Pensar en C++ *(Volumen 1)*

Traducción (cuasi-terminada) del libro Thinking in C++, Volumen 1

Bruce Eckel

12 de enero de 2012

Puede encontrar la versión actualizada de este libro e información adicional sobre el proyecto de traducción en
http://arco.esi.uclm.es/~david.villa/pensarC++.html

Pensar en C++ ***(Volumen 1)***
by Bruce Eckel

Copyright © 2000 Bruce Eckel

Índice general

1. Introducción a los Objetos **1**

1.1. El progreso de abstracción 1
1.2. Cada objeto tiene una interfaz 3
1.3. La implementación oculta 5
1.4. Reutilizar la implementación 6
1.5. Herencia: reutilización de interfaces 7
1.5.1. Relaciones es-un vs. es-como-un 10
1.6. Objetos intercambiables gracias al polimorfismo 11
1.7. Creación y destrucción de objetos 15
1.8. Gestión de excepciones: tratamiento de errores 16
1.9. Análisis y diseño 16
1.9.1. Fase 0: Hacer un plan 18
1.9.1.1. Declaración de objetivos 19
1.9.2. Fase 1: ¿Qué estamos haciendo? 19
1.9.3. Fase 2: ¿Cómo podemos construirlo? 22
1.9.3.1. Las cinco etapas del diseño de objetos 23
1.9.3.2. Directrices para desarrollo de objetos 24
1.9.4. Fase 3: Construir el núcleo 25
1.9.5. Fase 4: Iterar los casos de uso 25
1.9.6. Fase 5: Evolución 26
1.9.7. Los planes valen la pena 27
1.10. Programación Extrema 27
1.10.1. Escriba primero las pruebas 28
1.10.2. Programación en parejas 29
1.11. Porqué triunfa C++ 30
1.11.1. Un C mejor 30
1.11.2. Usted ya está en la curva de aprendizaje 30
1.11.3. Eficiencia 31
1.11.4. Los sistemas son más fáciles de expresar y entender 31

1.11.5. Aprovechamiento máximo con librerías 31
1.11.6. Reutilización de código fuente con plantillas 32
1.11.7. Manejo de errores 32
1.11.8. Programar a lo grande 32
1.12. Estrategias de transición 33
1.12.1. Directrices 33
1.12.1.1. Entrenamiento 33
1.12.1.2. Proyectos de bajo riesgo 33
1.12.1.3. Modelar desde el éxito 33
1.12.1.4. Use librerías de clases existentes 34
1.12.1.5. No reescriba en C++ código que ya existe 34
1.12.2. Obstáculos de la gestión 34
1.12.2.1. Costes iniciales 34
1.12.2.2. Cuestiones de rendimiento 35
1.12.2.3. Errores comunes de diseño 35
1.13. Resumen 36

2. Construir y usar objetos **37**
2.1. El proceso de traducción del lenguaje 37
2.1.1. Intérpretes 37
2.1.2. Compiladores 38
2.1.3. El proceso de compilación 39
2.1.3.1. Comprobación estática de tipos 40
2.2. Herramientas para compilación modular 40
2.2.1. Declaraciones vs definiciones 41
2.2.1.1. Sintaxis de declaración de funciones 41
2.2.1.2. Una puntualización 42
2.2.1.3. Definición de funciones 42
2.2.1.4. Sintaxis de declaración de variables 42
2.2.1.5. Incluir ficheros de cabecera 43
2.2.1.6. Formato de inclusión del estándar C++ 44
2.2.2. Enlazado 45
2.2.3. Uso de librerías 45
2.2.3.1. Cómo busca el enlazador una librería 46
2.2.3.2. Añadidos ocultos 46
2.2.3.3. Uso de librerías C plano 47
2.3. Su primer programa en C++ 47
2.3.1. Uso de las clases `iostream` 47

2.3.2. Espacios de nombres 48
2.3.3. Fundamentos de la estructura de los programa 49
2.3.4. «Hello, World!» 50
2.3.5. Utilizar el compilador 51
2.4. Más sobre iostreams 51
2.4.1. Concatenar vectores de caracteres 51
2.4.2. Leer de la entrada 52
2.4.3. Llamar a otros programas 53
2.5. Introdución a las cadenas 53
2.6. Lectura y escritura de ficheros 54
2.7. Introducción a los vectores 56
2.8. Resumen 59
2.9. Ejercicios 60

3. C en C++ **63**
3.1. Creación de funciones 63
3.1.1. Valores de retorno de las funciones 65
3.1.2. Uso de funciones de librerías C 66
3.1.3. Creación de librerías propias 66
3.2. Control de flujo 67
3.2.1. Verdadero y falso 67
3.2.2. `if-else` 67
3.2.3. `while` 68
3.2.4. `do-while` 69
3.2.5. `for` 70
3.2.6. Las palabras reservadas `break` y `continue` 70
3.2.7. `switch` 72
3.2.8. Uso y maluso de `goto` 73
3.2.9. Recursividad 74
3.3. Introducción a los operadores 74
3.3.1. Precedencia 75
3.3.2. Auto incremento y decremento 75
3.4. Introducción a los tipos de datos 76
3.4.1. Tipos predefinidos básicos 76
3.4.2. booleano, verdadero y falso 77
3.4.3. Especificadores 78
3.4.4. Introducción a punteros 79
3.4.5. Modificar objetos externos 82

3.4.6. Introducción a las referencias de C++ . . . 84
3.4.7. Punteros y Referencias como modificadores . . . 85
3.5. Alcance . . . 86
3.5.1. Definición de variables «al vuelo» . . . 87
3.6. Especificar la ubicación del espacio de almacenamiento . . . 89
3.6.1. Variables globales . . . 89
3.6.2. Variables locales . . . 90
3.6.2.1. Variables registro . . . 90
3.6.3. Static . . . 91
3.6.4. extern . . . 92
3.6.4.1. Enlazado . . . 93
3.6.5. Constantes . . . 93
3.6.5.1. Valores constantes . . . 94
3.6.6. Volatile . . . 95
3.7. Los operadores y su uso . . . 95
3.7.1. Asignación . . . 95
3.7.2. Operadores matemáticos . . . 96
3.7.2.1. Introducción a las macros del preprocesador . . . 97
3.7.3. Operadores relacionales . . . 97
3.7.4. Operadores lógicos . . . 97
3.7.5. Operadores para bits . . . 98
3.7.6. Operadores de desplazamiento . . . 98
3.7.7. Operadores unarios . . . 101
3.7.8. El operador ternario . . . 102
3.7.9. El operador coma . . . 102
3.7.10. Trampas habituales cuando se usan operadores . . . 103
3.7.11. Operadores de moldeado . . . 103
3.7.12. Los moldes explícitos de C++ . . . 104
3.7.12.1. `static_cast` . . . 105
3.7.12.2. `const_cast` . . . 106
3.7.12.3. `reinterpret_cast` . . . 107
3.7.13. `sizeof` - un operador en si mismo . . . 108
3.7.14. La palabra reservada `asm` . . . 108
3.7.15. Operadores explícitos . . . 109
3.8. Creación de tipos compuestos . . . 109
3.8.1. Creación de alias usando `typedef` . . . 109
3.8.2. Usar `struct` para combinar variables . . . 110
3.8.2.1. Punteros y estructuras . . . 112

3.8.3. Programas más claros gracias a `enum` . . . 112
3.8.3.1. Comprobación de tipos para enumerados . . . 114
3.8.4. Cómo ahorrar memoria con `union` . . . 114
3.8.5. Arrays . . . 115
3.8.5.1. Punteros y arrays . . . 117
3.8.5.2. El formato de punto flotante . . . 120
3.8.5.3. Aritmética de punteros . . . 121
3.9. Consejos para depuración . . . 124
3.9.1. Banderas para depuración . . . 124
3.9.1.1. Banderas de depuración para el preprocesador . . . 124
3.9.1.2. Banderas para depuración en tiempo de ejecución . . . 124
3.9.2. Convertir variables y expresiones en cadenas . . . 126
3.9.3. La macro C assert() . . . 126
3.10. Direcciones de función . . . 127
3.10.1. Definición de un puntero a función . . . 127
3.10.2. Declaraciones y definiciones complicadas . . . 128
3.10.3. Uso de un puntero a función . . . 129
3.10.4. Arrays de punteros a funciones . . . 129
3.11. Make: cómo hacer compilación separada . . . 130
3.11.1. Las actividades de Make . . . 131
3.11.1.1. Macros . . . 132
3.11.1.2. Reglas de sufijo . . . 132
3.11.1.3. Objetivos predeterminados . . . 133
3.11.2. Los Makefiles de este libro . . . 134
3.11.3. Un ejemplo de Makefile . . . 134
3.12. Resumen . . . 136
3.13. Ejercicios . . . 136

4. Abstracción de Datos **141**
4.1. Una librería pequeña al estilo C . . . 142
4.1.1. Asignación dinámica de memoria . . . 145
4.1.2. Malas suposiciones . . . 148
4.2. ¿Qué tiene de malo? . . . 149
4.3. El objeto básico . . . 150
4.4. ¿Qué es un objeto? . . . 156
4.5. Tipos abstractos de datos . . . 156
4.6. Detalles del objeto . . . 157
4.7. Conveciones para los ficheros de cabecera . . . 158

4.7.1. Importancia de los ficheros de cabecera . . . 158
4.7.2. El problema de la declaración múltiple . . . 160
4.7.3. Las directivas del preprocesador #define, #ifndef y #endif . . . 160
4.7.4. Un estándar para los ficheros de cabecera . . . 161
4.7.5. Espacios de nombres en los ficheros de cabecera . . . 162
4.7.6. Uso de los ficheros de cabecera en proyectos . . . 162
4.8. Estructuras anidadas . . . 163
4.8.1. Resolución de ámbito global . . . 166
4.9. Resumen . . . 166
4.10. Ejercicios . . . 167

5. Ocultar la implementación **171**
5.1. Establecer los límites . . . 171
5.2. Control de acceso en C++ . . . 172
5.2.1. `protected` . . . 173
5.3. Amigos (friends) . . . 173
5.3.1. Amigas anidadas . . . 175
5.3.2. ¿Es eso puro? . . . 177
5.4. Capa de objetos . . . 178
5.5. La clase . . . 178
5.5.1. Modificaciones en `Stash` para usar control de acceso . . . 181
5.5.2. Modificar `Stack` para usar control de acceso . . . 181
5.6. Manejo de clases . . . 182
5.6.1. Ocultar la implementación . . . 182
5.6.2. Reducir la recompilación . . . 183
5.7. Resumen . . . 185
5.8. Ejercicios . . . 185

6. Inicialización y limpieza **187**
6.1. Inicialización garantizada por el constructor . . . 188
6.2. Limpieza garantizada por el destructor . . . 189
6.3. Eliminación del bloque de definiciones . . . 191
6.3.1. Bucles `for` . . . 192
6.3.2. Alojamiento de memoria . . . 193
6.4. Stash con constructores y destructores . . . 194
6.5. Stack con constructores y destructores . . . 197
6.6. Inicialización de tipos agregados . . . 199
6.7. Constructores por defecto . . . 201
6.8. Resumen . . . 202

6.9. Ejercicios . . . 203

7. Sobrecarga de funciones y argumentos por defecto **205**

7.1. Más decoración de nombres . . . 206

7.1.1. Sobrecarga en el valor de retorno . . . 207

7.1.2. Enlace con FIXME:tipos seguros . . . 207

7.2. Ejemplo de sobrecarga . . . 208

7.3. Uniones . . . 211

7.4. Argumentos por defecto . . . 213

7.4.1. Argumentos de relleno . . . 214

7.5. Elección entre sobrecarga y argumentos por defecto . . . 215

7.6. Resumen . . . 219

7.7. Ejercicios . . . 220

8. Constantes **223**

8.1. Sustitución de valores . . . 223

8.1.1. `const` en archivos de cabecera . . . 224

8.1.2. constantes seguras . . . 225

8.1.3. Vectores . . . 226

8.1.4. Diferencias con C . . . 226

8.2. Punteros . . . 228

8.2.1. Puntero a constante . . . 228

8.2.2. Puntero constante . . . 228

8.2.2.1. Formato . . . 229

8.2.3. Asignación y comprobación de tipos . . . 229

8.2.3.1. Literales de cadena . . . 230

8.3. Argumentos de funciones y valores de retorno . . . 230

8.3.1. Paso por valor constante . . . 231

8.3.2. Retorno por valor constante . . . 231

8.3.2.1. Temporarios . . . 233

8.3.3. Paso y retorno de direcciones . . . 234

8.3.3.1. Criterio de paso de argumentos . . . 235

8.4. Clases . . . 236

8.4.1. `const` en las clases . . . 237

8.4.1.1. La lista de inicialización del constructor. . . . 237

8.4.1.2. Constructores para los tipos del lenguaje . . . 238

8.4.2. Constantes en tiempo de compilación dentro de clases . . . 239

8.4.2.1. El enumerado en codigo antiguo . . . 240

8.4.3. Objetos y métodos constantes . . . 241

8.4.3.1. mutable: constancia binaria vs. lógica 244
8.4.3.2. ROMability 245
8.5. Volatile 245
8.6. Resumen 247
8.7. Ejercicios 247

9. Funciones `inline` 251
9.1. Los peligros del preprocesador 251
9.1.1. Macros y acceso 254
9.2. Funciones `inline` 254
9.2.1. inline dentro de clases 255
9.2.2. Funciones de acceso 256
9.2.2.1. Accesores y mutadores 257
9.3. Stash y Stack con inlines 261
9.4. Funciones inline y el compilador 264
9.4.1. Limitaciones 265
9.4.2. Referencias adelantadas 265
9.4.3. Actividades ocultas en contructores y destructores 266
9.5. Reducir el desorden 267
9.6. Más características del preprocesador 268
9.6.1. Encolado de símbolos 269
9.7. Comprobación de errores mejorada 269
9.8. Resumen 272
9.9. Ejercicios 272

10. Control de nombres 275
10.1. Los elementos estáticos de C 275
10.1.1. Variables estáticas dentro de funciones 275
10.1.1.1. Objetos estáticos dentro de funciones 277
10.1.1.2. Destructores de objetos estáticos 277
10.1.2. Control del enlazado 279
10.1.2.1. Confusión 280
10.1.3. Otros especificadores para almacenamiento de clases 281
10.2. Espacios de nombres 281
10.2.1. Crear un espacio de nombres 281
10.2.1.1. Espacios de nombres sin nombre 282
10.2.1.2. Amigas 283
10.2.2. Cómo usar un espacio de nombres 283
10.2.2.1. Resolución del ámbito 283

10.2.2.2. La directiva `using` . . . 284
10.2.2.3. La declaración `using` . . . 286
10.2.3. El uso de los espacios de nombres . . . 287
10.3. Miembros estáticos en C++ . . . 287
10.3.1. Definición del almacenamiento para atributos estáticos . . . 288
10.3.1.1. Inicialización de vectores estáticos . . . 289
10.3.2. Clases anidadas y locales . . . 291
10.3.3. Métodos estáticos . . . 292
10.4. Dependencia en la inicialización de variables estáticas . . . 294
10.4.1. Qué hacer . . . 296
10.4.1.1. Técnica uno . . . 296
10.4.1.2. Técnica dos . . . 298
10.5. Especificaciones de enlazado alternativo . . . 302
10.6. Resumen . . . 302
10.7. Ejercicios . . . 303

11. Las referencias y el constructor de copia **307**
11.1. Punteros en C++ . . . 307
11.2. Referencias en C++ . . . 308
11.2.1. Referencias en las funciones . . . 308
11.2.1.1. Referencias constantes . . . 309
11.2.1.2. Referencias a puntero . . . 310
11.2.2. Consejos para el paso de argumentos . . . 311
11.3. El constructor de copia . . . 311
11.3.1. Paso y retorno por valor . . . 311
11.3.1.1. Paso y retorno de objetos grandes . . . 312
11.3.1.2. Marco de pila para llamadas a función . . . 313
11.3.1.3. Re-entrada . . . 313
11.3.1.4. Copia bit a bit vs. inicialización . . . 314
11.3.2. Construcción por copia . . . 316
11.3.2.1. Objetos temporales . . . 320
11.3.3. El constructor de copia por defecto . . . 320
11.3.4. Alternativas a la construcción por copia . . . 322
11.3.4.1. Evitando el paso por valor . . . 322
11.3.4.2. Funciones que modifican objetos externos . . . 323
11.4. Punteros a miembros . . . 323
11.4.1. Funciones . . . 325
11.4.1.1. Un ejemplo . . . 326

11.5. Resumen 327
11.6. Ejercicios 328

12. Sobrecarga de operadores **331**
12.1. Precaución y tranquilidad 331
12.2. Sintaxis 332
12.3. Operadores sobrecargables 333
12.3.1. Operadores unarios 333
12.3.1.1. Incremento y decremento 336
12.3.2. Operadores binarios 337
12.3.3. Argumentos y valores de retorno 346
12.3.3.1. Retorno por valor como constante 347
12.3.3.2. Optimización del retorno 347
12.3.4. Operadores poco usuales 348
12.3.4.1. El operador coma 348
12.3.4.2. El operador -> 349
12.3.4.3. Un operador anidado 351
12.3.4.4. Operador ->* 352
12.3.5. Operadores que no puede sobrecargar 354
12.4. Operadores no miembros 355
12.4.1. Directrices básicas 356
12.5. Sobrecargar la asignación 357
12.5.1. Comportamiento del `operador =` 358
12.5.1.1. Punteros en clases 359
12.5.1.2. Contabilidad de referencias 361
12.5.1.3. Creación automática del operador = ... 365
12.6. Conversión automática de tipos 366
12.6.1. Conversión por constructor 366
12.6.1.1. Evitar la conversión por constructor 367
12.6.2. Conversión por operador 368
12.6.2.1. Reflexividad 368
12.6.3. Ejemplo de conversión de tipos 370
12.6.4. Las trampas de la conversión automática de tipos 371
12.6.4.1. Actividades ocultas 372
12.7. Resumen 373
12.8. Ejercicios 373

13. Creación dinámica de objetos **377**
13.1. Creación de objetos 378

13.1.1. Asignación dinámica en C . . . 378
13.1.2. El operador `new` . . . 380
13.1.3. El operador `delete` . . . 380
13.1.4. Un ejemplo sencillo . . . 381
13.1.5. Trabajo extra para el gestor de memoria . . . 381
13.2. Rediseño de los ejemplos anteriores . . . 382
13.2.1. `delete` void* probablemente es un error . . . 382
13.2.2. Responsabilidad de la limpieza cuando se usan punteros . . . 383
13.2.3. Stash para punteros . . . 384
13.2.3.1. Una prueba . . . 386
13.3. `new` y `delete` para vectores . . . 388
13.3.1. Cómo hacer que un puntero sea más parecido a un vector . . . 389
13.3.2. Cuando se supera el espacio de almacenamiento . . . 389
13.3.3. Sobrecarga de los operadores `new` y `delete` . . . 390
13.3.3.1. Sobrecarga global de `new` y `delete` . . . 391
13.3.3.2. Sobrecarga de `new` y `delete` específica para una clase . . . 393
13.3.3.3. Sobrecarga de `new` y `delete` para vectores . . . 395
13.3.3.4. Llamadas al constructor . . . 397
13.3.3.5. Operadores `new` y `delete` de [FIXME emplazamiento (situación)] . . . 398
13.4. Resumen . . . 400
13.5. Ejercicios . . . 400

14. Herencia y Composición **403**
14.1. Sintaxis de la composición . . . 403
14.2. Sintaxis de la herencia . . . 405
14.3. Lista de inicializadores de un constructor . . . 406
14.3.1. Inicialización de objetos miembros . . . 407
14.3.2. Tipos predefinidos en la lista de inicializadores . . . 407
14.3.3. Combinación de composición y herencia . . . 408
14.3.3.1. Llamadas automáticas al destructor . . . 409
14.3.4. Orden de llamada de constructores y destructores . . . 409
14.4. Ocultación de nombres . . . 411
14.5. Funciones que no heredan automáticamente . . . 414
14.5.1. Herencia y métodos estáticos . . . 417
14.5.2. Composición vs. herencia . . . 417
14.5.2.1. Subtipado . . . 419
14.5.2.2. Herencia privada . . . 421

14.5.2.2.1. Publicar los miembros heredados de forma privada . . . 421
14.6. Protected . . . 422
14.6.1. Herencia protegida . . . 422
14.7. Herencia y sobrecarga de operadores . . . 423
14.8. Herencia múltiple . . . 424
14.9. Desarrollo incremental . . . 424
14.10Upcasting . . . 425
14.10.1.¿Por qué «upcasting»? . . . 426
14.10.2.FIXME Upcasting y el constructor de copia . . . 426
14.10.3.Composición vs. herencia FIXME (revisited) . . . 429
14.10.4.FIXME Upcasting de punteros y referencias . . . 430
14.10.5.Una crisis . . . 430
14.11Resumen . . . 431
14.12Ejercicios . . . 431

15. Polimorfismo y Funciones virtuales **435**
15.1. Evolución de los programadores de C++ . . . 435
15.2. Upcasting . . . 436
15.3. El problema . . . 437
15.3.1. Ligadura de las llamadas a funciones . . . 437
15.4. Funciones virtuales . . . 438
15.4.1. Extensibilidad . . . 439
15.5. Cómo implementa C++ la ligadura dinámica . . . 441
15.5.1. Almacenando información de tipo . . . 442
15.5.2. Pintar funciones virtuales . . . 443
15.5.3. Detrás del telón . . . 445
15.5.4. Instalar el vpointer . . . 446
15.5.5. Los objetos son diferentes . . . 446
15.6. ¿Por qué funciones virtuales? . . . 447
15.7. Clases base abstractas y funciones virtuales puras . . . 448
15.7.1. Definiciones virtuales puras . . . 452
15.8. Herencia y la VTABLE . . . 453
15.8.1. FIXME: Object slicing . . . 455
15.9. Sobrecargar y redefinir . . . 457
15.9.1. Tipo de retorno variante . . . 458
15.10funciones virtuales y constructores . . . 460
15.10.1.Orden de las llamadas a los constructores . . . 460

15.10.2.Comportamiento de las funciones virtuales dentro de los constructores . . . 461
15.10.3.Destructores y destructores virtuales . . . 462
15.10.4.Destructores virtuales puros . . . 463
15.10.5.Mecanismo virtual en los destructores . . . 465
15.10.6.Creación una jerarquía basada en objetos . . . 466
15.11Sobrecarga de operadores . . . 469
15.12Downcasting . . . 471
15.13Resumen . . . 473
15.14Ejercicios . . . 474

16. Introducción a las Plantillas **479**
16.1. Contenedores . . . 479
16.1.1. La necesidad de los contenedores . . . 481
16.2. Un vistazo a las plantillas . . . 481
16.2.1. La solución de la plantilla . . . 483
16.3. Sintaxis del Template . . . 484
16.3.1. Definiciones de función no inline . . . 486
16.3.1.1. Archivos cabecera . . . 486
16.3.2. IntStack como plantilla . . . 487
16.3.3. Constantes en los Templates . . . 488
16.4. Stack y Stash como Plantillas . . . 490
16.4.1. Cola de punteros mediante plantillas . . . 492
16.5. Activando y desactivando la propiedad . . . 496
16.6. Manejando objetos por valor . . . 498
16.7. Introducción a los iteradores . . . 500
16.7.1. Stack con iteradores . . . 507
16.7.2. PStash con iteradores . . . 510
16.8. Por qué usar iteradores . . . 515
16.8.1. Plantillas Función . . . 517
16.9. Resumen . . . 518
16.10Ejercicios . . . 519

A. Estilo de codificación **523**
A.1. General . . . 523
A.2. Nombres de fichero . . . 524
A.3. Marcas comentadas de inicio y fin . . . 524
A.4. Paréntesis, llaves e indentación . . . 525
A.5. Nombres para identificadores . . . 528

A.6. Orden de los #includes 529
A.7. Guardas de inclusión en ficheros de cabecera 529
A.8. Uso de los espacios de nombres 529
A.9. Utilización de `require()` y `assure()` 530

B. Directrices de Programación **531**

C. Lecturas recomendadas **541**
C.1. Sobre C 541
C.2. Sobre C++ en general 541
C.2.1. Mi propia lista de libros 541
C.3. Los rincones oscuros 542
C.4. Sobre Análisis y Diseño 543

Índice de figuras

1. Introducción a los Objetos

1.1. Clase `Luz` 4
1.2. Un coche tiene un motor 6
1.3. subclases 7
1.4. Jerarquía de `Figura` 8
1.5. Especialización de `Figura` 9
1.6. Reescritura de métodos 10
1.7. Relaciones 11
1.8. Polimorfismo 12
1.9. Upcasting 14
1.10. Diagramas de casos de uso 20
1.11. Un ejemplo de caso de uso 21

11. Las referencias y el constructor de copia

11.1. Llamada a una función 313

14. Herencia y Composición

14.1. Upcasting 426

15. Polimorfismo y Funciones virtuales

15.1. Funciones virtuales 443
15.2. Tabla de punteros virtuales 444
15.3. Clase abstracta 449
15.4. Una nueva función virtual 454
15.5. Object slicing 456

16. Introducción a las Plantillas

16.1. Contenedores 483

16.2. Herencia múltiple 483
16.3. Contenedor de objetos `Figura` 484

Índice de cuadros

3. C en C++

3.1. Expresiones que utilizan booleanos 77
3.2. Moldes explícitos de C++ 105
3.3. Nuevas palabras reservadas para operadores booleanos 109

12. Sobrecarga de operadores

12.1. Directrices para elegir entre miembro y no-miembro 357

Prólogo a la traducción

Este trabajo de traducción ha sido realizado íntegramente por voluntarios. Le agradecemos que nos comunique cualquier error de traducción o transcripción en el texto. También será bienvenido si desea colaborar más activamente en la traducción. Ayúdenos a hacer de esta traducción un trabajo de calidad.

Si desea saber más sobre este proyecto, obtener el segundo volumen, colaborar enviando informes de fallos, traduciendo o revisando, etc. visite la página web[1] o nuestro grupo Google[2].

El trabajo de traducción de este volumen prácticamente ha terminado, pero es posible que todavía queden muchos errores debido a que la revisión es trabajosa y contamos con pocos voluntarios. Le agradecemos su colaboración para corregir posibles erratas o fallos de cualquier tipo. En todo caso, el libro está completo y es perfectamente útil en su estado actual.

Este prólogo no forma parte del libro original y ha sido incluido como reseña y referencia de los trabajos de traducción que se han llevado a cabo. Este capítulo no lo daré por terminado hasta que concluya el proceso de traducción y revisión de este volumen al menos. La traducción del Volumen 2 ya está en marcha.

Licencia y normas de distribución

El equipo de traducción ha seguido al pié de la letra las directrices marcadas por Bruce Eckel, autor de *Thinking in C++* (el libro original), para la realización de traducciones y distribución de éstas. Si utiliza o distribuye este texto debe cumplirlas y advertir de su existencia a los posibles lectores. El equipo de traducción elude toda responsabilidad por la violación (por parte de terceros) de las citadas directrices[3]. Se incluyen a continuación respetando el idioma original para evitar eventuales interpretaciones incorrectas:

> In my contract with the publisher, I maintain all electronic publishing rights to the book, including translation into foreign languages. This means that the publisher still handles negotiations for translations that are printed (and I have nothing directly to do with that) but I may grant translation rights for electronic versions of the book.
>
> I have been granting such rights for «open-source» style translation projects. (Note that I still maintain the copyright on my material.) That is:
>
> - You must provide a web site or other medium whereby people may

[1] http://arco.esi.uclm.es/~david.villa/pensarC++.html
[2] http://groups.google.com/group/pensar-en-cpp
[3] El texto original de estas directrices está accesible en la página web del autor.

participate in the project (two easy possibilities are http://www.egroups.com or http://www.topica.com).

- You must maintain a downloadable version of the partially or fully translated version of the book.
- Someone must be responsible for the organization of the translation (I cannot be actively involved - I don't have the time).
- There should only be one language translation project for each book. We don't have the resources for a fork.
- As in an open-source project, there must be a way to pass responsibility to someone else if the first person becomes too busy.
- The book must be freely distributable.
- The book may be mirrored on other sites.
- Names of the translators should be included in the translated book.

Tecnicismos

Se han traducido la mayor parte de los términos específicos tanto de orientación a objetos como de programación en general. Para evitar confusiones o ambigüedades a los lectores que manejen literatura en inglés hemos incluido entre paréntesis el término original la primera vez que aparece traducido.

Para traducir tecnicismos especialmente complicados hemos utilizado como referencia la segunda edición de *El lenguaje de Programación C++* (en castellano) así como la Wikipedia.

En contadas ocasiones se ha mantenido el término original en inglés. En beneficio de la legibilidad, hemos preferido no hacer traducciones demasiado forzadas ni utilizar expresiones que pudieran resultar desconocidas en el argot o en los libros especializados disponibles en castellano. Nuestro propósito es tener un libro que pueda ser comprendido por hispano-hablantes. Es a todas luces imposible realizar una traducción rigurosa acorde con las normas lingüísticas de la RAE, puesto que, en algunos casos, el autor incluso utiliza palabras de su propia invención.

Código fuente

Por hacer

Producción

Todo el proceso de traducción, edición, formato y tipografía ha sido realizado íntegramente con software libre. Todo el software utilizado está disponible en la distribución Debian GNU/Linux, que es la que se ha utilizado principalmente para la actualización y mantenimiento de los documentos obtenidos como resultado.

El texto ha sido escrito en el lenguaje de marcado DocBook versión 4.5 en su variante XML. Cada capítulo está contenido en un fichero independiente y todos ellos se incluyen en un fichero «maestro» utilizando XInclude.

Debido a que muchos procesadores de DocBook no soportan adecuadamente la característica XInclude, se usa la herramienta xsltproc[4] para generar un único fiche-

[4] http://xmlsoft.org/XSLT/xsltproc2.html

ro XML que contiene el texto de todo el libro, y es ese fichero resultante el que se procesa.

Código fuente

También se utiliza XInclude para añadir en su lugar el contenido de los ficheros de código fuente escritos en C++. De ese modo, el texto de los listados que aparecen en el libro es idéntico a los ficheros C++ que distribuye el autor. De ese modo, la edición es mucha más limpia y sobretodo se evitan posibles errores de transcripción de los listados.

Utilizando un pequeño programa escrito en lenguaje Python[5], se substituyen los nombres etiquetados de los ficheros por la sentencia XInclude correspondiente:

```
//: V1C02:Hello.cpp
```

pasa a ser:

```
<example>
  <title>C02/Hello.cpp</title>
  <programlisting language="C++">
    <xi:include parse="text" href="./code_v1/C02/Hello.cpp"/>
  </programlisting>
</example>
```

Una ver realizada esta substitución, se utiliza de nuevo **xsltproc** para montar tanto el texto como los listados en un único fichero XML.

Convenciones tipográficas

- Palabras reservadas: `struct`
- Código fuente: `printf("Hello world");`
- Nombres de ficheros: `fichero.cpp`
- Aplicación o fichero binario: `make`
- Entrecomillado: «upcasting»

Esquemas y diagramas

Los dibujos y diagramas originales se han rehecho en formato `.svg` usando la herramienta inkscape[6]. A partir del fichero fuente `.svg` se generan versiones en formato `.png` para la versión HTML y `.pdf` para la versión PDF.

Generación de productos

A partir del documento completo en formato DocBook se generan dos resultados distintos;

[5] `./utils/fix_includes.py`
[6] http://inkscape.org/

HTML en una sola página Una página web XHTML. Para ello se utiliza también la herramienta **xsltproc** aplicando hojas de estilo XSLT que pueden encontrarse en el repositorio de fuentes del proyecto. Estas plantillas son modificaciones de las del proyecto de documentación del programa «The Gimp», que tienen licencia GPL.

Para el coloreado de los listados de código fuente se ha utilizado el programa **highlight**. Para ello, un pequeño programa Python marca los listados para su extracción, a continuación se colorean y por último se vuelven a insertar en la página HTML.

HTML (una página por sección) Un conjunto de páginas XHTML. Automáticamente se generan enlaces para navegar por el documento y tablas de contenidos.

PDF Un documento en formato PDF utilizando la aplicación dblatex[7]. Ha sido necesario crear una hoja de estilo específicamente para manipular el formato de página, títulos e índices. Para el resalte de sintaxis de los listados se ha utilizado el paquete LaTeX listings.

El equipo

Las siguientes personas han colaborado en mayor o menor medida en algún momento desde el comienzo del proyecto de traducción de *Pensar en C++*:

- David Villa Alises (coordinador) dvilla#gmx.net
- Míguel Ángel García miguelangel.garcia#gmail.com
- Javier Corrales García jcg#damir.iem.csic.es
- Bárbara Teruggi bwire.red#gmail.com
- Sebastián Gurin
- Gloria Barberán González globargon#gmail.com
- Fernando Perfumo Velázquez nperfumo#telefonica.net
- José María Gómez josemaria.gomez#gmail.com
- David Martínez Moreno ender#debian.org
- Cristóbal Tello ctg#tinet.org
- Jesús López Mollo (pre-Lucas)
- José María Requena López (pre-Lucas)
- Javier Fenoll Rejas (pre-Lucas)

Agradecimientos

Por hacer: LuCAS, spanglish@uma.es, docbook-ayuda@es.tldp.org

Utilidades

[7] http://dblatex.sourceforge.net/

Prefacio

Como cualquier lenguaje humano, C++ proporciona métodos para expresar conceptos. Si se utiliza de forma correcta, este medio de expresión será significativamente más sencillo y flexible que otras alternativas cuando los problemas aumentan en tamaño y complejidad.

No se puede ver C++ sólo como un conjunto de características, ya que algunas de esas características no tienen sentido por separado. Sólo se puede utilizar la suma de las partes si se está pensando en el diseño, no sólo en el código. Y para entender C++ de esta forma, se deben comprender los problemas existentes con C y con la programación en general. Este libro trata los problemas de programación, porque son problemas, y el enfoque que tiene C++ para solucionarlos. Además, el conjunto de características que explico en cada capítulo se basará en la forma en que yo veo un tipo de problema en particular y cómo resolverlo con el lenguaje. De esta forma espero llevar al lector, poco a poco, de entender C al punto en el que C++ se convierta en su propia lengua.

Durante todo el libro, mi actitud será pensar que el lector desea construir en su cabeza un modelo que le permita comprender el lenguaje bajando hasta sus raíces; si se tropieza con un rompecabezas, será capaz de compararlo con su modelo mental y deducir la respuesta. Trataré de comunicarle las percepciones que han reorientado mi cerebro para «Pensar en C++».

Material nuevo en la segunda edición

Este libro es una minuciosa reescritura de la primera edición para reflejar todos los cambios que han aparecido en C++ tras la finalización del estándar que lo rige, y también para reflejar lo que he aprendido desde que escribí la primera edición. He examinado y reescrito el texto completo, en ocasiones quitando viejos ejemplos, a veces cambiándolos, y también añadiendo muchos ejercicios nuevos. La reorganización y reordenación del material tuvo lugar para reflejar la disponibilidad de mejores herramientas, así como mi mejor comprensión de cómo la gente aprende C++. He añadido un nuevo capítulo, como introducción al resto del libro, una introducción rápida a los conceptos de C y a las características básicas de C++ para aquellos que no tienen experiencia en C. El CD-ROM incluido al final del libro en la edición en papel contiene un seminario: una introducción aún más ligera a los conceptos de C necesarios para comprender C++ (o Java). Chuck Allison lo escribió para mi empresa (MindView, Inc.), y se llama «Pensar en C: conceptos básicos de Java y C++». Presenta los aspectos de C que necesita conocer para poder cambiar a C++ o Java, abandonando los desagradables bits de bajo nivel con los que los programadores de C tratan a diario, pero que lenguajes como C++ y Java mantienen lejos (o

incluso eliminan, en el caso de Java).

Así que la respuesta corta a la pregunta «¿Qué es diferente en la segunda edición?» sería que aquello que no es completamente nuevo se ha reescrito, a veces hasta el punto en el que no podría reconocer los ejemplos y el material original de la primera edición.

¿Qué contiene el volumen 2 de este libro?

Con la conclusión del estándar de C++ también se añadieron algunas importantes bibliotecas nuevas, tales como string y los contenedores, y algoritmos de la Librería Estándar C++, y también se ha añadido complejidad a las plantillas. Éstos y otros temas más avanzados se han relegado al volumen 2 de este libro, incluyendo asuntos como la herencia múltiple, el manejo de excepciones, patrones de diseño, y material sobre la creación y depuración de sistemas estables.

Cómo obtener el volumen 2

Del mismo modo que el libro que lee en estos momentos, *Pensar en C++, Volumen 2* se puede descargar desde mi sitio web www.BruceEckel.com. Puede encontrar información en el sitio web sobre la fecha prevista para la impresión del Volumen 2.

El sitio web también contiene el código fuente de los listados para ambos libros, junto con actualizaciones e información sobre otros seminarios en CD-ROM que ofrece MidView Inc., seminarios públicos y formación interna, consultas, soporte y asistentes paso a paso.

Requisitos

En la primera edición de este libro, decidí suponer que otra persona ya le había enseñado C y que el lector tenía, al menos, un nivel aceptable de lectura del mismo. Mi primera intención fue hablar de lo que me resultó difícil: el lenguaje C++. En esta edición he añadido un capítulo como introducción rápida a C, acompañada del seminario en-CD *Thinking in C*, pero sigo asumiendo que el lector tiene algún tipo de experiencia en programación. Además, del mismo modo que se aprenden muchas palabras nuevas intuitivamente, viéndolas en el contexto de una novela, es posible aprender mucho sobre C por el contexto en el que se utiliza en el resto del libro.

Aprender C++

Yo me adentré en C++ exactamente desde la misma posición en la que espero que se encuentren muchos de los lectores de este libro: como un programador con una actitud muy sensata y con muchos vicios de programación. Peor aún, mi experiencia era sobre porgramación de sistemas empotrados a nivel hardware, en la que a veces se considera a C como un *lenguaje de alto nivel* y excesivamente ineficiente para ahorrar bits. Descubrí más tarde que nunca había sido un buen programador en C, camuflando así mi ignorancia sobre estructuras, `malloc()` y `free()`, `setjmp()` y `longjmp()`, y otros conceptos *sofisticados*, y muriéndome de vergüenza cuando estos términos entraban en una conversación, en lugar de investigar su utilidad.

Cuando comencé mi lucha por aprender C++, el único libro decente era la auto-

proclamada *Guía de expertos* de Bjarne Stroustrup [8] así que simplifiqué los conceptos básicos por mí mismo. Esto se acabó convirtiendo en mi primer libro de C++ [9] , que es esencialmente un reflejo de mi experiencia. Fue descrita como una guía de lectura para atraer a los programadores a C y C++ al mismo tiempo. Ambas ediciones [10] del libro consiguieron una respuesta entusiasta.

Más o menos al mismo tiempo que aparecía *Using C++*, comencé a enseñar el lenguaje en seminarios y presentaciones. Enseñar C++ (y más tarde, Java) se convirtió en mi profesión; llevo viendo cabezas asintiendo, caras pálidas, y expresiones de perplejidad en audiencias por todo el mundo desde 1989. Cuando comencé a dar formación interna a grupos más pequeños, descubrí algo durante los ejercicios. Incluso aquella gente que estaba sonriendo y asintiendo se encontraba equivocada en muchos aspectos. Creando y dirigiendo las pruebas de C++ y Java durante muchos años en la Conferencia de Desarrollo de Software, descubrí que tanto otros oradores como yo tendíamos a tocar demasiados temas, y todo demasiado rápido. Así que, de vez en cuando, a pesar de la variedad del nivel de la audiencia e independientemente de la forma en que se presentara el material, terminaría perdiendo alguna parte de mi público. Quizá sea pedir demasiado, pero como soy una de esas personas que se resisten a una conferencia tradicional (y para la mayoría de las personas, creo, esta resistencia está causada por el aburrimiento), quise intentar mantener a cada uno a su velocidad.

Durante un tiempo, estuve haciendo presentaciones en orden secuencial. De ese modo, terminé por aprender experimentando e iterando (una técnica que también funciona bien en el diseño de programas en C++). Al final, desarrollé un curso usando todo lo que había aprendido de mi experiencia en la enseñanza. Así, el aprendizaje se realiza en pequeños pasos, fáciles de digerir, y de cara a un seminario práctico (la situación ideal para el aprendizaje) hay ejercicios al final de cada presentación. Puede encontrar mis seminarios públicos en www.BruceEckel.com, y también puede aprender de los seminarios que he pasado a CD-ROM.

La primera edición de este libro se gestó a lo largo de dos años, y el material de este libro se ha usado de muchas formas y en muchos seminarios diferentes. Las reacciones que he percibido de cada seminario me han ayudado a cambiar y reorientar el material hasta que he comprobado que funciona bien como un medio de enseñanza. Pero no es sólo un manual para dar seminarios; he tratado de recopilar tanta información como he podido en estas páginas, intentando estructurarlas para atraer al lector hasta la siguiente materia. Más que nada, el libro está diseñado para servir al lector solitario que lucha con un lenguaje de programación nuevo.

Objetivos

Mis objetivos en este libro son:

1. Presentar el material paso a paso, de manera que el lector pueda digerir cada concepto fácilmente antes de continuar.

2. Usar ejemplos tan simples y cortos como sea posible. Esto a veces me impide manejar problemas del *mundo real*, pero he descubierto que los principiantes

[8] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986 (first edition).

[9] *Using C++*, Osborne/McGraw-Hill 1989.

[10] *Using C++ and C++ Inside & Out*, Osborne/McGraw-Hill 1993.

normalmente quedan más contentos cuando pueden comprender cada detalle de un ejemplo que siendo impresionados por el ámbito del problema que soluciona. Además, hay un límite en la cantidad de código que se puede asimilar en una clase. Por ello, a veces recibo críticas por usar *ejemplos de juguete*, pero tengo la buena voluntad de aceptarlas en favor de producir algo pedagógicamente útil.

3. La cuidadosa presentación secuencial de capacidades para que no se vea algo que no ha sido explicado. De acuerdo, esto no siempre es posible; en esos casos, se ofrece una breve descripción introductoria.

4. Indicarle lo que creo que es importante para que se comprenda el lenguaje, más que todo lo que sé. Creo que hay una "jerarquía de la importancia de la información", y hay algunos hechos que el 95 por ciento de los programadores nunca necesitará saber y que sólo podrían confundirles y afianzar su percepción de la complejidad del lenguaje. Tomando un ejemplo de C, si memoriza la tabla de precedencia de los operadores (yo nunca lo hice), puede escribir código más corto. Pero si lo piensa, esto confundirá al lector/mantenedor de ese código. Así que olvide la precedencia, y utilice paréntesis cuando las cosas no estén claras. Esta misma actitud la utilizaré con alguna otra información del lenguaje C++, que creo que es más importante para escritores de compiladores que para programadores.

5. Mantener cada sección suficientemente enfocada como para que el tiempo de lectura -y el tiempo entre bloques de ejercicios- sea razonable. Eso mantiene las mentes de la audiencia más activas e involucradas durante un seminario práctico, y además le da al lector una mayor sensación de avance.

6. Ofrecer a los lectores una base sólida de manera que puedan comprender las cuestiones lo suficientemente bien como para pasar a otros cursos y libros más difíciles (en concreto, el Volumen 2 de este libro).

7. He tratado de no utilizar ninguna versión de C++ de ningún proveedor en particular porque, para aprender el lenguaje, no creo que los detalles de una implementación concreta sean tan importantes como el lenguaje mismo. La documentación sobre las especificaciones de implementación propia de cada proveedor suele ser adecuada.

Capítulos

C++ es un lenguaje en el que se construyen características nuevas y diferentes sobre una sintaxis existente (por esta razón, nos referiremos a él como un lenguaje de programación orientado a objetos híbrido). Como mucha gente pasa por una curva de aprendizaje, hemos comenzado por adaptarnos a la forma en que los programadores pasan por las etapas de las cualidades del lenguaje C++. Como parece que la progresión natural es la de una mente entrenada de forma procedural, he decidido comprender y seguir el mismo camino y acelerar el proceso proponiendo y resolviendo las preguntas que se me ocurrieron cuando yo aprendía el lenguaje y también las que se les ocurrieron a la gente a la que lo enseñaba.

El curso fue diseñado con algo en mente: hacer más eficiente el proceso de aprender C++. La reacción de la audiencia me ayudó a comprender qué partes eran difíciles y necesitaban una aclaración extra. En las áreas en las que me volvía ambicioso e incluía demasiadas cosas de una vez, me dí cuenta -mediante la presentación de

material- de que si incluyes demasiadas características, tendrás que explicarlas todas, y es fácil que la confusión de los estudiantes se agrave. Como resultado, he tenido muchos problemas para introducir las características tan lentamente como ha sido posible; idealmente, sólo un concepto importante a la vez por capítulo.

Así pues, el objetivo en cada capítulo es enseñar un concepto simple, o un pequeño grupo de conceptos asociados, en caso de que no haya más conceptos adicionales. De esa forma puede digerir cada parte en el contexto de su conocimiento actual antes de continuar. Para llevarlo a cabo, dejé algunas partes de C para más adelante de lo que me hubiese gustado. La ventaja es que se evita la confusión al no ver todas las características de C++ antes de que éstas sean explicadas, así su introducción al lenguaje será tranquila y reflejará la forma en que asimile las características que dejo en sus manos.

He aquí una breve descripción de los capítulos que contiene este libro:

Capítulo 1: Introducción a los objetos. Cuando los proyectos se vuelven demasiado grandes y difíciles de mantener, nace la «crisis del software», que es cuando los programadores dicen: «¡No podemos terminar los proyectos, y cuando podemos, son demasiado caros!». Eso provocó gran cantidad de reacciones, que se discuten en este capítulo mediante las ideas de Programación Orientada a Objetos (POO) y cómo intenta ésta resolver la crisis del software. El capítulo le lleva a través de las características y conceptos básicos de la POO y también introduce los procesos de análisis y diseño. Además, aprenderá acerca de los beneficios y problemas de adaptar el lenguaje, y obtendrá sugerencias para adentrarse en el mundo de C++.

Capítulo 2: Crear y usar objetos. Este capítulo explica el proceso de construir programas usando compiladores y librerías. Presenta el primer programa C++ del libro y muestra cómo se construyen y compilan los programas. Después se presentan algunas de las librerías de objetos básicas disponibles en C++ Estándar. Para cuando acabe el capítulo, dominará lo que se refiere a escribir un programa C++ utilizando las librerías de objetos predefinidas.

Capítulo 3: El C de C++. Este capítulo es una densa vista general de las características de C que se utilizan en C++, así como gran número de características básicas que sólo están disponibles en C++. Además introduce la utilidad make, que es habitual en el desarrollo software de todo el mundo y que se utiliza para construir todos los ejemplos de este libro (el código fuente de los listados de este libro, que está disponible en www.BruceEckel.com, contiene los `makefiles` correspondientes a cada capítulo). En el capítulo 3 supongo que el lector tiene unos conocimientos básicos sólidos en algún lenguaje de programación procedural como Pascal, C, o incluso algún tipo de Basic (basta con que haya escrito algo de código en ese lenguaje, especialmente funciones). Si encuentra este capítulo demasiado difícil, debería mirar primero el seminario *Pensar en C* del CD que acompaña este libro (también disponible en www.BruceEckel.com).

Capítulo 4: Abstracción de datos. La mayor parte de las características de C++ giran entorno a la capacidad de crear nuevos tipos de datos. Esto no sólo ofrece una mayor organización del código, también es la base preliminar para las capacidades de POO más poderosas. Verá cómo esta idea es posible por el simple hecho de poner funciones dentro de las estructuras, los detalles de cómo hacerlo, y qué tipo de código se escribe. También aprenderá la mejor manera de organizar su código mediante archivos de cabecera y archivos de implementación.

Capítulo 5: Ocultar la implementación. El programador puede decidir que algunos de los datos y funciones de su estructura no estén disponibles para el usuario del nuevo tipo haciéndolas *privadas*. Eso significa que se puede separar la implementación principal de la interfaz que ve el programador cliente, y de este modo permitir

que la implementación se pueda cambiar fácilmente sin afectar al código del cliente. La palabra clave `class` también se presenta como una manera más elaborada de describir un tipo de datos nuevo, y se desmitifica el significado de la palabra «objeto» (es una variable elaborada).

Capítulo 6: Inicialización y limpieza. Uno de los errores más comunes en C se debe a las variables no inicializadas. El *constructor* de C++ permite garantizar que las variables de su nuevo tipo de datos («objetos de su clase») siempre se inicializarán correctamente. Si sus objetos también requieren algún tipo de reciclado, usted puede garantizar que ese reciclado se realice siempre mediante el *destructor* C++.

Capítulo 7: Sobrecarga de funciones y argumentos por defecto. C++ está pensado para ayudar a construir proyectos grandes y complejos. Mientras lo hace, puede dar lugar a múltiples librerías que utilicen el mismo nombre de función, y también puede decidir utilizar un mismo nombre con diferentes significados en la misma biblioteca. Con C++ es sencillo gracias a la «sobrecarga de funciones», lo que le permite reutilizar el mismo nombre de función siempre que la lista de argumentos sea diferente. Los argumentos por defecto le permiten llamar a la misma función de diferentes maneras proporcionando, automáticamente, valores por defecto para algunos de sus argumentos.

Capítulo 8: Constantes. Este capítulo cubre las palabras reservadas `const` y `volatile`, que en C++ tienen un significado adicional, especialmente dentro de las clases. Aprenderá lo que significa aplicar `const` a una definición de puntero. El capítulo también muestra cómo varía el significado de `const` según se utilice dentro o fuera de las clases y cómo crear constantes dentro de clases en tiempo de compilación.

Capítulo 9: Funciones inline. Las macros del preprocesador eliminan la sobrecarga de llamada a función, pero el preprocesador también elimina la valiosa comprobación de tipos de C++. La función `inline` le ofrece todos los beneficios de una macro de preprocesador además de los beneficios de una verdadera llamada a función. Este capítulo explora minuciosamente la implementación y uso de las funciones `inline`.

Capítulo 10: Control de nombres. La elección de nombres es una actividad fundamental en la programación y, cuando un proyecto se vuelve grande, el número de nombres puede ser arrollador. C++ le permite un gran control de los nombres en función de su creación, visibilidad, lugar de almacenamiento y enlazado. Este capítulo muestra cómo se controlan los nombres en C++ utilizando dos técnicas. Primero, la palabra reservada `static` se utiliza para controlar la visibilidad y enlazado, y se explora su significado especial para clases. Una técnica mucho más útil para controlar los nombres a nivel global es el `namespace` de C++, que le permite dividir el espacio de nombres global en distintas regiones.

Capítulo 11: Las referencias y el constructor de copia. Los punteros de C++ trabajan como los punteros de C con el beneficio adicional de la comprobación de tipos más fuerte de C++. C++ también proporciona un método adicional para manejar direcciones: C++ imita la *referencia* de Algol y Pascal, que permite al compilador manipular las direcciones, pero utilizando la notación ordinaria. También encontrará el constructor-de-copia, que controla la manera en que los objetos se pasan por valor hacia o desde las funciones. Finalmente, se explica el puntero-a-miembro de C++.

Capítulo 12: Sobrecarga de operadores. Esta característica se llama algunas veces «azúcar sintáctico»; permite dulcificar la sintaxis de uso de su tipo permitiendo operadores así como llamadas a funciones. En este capítulo aprenderá que la sobrecarga de operadores sólo es un tipo de llamada a función diferente y aprenderá cómo escribir sus propios operadores, manejando el -a veces confuso- uso de los argumentos, devolviendo tipos, y la decisión de si implementar el operador como método o

función amiga.

Capítulo 13: Creación dinámica de objetos. ¿Cuántos aviones necesitará manejar un sistema de tráfico aéreo? ¿Cuántas figuras requerirá un sistema CAD? En el problema de la programación genérica, no se puede saber la cantidad, tiempo de vida o el tipo de los objetos que necesitará el programa una vez lanzado. En este capítulo aprenderá cómo `new` y `delete` solventan de modo elegante este problema en C++ creando objetos en el montón. También verá cómo `new` y `delete` se pueden sobrecargar de varias maneras, de forma que puedan controlar cómo se asigna y se recupera el espacio de almacenamiento.

Capítulo 14: Herencia y composición. La abstracción de datos le permite crear tipos nuevos de la nada, pero con composición y herencia, se puede crear tipos nuevos a partir de los ya existentes. Con la composición, se puede ensamblar un tipo nuevo utilizando otros tipos como piezas y, con la herencia, puede crear una versión más específica de un tipo existente. En este capítulo aprenderá la sintaxis, cómo redefinir funciones y la importancia de la construcción y destrucción para la herencia y la composición.

Capítulo 15: Polimorfismo y funciones virtuales. Por su cuenta, podría llevarle nueve meses descubrir y comprender esta piedra angular de la POO. A través de ejercicios pequeños y simples, verá cómo crear una familia de tipos con herencia y manipular objetos de esa familia mediante su clase base común. La palabra reservada `virtual` le permite tratar todos los objetos de su familia de forma genérica, lo que significa que el grueso del código no depende de información de tipo específica. Esto hace extensibles sus programas, de manera que construir programas y mantener el código sea más sencillo y más barato.

Capítulo 16: Introducción a las plantillas. La herencia y la composición permiten reutilizar el código objeto, pero eso no resuelve todas las necesidades de reutilización. Las plantillas permiten reutilizar el código fuente proporcionando al compilador un medio para sustituir el nombre de tipo en el cuerpo de una clase o función. Esto da soporte al uso de bibliotecas de *clase contenedor*, que son herramientas importantes para el desarrollo rápido y robusto de programas orientados a objetos (la Biblioteca Estándar de C++ incluye una biblioteca significativa de clases contenedor). Este capítulo ofrece una profunda base en este tema esencial.

Temas adicionales (y materias más avanzadas) están disponibles en el Volumen 2 del libro, que se puede descargar del sitio web www.BruceEckel.com.

Ejercicios

He descubierto que los ejercicios son excepcionalmente útiles durante un seminario para completar la comprensión de los estudiantes, así que encontrará algunos al final de cada capítulo. El número de ejercicios ha aumentado enormemente respecto a la primera edición.

Muchos de los ejercicios son suficientemente sencillos como para que puedan terminarse en una cantidad de tiempo razonable en una clase o apartado de laboratorio mientras el profesor observa, asegurándose de que todos los estudiantes asimilan el material. Algunos ejercicios son un poco más complejos para mantener entretenidos a los estudiantes avanzados. El grueso de los ejercicios están orientados para ser resueltos en poco tiempo y se intenta sólo probar y pulir sus conocimientos más que presentar retos importantes (seguramente ya los encontrará por su cuenta -o mejor dicho-, ellos lo encontrarán a usted).

Soluciones a los ejercicios

Las soluciones a los ejercicios seleccionados pueden encontrarse en el documento electrónico *El Solucionario de Pensar en C++*, disponible por una pequeña cantidad en www.BruceEckel.com.

Código fuente

El código fuente de los listados de este libro está registrado como *freeware*, distribuido mediante el sitio Web www.BruceEckel.com. El copyright le impide publicar el código en un medio impreso sin permiso, pero se le otorga el derecho de usarlo de muchas otras maneras (ver más abajo).

El código está disponible en un fichero comprimido, destinado a extraerse desde cualquier plataforma que tenga una utilidad **zip** (puede buscar en Internet para encontrar una versión para su platarforma si aún no tiene una instalada). En el directorio inicial donde desempaquete el código encontrará la siguiente nota sobre derechos de copia:

```
Copyright (c) 2000, Bruce Eckel
Source code file from the book "Thinking in C++"
All rights reserved EXCEPT as allowed by the
following statements: You can freely use this file
for your own work (personal or commercial),
including modifications and distribution in
executable form only. Permission is granted to use
this file in classroom situations, including its
use in presentation materials, as long as the book
"Thinking in C++" is cited as the source.
Except in classroom situations, you cannot copy
and distribute this code; instead, the sole
distribution point is http://www.BruceEckel.com
(and official mirror sites) where it is
available for free. You cannot remove this
copyright and notice. You cannot distribute
modified versions of the source code in this
package. You cannot use this file in printed
media without the express permission of the
author. Bruce Eckel makes no representation about
the suitability of this software for any purpose.
It is provided "as is" without express or implied
warranty of any kind, including any implied
warranty of merchantability, fitness for a
particular purpose, or non-infringement. The entire
risk as to the quality and performance of the
software is with you. Bruce Eckel and the
publisher shall not be liable for any damages
suffered by you or any third party as a result of
using or distributing this software. In no event
will Bruce Eckel or the publisher be liable for
any lost revenue, profit, or data, or for direct,
indirect, special, consequential, incidental, or
punitive damages, however caused and regardless of
```

```
the theory of liability, arising out of the use of
or inability to use software, even if Bruce Eckel
and the publisher have been advised of the
possibility of such damages. Should the software
prove defective, you assume the cost of all
necessary servicing, repair, or correction. If you
think you've found an error, please submit the
correction using the form you will find at
www.BruceEckel.com. (Please use the same
form for non-code errors found in the book.)
```

Se puede usar el código en proyectos y clases siempre y cuando se mantenga la nota de copyright.

Estándares del lenguaje

Durante todo el libro, cuando se haga referencia al estándar de C ISO, generalmente se dirá «C». Sólo si se necesita distinguir entre C estándar y otros más viejos, versiones previas al estándar de C, se hará una distinción.

Cuando se escribió este libro, el Comité de Estándares de C++ ya había terminado de trabajar en el lenguaje. Por eso, se usará el término *C++ Estándar* para referirse al lenguaje estandarizado. Si se hace referencia simplemente a C++, debería asumir que se quiere decir «C++ Estándar».

Hay alguna confusión sobre el nombre real del Comité de Estándares de C++ y el nombre del estándar mismo. Steve Clamage, el presidente del comité, clarificó esto:

> Hay dos comités de estandarización de C++: El comité NCITS (antiguamente X3) J16 y el comité ISO JTC1/SC22/WG14. ANSI alquila NCITS para crear comités técnicos para desarrollar estándares nacionales americanos.
>
> J16 fue alquilado en 1989 para crear un estándar americano para C++. Por el año 1991 se alquiló WG14 para crear un estándar internacional. El proyecto J16 se convirtió en un proyecto «Tipo I» (Internacional) y se subordinó al esfuerzo de estandarización de ISO.
>
> Los dos comités se encontraban al mismo tiempo en el mismo sitio, y el voto de J16 constituye el voto americano con WG14. WG14 delega el trabajo técnico a J16. WG14 vota por el trabajo técnico de J16.
>
> El estándar de C++ fue creado originalmente como un estándar ISO. ANSI votó más tarde (como recomendaba J16) para adoptar el estándar de C++ ISO como el estándar americano para C++.

Por eso, «ISO» es la forma correcta de referirse al Estándar C++.

Soporte del lenguaje

Puede que su compilador no disponga de todas las características discutidas en este libro, especialmente si no tiene la versión más recente del compilador. Imple-

mentar un lenguaje como C++ es una tarea hercúlea, y puede esperar que las características aparecerán poco a poco en lugar de todas a la vez. Pero si prueba uno de los ejemplos del libro y obtiene un montón de errores del compilador, no es necesariamente un error en el código o en el compilador; simplemente puede no estar implementado aún en su compilador particular.

El CD-ROM del libro

El contenido principal del CD-ROM empaquetado al final de este libro es un «seminario en CD-ROM» titulado *Pensar en C: Fundamentos para Java y C++* obra de Chuck Allison (publicado por MindView, Inc., y también disponible en www.BruceEckel.com). Contiene muchas horas de grabaciones y transparencias, que pueden mostrarse en la mayoría de las computadoras que dispongan de lector de CD-ROM y sistema de sonido.

El objetivo de *Pensar en C* es llevarle cuidadosamente a través de los fundamentos del lenguaje C. Se centra en el conocimiento que necesita para poder pasarse a C++ o Java en lugar de intentar hacerle un experto en todos los recovecos de C (una de las razones de utilizar un lenguaje de alto nivel como C++ o Java es, precisamente, que se pueden evitar muchos de esos recovecos). También contiene ejercicios y soluciones guiadas. Téngalo en cuenta porque el Capítulo 3 de este libro va más allá del CD de *Pensar en C*, el CD no es una alternativa a este capítulo, sino que debería utilizarse como preparación para este libro.

Por favor, tenga en cuenta que el CD-ROM está basado en navegador, por lo que debería tener un navegador Web instalado en su máquina antes de utilizarlo.

CD-ROMs, seminarios, y consultoría

Hay seminarios en CD-ROM planeados para cubrir el Volumen 1 y el Volumen 2 de este libro. Comprenden muchas horas de grabaciones mías que acompañan las transparencias que cubren el material seleccionado de cada capítulo del libro. Se pueden ver en la mayoría de las computadoras que disponen de lector de CDROM y sistema de sonido. Estos CDs pueden comprarse en www.BruceEckel.com, donde encontrará más información y lecturas de ejemplo.

Mi compañía, MindView, Inc., proporciona seminarios públicos de preparación práctica basados en el material de este libro y también en temas avanzados. El material seleccionado de cada capítulo representa una lección, que se continúa con un periodo de ejercicios monitorizados para que cada estudiante reciba atención personal. También proporcionamos preparación «in situ», consultoría, tutorización, diseño y asistentes de código. Puede encontrar la información y los formularios para los próximos seminarios, así como otra información de contacto, en www.BruceEckel.com.

A veces me encuentro disponible para consultas de diseño, evaluación de procesos y asistencia. Cuando comencé a escribir sobre computadoras, mi motivación principal fue incrementar mis actividades de consultoría, porque encontraba que la consultoría era competitiva, educacional, y una de mis experiencias profesionales más valiosas. Así que haré todo lo que pueda para incluirle a usted en mi agenda, o para ofrecerle uno de mis socios (que son gente que conozco bien y con la que he tratado, y a menudo co-desarrollan e imparten seminarios conmigo).

Errores

No importa cuántos trucos emplee un escritor para detectar los errores, algunos siempre se escapan y saltan del papel al lector atento. Si encuentra algo que crea que es un error, por favor, utilice el formulario de correcciones que encontrará en www.BruceEckel.com. Se agradece su ayuda.

Sobre la portada

La primera edición de este libro tenía mi cara en la portada, pero para la segunda edición yo quería desde el principio una portada que se pareciera más una obra de arte, como la portada de *Pensar en Java*. Por alguna razón, C++ parece sugerirme Art Decó con sus curvas simples y pinceladas cromadas. Tenía en mente algo como esos carteles de barcos y aviones con cuerpos largos.

Mi amigo Daniel Will-Harris, (www.Will-Harris.com) a quien conocí en las clases del coro del instituto, iba a llegar a ser un diseñador y escritor de talla mundial. Él ha hecho prácticamente todos mis diseños, incluída la portada para la primera edición de este libro. Durante el proceso de diseño de la portada, Daniel, insatisfecho con el progreso que realizábamos, siempre preguntaba: «¿Qué relación hay entre las personas y las computadoras?». Estábamos atascados.

Como capricho, sin nada en mente, me pidió que pusiera mi cara en el escáner. Daniel tenía uno de sus programas gráficos (Corel Xara, su favorito) que «autotrazó» mi cara escaneada. Él lo describe de la siguente manera: «El autotrazado es la forma en la que la computadora transforma un dibujo en los tipos de líneas y curvas que realmente le gustan». Entonces jugó con ello hasta que obtuvo algo que parecía un mapa topográfico de mi cara, una imagen que podría ser la manera en que la computadora ve a la gente.

Cogí esta imagen y la fotocopié en papel de acuarela (algunas copiadoras pueden manejar papeles gruesos), y entonces comenzó a realizar montones de experimentos añadiendo acuarela a la imagen. Seleccionamos las que nos gustaban más, entonces Daniel las volvió a escanear y las organizó en la portada, añadiendo el texto y otros elementos de diseño. El proceso total requirió varios meses, mayormente a causa del tiempo que me tomó hacer las acuarelas. Pero me he divertido especialmente porque conseguí participar en el arte de la portada, y porque me dio un incentivo para hacer más acuarelas (lo que dicen sobre la práctica realmente es cierto).

Diseño del libro y producción

El diseño del interior del libro fue creado por Daniel Will-Harris, que solía jugar con letras (FIXME:rub-on) en el instituto mientras esperaba la invención de las computadoras y la publicación de escritorio. De todos modos, yo mismo produje las páginas para impresión (*camera-ready*), por lo que los errores tipográficos son míos. Se utilizó Microsoft® Word para Windows Versiones 8 y 9 para escribir el libro y crear la versión para impresión, incluyendo la generación de la tabla de contenidos y el índice (creé un servidor automatizado COM en Python, invocado desde las macros VBA de Word, para ayudarme en el marcado de los índices). Python (vea www.python.com) se utilizó para crear algunas de las herramientas para comprobar el código, y lo habría utilizado como herramienta de extracción de código si lo hubiese descubierto antes.

Creé los diagramas utilizando Visio®. Gracias a Visio Corporation por crear una

herramienta tan útil.

El tipo de letra del cuerpo es Georgia y los títulos utilizan Verdana. La versión definitiva se creó con Adobe® Acrobat 4 y el fichero generado se llevó directamente a la imprenta - muchas gracias a Adobe por crear una herramienta que permite enviar documentos listos para impresión por correo electrónico, así como permitir que se realicen múltiples revisiones en un único día en lugar de recaer sobre mi impresora láser y servicios rápidos 24 horas (probamos el proceso Acrobat por primera vez con *Pensar en Java*, y fui capaz de subir la versión final de ese libro a la imprenta de U.S. desde Sudáfrica).

La versión HTML se creó exportando el documento Word a RTF, y utilizando entonces RTF2HTML (ver http://www.sunpack.com/RTF/) para hacer la mayor parte del trabajo de la conversión HTML (gracias a Chris Hector por hacer una herramienta tan útil y especialmente fiable). Los ficheros resultantes se limpiaron utilizando un programa Python que truqué, y los WMFs se transformaron en GIFs utilizando el PaintShop Pro 6 de JASC® y su herramienta de conversión por lotes (gracias a JASC por resolver tantos de mis problemas con su excelente producto). El realce del color de la sintaxis se añadió con un script Perl amablemente cedido por Zafir Anjum.

Agradecimientos

Lo primero, agradecer a todo aquel que presentó correcciones y sugerencias desde Internet; han sido de tremenda ayuda para mejorar la calidad de este libro, y no podría haberlo hecho sin ustedes. Gracias en especial a John Cook.

Las ideas y comprensión de este libro han llegado de varias fuentes: amigos como Chuck Allison, Andrea Provaglio, Dans Sakx, Scott Meyers, Charles Petzold y Michael Wilk; pioneros del lenguaje como Bjarne Stroustrup, Andrew Koenig y Rob Murray; miembros del Comité de Estándares de C++ como Nathan Myers (que fue de particular ayuda y generosidad con sus percepciones), Bill Plauger, Reg Charney, Tom Penello, Tom Plum, Sam Druker y Uwe Steinmueller; gente que ha hablado en mis charlas de C++ en la Conferencia de Desarrollo de Software; y a menudo estudiantes de mis seminarios, que preguntan aquello que necesito oír para aclarar el material.

Enormes agradecimientos para mi amigo Gen Kiyooka, cuya compañía Digigami me proporcionó un servidor web.

Mi amigo Richard Hale Shaw y yo hemos enseñado C++ juntos; las percepciones de Richard y su apoyo han sido muy útiles (y las de Kim también). Gracias también a DoAnn Vikoren, Eric Faurot, Jennifer Jessup, Tara Arrowood, Marco Pardi, Nicole Freeman, Barbara Hanscome, Regina Ridley, Alex Dunne y el resto del reparto y plantilla de MFI.

Un agradecimiento especial para todos mis profesores y todos mis estudiantes (que también son profesores).

Y para mis escritores favoritos, mi más profundo aprecio y simpatía por vuestros esfuerzos: John Irving, Neal Stephenson, Robertson Davies (te echaremos de menos), Tom Robbins, William Gibson, Richard Bach, Carlos Castaneda y Gene Wolfe.

A Guido van Rossum, por inventar Python y donarlo desinteresadamente al mundo. Has enriquecido mi vida con tu contribución.

Gracias a la gente de Prentice Hall: Alan Apt, Ana Terry, Scott Disanno, Toni Holm y mi editora de copias electrónicas Stephanie English. En márqueting, Bryan Gambrel y Jennie Burger.

Sonda Donovan me ayudó con la producción del CD ROM. Daniel Will-Harris (por supuesto) creó el diseño de la portada que se encuentra en el propio CD.

Para todos los grandes amigos de Crested Butte, gracias por hacer de él un lugar mágico, especialmente a Al Smith (creador del maravilloso Camp4 Coffee Garden), mis vecinos Dave y Erika, Marsha de la librería Heg's Place, Pat y John de Teocalli Temale, Sam de Barkery Café, y a Tiller por su ayuda con la investigación en audio. Y a toda la gente fenomenal que anda por Camp4 y hace interesantes mis mañanas.

La lista de amigos que me han dado soporte incluye, pero no está limitada, a Zack Urlocker, Andrew Binstock, Neil Rubenking, Kraig Brocschmidt, Steve Sinofsky, JD Hildebrandt, Brian McElhinney, Brinkey Barr, Larry O'Brien, Bill Gates en *Midnight Engineering Magazine*, Larry Constantine, Lucy Lockwood, Tom Keffer, Dan Putterman, Gene Wang, Dave Mayer, David Intersimone, Claire Sawyers, los Italianos (Andrea Provaglio, Rossella Gioia, Laura Fallai, Marco & Lella Cantu, Corrado, Ilsa y Christina Giustozzi), Chris y Laura Strand (y Parker), los Alquimistas, Brad Jerbic, Marilyn Cvitanic, el Mabrys, el Halflingers, los Pollocks, Peter Vinci, los Robbins, los Moelters, Dave Stoner, Laurie Adams, los Cranstons, Larry Fogg, Mike y karen Sequeira, Gary Entsminger y Allison Brody, Kevin, Sonda & Ella Donovan, Chester y Shannon Andersen, Joe Lordi, Dave y Brenda Barlett, los Rentschlers, Lynn y Todd y sus familias. Y por supuesto, a Mamá y Papá.

1: Introducción a los Objetos

El origen de la revolución informática ocurrió dentro de una máquina. Por tanto, el origen de nuestros lenguajes de programación tiende a parecerse a esa máquina.

Pero los ordenadores no son tanto máquinas como herramientas de amplificación de la mente («bicicletas para la mente», como le gusta decir a Steve Jobs) y un medio de expresión diferente. Como resultado, las herramientas empiezan a parecerse menos a las máquinas y más a partes de nuestra mente, y también a otros medios de expresión como la escritura, la pintura, la escultura, la animación y la cinematografía. La programación orientada a objetos es parte de este movimiento hacia un uso del ordenador como medio de expresión.

Este capítulo le servirá de introducción a los conceptos básicos de la programación orientada a objetos (POO), incluyendo un resumen de los métodos de desarrollo de la POO. Este capítulo, y este libro, presuponen que el lector ya tiene experiencia con un lenguaje de programación procedural, aunque no tiene porqué ser C. Si cree que necesita más preparación en programación y en la sintaxis de C antes de abordar este libro, debería leer el CD-ROM de entrenamiento *Thinking in C: Foundations for C++ and Java*, que acompaña a este libro, y está disponible también en www.BruceEckel.com.

Este capítulo contiene material básico y suplementario. Mucha gente no se siente cómoda adentrándose en la programación orientada a objetos sin tener antes una visión global. Por eso, aquí se introducen muchos conceptos que intentan darle una visión sólida de la POO. Sin embargo, muchas personas no captan los conceptos globales hasta que no han visto primero parte de la mecánica; puede que se atasquen o se pierdan si no hay ningún trozo de código al que ponerle las manos encima. Si usted pertenece a este último grupo, y está ansioso por llegar a las especificaciones del lenguaje, siéntase libre de saltar este capítulo; eso no le impedirá escribir programas o aprender el lenguaje. Sin embargo, quizá quiera volver a este capítulo para completar sus conocimientos y poder comprender porqué son importantes los objetos y cómo diseñar con ellos.

1.1. El progreso de abstracción

Todos los lenguajes de programación proporcionan abstracciones. Se puede afirmar que la complejidad de los problemas que se pueden resolver está directamente relacionada con el tipo y calidad de la abstracción. Por «tipo» me refiero a «¿Qué es lo que está abstrayendo?». El lenguaje ensamblador es una pequeña abstracción de la máquina subyacente. Muchos lenguajes llamados «imperativos» que siguieron (como Fortran, BASIC y C) eran abstracciones del lenguaje ensamblador. Estos

lenguajes suponen grandes mejoras con respecto al lenguaje ensamblador, pero su abstracción primaria todavía requiere pensar en términos de la estructura del ordenador, en lugar de la estructura del problema que intenta resolver. El programador debe establecer la asociación entre el modelo de la máquina (en el «espacio de soluciones», que es el lugar donde está modelando ese problema, como un ordenador) y el modelo del problema que se está resolviendo (en el «espacio de problemas», que es el lugar donde existe el problema). El esfuerzo requerido para realizar esta correspondencia, y el hecho de que sea extrínseco al lenguaje de programación, produce programas difíciles de escribir y caros de mantener y, como efecto secundario, creó toda la industria de «métodos de programación».

La alternativa a modelar la máquina es modelar el problema que está intentando resolver. Los primeros lenguajes como LISP y APL eligieron concepciones del mundo particulares («Todos los problemas son listas en última instancia», o «Todos los problemas son algorítmicos»). PROLOG reduce todos los problemas a cadenas de decisiones. Se han creado lenguajes para programación basados en restricciones y para programar manipulando exclusivamente símbolos gráficos (lo último demostró ser demasiado restrictivo). Cada uno de estos métodos es una buena solución para el tipo particular de problema para el que fueron diseñados, pero cuando uno sale de ese dominio se hacen difíciles de usar.

El método orientado a objetos va un paso más allá, proporcionando herramientas para que el programador represente los elementos en el espacio del problema. Esta representación es lo suficientemente general como para que el programador no esté limitado a un tipo particular de problema. Nos referimos a los elementos en el espacio del problema, y a sus representaciones en el espacio de la solución, como «objetos» (por supuesto, necesitará otros objetos que no tengan analogías en el espacio del problema). La idea es que permita al programa adaptarse al lenguaje del problema añadiendo nuevos tipos de objetos de modo que cuando lea el código que describe la solución, esté leyendo palabras que además expresan el problema. Es un lenguaje de abstracción más flexible y potente que los que haya usado antes. De esta manera, la POO permite describir el problema en términos del problema, en lugar de usar términos de la computadora en la que se ejecutará la solución. Sin embargo, todavía existe una conexión con la computadora. Cada objeto se parece un poco a una pequeña computadora; tiene un estado y operaciones que se le puede pedir que haga. Sin embargo, no parece una mala analogía a los objetos en el mundo real; todos ellos tienen características y comportamientos.

Algunos diseñadores de lenguajes han decidido que la programación orientada a objetos en sí misma no es adecuada para resolver fácilmente todos los problemas de programación, y abogan por una combinación de varias aproximaciones en lenguajes de programación *multiparadigma*. [1]

Alan Kay resumió las cinco características básicas de Smalltalk, el primer lenguaje orientado a objetos con éxito y uno de los lenguajes en los que está basado C++. Esas características representan una aproximación a la programación orientada a objetos:

1. Todo es un objeto. Piense en un objeto como una variable elaborada; almacena datos, pero puede «hacer peticiones» a este objeto, solicitando que realice operaciones en sí mismo. En teoría, puede coger cualquier componente conceptual del problema que está intentando resolver (perros, edificios, servicios, etc.) y representarlos como un objeto en su programa.

[1] Ver *Multiparadigm Programming in Leda* de Timothy Budd (Addison-Wesley 1995).

2. Un programa es un grupo de objetos enviando mensajes a otros para decirles qué hacer. Para hacer una petición a un objeto, «envía un mensaje» a ese objeto. Más concretamente, puede pensar en un mensaje como una petición de invocación a una función que pertenece a un objeto particular.

3. Cada objeto tiene su propia memoria constituida por otros objetos. Visto de otra manera, puede crear un nuevo tipo de objeto haciendo un paquete que contenga objetos existentes. Por consiguiente, puede hacer cosas complejas en un programa ocultando la complejidad de los objetos.

4. Cada objeto tiene un tipo. Usando el argot, cada objeto es una instancia de una clase, en el que «clase» es sinónimo de «tipo». La característica más importante que lo distingue de una clase es «¿Qué mensajes puede enviarle?»

5. Todos los objetos de un tipo particular pueden recibir los mismos mensajes. En realidad es una frase con doble sentido, como verá más tarde. Como un objeto de tipo `círculo` es también un objeto de tipo `figura`, está garantizado que un círculo aceptará los mensajes de figura. Esto significa que puede escribir código que habla con objetos `figura` y automáticamente funcionará con cualquier otro objeto que coincida con la descripción de `figura`. Esta *sustituibilidad* es uno de los conceptos más poderosos en la POO.

1.2. Cada objeto tiene una interfaz

Aristóteles fue probablemente el primero en hacer un estudio minucioso del concepto de *tipo*; él habló de «la clase de peces y la clase de pájaros». La idea de que todos los objetos, aún siendo únicos, también son parte de una clase de objetos que tienen características y comportamientos comunes se utilizó directamente en el primer lenguaje orientado a objetos, Simula-67, con su palabra reservada `class` que introduce un nuevo tipo en un programa.

Simula, como su nombre indica, fue creado para desarrollar simulaciones como el clásico «problema del cajero» [2]. Tiene un grupo de cajeros, clientes, cuentas, transacciones, y unidades de moneda - un montón de «objetos». Los objetos idénticos, exceptuando su estado durante la ejecución del programa, se agrupan en «clases de objetos» y de ahí viene la palabra reservada `class`. Crear tipos de datos abstractos (clases) es un concepto fundamental en la programación orientada a objetos. Los tipos de datos abstractos trabajan casi exactamente como tipos predefinidos: puede crear variables de un tipo (llamadas *objetos* o *instancias* en el argot de la programación orientada a objetos) y manipular estas variables (llamado *envío de mensajes* o *peticiones*; envía un mensaje y el objeto decide qué hacer con él). Los miembros (elementos) de cada clase tienen algo en común: cada cuenta tiene un balance, cada cajero puede aceptar un depósito, etc. Al mismo tiempo, cada miembro tiene su propio estado, cada cuenta tiene un balance diferente, cada cajero tiene un nombre. De este modo, cada cajero, cliente, cuenta, transacción, etc., se puede representar con una única entidad en el programa de computador. Esta entidad es un objeto, y cada objeto pertenece a una clase particular que define sus características y comportamientos.

Por eso, lo que hace realmente un programa orientado a objetos es crear nuevos tipos de datos, prácticamente todos los lenguajes de programación orientados a objetos usan la palabra reservada `class`. Cuando vea la palabra «type», piense en

[2] Puede encontrar una implementación interesante de este problema en el Volumen 2 de este libro, disponible en www.BruceEckel.com

«class» y viceversa [3].

Dado que una clase describe un conjunto de objetos que tienen idénticas características (elementos de datos) y comportamientos (funcionalidad), una clase es realmente un tipo de datos porque un número de punto flotante, por ejemplo, también tiene un conjunto de características y comportamientos. La diferencia está en que el programador define una clase para resolver un problema en lugar de estar obligado a usar un tipo de dato existente diseñado para representar una unidad de almacenamiento en una máquina. Amplía el lenguaje de programación añadiendo nuevos tipos de datos específicos según sus necesidades. El sistema de programación acoge las nuevas clases y les presta toda la atención y comprobación de tipo que da a los tipos predefinidos.

El enfoque orientado a objetos no está limitado a la construcción de simulaciones. Esté o no de acuerdo con que cualquier problema es una simulación del sistema que está diseñando, el uso de técnicas POO puede reducir fácilmente un amplio conjunto de problemas a una solución simple.

Una vez establecida una clase, puede hacer tantos objetos de esta clase como quiera, y manipularlos como si fueran elementos que existen en el problema que está intentando resolver. De hecho, uno de los desafíos de la programación orientada a objetos es crear una correspondencia unívoca entre los elementos en el espacio del problema y objetos en el espacio de la solución.

Pero, ¿cómo se consigue que un objeto haga algo útil por usted? Debe haber una forma de hacer una petición al objeto para que haga algo, como completar una transacción, dibujar algo en la pantalla o activar un interruptor. Y cada objeto puede satisfacer sólo ciertas peticiones. Las peticiones que puede hacer un objeto están definidas por su *intefaz*, y es el tipo lo que determina la interfaz. Un ejemplo simple puede ser una representación de una bombilla:

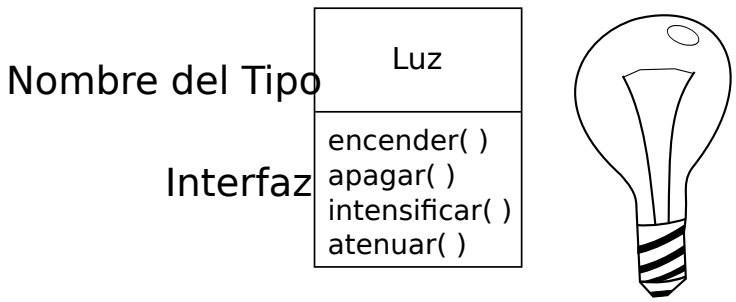

Figura 1.1: Clase `Luz`

```
Luz luz1;
luz1.encender();
```

La interfaz establece qué peticiones se pueden hacer a un objeto particular. Sin embargo, se debe codificar en algún sitio para satisfacer esta petición. Ésta, junto con los datos ocultos, constituyen la *implementación*. Desde el punto de vista de la programación procedural, no es complicado. Un tipo tiene una función asociada para cada posible petición, y cuando se hace una petición particular a un objeto, se llama a esa

[3] Hay quien hace una distinción, afirmando que `type` determina la interfaz mientras `class` es una implementación particular de esta interfaz.

función. Este proceso normalmente se resume diciendo que ha «enviado un mensaje» (ha hecho una petición) a un objeto, y el objeto sabe qué hacer con este mensaje (ejecuta código).

Aquí, el nombre del tipo/clase es `Luz`, el nombre de este objeto particular de `Luz` es `luz1`, y las peticiones que se le pueden hacer a un objeto `Luz` son encender, apagar, intensificar o atenuar. Puede crear un objeto `Luz` declarando un nombre (`luz1`) para ese objeto. Para enviar un mensaje al objeto, escriba el nombre del objeto y conéctelo al mensaje de petición con un punto. Desde el punto de vista del usuario de una clase predefinida, eso es prácticamente todo lo que necesita para programar con objetos.

El diagrama mostrado arriba sigue el formato del Lenguaje Unificado de Modelado (UML). Cada clase se representa con una caja, con el nombre del tipo en la parte de arriba, los atributos que necesite describir en la parte central de la caja, y los *métodos* (las funciones que pertenecen a este objeto, que reciben cualquier mensaje que se envíe al objeto) en la parte inferior de la caja. A menudo, en los diagramas de diseño UML sólo se muestra el nombre de la clase y el nombre de los métodos públicos, y por eso la parte central no se muestra. Si sólo está interesado en el nombre de la clase, tampoco es necesario mostrar la parte inferior.

1.3. La implementación oculta

Es útil distinguir entre los *creadores de clases* (aquellos que crean nuevos tipos de datos) y los *programadores clientes* [4] (los consumidores de clases que usan los tipos de datos en sus aplicaciones). El objetivo del programador cliente es acumular una caja de herramientas llena de clases que poder usar para un desarrollo rápido de aplicaciones. El objetivo del creador de clases es construir una clase que exponga sólo lo necesario para el programador cliente y mantenga todo lo demás oculto. ¿Por qué? Porque si está oculto, el programador cliente no puede usarlo, lo cual significa que el creador de clases puede cambiar la parte oculta sin preocuparse de las consecuencias sobre lo demás. La parte oculta suele representar las interioridades delicadas de un objeto que podría fácilmente corromperse por un programador cliente descuidado o desinformado, así que ocultando la implementación se reducen los errores de programación. No se debe abusar del concepto de implementación oculta.

En cualquier relación es importante poner límites que sean respetados por todas las partes involucradas. Cuando se crea una librería, se establece una relación con el programador cliente, quien también es programador, porque puede estar utilizando la librería para crear a su vez una librería mayor.

Si todos los miembros de una clase están disponibles para cualquiera, entonces el programador cliente puede hacer cualquier cosa con la clase y no hay forma de imponer las reglas. Incluso si quisiera que el programador cliente no manipulase directamente algunos de los miembros de su clase, sin control de acceso no hay forma de impedirlo. Nadie está a salvo.

Por eso la principal razón del control de acceso es impedir que el cliente toque las partes que no debería (partes que son necesarias para los mecanismos internos de los tipos de datos), pero no la parte de la interfaz que los usuarios necesitan para resolver sus problemas particulares. En realidad, ésto es un servicio para los usuarios porque pueden ver fácilmente lo qué es importante para ellos y qué pueden ignorar.

La segunda razón para el control de acceso es permitir al diseñador de la libre-

[4] Agradezco este término a mi amigo Scott Meyers.

ría cambiar la implementación interna de la clase sin preocuparse de cómo afectará a los programadores clientes. Por ejemplo, podría implementar una clase particular de una manera sencilla para un desarrollo fácil, y más tarde descubrir que necesita reescribirla para hacerla más rápida. Si la interfaz y la implementación están claramente separadas y protegidas, puede lograrlo fácilmente y sólo requiere que el usuario vuelva a enlazar la aplicación.

C++ utiliza tres palabras reservadas explícitas para poner límites en una clase: `public`, `private`, y `protected`. Su uso y significado son bastante sencillos. Estos *especificadores de acceso* determinan quién puede usar las definiciones que siguen. `public` significa que las definiciones posteriores están disponibles para cualquiera. La palabra reservada `private`, por otro lado, significa que nadie puede acceder a estas definiciones excepto el creador del tipo, es decir, los métodos internos de la clase. `private` es una pared entre el creador de la clase y el programador cliente. Si alguien intenta acceder a un miembro privado, obtendrá un error al compilar. `protected` actúa como `private`, con la excepción de que las clases derivadas tienen acceso a miembros protegidos, pero no a los privados. La herencia se explicará en breve.

1.4. Reutilizar la implementación

Una vez que una clase se ha creado y probado, debería constituir (idealmente) una unidad útil de código. Sin embargo, esta reutilización no es tan fácil de conseguir como muchos esperarían; producir un buen diseño requiere experiencia y conocimientos. Pero una vez que lo tiene, pide ser reutilizado. El código reutilizado es una de las mejores ventajas de los lenguajes para programación orientada a objetos.

La forma más fácil de reutilizar una clase es precisamente utilizar un objeto de esa clase directamente, pero también puede colocar un objeto de esta clase dentro de una clase nueva. Podemos llamarlo «crear un objeto miembro». Su nueva clase puede estar compuesta de varios objetos de cualquier tipo, en cualquier combinación que necesite para conseguir la funcionalidad deseada en su nueva clase. Como está componiendo una nueva clase a partir de clases existentes, este concepto se llama *composición* (o de forma más general, *agregación*). A menudo nos referimos a la composición como una relación «tiene-un», como en «un coche tiene-un motor».

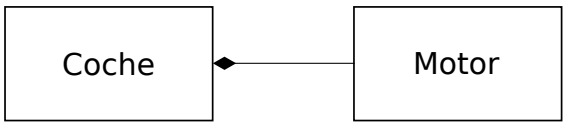

Figura 1.2: Un coche tiene un motor

(El diagrama UML anterior indica composición con el rombo relleno, lo cual implica que hay un coche. Típicamente usaré una forma más simple: sólo una línea, sin el rombo, para indicar una asociación. [5])

La composición es un mecanismo muy flexible. Los objetos miembro de su nueva clase normalmente son privados, haciéndolos inaccesibles para los programadores clientes que están usando la clase. Eso permite cambiar esos miembros sin perturbar al código cliente existente. También puede cambiar los miembros del objeto en tiem-

[5] Normalmente esto es suficiente para la mayoría de los diagramas y no necesita especificar si está usando agregación o composición.

po de ejecución, para cambiar dinámicamente el comportamiento de su programa. La herencia, descrita más adelante, no tiene esta flexibilidad dado que el compilador debe imponer restricciones durante la compilación en clases creadas con herencia.

Como la herencia es tan importante en la programación orientada a objetos, se suele enfatizar mucho su uso, y puede que el programador novel tenga la idea de que la herencia se debe usar en todas partes. Eso puede dar como resultado diseños torpes y demasiado complicados. En lugar de eso, debería considerar primero la composición cuando tenga que crear nuevas clases, ya que es más simple y flexible. Si acepta este enfoque, sus diseños serán más limpios. Una vez que tenga experiencia, los casos en los que necesite la herencia resultarán evidentes.

1.5. Herencia: reutilización de interfaces

En sí misma, la idea de objeto es una herramienta útil. Permite empaquetar datos y funcionalidad junto al propio concepto, además puede representar una idea apropiada del espacio del problema en vez de estar forzado a usar el vocabulario de la máquina subyacente. Esos conceptos se expresan como unidades fundamentales en el lenguaje de programación mediante la palabra reservada `class`.

Sin embargo, es una pena tomarse tantas molestias en crear una clase y verse obligado a crear una más para un nuevo tipo que tiene una funcionalidad similar. Es más sencillo si se puede usar la clase existente, clonarla, y hacerle añadidos y modificaciones a ese clon. Esto es justamente lo que hace la *herencia*, con la excepción de que si cambia la clase original (llamada clase *base*, *super* o *padre*), el «clon» modificado (llamado clase *derivada*, *heredada*, *sub* o *hija*) también refleja esos cambios.

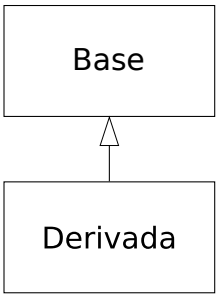

Figura 1.3: subclases

(En el diagrama UML anterior, la flecha apunta desde la clase derivada hacia la clase base. Como puede ver, puede haber más de una clase derivada.)

Un tipo hace algo más que describir las restricciones de un conjunto de objetos; también tiene una relación con otros tipos. Dos tipos pueden tener características y comportamientos en común, pero un tipo puede contener más características que otro y también puede manipular más mensajes (o hacerlo de forma diferente). La herencia lo expresa de forma similar entre tipos usando el concepto de tipos base y tipos derivados. Un tipo base contiene todas las características y comportamientos compartidos entre los tipos derivados de él. Cree un tipo base para representar lo esencial de sus ideas sobre algunos objetos en su sistema. A partir del tipo base, derive otros tipos para expresar caminos diferentes que puede realizar esa parte común.

Por ejemplo, una máquina de reciclado de basura clasifica piezas de basura. El tipo base es «basura», y cada pieza de basura tiene un peso, un valor, y también, se puede triturar, fundir o descomponer. A partir de ahí, se obtienen más tipos específicos de basura que pueden tener características adicionales (una botella tiene un color) o comportamientos (el aluminio puede ser aplastado, el acero puede ser magnético). Además, algunos comportamientos pueden ser diferentes (el valor del papel depende del tipo y condición). Usando la herencia, se puede construir una jerarquía de tipos que exprese el problema que se intenta resolver en términos de sus tipos.

Un segundo ejemplo es el clásico ejemplo «figura», tal vez usado en un sistema de diseño asistido por computador o juegos de simulación. El tipo base es `figura`, y cada figura tiene un tamaño, un color, una posición y así sucesivamente. Cada figura se puede dibujar, borrar, mover, colorear, etc. A partir de ahí, los tipos específicos de figuras derivan (heredan) de ella: círculo, cuadrado, triángulo, y así sucesivamente, cada uno de ellos puede tener características y comportamientos adicionales. Ciertas figuras pueden ser, por ejemplo, rotadas. Algunos comportamientos pueden ser diferentes, como cuando se quiere calcular el área de una figura. La jerarquía de tipos expresa las similitudes y las diferencias entre las figuras.

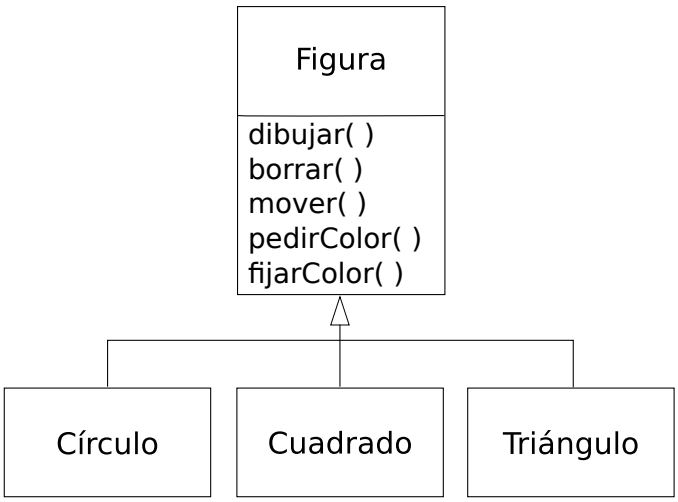

Figura 1.4: Jerarquía de `Figura`

Modelar la solución en los mismos términos que el problema es tremendamente beneficioso porque no se necesitan un montón de modelos intermedios para transformar una descripción del problema en una descripción de la solución. Con objetos, la jerarquía de tipos es el principal modelo, lleva directamente desde la descripción del sistema en el mundo real a la descripción del sistema en código. Efectivamente, una de las dificultades que la gente tiene con el diseño orientado a objetos es que es demasiado fácil ir desde el principio hasta el final. Una mente entrenada para buscar soluciones complejas a menudo se confunde al principio a causa de la simplicidad.

Cuando se hereda de un tipo existente, se está creando un tipo nuevo. Este nuevo tipo contiene no sólo todos los miembros del tipo base (aunque los datos privados `private` están ocultos e inaccesibles), sino que además, y lo que es más importante, duplica la interfaz de la clase base. Es decir, todos los mensajes que se pueden enviar a los objetos de la clase base se pueden enviar también a los objetos de la clase derivada. Dado que se conoce el tipo de una clase por los mensajes que se le pue-

den enviar, eso significa que la clase derivada *es del mismo tipo que la clase base*. En el ejemplo anterior, «un círculo es una figura». Esta equivalencia de tipos vía herencia es uno de las claves fundamentales para comprender la programación orientada a objetos.

Por lo que tanto la clase base como la derivada tienen la misma interfaz, debe haber alguna implementación que corresponda a esa interfaz. Es decir, debe haber código para ejecutar cuando un objeto recibe un mensaje particular. Si simplemente hereda de una clase y no hace nada más, los métodos de la interfaz de la clase base están disponibles en la clase derivada. Esto significa que los objetos de la clase derivada no sólo tienen el mismo tipo, también tienen el mismo comportamiento, lo cual no es particularmente interesante.

Hay dos caminos para diferenciar la nueva clase derivada de la clase base original. El primero es bastante sencillo: simplemente hay que añadir nuevas funciones a la clase derivada. Estas nuevas funciones no son parte de la interfaz de la clase base. Eso significa que la clase base simplemente no hace todo lo que necesitamos, por lo que se añaden más funciones. Este uso simple y primitivo de la herencia es, a veces, la solución perfecta a muchos problemas. Sin embargo, quizá debería pensar en la posibilidad de que su clase base puede necesitar también funciones adicionales. Este proceso de descubrimiento e iteración de su diseño ocurre regularmente en la programación orientada a objetos.

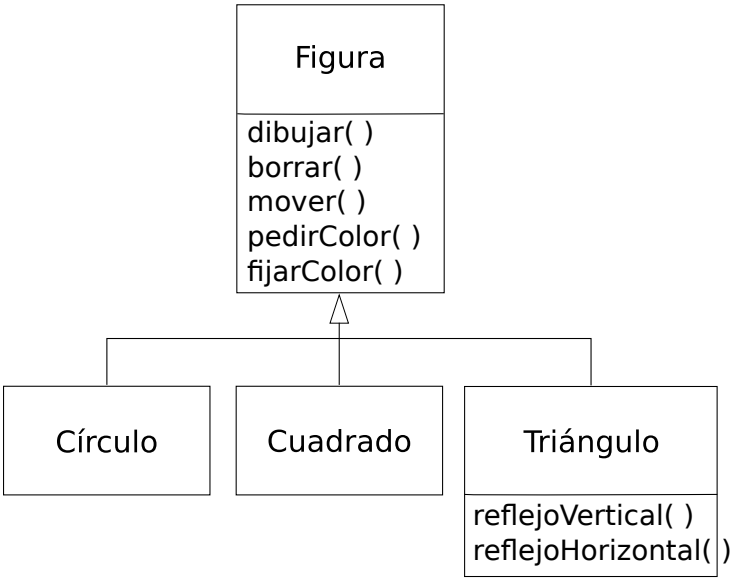

Figura 1.5: Especialización de `Figura`

Aunque la herencia algunas veces supone que se van a añadir nuevas funciones a la interfaz, no es necesariamente cierto. El segundo y más importante camino para diferenciar su nueva clase es *cambiar* el comportamiento respecto de una función de una clase base existente. A esto se le llama *reescribir* (*override*) una función.

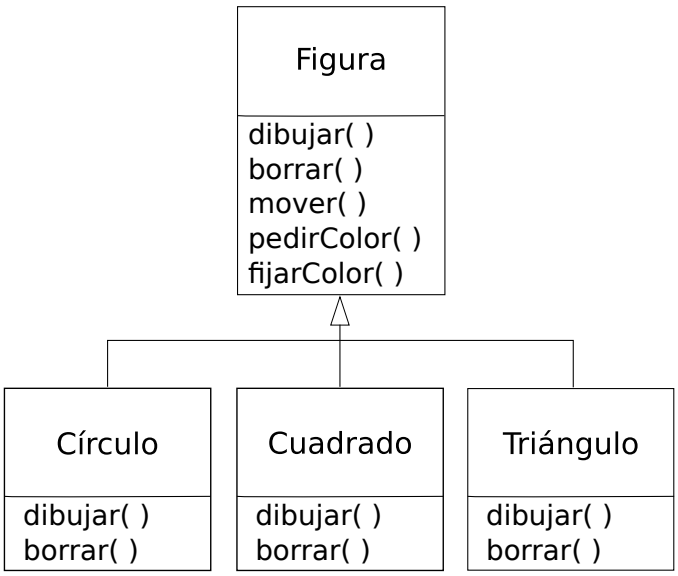

Figura 1.6: Reescritura de métodos

Para reescribir una función, simplemente hay que crear una nueva definición para esa función en la clase derivada. Está diciendo, «Estoy usando la misma función de interfaz aquí, pero quiero hacer algo diferente para mi nuevo tipo».

1.5.1. Relaciones es-un vs. es-como-un

Hay cierta controversia que puede ocurrir con la herencia: ¿la herencia debería limitarse a anular *sólo* funciones de la clase base (y no añadir nuevos métodos que no estén en la clase base)? Esto puede significar que el tipo derivado es *exactamente* el mismo tipo que la clase base dado que tiene exactamente la misma interfaz. Como resultado, se puede sustituir un objeto de una clase derivada por un objeto de la clase base. Se puede pensar como una *sustitución pura*, y se suele llamar *principio de sustitución*. En cierto modo, esta es la forma ideal de tratar la herencia. A menudo nos referimos a las relaciones entre la clase base y clases derivadas en este caso como una relación es-un, porque se dice «un círculo es una figura». Un modo de probar la herencia es determinar si se puede considerar la relación es-un sobre las clases y si tiene sentido.

Hay ocasiones en las que se deben añadir nuevos elementos a la interfaz de un tipo derivado, de esta manera se amplía la interfaz y se crea un tipo nuevo. El nuevo tipo todavía puede ser sustituido por el tipo base, pero la sustitución no es perfecta porque sus nuevas funciones no son accesibles desde el tipo base. Esta relación se conoce como *es-como-un*; el nuevo tipo tiene la interfaz del viejo tipo, pero también contiene otras funciones, por lo que se puede decir que es exactamente el mismo. Por ejemplo, considere un aire acondicionado. Suponga que su casa está conectada con todos los controles para refrigerar; es decir, tiene una interfaz que le permite controlar la temperatura. Imagine que el aire acondicionado se avería y lo reemplaza por una bomba de calor, la cual puede dar calor y frío. La bomba de calor *es-como-un* aire acondicionado, pero puede hacer más cosas. Como el sistema de control de su casa está diseñado sólo para controlar el frío, está rentringida a comunicarse sólo con

la parte de frío del nuevo objeto. La interfaz del nuevo objeto se ha extendido, y el sistema existente no conoce nada excepto la interfaz original.

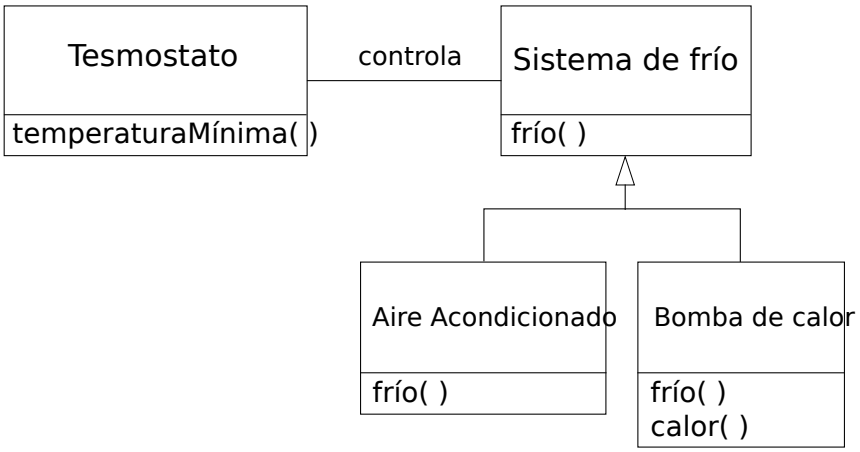

Figura 1.7: Relaciones

Por supuesto, una vez que vea este diseño queda claro que la clase base «sistema de frío» no es bastante general, y se debería renombrar a «sistema de control de temperatura», además también puede incluir calor, en este punto se aplica el principio de sustitución. Sin embargo, el diagrama de arriba es un ejemplo de lo que puede ocurrir en el diseño y en el mundo real.

Cuando se ve el principio de sustitución es fácil entender cómo este enfoque (sustitución pura) es la única forma de hacer las cosas, y de hecho es bueno para que sus diseños funcionen de esta forma. Pero verá que hay ocasiones en que está igualmente claro que se deben añadir nuevas funciones a la interfaz de la clase derivada. Con experiencia, ambos casos puede ser razonablemente obvios.

1.6. Objetos intercambiables gracias al polimorfismo

Cuando se manejan jerarquías de tipos, se suele tratar un objeto no como el tipo específico si no como su tipo base. Esto le permite escribir código que no depende de los tipos específicos. En el ejemplo de la figura, las funciones manipulan figuras genéricas sin preocuparse de si son círculos, cuadrados, triángulos, etc. Todas las figuras se pueden dibujar, borrar y mover, pero estas funciones simplemente envían un mensaje a un objeto figura, sin preocuparse de cómo se las arregla el objeto con cada mensaje.

Semejante código no está afectado por la adición de nuevos tipos, y añadir nuevos tipos es la forma más común de extender un programa orientado a objetos para tratar nuevas situaciones. Por ejemplo, puede derivar un nuevo subtipo de figura llamado `pentágono` sin modificar las funciones que tratan sólo con figuras genéricas. Esta habilidad para extender un programa fácilmente derivando nuevos subtipos es importante porque mejora enormemente los diseños al mismo tiempo que reduce el coste del mantenimiento del software.

Hay un problema, no obstante, con intentar tratar un tipo derivado como sus tipos base genéricos (círculos como figuras, bicicletas como vehículos, cormoranes como pájaros, etc). Si una función va a indicar a una figura genérica que se dibuje a sí misma, o a un vehículo genérico que se conduzca, o a un pájaro genérico que se mueva, el compilador en el momento de la compilación no sabe con precisión qué pieza del código será ejecutada. Este es el punto clave - cuando el mensaje se envía, el programador no *quiere* saber qué pieza de código será ejecutada; la función `dibujar()` se puede aplicar a un círculo, un cuadrado, o un triángulo, y el objeto ejecutará el código correcto dependiendo de tipo específico. Si no sabe qué pieza del código se ejecuta, ¿qué hace? Por ejemplo, en el siguiente diagrama el objeto `ControladorDePájaro` trabaja con los objetos genéricos `Pájaro`, y no sabe de qué tipo son exactamente. Esto es conveniente desde la perspectiva del `ControladorDePájaro`, porque no hay que escribir código especial para determinar el tipo exacto de `Pájaro` con el que está trabajando, o el comportamiento del `Pájaro`. Entonces, ¿qué hace que cuando se invoca `mover()` ignorando el tipo específico de `Pájaro`, puede ocurrir el comportamiento correcto (un `Ganso` corre, vuela, o nada, y un `Pingüino` corre o nada)?

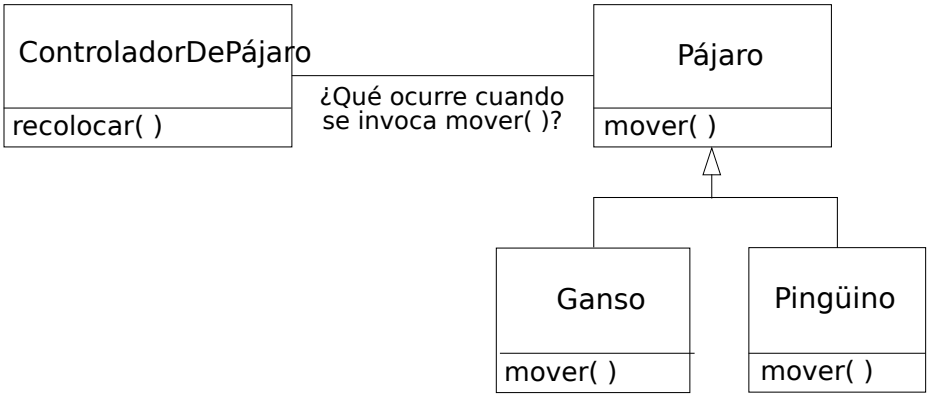

Figura 1.8: Polimorfismo

La respuesta es el primer giro en programación orientada a objetos: el compilador no hace una llamada a la función en el sentido tradicional. La llamada a función generada por un compilador no-OO provoca lo que se llama una *ligadura temprana* (*early binding*), un término que quizá no haya oído antes porque nunca ha pensado en que hubiera ninguna otra forma. Significa que el compilador genera una llamada al nombre de la función específica, y el enlazador resuelve esta llamada con la dirección absoluta del código que se ejecutará. En POO, el programa no puede determinar la dirección del código hasta el momento de la ejecución, de modo que se necesita algún otro esquema cuando se envía un mensaje a un objeto genérico.

Para resolver el problema, los lenguajes orientados a objetos usan el concepto de *ligadura tardía* (*late binding*). Cuando envía un mensaje a un objeto, el código invocado no está determinado hasta el momento de la ejecución. El compilador se asegura de que la función existe y realiza una comprobación de tipo de los argumentos y el valor de retorno (el lenguaje que no realiza esta comprobación se dice que es *débilmente tipado*), pero no sabe el código exacto a ejecutar.

Para llevar a cabo la ligadura tardía, el compilador de C++ inserta un trozo especial de código en lugar de la llamada absoluta. Este código calcula la dirección del

cuerpo de la función, usando información almacenada en el objeto (este proceso se trata con detalle en el Capítulo 15). De este modo, cualquier objeto se puede comportar de forma diferente de acuerdo con el contenido de este trozo especial de código. Cuando envía un mensaje a un objeto, el objeto comprende realmente qué hacer con el mensaje.

Es posible disponer de una función que tenga la flexibilidad de las propiedades de la ligadura tardía usando la palabra reservada `virtual`. No necesita entender el mecanismo de `virtual` para usarla, pero sin ella no puede hacer programación orientada a objetos en C++. En C++, debe recordar añadir la palabra reservada `virtual` porque, por defecto, los métodos *no* se enlazan dinámicamente. Los métodos virtuales le permiten expresar las diferencias de comportamiento en clases de la misma familia. Estas diferencias son las que causan comportamientos polimórficos.

Considere el ejemplo de la figura. El diagrama de la familia de clases (todas basadas en la misma interfaz uniforme) apareció antes en este capítulo. Para demostrar el polimorfismo, queremos escribir una única pieza de código que ignore los detalles específicos de tipo y hable sólo con la clase base. Este código está *desacoplado* de la información del tipo específico, y de esa manera es más simple de escribir y más fácil de entender. Y, si tiene un nuevo tipo - un `Hexágono`, por ejemplo - se añade a través de la herencia, el código que escriba funcionará igual de bien para el nuevo tipo de `Figura` como para los tipos anteriores. De esta manera, el programa es *extensible*.

Si escribe una función C++ (podrá aprender dentro de poco cómo hacerlo):

```
void hacerTarea(Figura& f) {
    f.borrar();
    // ...
    f.dibujar();
}
```

Esta función se puede aplicar a cualquier `Figura`, de modo que es independiente del tipo específico del objeto que se dibuja y borra (el «&» significa «toma la dirección del objeto que se pasa a `hacerTarea()`», pero no es importante que entienda los detalles ahora). Si en alguna otra parte del programa usamos la función `hacerTarea()`:

```
Circulo c;
Triangulo t;
Linea l;
hacerTarea(c);
hacerTarea(t);
hacerTarea(l);
```

Las llamadas a `hacerTarea()` funcionan bien automáticamente, a pesar del tipo concreto del objeto.

En efecto es un truco bonito y asombroso. Considere la línea:

```
hacerTarea(c);
```

Lo que está ocurriendo aquí es que está pasando un `Círculo` a una función que espera una `Figura`. Como un `Círculo` es una `Figura` se puede tratar como tal por parte de `hacerTarea()`. Es decir, cualquier mensaje que pueda enviar `hacerTarea()` a una `Figura`, un `Círculo` puede aceptarlo. Por eso, es algo completamente

lógico y seguro.

A este proceso de tratar un tipo derivado como si fuera su tipo base se le llama *upcasting* (*moldeado hacia arriba*[6]). El nombre *cast* (molde) se usa en el sentido de adaptar a un molde y es *hacia arriba* por la forma en que se dibujan los diagramas de clases para indicar la herencia, con el tipo base en la parte superior y las clases derivadas colgando debajo. De esta manera, moldear un tipo base es moverse hacia arriba por el diagrama de herencias: «upcasting»

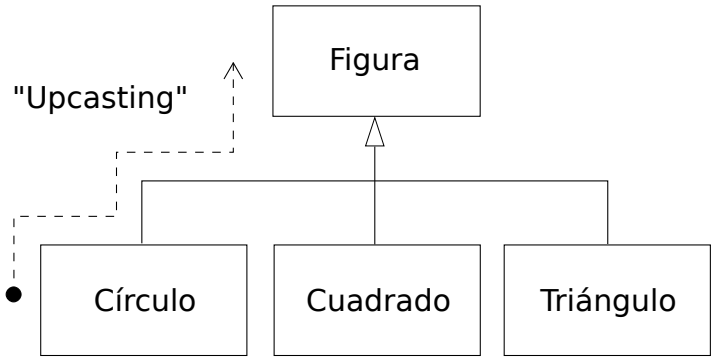

Figura 1.9: Upcasting

Todo programa orientado a objetos tiene algún upcasting en alguna parte, porque así es como se despreocupa de tener que conocer el tipo exacto con el que está trabajando. Mire el código de `hacerTarea()`:

```
f.borrar();
// ...
f.dibujar();
```

Observe que no dice «Si es un `Círculo`, haz esto, si es un `Cuadrado`, haz esto otro, etc.». Si escribe un tipo de código que comprueba todos los posibles tipos que una `Figura` puede tener realmente, resultará sucio y tendrá que cambiarlo cada vez que añada un nuevo tipo de `Figura`. Aquí, sólo dice «Eres una figura, sé que te puedes `borrar()` y `dibujar()` a ti misma, hazlo, y preocúpate de los detalles».

Lo impresionante del código en `hacerTarea()` es que, de alguna manera, funciona bien. Llamar a `dibujar()` para un `Círculo` ejecuta diferente código que cuando llama a `dibujar()` para un `Cuadrado` o una `Línea`, pero cuando se envía el mensaje `dibujar()` a un `Figura` anónima, la conducta correcta sucede en base en el tipo real de `Figura`. Esto es asombroso porque, como se mencionó anteriormente, cuando el compilador C++ está compilando el código para `hacerTarea()`, no sabe exactamente qué tipos está manipulando. Por eso normalmente, es de esperar que acabe invocando la versión de `borrar()` y `dibujar()` para `Figura`, y no para el `Círculo`, `Cuadrado`, o `Línea` específico. Y aún así ocurre del modo correcto a causa del polimorfismo. El compilador y el sistema se encargan de los detalles; todo lo que necesita saber es que esto ocurre y lo que es más importante, cómo utilizarlo en sus diseños. Si un método es `virtual`, entonces cuando envíe el mensaje a

[6] N. de T: En el libro se utilizará el término original en inglés debido a su uso común, incluso en la literatura en castellano.

un objeto, el objeto hará lo correcto, incluso cuando esté involucrado el upcasting.

1.7. Creación y destrucción de objetos

Técnicamente, el dominio de la POO son los tipos abstractos de datos, la herencia y el polimorfismo, pero otros asuntos pueden ser al menos igual de importantes. Esta sección ofrece una visión general de esos asuntos.

Es especialmente importante la forma en que se crean y se destruyen los objetos. ¿Dónde está el dato para un objeto y cómo se controla la vida de este objeto? Diferentes lenguajes de programación usan distintas filosofías al respecto. C++ adopta el enfoque de que el control de eficiencia es la cuestión más importante, pero eso delega la elección al programador. Para una velocidad máxima de ejecución, el almacenamiento y la vida se determinan mientras el programa se escribe, colocando los objetos en la pila o en almacenamiento estático. La pila es un área de memoria usada directamente por el microprocesador para almacenar datos durante la ejecución del programa. A veces las variables de la pila se llaman variables *automáticas* o *de ámbito* (*scoped*). El área de almacenamiento estático es simplemente un parche fijo de memoria alojado antes de que el programa empiece a ejecutarse. Usar la pila o el área de almacenamiento estático fija una prioridad en la rapidez de asignación y liberación de memoria, que puede ser valioso en algunas situaciones. Sin embargo, se sacrifica flexibilidad porque se debe conocer la cantidad exacta, vida, y tipo de objetos *mientras* el programador escribe el programa. Si está intentando resolver un problema más general, como un diseño asistido por computadora, gestión de almacén, o control de tráfico aéreo, eso también es restrictivo.

El segundo enfoque es crear objetos dinámicamente en un espacio de memoria llamado *montículo* (*heap*). En este enfoque no se sabe hasta el momento de la ejecución cuántos objetos se necesitan, cuál será su ciclo de vida, o su tipo exacto. Estas decisiones se toman de improviso mientras el programa está en ejecución. Si necesita un nuevo objeto, simplemente creelo en el montículo cuando lo necesite, usando la palabra reservada `new`. Cuando ya no necesite ese espacio de almacenamiento, debe liberarlo usando la palabra reservada `delete`.

Como la memoria se administra dinámicamente en tiempo de ejecución, la cantidad de tiempo requerido para reservar espacio en el montículo es considerablemente mayor que el tiempo para manipular la pila (reservar espacio en la pila a menudo es una única instrucción del microprocesador para mover el puntero de la pila hacia abajo, y otro para moverlo de nuevo hacia arriba). El enfoque dinámico asume que los objetos tienden a ser complicados, por eso la sobrecarga extra de encontrar espacio para alojarlos y después liberarlos, no tiene un impacto importante en la creación de un objeto. Además, el aumento de flexibilidad es esencial para resolver problemas generales de programación.

Hay otra cuestión, sin embargo, y es el tiempo de vida de un objeto. Si crea un objeto en la pila o en espacio estático, el compilador determina cuánto tiempo dura el objeto y puede destruirlo automáticamente. Pero si lo crea en el montículo, el compilador no tiene conocimiento de su tiempo de vida. En C++, el programador debe determinar programáticamente cuándo destruir el objeto, y entonces llevar a cabo la destrucción usando la palabra reservada `delete`. Como alternativa, el entorno puede proporcionar una característica llamada *recolector de basura* (*garbage collector*) que automáticamente descubre qué objetos ya no se usan y los destruye. Naturalmente, escribir programas usando un recolector de basura es mucho más conveniente, pero requiere que todas las aplicaciones sean capaces de tolerar la existencia del recolector de basura y la sobrecarga que supone. Eso no encaja en los requisitos del diseño del

lenguaje C++ por lo que no se incluye, aunque existen recolectores de basura para C++, creados por terceros.

1.8. Gestión de excepciones: tratamiento de errores

Desde los inicios de los lenguajes de programación, la gestión de errores ha sido uno de los asuntos más difíciles. Es tan complicado diseñar un buen esquema de gestión de errores, que muchos lenguajes simplemente lo ignoran, delegando el problema en los diseñadores de la librería, que lo resuelven a medias, de forma que puede funcionar en muchas situaciones, pero se pueden eludir, normalmente ignorándolos. El problema más importante de la mayoría de los esquemas de gestión de errores es que dependen de que el programador se preocupe en seguir un convenio que no está forzado por el lenguaje. Si los programadores no se preocupan, cosa que ocurre cuando se tiene prisa, esos esquemas se olvidan fácilmente.

La gestión de excepciones «conecta» la gestión de errores directamente en el lenguaje de programación y a veces incluso en el sistema operativo. Una excepción es un objeto que se «lanza» desde el lugar del error y puede ser «capturado» por un *manejador de excepción* apropiado diseñado para manipular este tipo particular de error. Es como si la gestión de errores fuera una ruta de ejecución diferente y paralela que se puede tomar cuando las cosas van mal. Y como usa un camino separado de ejecución, no necesita interferir con el código ejecutado normalmente. Eso hace que el código sea más simple de escribir ya que no se fuerza al programador a comprobar los errores constantemente. Además, una excepción no es lo mismo que un valor de error devuelto por una función o una bandera fijada por una función para indicar una condición de error, que se puede ignorar. Una excepción no se puede ignorar, de modo que está garantizado que habrá que tratarla en algún momento. Finalmente, las excepciones proporcionan una forma para recuperar una situación consistente. En lugar de salir simplemente del programa, a menudo es posible arreglar las cosas y restaurar la ejecución, lo que produce sistemas más robustos.

Merece la pena tener en cuenta que la gestión de excepciones no es una característica orientada a objetos, aunque en lenguajes orientados a objetos las excepciones normalmente se representan con objetos. La gestión de excepciones existía antes que los lenguajes orientados a objetos.

En este Volumen se usa y explica la gestión de excepciones sólo por encima; el Volúmen 2 (disponible en www.BruceEckel.com) cubre con más detalle la gestión de excepciones.

1.9. Análisis y diseño

El paradigma orientado a objetos es una nueva forma de pensar sobre programación y mucha gente tiene problemas la primera vez que escucha cómo se aborda un proyecto POO. Una vez que se sabe que, supuestamente, todo es un objeto, y cómo aprender a pensar al estilo orientado a objetos, puede empezar a crear «buenos» diseños que aprovechen las ventajas de todos los beneficios que ofrece la POO.

Un *método* (llamado a menudo *metodología*) es un conjunto de procesos y heurísticas usados para tratar la complejidad de un problema de programación. Desde el comienzo de la programación orientada a objetos se han formulado muchos métodos. Esta sección le dará una idea de cuál es el objetivo que se intenta conseguir

cuando se usa una metodología.

Especialmente en POO, la metodología es un campo de muchos experimentos, así que antes de elegir un método, es importante que comprenda cuál es el problema que resuelve. Eso es particularmente cierto con C++, en el que el lenguaje de programación pretende reducir la complejidad (comparado con C) que implica expresar un programa. De hecho, puede aliviar la necesidad de metodologías aún más complejas. En cambio, otras más simples podrían ser suficientes en C++ para muchos tipos de problemas grandes que podría manejar usando metodologías simples con lenguajes procedurales.

También es importante darse cuenta de que el término «metodología» a menudo es demasiado grande y prometedor. A partir de ahora, cuando diseñe y escriba un programa estará usando una metodología. Puede ser su propia metodología, y puede no ser consciente, pero es un proceso por el que pasa cuando crea un programa. Si es un proceso efectivo, puede que sólo necesite un pequeño ajuste para que funcione con C++. Si no está satisfecho con su productividad y con el camino que sus programas han tomado, puede considerar adoptar un método formal, o elegir trozos de entre muchos métodos formales.

Mientras pasa por el proceso de desarrollo, el uso más importante es éste: no perderse. Eso es fácil de hacer. La mayoría de los análisis y métodos de diseño pretenden resolver los problemas más grandes. Recuerde que la mayoría de los proyectos no encajan en esta categoría, normalmente puede tener un análisis y diseño exitoso con un subconjunto relativamente pequeño de lo que recomienda el método [7] . Pero muchos tipos de procesos, sin importar lo limitados que sean, generalmente le ofrecerán un camino mucho mejor que simplemente empezar a codificar.

También es fácil quedarse estancado, caer en *análisis-parálisis*, donde sentirá que no puede avanzar porque en la plataforma que está usando no está especificado cada pequeño detalle. Recuerde, no importa cuánto análisis haga, hay algunas cosas sobre el sistema que no se revelan hasta el momento del diseño, y más cosas que no se revelarán hasta que esté codificando, o incluso hasta que el programa esté funcionando. Por eso, es crucial moverse bastante rápido durante del análisis y diseño, e implementar un test del sistema propuesto.

Este punto merece la pena enfatizarlo. Debido a nuestra experiencia con los lenguajes procedurales, es encomiable que un equipo quiera proceder con cuidado y entender cada pequeño detalle antes de pasar al diseño y a la implementación. Desde luego, cuando crea un SGBD (Sistema Gestor de Bases de Datos), conviene entender la necesidad de un cliente a fondo. Pero un SGBD está en una clase de problemas que son muy concretos y bien entendidos; en muchos programas semejantes, la estructura de la base de datos *es* el problema que debe afrontarse. El tipo de problema de programación tratado en este capítulo es de la variedad «comodín» (con mis palabras), en el que la solución no es simplemente adaptar una solución bien conocida, en cambio involucra uno o más «factores comodín» -elementos para los que no hay solución previa bien entendida, y para los que es necesario investigar [8] . Intentar analizar minuciosamente un problema comodín antes de pasar al diseño y la implementación provoca un análisis-parálisis porque no se tiene suficiente información para resolver este tipo de problema durante la fase de análisis. Resolver estos problemas requiere

[7] Un ejemplo excelente es *UML Distilled*, de Martin Fowler (Addison-Wesley 2000), que reduce el, a menudo, insoportable proceso UML a un subconjunto manejable.

[8] Mi regla general para el cálculo de semejantes proyectos: Si hay más de un comodín, no intente planear cuánto tiempo le llevará o cuánto costará hasta que haya creado un prototipo funcional. También hay muchos grados de libertad.

interacción a través del ciclo completo, y eso requiere comportamientos arriesgados (lo cual tiene sentido, porque está intentando hacer algo nuevo y los beneficios potenciales son mayores). Puede parecer que el riesgo está compuesto por «prisas» en una implementación preliminar, pero en cambio puede reducir el riesgo en un proyecto comodín porque está descubriendo pronto si es viable un enfoque particular para el problema. El desarrollo del producto es gestión de riesgos.

A menudo se propone que «construya uno desechable». Con la POO, todavía debe andar *parte* de este camino, pero debido a que el código está encapsulado en clases, durante la primera iteración inevitablemente producirá algunos diseños de clases útiles y desarrollará algunas ideas válidas sobre el diseño del sistema que no necesariamente son desechables. De esta manera, la primera pasada rápida al problema no produce sólo información crítica para la siguiente iteración de análisis, diseño, e implementación, sino que además crea el código base para esa iteración.

Es decir, si está buscando una metodología que contenga detalles tremendos y sugiera muchos pasos y documentos, es aún más difícil saber cuándo parar. Tenga presente lo que está intentando encontrar:

1. ¿Cuáles son los objetos? (¿Cómo divide su proyecto en sus partes componentes?)
2. ¿Cuáles son sus interfaces? (¿Qué mensajes necesita enviar a otros objetos?)

Si sólo cuenta con los objetos y sus interfaces, entonces puede escribir un programa. Por varias razones podría necesitar más descripciones y documentos, pero no puede hacerlo con menos.

El proceso se puede realizar en cinco fases, y una fase 0 que es simplemente el compromiso inicial de usar algún tipo de estructura.

1.9.1. Fase 0: Hacer un plan

Primero debe decidir qué pasos va a dar en su proceso. Parece fácil (de hecho, todo esto parece fácil) y sin embargo la gente a menudo no toma esta decisión antes de ponerse a programar. Si su plan es «ponerse directamente a programar», de acuerdo (a veces es adecuado cuando es un problema bien conocido). Al menos estará de acuerdo en que eso es el plan.

También debe decidir en esta fase si necesita alguna estructura de proceso adicional, pero no las nueve yardas completas. Bastante comprensible, algunos programadores prefieren trabajar en «modo vacaciones» en cuyo caso no se impone ninguna estructura en el proceso de desarrollo de su trabajo; «Se hará cuando se haga». Eso puede resultar atractivo durante un tiempo, pero se ha descubierto que tener unos pocos hitos a lo largo del camino ayuda a enfocar e impulsar sus esfuerzos en torno a esos hitos en lugar de empezar a atascarse con el único objetivo de «finalizar el proyecto». Además, divide el proyecto en piezas más pequeñas y hace que dé menos miedo (y además los hitos ofrecen más oportunidades para celebraciones).

Cuando empecé a estudiar la estructura de la historia (por eso algún día escribiré una novela) inicialmente me resistía a la idea de una estructura, sentía que cuando escribía simplemente permitía que fluyera en la página. Pero más tarde me di cuenta de que cuando escribo sobre computadoras la estructura es bastante clara, pero no pienso mucho sobre ello. Pero aún así estructuro mi trabajo, aunque sólo semi-inconscientemente en mi cabeza. Si aún piensa que su plan es sólo ponerse a codificar, de algún modo, usted pasará por las posteriores fases mientras pregunta y responde ciertas cuestiones.

Declaración de objetivos

Cualquier sistema construido, no importa cuan complicado sea, tiene un propósito fundamental, el negocio que hay en él, la necesidad básica que satisface. Si puede ver la interfaz de usuario, el hardware o los detalles específicos del sistema, los algoritmos de codificación y los problemas de eficiencia, finalmente encontrará el núcleo de su existencia, simple y sencillo. Como el así llamado *concepto de alto nivel* de una película de Hollywood, puede describirlo en una o dos frases. Esta descripción pura es el punto de partida.

El concepto de alto nivel es bastante importante porque le da el tono a su proyecto; es una declaración de principios. No tiene porqué conseguirlo necesariamente la primera vez (podría tener que llegar a una fase posterior del proyecto antes de tenerlo completamente claro), pero siga intentándolo hasta que lo consiga. Por ejemplo, en un sistema de control de tráfico aéreo puede empezar con un concepto de alto nivel centrado en el sistema que está construyendo: «El programa de la torre sigue la pista a los aviones». Pero considere qué ocurre cuando adapta el sistema para un pequeño aeropuerto; quizá sólo haya un controlador humano o ninguno. Un modelo más útil no se preocupará de la solución que está creando tanto como la descripción del problema: «Llega un avión, descarga, se revisa y recarga, y se marcha».

1.9.2. Fase 1: ¿Qué estamos haciendo?

En la generación previa de diseño de programas (llamado *diseño procedural*), esto se llamaba «crear el *análisis de requisitos* y *especificación del sistema*». Éstos, por supuesto, eran lugares donde perderse; documentos con nombres intimidantes que podrían llegar a ser grandes proyectos en sí mismos. Sin embargo, su intención era buena. El análisis de requisitos dice: «Haga una lista de las directrices que usará para saber cuándo ha hecho su trabajo y el cliente estará satisfecho». La especificación del sistema dice: «Hay una descripción de *lo que* hará el programa (no *cómo*) por satisfacer los requisitos». El análisis de requisitos es realmente un contrato entre usted y el cliente (incluso si el cliente trabaja dentro de su compañía o es algún otro objeto o sistema). Las especificaciones del sistema son una exploración de alto nivel del problema y en algún sentido un descubrimiento de si se puede hacer y cuánto se tardará. Dado que ambos requerirán consenso entre la gente (y porque suelen cambiar todo el tiempo), creo que es mejor mantenerlos todo lo escueto posible -en el mejor de los casos, listas y diagramas básicos- para ahorrar tiempo. Podría tener otras restricciones que le exijan ampliarla en documentos más grandes, pero manteniendo el documento inicial pequeño y conciso, puede crearse en algunas sesiones de tormentas de ideas de grupo con un líder que cree la descripción dinámicamente. Esto no sólo solicita participación de todos, también fomenta aprobación inicial y llegar a acuerdos entre todos. Quizá lo más importante sea empezar el proyecto con mucho entusiasmo.

Es necesario no perder de vista lo que está intentando conseguir en esta fase: determinar el sistema que se supone que quiere hacer. La herramienta más valiosa para eso es una colección de los llamados «casos de uso». Los casos de uso identifican características clave en el sistema que pueden revelar algunas de las clases fundamentales que se usarán. En esencia son respuestas descriptivas a preguntas como: [9]:

1. «¿Quién usará el sistema?»
2. «¿Qué pueden hacer estos actores con el sistema?»

[9] Gracias a James H Jarrett por su ayuda.

3. «¿Cómo puede este actor hacer eso con este sistema?»
4. «¿Cómo podría alguien más hacer este trabajo si alguien más estuviera haciéndolo, o si el mismo actor tuviera un objetivo diferente?» (para revelar variaciones).
5. «¿Qué problemas podrían ocurrir mientras hace esto con el sistema?» (para revelar excepciones).

Si está diseñando un cajero automático, por ejemplo, el caso de uso para un aspecto particular de la funcionalidad del sistema es poder describir qué hace el contestador automático en todas las situaciones posibles. Cada una de esas «situaciones» se denomina *escenario*, y se puede considerar que un caso de uso es una colección de escenarios. Puede pensar en un escenario como una pregunta que comienza con: «¿Qué hace el sistema si...?» Por ejemplo, «¿Qué hace el cajero automático si un cliente ingresa un cheque dentro de las 24 horas y no hay suficiente en la cuenta para proporcionar la nota para satisfacer el cargo?»

Los diagramas de caso de uso son intencionadamente simples para impedir que se atasque con los detalles de implementación del sistema demasiado pronto:

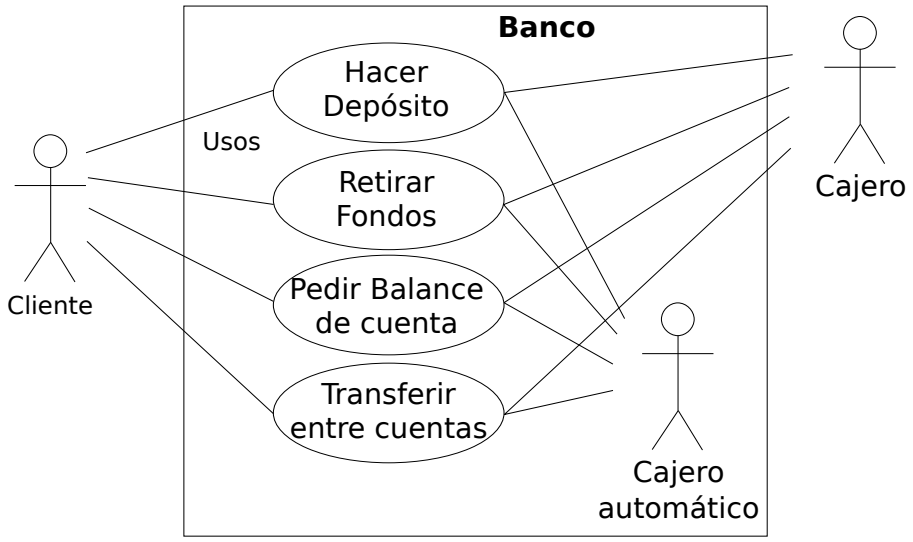

Figura 1.10: Diagramas de casos de uso

Cada monigote representa un «actor», que típicamente es un humano o algún otro tipo de agente libre. (Incluso puede ser otro sistema de computación, como es el caso del «ATM»). La caja representa el límite del sistema. Las elipses representan los casos de uso, los cuales son descripciones de trabajo válido que se puede llevar a cabo con el sistema. Las líneas entre los actores y los casos de uso representan las interacciones.

No importa cómo está implementado realmente el sistema, mientras se lo parezca al usuario.

Un caso de uso no necesita ser terriblemente complejo, incluso si el sistema subyacente es complejo. Lo único que se persigue es mostrar el sistema tal como aparece ante el usuario. Por ejemplo:

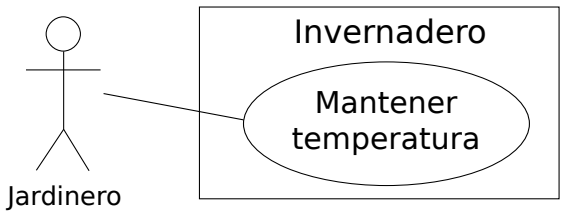

Figura 1.11: Un ejemplo de caso de uso

Los casos de uso producen las especificaciones de requisitos determinando todas las interacciones que el usuario puede tener con el sistema. Intente descubrir una serie completa de casos de uso para su sistema, y una vez que lo haya hecho tendrá lo esencial sobre lo que se supone que hace su sistema. Lo bueno de centrarse en casos de uso es que siempre le lleva de vuelta a lo esencial y le mantiene alejado de los asuntos no críticos para conseguir terminar el trabajo. Es decir, si tiene una serie completa de casos de uso puede describir su sistema y pasar a la siguiente fase. Probablemente no lo hará todo perfectamente en el primer intento, pero no pasa nada. Todo le será revelado en su momento, y si pide una especificación del sistema perfecta en este punto se atascará.

Si se ha atascado, puede reactivar esta fase usando una herramienta tosca de aproximación: describir el sistema en pocos párrafos y después buscar sustantivos y verbos. Los nombres pueden sugerir actores, contexto del caso de uso (ej. «lobby»), o artefactos manipulados en el caso de uso. Los verbos pueden sugerir interacción entre actores y casos de uso, y pasos específicos dentro del caso de uso. Además descubrirá que nombres y verbos producen objetos y mensajes durante la fase de diseño (y observe que los casos de uso describen interacciones entre subsistemas, así que la técnica «nombre y verbo» sólo se puede usar como una herramienta de lluvia de ideas puesto que no genera casos de uso) [10].

El límite entre un caso de uso y un actor puede mostrar la existencia de una interfaz de usuario, pero no la define. Si le interesa el proceso de definición y creación de interfaces de usuario, vea *Software for Use* de Larry Constantine y Lucy Lockwood, (Addison Wesley Longman, 1999) o vaya a www.ForUse.com.

Aunque es un arte oscuro, en este punto es importante hacer algún tipo de estimación de tiempo básica. Ahora tiene una visión general de qué está construyendo así que probablemente será capaz de tener alguna idea de cuánto tiempo llevará. Aquí entran en juego muchos factores. Si hace una estimación a largo plazo entonces la compañía puede decidir no construirlo (y usar sus recursos en algo más razonable -eso es *bueno*). O un gerente puede tener ya decidido cuánto puede durar un proyecto e intentar influir en su estimación. Pero es mejor tener una estimación honesta desde el principio y afrontar pronto las decisiones difíciles. Ha habido un montón de intentos de crear técnicas de estimación precisas (como técnicas para predecir la bolsa), pero probablemente la mejor aproximación es confiar en su experiencia e intuición. Utilice su instinto para predecir cuánto tiempo llevará tenerlo terminado, entonces multiplique por dos y añada un 10%. Su instinto visceral probablemente sea correcto; *puede* conseguir algo contando con este tiempo. El «doble» le permitirá convertirlo en algo decente, y el 10% es para tratar los refinamientos y detalles fi-

[10] Puede encontar más información sobre casos de uso en *Applying Use Cases* de Schneider & Winters (Addison-Wesley 1998) y *Use Case Driven Object Modeling with UML* de Rosenberg (Addison-Wesley 1999).

nales [11]. Sin embargo, usted quiere explicarlo, y a pesar de quejas y manipulaciones que ocurren cuando publique la estimación, parece que esta regla funciona.

1.9.3. Fase 2: ¿Cómo podemos construirlo?

En esta fase debe aparecer un diseño que describa qué clases hay y cómo interactúan. Una técnica excelente para determinar clases es la tarjeta *Clase-Responsabilidad-Colaboración* (*Class-Responsibility-Collaboration*) o CRC. Parte del valor de esta herramienta es que es baja-tecnología: empieza con una colección de 3 a 5 tarjeta en blanco, y se escribe sobre ellas. Cada tarjeta representa una única clase, y en ella se escribe:

1. El nombre de la clase. Es importante que el nombre refleje la esencia de lo que hace la clase, así todo tiene sentido con un simple vistazo.
2. Las «responsabilidades» de la clase: qué debe hacer. Típicamente se puede resumir por la misma declaración de las funciones miembro o métodos (ya que esos nombres pueden ser descritos en un buen diseño), pero no descarte otras notas. Si necesita hacer una selección previa, mire el problema desde un punto de vista de programador perezoso: ¿Qué objetos quiere que aparezcan por arte de magia para resolver su problema?
3. Las «colaboraciones» de la clase: ¿qué otras clases interactúan con ésta? «Interacción» es un término amplio a propósito; puede significar agregación o simplemente que algún otro objeto que lleva a cabo servicios para un objeto de la clase. Las colaboraciones deberían considerar también la audiencia para esta clase. Por ejemplo, si crea una clase `Petardo`, ¿quién va a observarlo, un `Químico` o un `Espectador`? El primero puede querer saber qué componentes químicos se han usado en su construcción, y el último responderá a los colores y figuras que aparezcan cuando explote.

Puede creer que las fichas pueden ser más grandes por toda la información que pondrá en ellas, pero son pequeñas a propósito, no sólo para que las clases se mantengan pequeñas también para evitar tener que manejar demasiados detalles demasiado pronto. Si no puede apuntar todo lo que necesita saber sobre una clase en una ficha pequeña, la clase es demasiado compleja (a está poniendo demasiados detalles, o debería crear más de una clase). La clase ideal se entiende con un vistazo. La idea de las fichas CRC es ayudarle a realizar un acercamiento con un primer corte del diseño y que pueda obtener una visión global y después refinar su diseño.

Uno de los mayores beneficios de las tarjetas CRC es la comunicación. Se hace mejor en tiempo-real, en grupo, sin computadores. Cada persona es responsable de varias clases (que al principio no tienen nombres ni otra información). Haga una simulación en vivo resolviendo un escenario cada vez, decidiendo qué mensajes envía a varios objetos para satisfacer las necesidades de cada escenario. Al pasar por este proceso, descubrirá las clases que necesita con sus responsabilidades y colaboraciones, rellene las tarjetas del mismo modo. Cuando haya pasado por todos los casos de uso, debería disponer de un primer corte bastante completo su diseño.

Antes de empezar a usar fichas CRC, las mayoría de las experiencias de consultoría exitosas las tuve cuando me enfrentaba con un diseño inicial complicado estando

[11] Últimamente mi idea respeto a esto ha cambiado. Doblar y añadir un 10% puede darle una estimación bastante acertada (asumiendo que no hay demasiados factores comodín), pero debe trabajar con bastante diligencia para acabar a tiempo. Si realmente quiere tiempo para hacerlo de forma elegante y estar orgulloso del proceso, el multiplicador correcto es más bien tres o cuatro veces, creo yo.

al frente de un equipo, que no había construido un proyecto POO antes, y dibujando objetos en un pizarra blanca. Hablábamos sobre cómo los objetos deberían comunicarse unos con otros, y borrábamos algunos de ellos para reemplazarlos por otros objetos. Efectivamente, yo gestionaba todas las«tarjetas CRC» en la pizarra. Realmente, el equipo (que conocía lo que el proyecto se suponía tenía que hacer) creó el diseño; ellos «poseían» el diseño en lugar de tener que dárselo. Todo lo que yo hacía era guiar el proceso haciendo las preguntas correctas, poniendo a prueba los suposiciones, y llevando la retroalimentación del equipo para modificar esas suposiciones. La verdadera belleza del proceso era que el equipo aprendía cómo hacer diseños orientado a objetos no revisando ejemplos abstractos, sino trabajando sobre un diseño que era más interesante para ellos en ese momento: los suyos.

Una vez que tenga con una serie de tarjetas CRC, quizá quiera crear una descripción más formal de su diseño usando UML [12]. No necesita usar UML, pero puede servirle de ayuda, especialmente si quiere poner un diagrama en la pared para que todo el mundo lo tenga en cuenta, lo cual es una buena idea. Una alternativa a UML es una descripción textual de los objetos y sus interfaces, o, dependiendo de su lenguaje de programación, el propio código [13].

UML también proporciona una notación de diagramas adicional para describir el modelo dinámico de su sistema. Eso es útil en situaciones en las que las transiciones de estado de un sistema o subsistema son bastante más dominantes de lo que necesitan sus propios diagramas (como en un sistema de control). También puede necesitar describir las estructuras de datos, para sistemas o subsistemas en los que los propios datos son un factor dominante (como una base de datos).

Sabrá qué está haciendo con la fase 2 cuando haya descrito los objetos y sus interfaces. Bien, en muchos de ellos hay algunos que no se pueden conocer hasta la fase 3. Pero está bien. Todo lo que le preocupa es que eventualmente descubra todo sobre sus objetos. Es bueno descubrirlos pronto pero la POO proporciona suficiente estructura de modo que no es grave si los descubre más tarde. De hecho, el diseño de un objeto suele ocurrir en cinco etapas, durante todo el proceso de desarrollo del programa.

Las cinco etapas del diseño de objetos

La vida del diseño de un objeto no se limita a la escritura del programa. En cambio, el diseño de un objeto ocurre en una secuencia de etapas. Es útil tener esta perspectiva porque no debería esperar alcanzar la perfección enseguida; en lugar de eso, se dará cuenta que entender lo que hace un objeto y a qué se debería que ocurre con el tiempo. Esta vista también se aplica al diseño de varios tipos de programas; el patrón para un tipo particular de programas surge a fuerza de pelearse una y otra vez con ese problema (los *Patrones de Diseño* se desarrollan en el Volumen 2). Los objetos, también, tienen sus patrones que surgen del entendimiento, uso y reutilización.

1. Descubrimiento de objetos. Esta etapa ocurre durante el análisis inicial de un programa. Los objetos pueden descubrirse viendo los factores externos y los límites, duplicación de elementos en el sistema, y las unidades conceptuales más pequeñas. Algunos objetos son obvios si se dispone de un conjunto de librerías de clases. Las partes comunes entre clases pueden sugerir clases base y herencia que pueden aparecer pronto, o más tarde en el proceso de diseño.

[12] Para novatos, recomiendo el mencionado *UML Distilled*.

[13] Python (www.python.org) suele utilizarse como «pseudocódigo ejecutable».

2. Montaje de objetos. Si está construyendo un objeto descubrirá la necesidad de nuevos miembros que no aparecen durante la fase de descubrimiento. Las necesidades internas del objeto pueden requerir otras clases que le den soporte.

3. Construcción del sistema. Una vez más, pueden aparecer más requisitos para un objeto a lo largo de esta etapa. Conforme aprende, evoluciona sus objetos. La necesidad de comunicación e interconexión con otros objetos en el sistema puede cambiar las necesidades de sus clases o requerir clases nuevas. Por ejemplo, puede descubrir la necesidad de clases utilería o ayudantes (*helper*), como una lista enlazada, que contienen o no una pequeña información de estado y que simplemente ayudan a la función de otras clases.

4. Extensión del sistema. Cuando añada nuevas características a un sistema puede descubrir que su diseño previo no soportaba extensiones sencillas del sistema. Con esta nueva información, puede reestructurar partes del sistema, posiblemente añadiendo nuevas clases o jerarquía de clases.

5. Reutilización de objetos. Esta es la verdadera prueba de estrés para una clase. Si alguien intenta reutilizarla en una situación completamente nueva, probablemente descubrirá algunos defectos. Si cambia una clase para adaptarla a nuevos programas, los principios generales de la clase se verán más claros, hasta que consiga un tipo verdaderamente reutilizable. Sin embargo, no espere que muchos objetos del diseño de un sistema sean reutilizables -es perfectamente aceptable que la mayor parte de los objetos sean específicos para el sistema. Los tipos reutilizables tienden a ser menos comunes, y deben resolver problemas más generales para ser reutilizables.

Directrices para desarrollo de objetos

Estas etapas sugieren algunas directrices cuando se piensa sobre el desarrollo de clases:

1. Permita que un problema específico dé lugar a una clase, después deje que la clase crezca y madure durante la solución de otros problemas.

2. Recuerde, descubrir las clases que necesita (y sus interfaces) supone la mayor parte del diseño del sistema. Si ya tenía esas clases, será un proyecto fácil.

3. No se esfuerce por saber todo desde el principio; aprenda conforme avanza. Ocurrirá así de todos modos.

4. Comience a programar; consiga tener algo funcionando para poder aprobar o desaprobar su diseño. No tenga miedo a que acabe haciendo código procedural espagueti -las clases dividen el problema y ayudan a controlar la anarquía y la entropía. Las clases malas no estropean las buenas.

5. Manténgalo simple. Pequeños objetos claros con utilidades obvias son mejores que grandes interfaces complicadas. Cuando aparezcan los puntos de decisión, aplique el principio de la Navaja de Occam: Considere las alternativas y elija la más simple, porque las clases simples casi siempre son mejores. Empiece con clases pequeñas y sencillas, y podrá ampliar la interfaz cuando la entienda mejor, pero cuando esto ocurra, será difícil eliminar elementos de la clase.

1.9.4. Fase 3: Construir el núcleo

Esta es la conversión inicial desde el diseño rudo al cuerpo del código compilable y ejecutable que se puede probar, y que aprobará y desaprobará su arquitectura. No es un proceso en un solo paso, más bien es el principio de una serie de pasos que iterativamente construirán el sistema, como verá en la fase 4.

Su objetivo es encontrar el núcleo de la arquitectura de su sistema que hay que implementar para generar un sistema funcional, sin importar lo incompleto que esté el sistema en la pasada inicial. Está creando una estructura que se puede construir con más iteraciones. También está llevando a cabo la primera de muchas integraciones del sistema y pruebas, y dando a los clientes realimentación sobre cómo serán y cómo progresan sus sistemas. Idealmente, también expone algunos de los riesgos críticos. Probablemente descubrirá cambios y mejoras que se pueden hacer en la arquitectura original - cosas que podría no haber aprendido sin implementar el sistema.

Parte de la construcción del sistema es la dosis de realidad que se obtiene al probar su análisis de requisitos y su especificación del sistema (existe de cualquier forma). Asegúrese de que sus pruebas verifican los requisitos y los casos de uso. Cuando el núcleo de su sistema sea estable, estará preparado para progresar y añadir más funcionalidad.

1.9.5. Fase 4: Iterar los casos de uso

Una vez que la estructura del núcleo está funcionando, cada conjunto de características que añade es un pequeño proyecto en sí mismo. Añada una colección de características durante cada *iteración*, un periodo razonablemente corto de desarrollo.

¿Cómo de grande es una iteración? Idealmente, cada iteración dura unas tres semanas (puede cambiar dependiendo del lenguaje de implementación). Al final de ese periodo, tendrá un sistema probado e integrado con más funcionalidades de las que tenía antes. Pero lo que es particularmente interesante son las bases de la iteración: un único caso de uso. Cada caso de uso es un paquete de funcionalidades relacionadas que se puede construir en su sistema de una vez, a lo largo de una iteración. No sólo le da una mejor idea de qué alcance debería tener, también le da más valor a la idea un caso de uso, ya que el concepto no se descarta después del análisis y diseño, sino que es una unidad fundamental de desarrollo durante el proceso de construcción de software.

Se deja de iterar cuando se consigue la funcionalidad deseada o se acaba el plazo impuesto y el cliente está satisfecho con la versión actual. (Recuerde, el software es una subscripción de negocios). Como el proceso es iterativo, tiene muchas oportunidades para enviar un producto en lugar de un simple punto final; los proyectos de software libre trabajan exclusivamente en un entorno iterativo con alta realimentación, que es precisamente la clave de su éxito.

Un proceso de desarrollo iterativo es valioso por muchas razones. Puede mostrar y resolver pronto riesgos críticos, los clientes tienen abundantes oportunidades de cambiar sus opiniones, la satisfacción del programador es más alta, y el proyecto puede dirigirse con más precisión. Pero un beneficio adicional importante es la realimentación para los clientes, los cuales pueden ver en el estado actual del producto exactamente donde se encuentra todo. Esto puede reducir o eliminar la necesidad de abrumadoras reuniones de control y aumentar la confianza y el apoyo de los clientes.

1.9.6. Fase 5: Evolución

Este es el punto en el ciclo de desarrollo que se conoce tradicionalmente como «mantenimiento», un término amplio que puede significar de todo, desde «conseguir que funcione como se supone que debió hacerlo desde el principio» hasta «añadir características que el cliente olvidó mencionar» pasando por el tradicional «arreglar errores que han ido apareciendo» y «añadir nuevas características según se presentan las necesidades». Se han aplicado algunas ideas equivocadas al término «mantenimiento» que se ha tomado en calidad de pequeño engaño, en parte porque sugiere que realmente ha construido un programa primitivo y todo lo que necesita hacer es cambiar partes, engrasarlo, e impedir que se oxide. Quizá haya un término mejor para describir esa tarea.

Yo usaré el término *evolución* [14]. Es decir, «no podrá hacerlo bien la primera vez, pero le dará la oportunidad de aprender y volver atrás y hacer cambios». Puede que necesite hacer muchos cambios hasta que aprenda y entienda el problema con mayor profundidad. La elegancia que obtendrá si evoluciona hasta hacerlo bien valdrá la pena, tanto a corto como a largo plazo. La evolución es donde su programa pasa de bueno a fenomenal, y donde estos usos, que realmente no entiende en un primer momento, pasan a ser más claros después. Es también donde sus clases pueden evolucionar de un uso de único-proyecto a recursos reutilizables.

«Hacerlo bien» no significa sólo que el programa funcione según los requisitos y los casos de uso. Significa que la estructura interna del código tiene sentido, y parece que encaja bien, sin sintaxis difícil, objetos sobredimensionados, o pedazos de código desgarbados. Además, debe tener la sensación de que la estructura del programa sobrevivirá a los cambios que inevitablemente habrá durante su ciclo de vida, y estos cambios pueden hacerse fácil y limpiamente. No es una tarea sencilla. No sólo debe entender lo que está construyendo, sino también cómo evolucionará el programa (lo que yo llamo el *vector de cambio* [15]. Afortunadamente, los lenguajes de programación orientados a objetos son particularmente adecuados para dar soporte a este tipo de modificaciones continuas - los límites creados por los objetos son los que tienden a conservar la estructura frente a roturas. También le permiten hacer cambios - algunos pueden parecer drásticos en un programa procedural - sin causar terremotos en todo su código. En realidad, el soporte para la evolución puede que sea el beneficio más importante de la POO.

Con la evolución, el programador crea algo que al menos se aproxima a lo que piensa que está construyendo, y luego busca defectos, lo compara con sus requisitos y ve lo que falta. Entonces puede volver y arreglarlo rediseñando y re-implementando las porciones del programa que no funcionen bien [16]. Realmente puede necesitar resolver el problema, o un aspecto del mismo, varias veces antes de dar con la solución correcta. (Un estudio de los *Patrones de Diseño*, descrito en el Volumen 2, normalmente resulta útil aquí).

La evolución también ocurre cuando construye un sistema, ve que encaja con

[14] Por lo menos un aspecto de evolución se explica en el libro *Refactoring: improving the design of existing code* (Addison-Wesley 1999) de Martin Fowler. Tenga presente que este libro usa exlusivamente ejemplos en Java.

[15] Este término se explica en el capítulo *Los patrones de diseño* en el Volumen 2

[16] Esto es algo como «prototipado rápido», donde se propone construir un borrador de la versión rápida y sucia que se puede utilizar para aprender sobre el sistema, y entonces puede tirar su prototipo y construir el bueno. El problema con el prototipado rápido es que la gente no tiró el prototipo, y construyó sobre él. Combinado con la falta de estructura en la programación procedural, esto producía a menudo sistemas desordenados que eran difíciles de mantener.

sus requisitos, y entonces descubre que no era realmente lo que buscaba. Cuando ve el sistema en funcionamiento, descubre que realmente quería resolver era problema diferente. Si piensa que este tipo de evolución le va a ocurrir, entonces debe construir su primera versión lo más rápidamente posible para que pueda darse cuenta de si es eso lo que quiere.

Quizás lo más importante a recordar es que por defecto -por definición, realmente- si modifica una clase entonces su superclase -y subclases- seguirán funcionando. Necesita perder el miedo a los cambios (especialmente si tiene un conjunto predefinido de pruebas unitarias para verificar la validez de sus cambios). La modificación no romperá necesariamente el programa, y ningún cambio en el resultado estará limitado a las subclases y/o colaboradores específicos de la clase que cambie.

1.9.7. Los planes valen la pena

Por supuesto, no construiría una casa sin un montón de planos cuidadosamente dibujados. Si construye un piso o una casa para el perro, sus planos no serán muy elaborados pero probablemente empezará con algún tipo de esbozo para guiarle en su camino. El desarrollo de software ha llegado a extremos. Durante mucho tiempo, la gente tenía poca estructura en sus desarrollos, pero entonces grandes proyectos empezaron a fracasar. Como resultado, se acabó utilizando metodologías que tenían una cantidad abrumadora de estructura y detalle, se intentó principalmente para esos grandes proyectos. Estas metodologías eran muy complicadas de usar - la sensación era que se estaba perdiendo todo el tiempo escribiendo documentos y no programando (a menudo era así). Espero haberle mostrado aquí sugerencias a medio camino - una escala proporcional. Usar una propuesta que se ajusta a sus necesidades (y a su personalidad). No importa lo pequeño que desee hacerlo, *cualquier* tipo de plan supondrá una gran mejora en su proyecto respecto a no planear nada. Recuerde que, según la mayoría de las estimaciones, alrededor del 50 % de proyectos fracasan (¡algunas estimaciones superan el 70 %!).

Seguir un plan - preferiblemente uno simple y breve - y esbozar la estructura del diseño antes de empezar a codificar, descubrirá qué cosas caen juntas más fácilmente que si se lanza a programar, y también alcanzará un mayor grado de satisfacción. Mi experiencia me dice que llegar a una solución elegante es profundamente satisfactorio en un nivel completamente diferente; parece más arte que tecnología. Y la elegancia siempre vale la pena; no es una búsqueda frívola. No sólo le permite tener un programa fácil de construir y depurar, también es más fácil de comprender y mantener, y ahí es donde recae su valor económico.

1.10. Programación Extrema

He estudiado técnicas de análisis y diseño, por activa y por pasiva, desde mis estudios universitarios. El concepto de *Programación Extrema* (XP) es el más radical y encantador que he visto nunca. Puede encontrar una crónica sobre el tema en *Extreme Programming Explained* de Kent Beck (Addison-Wesley 2000) y en la web www.xprogramming.com

XP es una filosofía sobre el trabajo de programación y también un conjunto de directrices para hacerlo. Alguna de estas directrices se reflejan en otras metodologías recientes, pero las dos contribuciones más importantes y destacables, en mi opinión, son «escribir primero las pruebas» y la «programación en parejas». Aunque defiende con fuerza el proceso completo, Benk señala que si adopta únicamente estas dos

prácticas mejorará sensiblemente su productividad y fiabilidad.

1.10.1. Escriba primero las pruebas

El proceso de prueba se ha relegado tradicionalmente a la parte final del proyecto, después de que «consiga tener todo funcionando, pero necesite estar seguro». Implícitamente ha tenido una prioridad baja, y la gente que se especializa en ello nunca ha tenido estatus y suele trabajar en el sótano, lejos de los «programadores reales». Los equipos de pruebas han respondido al estereotipo, vistiendo trajes negros y hablando con regocijo siempre que encontraban algo (para ser honesto, yo tenía esa misma sensación cuando encontraba fallos en los compiladores de C++).

XP revoluciona completamente el concepto del proceso de prueba dándole la misma (o incluso mayor) prioridad que al código. De hecho, se escriben las pruebas *antes* de escribir el código que está probando, y las pruebas permanecen con el código siempre. Las pruebas se deben ejecutar con éxito cada vez que hace una integración del proyecto (algo que ocurre a menudo, a veces más de una vez al día).

Escribir primero las pruebas tiene dos efectos extremadamente importantes.

Primero, fuerza una definición clara de la interfaz de la clase. A menudo sugiero que la gente «imagine la clase perfecta para resolver un problema particular» como una herramienta cuando intenta diseñar el sistema. La estrategia del proceso de prueba de XP va más lejos que eso - especifica exactamente cual es el aspecto de la clase, para el consumidor de esa clase, y exactamente cómo debe comportarse la clase. En ciertos términos. Puede escribir toda la prosa, o crear todos los diagramas donde quiera describir cómo debe comportarse una clase y qué aspecto debe tener, pero nada es tan real como un conjunto de pruebas. Lo primero es una lista de deseos, pero las pruebas son un contrato forzado por el compilador y el programa. Es difícil imaginar una descripción más concreta de una clase que las pruebas.

Mientras se crean las pruebas, el programador está completamente forzado a elaborar la clase y a menudo descubrirá necesidades de funcionalidad que habrían sido omitidas durante los experimentos de diagramas UML, tarjetas CRC, casos de uso, etc.

El segundo efecto importante de escribir las pruebas primero procede de la propia ejecución de las pruebas cada vez que hace una construcción del software. Esta actividad le ofrece la otra mitad del proceso de prueba que es efectuado por el compilador. Si mira la evolución de los lenguajes de programación desde esta perspectiva, verá que las mejoras reales en la tecnología giran realmente alrededor del proceso de prueba. El lenguaje ensamblador sólo se fija en la sintaxis, pero C impone algunas restricciones de semántica, y éstas le impiden cometer ciertos tipos de errores. Los lenguajes POO imponen incluso más restricciones semánticas, si lo piensa son realmente formas del proceso de prueba. «¿Se utiliza apropiadamente este tipo de datos? ¿Se invoca esta función del modo correcto?» son el tipo de pruebas que se llevan a cabo por el compilador en tiempo de ejecución del sistema. Se han visto los resultados de tener estas pruebas incorporadas en el lenguaje: la gente ha sido capaz de escribir sistemas más complejos, y han funcionado, con mucho menos tiempo y esfuerzo. He tratado de comprender porqué ocurre eso, pero ahora me doy cuenta de que son las pruebas: el programador hace algo mal, y la red de seguridad de las pruebas incorporadas le dice que hay un problema y le indica dónde.

Pero las pruebas incorporadas que proporciona el diseño del lenguaje no pueden ir más lejos. En este punto, el programador debe intervenir y añadir el resto de las pruebas que producen un juego completo (en cooperación con el compilador y el

tiempo de ejecución del sistema) que verifica el programa completo. Y, del mismo modo que tiene un compilador vigilando por encima de su hombro, ¿no querría que estas pruebas le ayudaran desde el principio? Por eso se escriben primero, y se ejecutan automáticamente con cada construcción del sistema. Sus pruebas se convierten en una extensión de la red de seguridad proporcionada por el lenguaje.

Una de las cosas que he descubierto sobre el uso de lenguajes de programación cada vez más poderosos es que estoy dispuesto a probar experimentos más descarados, porque sé que el lenguaje me ahorra la pérdida de tiempo que supone estar persiguiendo errores. El esquema de pruebas de XP hace lo mismo para el proyecto completo. Como el programador conoce sus pruebas siempre cazará cualquier problema que introduzca (y regularmente se añadirán nuevas pruebas), puede hacer grandes cambios cuando lo necesite sin preocuparse de causar un completo desastre. Eso es increíblemente poderoso.

1.10.2. Programación en parejas

Programar en parejas va en contra del duro individualismo en el que hemos sido adoctrinados desde el principio, a través de la facultad (donde triunfábamos o fracasábamos por nosotros mismos, y trabajar con nuestros vecinos se consideraba «engañoso») y los medios de comunicación, especialmente las películas de Hollywood donde el héroe normalmente lucha contra la estúpida conformidad [17]. Los programadores también se consideran dechados de individualismo -«cowboy coders» como le gusta decir a Larry Constantine. XP, que es en si mismo una batalla contra el pensamiento convencional, dice que el código debería ser escrito por dos personas por estación de trabajo. Y eso se puede hacer en una área con un grupo de estaciones de trabajo, sin las barreras a las que la gente de diseño de infraestructuras tiene tanto cariño. De hecho, Beck dice que la primera tarea de pasarse a XP es llegar con destornilladores y llaves Allen y desmontar todas esas barreras [18]. (Esto requerirá un director que pueda afrontar la ira del departamento de infraestructuras).

El valor de la programación en parejas está en que mientras una persona escribe el código la otra está pensando. El pensador mantiene un visión global en su cabeza, no sólo la imagen del problema concreto, también las pautas de XP. Si dos personas están trabajando, es menos probable que uno de ellos acabe diciendo, «No quiero escribir las pruebas primero», por ejemplo. Y si el programador se atasca, pueden cambiar los papeles. Si ambos se atascan, sus pensamientos pueden ser escuchados por otro en el área de trabajo que puede contribuir. Trabajar en parejas mantiene las cosas en movimiento y sobre la pista. Y probablemente más importante, hace que la programación sea mucho más social y divertida.

He empezado a usar programación en parejas durante los periodos de ejercicio en algunos de mis seminarios y parece mejorar considerablemente la experiencia de todo el mundo.

[17] Aunque esto puede ser una perspectiva americana, las historias de Hollywood llegan a todas partes.

[18] Incluyendo (especialmente) el sistema PA. Una vez trabajé en una compañía que insistía en anunciar públicamente cada llamada de teléfono que llegaba a los ejecutivos, y constantemente interrumpía nuestra productividad (pero los directores no concebían el agobio como un servicio importante de PA). Finalmente, cuando nadie miraba empecé a cortar los cables de los altavoces.

1.11. Porqué triunfa C++

Parte de la razón por la que C++ ha tenido tanto éxito es que la meta no era precisamente convertir C en un lenguaje de POO (aunque comenzó de ese modo), sino también resolver muchos otros problemas orientados a los desarrolladores de hoy en día, especialmente aquellos que tienen grandes inversiones en C. Tradicionalmente, los lenguajes de POO han sufrido de la postura de que debería abandonar todo lo que sabe y empezar desde cero, con un nuevo conjunto de conceptos y una nueva sintaxis, argumentando que es mejor a largo plazo todo el viejo equipaje que viene con los lenguajes procedurales. Puede ser cierto, a largo plazo. Pero a corto plazo, mucho de este equipaje era valioso. Los elementos más valiosos podían no estar en el código base existente (el cual, con las herramientas adecuadas, se podría traducir), sino en el *conocimiento adquirido*. Si usted es un programador C y tiene que tirar todo lo que sabe sobre C para adoptar un nuevo lenguaje, inmediatamente será mucho menos productivo durante muchos meses, hasta que su mente su ajuste al nuevo paradigma. Mientras que si puede apoyarse en su conocimiento actual de C y ampliarlo, puede continuar siendo productivo con lo que realmente sabe mientras se pasa al mundo de la programación orientada a objetos. Como todo el mundo tiene su propio modelo mental de la programación, este cambio es lo suficientemente turbio sin el gasto añadido de volver a empezar con un nuevo modelo de lenguaje. Por eso, la razón del éxito de C++, en dos palabras: es económico. Sigue costando cambiarse a la POO, pero con C++ puede costar menos [19].

La meta de C++ es mejorar la productividad. Ésta viene por muchos caminos, pero el lenguaje está diseñado para ayudarle todo lo posible, y al mismo tiempo dificultarle lo menos posible con reglas arbitrarias o algún requisito que use un conjunto particular de características. C++ está diseñado para ser práctico; las decisiones de diseño del lenguaje C++ estaban basadas en proveer los beneficios máximos al programador (por lo menos, desde la visión del mundo de C).

1.11.1. Un C mejor

Se obtiene una mejora incluso si continúa escribiendo código C porque C++ ha cerrado muchos agujeros en el lenguaje C y ofrece mejor control de tipos y análisis en tiempo de compilación. Está obligado a declarar funciones de modo que el compilador pueda controlar su uso. La necesidad del preprocesador ha sido prácticamente eliminada para sustitución de valores y macros, que eliminan muchas dificultades para encontrar errores. C++ tiene una característica llamada *referencias* que permite un manejo más conveniente de direcciones para argumentos de funciones y retorno de valores. El manejo de nombres se mejora a través de una característica llamada *sobrecarga de funciones*, que le permite usar el mismo nombre para diferentes funciones. Una característica llamada *namespaces* (espacios de nombres) también mejora la seguridad respecto a C.

1.11.2. Usted ya está en la curva de aprendizaje

El problema con el aprendizaje de un nuevo lenguaje es la productividad. Ninguna empresa puede permitirse de repente perder un ingeniero de software productivo porque está aprendiendo un nuevo lenguaje. C++ es una extensión de C, no una nue-

[19] Dije «puede» porque, debido a la complejidad de C++, realmente podría ser más económico cambiarse a Java. Pero la decisión de qué lenguaje elegir tiene muchos factores, y en este libro asumiré que el lector ha elegido C++.

va sintaxis completa y un modelo de programación. Le permite continuar creando código útil, usando las características gradualmente según las va aprendiendo y entendiendo. Puede que ésta sea una de las razones más importantes del éxito de C++.

Además, todo su código C es todavía viable en C++, pero como el compilador de C++ es más delicado, a menudo encontrará errores ocultos de C cuando recompile su código con C++.

1.11.3. Eficiencia

A veces es apropiado intercambiar velocidad de ejecución por productividad de programación. Un modelo económico, por ejemplo, puede ser útil sólo por un periodo corto de tiempo, pero es más importante crear el modelo rápidamente. No obstante, la mayoría de las aplicaciones requieren algún grado de eficiencia, de modo que C++ siempre yerra en la parte de mayor eficiencia. Como los programadores de C tienden a ser muy concienzudos con la eficiencia, ésta es también una forma de asegurar que no podrán argumentar que el lenguaje es demasiado pesado y lento. Algunas características en C++ intentan facilitar el afinado del rendimiento cuando el código generado no es lo suficientemente eficiente.

No sólo se puede conseguir el mismo bajo nivel de C (y la capacidad de escribir directamente lenguaje ensamblador dentro de un programa C++), además la experiencia práctica sugiere que la velocidad para un programa C++ orientado a objetos tiende a ser $\pm 10\%$ de un programa escrito en C, y a menudo mucho menos [20]. El diseño producido por un programa POO puede ser realmente más eficiente que el homólogo en C.

1.11.4. Los sistemas son más fáciles de expresar y entender

Las clases diseñadas para encajar en el problema tienden a expresarlo mejor. Esto significa que cuando escribe el código, está describiendo su solución en los términos del espacio del problema («ponga el FIXME:plástico en el cubo») mejor que en los términos de la computadora, que están en el espacio de la solución («active el bit para cerrar el relé »). Usted maneja conceptos de alto nivel y puede hacer mucho más con una única línea de código.

El otro beneficio de esta facilidad de expresión es el mantenimiento, que (si informa se puede creer) implica una enorme parte del coste del tiempo de vida del programa. Si un programa es más fácil de entender, entonces es más fácil de mantener. También puede reducir el coste de crear y mantener la documentación.

1.11.5. Aprovechamiento máximo con librerías

El camino más rápido para crear un programa es usar código que ya está escrito: una librería. Un objetivo primordial de C++ es hacer más sencillo el uso de las librerías. Esto se consigue viendo las librerías como nuevos tipos de datos (clases), así que crear librerías significa añadir nuevos tipos al lenguaje. Como el compilador C++ se preocupa del modo en que se usa la librería - garantizando una inicialización y limpieza apropiadas, y asegurando que las funciones se llamen apropiadamente - puede centrarse en lo que hace la librería, no en cómo tiene que hacerlo.

[20] Sin embargo, mire en las columnas de Dan Saks en *C/C++ User's Journal* sobre algunas investigaciones importantes sobre el rendimiento de librerías C++.

Como los nombres están jerarquizados según las partes de su programa por medio de los espacios de nombres de C++, puede usar tantas librerías como quiera sin los conflictos de nombres típicos de C.

1.11.6. Reutilización de código fuente con plantillas

Hay una categoría significativa de tipos que requiere modificaciones del código fuente para lograr una reutilización efectiva. Las plantillas de C++ llevan a cabo la modificación del código fuente automáticamente, convirtiéndola en una herramienta especialmente potente para la reutilización del código de las librerías. Si se diseña un tipo usando plantillas funcionará fácilmente con muchos otros tipos. Las plantillas son especialmente interesantes porque ocultan al programador cliente la complejidad de esta forma de reutilizar código.

1.11.7. Manejo de errores

La gestión de errores en C es un problema muy conocido, y a menudo ignorado - cruzando los dedos. Si está construyendo un programa complejo y grande, no hay nada peor que tener un error enterrado en cualquier lugar sin la menor idea de cómo llegó allí. La gestión de excepciones de C++ (introducida en este volumen, y explicada en detalle en el Volumen 2, que se puede descargar de www.BruceEckel.com) es un camino para garantizar que se notifica un error y que ocurre algo como consecuencia.

1.11.8. Programar a lo grande

Muchos lenguajes tradicionales tienen limitaciones propias para hacer programas grandes y complejos. BASIC, por ejemplo, puede valer para solucionar ciertas clases de problemas rápidamente, pero si el programa tiene más de unas cuantas páginas o se sale del dominio de problemas de ese lenguaje, es como intentar nadar a través de un fluido cada vez más viscoso. C también tiene estas limitaciones. Por ejemplo, cuando un programa tiene más de 50.000 líneas de código, los conflictos de nombres empiezan a ser un problema - efectivamente, se queda sin nombres de funciones o variables. Otro problema particularmente malo son los pequeños agujeros en el lenguaje C - errores enterrados en un programa grande que pueden ser extremadamente difíciles de encontrar.

No hay una línea clara que diga cuando un lenguaje está fallando, y si la hubiese, debería ignorarla. No diga: «Mi programa BASIC se ha hecho demasiado grande; ¡lo tendré que reescribir en C!» En su lugar, intente calzar unas cuantas líneas más para añadirle una nueva característica. De ese modo, el coste extra lo decide usted.

C++ está diseñado para ayudarle a *programar a lo grande*, es decir, eliminar las diferencias de complejidad entre un programa pequeño y uno grande. Ciertamente no necesita usar POO, plantillas, espacios de nombres ni manejadores de excepciones cuando esté escribiendo un programa tipo «hola mundo», pero estas prestaciones están ahí para cuando las necesite. Y el compilador es agresivo en la detección de errores tanto para programas pequeños como grandes.

1.12. Estrategias de transición

Si acepta la POO, su próxima pregunta seguramente será: «¿cómo puedo hacer que mi jefe, mis colegas, mi departamento, mis compañeros empiecen a utilizar objetos?» Piense sobre cómo usted -un programador independiente- puede ir aprendiendo a usar un nuevo lenguaje y un nuevo paradigma de programación. Ya lo ha hecho antes. Primero viene la educación y los ejemplos; entonces llega un proyecto de prueba que le permita manejar los conceptos básicos sin que se vuelva demasiado confuso. Después llega un proyecto del «mundo real» que realmente hace algo útil. Durante todos sus primeros proyectos continúa su educación leyendo, preguntando a expertos, e intercambiando consejos con amigos. Este es el acercamiento que sugieren muchos programadores experimentados para el cambio de C a C++. Por supuesto, cambiar una compañía entera introduce ciertas dinámicas de grupo, pero puede ayudar en cada paso recordar cómo lo haría una persona.

1.12.1. Directrices

Aquí hay algunas pautas a considerar cuando se hace la transición a POO y C++:

Entrenamiento

El primer paso es algún tipo de estudio. Recuerde la inversión que la compañía tiene en código C, e intente no tenerlo todo desorganizado durante seis o nueve meses mientras todo el mundo alucina con la herencia múltiple. Elija un pequeño grupo para formarlo, preferiblemente uno compuesto de gente que sea curiosa, trabaje bien junta, y pueda funcionar como su propia red de soporte mientras están aprendiendo C++.

Un enfoque alternativo que se sugiere a veces es la enseñanza a todos los niveles de la compañía a la vez, incluir una visión general de los cursos para gerentes estratégicos es tan bueno como cursos de diseño y programación para trabajadores de proyectos. Es especialmente bueno para compañías más pequeñas al hacer cambios fundamentales en la forma en la que se hacen cosas, o en la división de niveles en compañías más grandes. Como el coste es mayor, sin embargo, se puede cambiar algo al empezar con entrenamiento de nivel de proyecto, hacer un proyecto piloto (posiblemente con un mentor externo), y dejar que el equipo de trabajo se convierta en los profesores del resto de la compañía.

Proyectos de bajo riesgo

Pruebe primero con un proyecto de bajo riesgo que permita errores. Una vez que adquiera alguna experiencia, puede acometer cualquier otro proyecto con miembros del primer equipo o usar los miembros del equipo como una plantilla de soporte técnico de POO. Este primer proyecto puede que no funcione bien la primera vez, pero no debería ser una tarea crítica para la compañía. Debería ser simple, autocontenido, e instructivo; eso significa que suele implicar la creación de clases que serán significativas para otros programadores en la compañía cuando les llegue el turno de aprender C++.

Modelar desde el éxito

Buscar ejemplos de un buen diseño orientado a objetos antes de partir de cero. Hay una gran probabilidad de que alguien ya haya resuelto su problema, y si ellos no lo han resuelto probablemente puede aplicar lo que ha aprendido sobre abstracción

para modificar un diseño existente y adecuarlo a sus necesidades. Este es el concepto general de los patrones de diseño, tratado en el Volumen 2.

Use librerías de clases existentes

La primera motivación económica para cambiar a POO es el fácil uso de código existente en forma de librerías de clases (en particular, las librerías Estándar de C++, explicadas en profundidad en el Volumen 2 de este libro). El ciclo de desarrollo de aplicación más corto ocurrirá cuando sólo tenga que escribir la función `main()`, creando y usando objetos de las librerías de fábrica. No obstante, algunos programadores nuevos no lo entienden, no son conscientes de la existencia de librerías de clases, o, a través de la fascinación con el lenguaje, desean escribir clases que ya existen. Su éxito con POO y C++ se optimizará si hace un esfuerzo por buscar y reutilizar código de otras personas desde el principio del proceso de transición.

No reescriba en C++ código que ya existe

Aunque compilar su código C con un compilador de C++ normalmente produce (de vez en cuando tremendos) beneficios encontrando problemas en el viejo código, normalmente coger código funcional existente y reescribirlo en C++ no es la mejor manera de aprovechar su tiempo. (Si tiene que convertirlo en objetos, puede «envolver» el código C en clases C++). Hay beneficios incrementales, especialmente si es importante reutilizar el código. Pero esos cambios no le van a mostrar los espectaculares incrementos en productividad que espera para sus primeros proyectos a menos que ese proyecto sea nuevo. C++ y la POO destacan más cuando un proyecto pasa del concepto a la realidad.

1.12.2. Obstáculos de la gestión

Si es gerente, su trabajo es adquirir recursos para su equipo, para superar las barreras en el camino del éxito de su equipo, y en general para intentar proporcionar el entorno más productivo y agradable de modo que sea más probable que su equipo realice esos milagros que se le piden siempre. Cambiar a C++ cae en tres de estas categorías, y puede ser maravilloso si no le costara nada. Aunque cambiar a C++ puede ser más económico - dependiendo de sus restricciones [21] - como las alternativas de la POO para un equipo de programadores de C (y probablemente para programadores en otros lenguajes procedurales), no es gratis, y hay obstáculos que debería conocer antes de intentar comunicar el cambio a C++ dentro de su compañía y embarcarse en el cambio usted mismo.

Costes iniciales

El coste del cambio a C++ es más que solamente la adquisición de compiladores C++ (el compilador GNU de C++, uno de los mejores, es libre y gratuito). Sus costes a medio y largo plazo se minimizarán si invierte en formación (y posiblemente un mentor para su primer proyecto) y también si identifica y compra librerías de clases que resuelvan su problema más que intentar construir las librerías usted mismo. Hay costes que se deben proponer en un proyecto realista. Además, están los costes ocultos en pérdidas de productividad mientras se aprende el nuevo lenguaje y posiblemente un nuevo entorno de programación. Formar y orientar puede minimizar ese efecto, pero los miembros del equipo deben superar sus propios problemas pa-

[21] Para mejora de la productividad, debería considerar también el lenguaje Java.

ra entender la nueva tecnología. A lo largo del proceso ellos cometerán más errores (esto es una ventaja, porque los errores reconocidos son el modo más rápido para aprender) y ser menos productivos. Incluso entonces, con algunos tipos de problemas de programación, las clases correctas y el entorno de programación adecuado, es posible ser más productivo mientras se está aprendiendo C++ (incluso considerando que está cometiendo más errores y escribiendo menos líneas de código por día) que si estuviera usando C.

Cuestiones de rendimiento

Una pregunta común es, «¿La POO no hace automáticamente mis programas mucho más grandes y lentos?» La respuesta es: «depende». Los lenguajes de POO más tradicionales se diseñaron con experimentación y prototipado rápido más que pensando en la eficiencia. De esta manera, prácticamente garantiza un incremento significativo en tamaño y una disminución en velocidad. C++ sin ambargo, está diseñado teniendo presente la producción de programación. Cuando su objetivo es un prototipado rápido, puede lanzar componentes juntos tan rápido como sea posible ignorando las cuestiones de eficiencia. Si está usando una librerías de otros, normalmente ya están optimizadas por sus vendedores; en cualquier caso no es un problema mientras está en un modo de desarrollo rápido. Cuando tenga el sistema que quiere, si es bastante pequeño y rápido, entonces ya está hecho. Si no, lo puede afinar con una herramienta de perfilado, mire primero las mejoras que puede conseguir aplicando las características que incorpora C++. Si esto no le ayuda, mire las modificaciones que se pueden hacer en la implementación subyacente de modo que no sea necesario cambiar ningún código que utilice una clase particular. Únicamente si ninguna otra cosa soluciona el problema necesitará cambiar el diseño. El hecho de que el rendimiento sea tan crítico en esta fase del diseño es un indicador de que debe ser parte del criterio del diseño principal. FIXME:Usar un desarrollo rápido tiene la ventaja de darse cuenta rápidamente.

Como se mencionó anteriormente, el número dado con más frecuencia para la diferencia en tamaño y velocidad entre C y C++ es 10%, y a menudo menor. Incluso podría conseguir una mejora significativa en tamaño y velocidad cuando usa C++ más que con C porque el diseño que hace para C++ puede ser bastante diferente respecto al que hizo para C.

La evidencia entre las comparaciones de tamaño y velocidad entre C y C++ tienden a ser anecdóticas y es probable que permanezcan así. A pesar de la cantidad de personas que sugiere que una compañía intenta el mismo proyecto usando C y C++, probablemente ninguna compañía quiere perder dinero en el camino a no ser que sea muy grande y esté interesada en tales proyectos de investigación. Incluso entonces, parece que el dinero se puede gastar mejor. Casi universalmente, los programadores que se han cambiado de C (o cualquier otro lenguaje procedural) a C++ (o cualquier otro lenguaje de POO) han tenido la experiencia personal de una gran mejora en su productividad de programación, y es el argumento más convincente que pueda encontrar.

Errores comunes de diseño

Cuando su equipo empieza con la POO y C++, típicamente los programadores pasan por una serie de errores de diseño comunes. Esto ocurre a menudo porque hay poca realimentación de expertos durante el diseño e implementación de los proyectos iniciales, porque ningún experto ha sido desarrollador dentro de la compañía y puede haber resistencia a contratar consultores. Es fácil pensar que se entiende la POO demasiado pronto en el ciclo y se va por el mal camino. Algo que es obvio para

alguien experimentado con el lenguaje puede ser un tema de gran debate interno para un novato. La mayor parte de este trauma se puede olvidar usando un experto externo para enseñar y tutorizar.

Por otro lado, el hecho de que estos errores de diseño son fáciles de cometer, apunta al principal inconveniente de C++: su compatibilidad con C (por supuesto, también es su principal fortaleza). Para llevar a cabo la hazaña de ser capaz de compilar código C, el lenguaje debe cumplir algunos compromisos, lo que ha dado lugar a algunos «rincones oscuros». Esto es una realidad, y comprende gran parte de la curva de aprendizaje del lenguaje. En este libro y en el volumen posterior (y en otros libros; ver el Apéndice C), intento mostrar la mayoría de los obstáculos que probablemente encontrará cuando trabaje con C++. Debería ser consciente siempre de que hay algunos agujeros en la red de seguridad.

1.13. Resumen

Este capítulo intenta darle sentido a los extensos usos de la programación orientada a objetos y C++, incluyendo el porqué de que la POO sea diferente, y porqué C++ en particular es diferente, conceptos de metodología de POO, y finalmente los tipos de cuestiones que encontrará cuando cambie su propia compañía a POO y C++.

La POO y C++ pueden no ser para todos. Es importante evaluar sus necesidades y decidir si C++ satisfará de forma óptima sus necesidades, o si podría ser mejor con otros sistemas de programación (incluido el que utiliza actualmente). Si sabe que sus necesidades serán muy especializadas en un futuro inmediato y tiene restricciones específicas que no se pueden satisfacer con C++, entonces debe investigar otras alternativas [22]. Incluso si finalmente elige C++ como su lenguaje, por lo menos entenderá qué opciones había y tendrá una visión clara de porqué tomó esa dirección.

El lector conoce el aspecto de un programa procedural: definiciones de datos y llamadas a funciones. Para encontrar el significado de un programa tiene que trabajar un poco, revisando las llamadas a función y los conceptos de bajo nivel para crear un modelo en su mente. Esta es la razón por la que necesitamos representaciones intermedias cuando diseñamos programas procedurales - por eso mismo, estos programas tienden a ser confusos porque los términos de expresión están orientados más hacia la computadora que a resolver el problema.

Como C++ añade muchos conceptos nuevos al lenguaje C, puede que su asunción natural sea que el `main()` en un programa de C++ será mucho más complicado que el equivalente del programa en C. En eso, quedará gratamente sorprendido: un programa C++ bien escrito es generalmente mucho más simple y mucho más sencillo de entender que el programa equivalente en C. Lo que verá son las definiciones de los objetos que representan conceptos en el espacio de su problema (en lugar de cuestiones de la representación en el computador) y mensajes enviados a otros objetos para representar las actividades en este espacio. Ese es uno de los placeres de la programación orientada a objetos, con un programa bien diseñado, es fácil entender el código leyéndolo. Normalmente hay mucho menos código, en parte, porque muchos de sus problemas se resolverán utilizando código de librerías existentes.

[22] En particular, recomiendo mirar Java http://java.sun.com y Python http://www.python.org.

2: Construir y usar objetos

Este capítulo presenta la suficiente sintaxis y los conceptos de construcción de programas de C++ como para permitirle crear y ejecutar algunos programas simples orientados a objetos. El siguiente capítulo cubre la sintaxis básica de C y C++ en detalle.

Leyendo primero este capítulo, le cogerá el gustillo a lo que supone programar con objetos en C++, y también descubrirá algunas de las razones por las que hay tanto entusiasmo alrededor de este lenguaje. Debería ser suficiente para pasar al Capítulo 3, que puede ser un poco agotador debido a que contiene la mayoría de los detalles del lenguaje C.

Los tipos de datos definidos por el usuario, o *clases* es lo que diferencia a C++ de los lenguajes procedimentales tradicionales. Una clase es un nuevo tipo de datos que usted o alguna otra persona crea para resolver un problema particular. Una vez que se ha creado una clase, cualquiera puede utilizarla sin conocer los detalles de su funcionamiento, o incluso de la forma en que se han construído. Este capítulo trata las clases como si sólo fueran otro tipo de datos predefinido disponible para su uso en programas.

Las clases creadas por terceras personas se suelen empaquetar en librerías. Este capítulo usa algunas de las librerías que vienen en todas las implementaciones de C++. Una librería especialmente importante es FIXME:iostreams, que le permite (entre otras cosas) leer desde ficheros o teclado, y escribir a ficheros o pantalla. También verá la clase `string`, que es muy práctica, y el contenedor `vector` de la Libreria Estándar de C++. Al final del capítulo, verá lo sencillo que resulta utilizar una librería de clases predefinida.

Para que pueda crear su primer programa debe conocer primero las herramientas utilizadas para construir aplicaciones.

2.1. El proceso de traducción del lenguaje

Todos los lenguajes de programación se traducen de algo que suele ser fácilmente entendible por una persona (*código fuente*) a algo que es ejecutado por una computadora (*código máquina*). Los traductores se dividen tradicionalmente en dos categorías: *intérpretes* y *compiladores*.

2.1.1. Intérpretes

Un intérprete traduce el código fuente en actividades (las cuales pueden comprender grupos de instrucciones máquina) y ejecuta inmediatamente estas activida-

des. El BASIC, por ejemplo, fue un lenguaje interpretado bastante popular. Los intérpretes de BASIC tradicionales traducen y ejecutan una línea cada vez, y después olvidan la línea traducida. Esto los hace lentos debido a que deben volver a traducir cualquier código que se repita. BASIC también ha sido compilado para ganar en velocidad. La mayoría de los intérpretes modernos, como los de Python, traducen el programa entero en un lenguaje intermedio que es ejecutable por un intérprete mucho más rápido [1].

Los intérpretes tienen muchas ventajas. La transición del código escrito al código ejecutable es casi inmediata, y el código fuente está siempre disponible, por lo que el intérprete puede ser mucho más específico cuando ocurre un error. Los beneficios que se suelen mencionar de los intérpretes es la facilidad de interacción y el rápido desarrollo (pero no necesariamente ejecución) de los programas.

Los lenguajes interpretados a menudo tienen severas limitaciones cuando se construyen grandes proyectos (Python parece ser una excepción). El intérprete (o una versión reducida) debe estar siempre en memoria para ejecutar el código e incluso el intérprete más rápido puede introducir restricciones de velocidad inaceptables. La mayoría de los intérpretes requieren que todo el código fuente se les envíe de una sola vez. Esto no sólo introduce limitaciones de espacio, sino que puede causar errores difíciles de detectar si el lenguaje no incluye facilidades para localizar el efecto de las diferentes porciones de código.

2.1.2. Compiladores

Un compilador traduce el código fuente directamente a lenguaje ensamblador o instrucciones máquina. El producto final suele ser uno o varios ficheros que contienen código máquina. La forma de realizarlo suele ser un proceso que consta de varios pasos. La transición del código escrito al código ejecutable es significativamente más larga con un compilador.

Dependiendo de la perspicacia del escritor del compilador, los programas generados por un compilador tienden a requerir mucho menos espacio para ser ejecutados, y se ejecutan mucho más rápido. Aunque el tamaño y la velocidad son probablemente las razones más citadas para usar un compilador, en muchas situaciones no son las más importantes. Algunos lenguajes (como el C) están diseñados para admitir trozos de programas compilados independientemente. Estas partes terminan combinando en un programa *ejecutable* final mediante una herramienta llamada *enlazador* (*linker*). Este proceso se conoce como *compilación separada*.

La compilación separada tiene muchos beneficios. Un programa que, tomado de una vez, excedería los límites del compilador o del entorno de compilación puede ser compilado por piezas. Los programas se pueden ser construir y probar pieza a pieza. Una vez que una parte funciona, se puede guardar y tratarse como un bloque. Los conjuntos de piezas ya funcionales y probadas se pueden combinar en *librerías* para que otros programadores puedan usarlos. Como se crean piezas, la complejidad de las otras piezas se mantiene oculta. Todas estas características ayudan a la creación de programas grandes, [2].

Las características de depuración del compilador han mejorado considerable-

[1] Los límites entre los compiladores y los intérpretes tienden a ser difusos, especialmente con Python, que tiene muchas de las caractéristicas y el poder de un lenguaje compilado pero también tiene parte de las ventajas de los lenguajes interpretados.

[2] Python vuelve a ser una excepción, debido a que permite compilación separada.

mente con el tiempo. Los primeros compiladores simplemente generaban código máquina, y el programador insertaba sentencias de impresión para ver qué estaba ocurriendo, lo que no siempre era efectivo. Los compiladores modernos pueden insertar información sobre el código fuente en el programa ejecutable. Esta información se usa por poderosos *depuradores a nivel de código* que muestran exactamente lo que pasa en un programa rastreando su progreso mediante su código fuente.

Algunos compiladores solucionan el problema de la velocidad de compilación mediante *compilación en memoria*. La mayoría de los compiladores trabajan con ficheros, leyéndolos y escribiéndolos en cada paso de los procesos de compilación. En la compilación en memoria el compilador se mantiene en RAM. Para programas pequeños, puede parecerse a un intérprete.

2.1.3. El proceso de compilación

Para programar en C y en C++, es necesario entender los pasos y las herramientas del proceso de compilación. Algunos lenguajes (C y C++, en particular) empiezan la compilación ejecutando un *preprocesador* sobre el código fuente. El preprocesador es un programa simple que sustituye patrones que se encuentran en el código fuente con otros que ha definido el programador (usando las *directivas de preprocesado*). Las directivas de preprocesado se utilizan para ahorrar escritura y para aumentar la legibilidad del código (posteriormente en este libro, aprenderá cómo el diseño de C++ desaconseja en gran medida el uso del preprocesador, ya que puede causar errores sutiles). El código preprocesado se suele escribir en un fichero intermedio.

Normalmente, los compiladores hacen su trabajo en dos pasadas. La primera pasada consiste en analizar sintácticamente el código generado por el preprocesador. El compilador trocea el código fuente en pequeñas partes y lo organiza en una estructura llamada *árbol*. En la expresión FIXME:«A+B», los elementos «A», «+», «B» son hojas del árbol.

A menudo se utiliza un *optimizador global* entre el primer y el segundo paso para producir código más pequeño y rápido.

En la segunda pasada, el *generador de código* recorre el árbol sintáctico y genera lenguaje ensamblador o código máquina para los nodos del árbol. Si el generador de código crea lenguaje ensamblador, entonces se debe ejecutar el programa ensamblador. El resultado final en ambos casos es un módulo objeto (un fichero que típicamente tiene una extensión de `.o` o `.obj`. A veces se utiliza un *optimizador de mirilla* en esta segunda pasada para buscar trozos de código que contengan sentencias redundantes de lenguaje ensamblador.

Usar la palabra «objeto» para describir pedazos de código máquina es un hecho desafortunado. La palabra comenzó a usarse antes de que la programación orientada a objetos tuviera un uso generalizado. «Objeto» significa lo mismo que «FIXME:meta» en este contexto, mientras que en la programación orientada a objetos significa «una cosa con límites».

El *enlazador* combina una lista de módulos objeto en un programa ejecutable que el sistema operativo puede cargar y ejecutar. Cuando una función en un módulo objeto hace una referencia a una función o variable en otro módulo objeto, el enlazador resuelve estas referencias; se asegura de que todas las funciones y los datos externos solicitados durante el proceso de compilación existen realmente. Además, el enlazador añade un módulo objeto especial para realizar las actividades de inicialización.

El enlazador puede buscar en unos archivos especiales llamados *librerías* para resolver todas sus referencias. Una librería contiene una colección de módulos objeto

en un único fichero. Una librería se crea y mantiene por un programa conocido como *bibliotecario* (*librarian*).

Comprobación estática de tipos

El compilador realiza una *comprobación de tipos* durante la primera pasada. La comprobación de tipos asegura el correcto uso de los argumentos en las funciones y previene muchos tipos de errores de programación. Como esta comprobación de tipos ocurre se hace la compilación y no cuando el programa se está ejecutado, se conoce como *comprobación estática de tipos*.

Algunos lenguajes orientados a objetos (Java por ejemplo) realizan comprobaciones en tiempo de ejecución (*comprobación dinámica de tipos*). Si se combina con la estática, la comprobación dinámica es más potente que sólo la estática. Sin embargo, añade una sobrecarga a la ejecución del programa.

C++ usa la comprobación estática de tipos debido a que el lenguaje no puede asumir ningún soporte particular durante la ejecución. La comprobación estática de tipos notifica al programador malos usos de los tipos durante la compilación, y así maximiza la velocidad de ejecución. A medida que aprenda C++, comprobará que la mayoría de las decisiones de diseño del lenguaje están tomadas en favor de la mejora del rendimiento, motivo por el cual C es famoso en la programación orientada a la producción.

Se puede deshabilitar la comprobación estática de tipos en C++, e incluso permite al programador usar su propia comprobación dinámica de tipos - simplemente necesita escribir el código.

2.2. Herramientas para compilación modular

La compilación modular es particularmente importante cuando se construyen grandes proyectos. En C y en C++, un programa se puede crear en pequeñas piezas, manejables y comprobables de forma independiente. La herramienta más importante para dividir un programa en piezas más pequeñas es la capacidad de crear subrutinas o subprogramas que tengan un nombre que las identifique. En C y en C++, estos subprogramas se llamana *funciones*, que son las piezas de código que se pueden almacenar en diferentes ficheros, permitiendo la compilación separada. Dicho de otra forma, una función es la unidad atómica de código, debido a que no se puede tener una parte de una función en un fichero y el resto en otro (aunque los ficheros pueden contener más de una función).

Cuando se invoca una función, se le suelen pasar una serie de *argumentos*, que son valores que desea que la función utilice durante su ejecución. Cuando la función termina, normalmente devuelve un *valor de retorno*, que equivale al resultado. También es posible crear funciones que no tengan ni argumentos ni valor de retorno.

Para crear un programa con múltiples ficheros, las funciones de un fichero deben acceder a las funciones y los datos de otros ficheros. Cuando se compila un fichero, el compilador de C o C++ debe conocer las funciones y los datos de los otros ficheros, en particular sus nombres y su uso apropiado. El compilador asegura que las funciones y los datos son usados correctamente. El proceso de "decirle al compilador" los nombres de las funciones externas y los datos que necesitan es conocido como *declaración*. Una vez declarada una función o una variable, el compilador sabe cómo comprobar que la función se utiliza adecuadamente.

2.2.1. Declaraciones vs definiciones

Es importante comprender la diferencia entre *declaraciones* y *definiciones* porque estos términos se usarán de forma precisa en todo el libro. Básicamente todos los programas escritos en C o en C++ requieren declaraciones. Antes de poder escribir su primer programa, necesita comprender la manera correcta de escribir una declaración.

Una *declaración* presenta un nombre -identificador- al compilador. Le dice al compilador «Esta función o esta variable existe en algún lugar, y éste es el aspecto que debe tener». Una *definición*, sin embargo, dice: «Crea esta variable aquí» o «Crea esta función aquí». Eso reserva memoria para el nombre. Este significado sirve tanto para una variable que para una función; en ambos casos, el compilador reserva espacio en el momento de la definición. Para una variable, el compilador determina su tamaño y reserva el espacio en memoria para contener los datos de la variable. Para una función, el compilador genera el código que finalmente ocupará un espacio en memoria.

Se puede declarar una variable o una función en muchos sitios diferentes, pero en C o en C++ sólo se puede definir una vez (se conoce a veces como Regla de Definición Única (ODR) [3]). Cuando el enlazador une todos los módulos objeto, normalmente se quejará si encuentra más de una definición para la misma función o variable.

Una definición puede ser también una declaración. Si el compilador no ha visto antes el nombre `x` y hay una definición `int x;`, el compilador ve el nombre también como una declaración y asigna memoria al mismo tiempo.

Sintaxis de declaración de funciones

La declaración de una función en C y en C++ consiste en escribir el nombre de la función, los tipos de argumentos que se pasan a la función, y el valor de retorno de la misma. Por ejemplo, aquí tenemos la declaración de una función llamada `func-1()` que toma dos enteros como argumentos (en C/C++ los enteros se denotan con la palabra reservada int) y que devuelve un entero:

```
int func1(int, int);
```

La primera palabra reservada es el valor de retorno: `int`. Los argumentos están encerrados entre paréntesis después del nombre de la función en el orden en que se utilizan. El punto y coma indica el final de la sentencia; en este caso le dice al compilador «esto es todo - ¡aquí no está la definición de la función!».

Las declaraciones en C y C++ tratan de mimetizar la forma en que se utilizará ese elemento. Por ejemplo, si `a` es otro entero la función de arriba se debería usar de la siguiente manera:

```
a = func1(2, 3);
```

Como `func1()` devuelve un entero, el compilador de C/C++ comprobará el uso de `func1()` para asegurarse que `a` puede aceptar el valor devuelto y que los argumentos son válidos.

Los argumentos de las declaraciones de funciones pueden tener nombres. El compilador los ignora pero pueden ser útilies como nemotécnicos para el usuario. Por

[3] *One Definition Rule*

ejemplo, se puede declarar `func1()` con una apariencia diferente pero con el mismo significado:

```
int func1(int length, int width);
```

Una puntualización

Existe una diferencia significativa entre C y el C++ para las funciones con lista de argumentos vacía. En C, la declaración:

```
int func2();
```

significa «una funcion con cualquier número y tipo de argumentos», lo cual anula la comprobación de tipos. En C++, sin embargo, significa «una función sin argumentos».

Definición de funciones

La definición de funciones se parece a la declaración excepto en que tienen cuerpo. Un cuerpo es un conjunto de sentencias encerradas entre llaves. Las llaves indican el comienzo y el final del código. Para dar a `func1()` una definición con un cuerpo vacío (un cuerpo que no contiene código), escriba:

```
int func1(int ancho, int largo) {}
```

Note que en la definición de la función las llaves sustituyen el punto y coma. Como las llaves contienen una sentencia o grupo de sentencias, no es necesario un punto y coma. Tenga en cuenta además que los argumentos en la definición de la función deben nombres si los quiere usar en el cuerpo de la función (como aquí no se usan, son opcionales).

Sintaxis de declaración de variables

El significado atribuido a la frase «declaración de variables» históricamente ha sido confuso y contradictorio, y es importante que entienda el significado correcto para poder leer el código correctamente. Una declaración de variable dice al compilador cómo es la variable. Dice al compilador, «Sé que no has visto este nombre antes, pero te prometo que existe en algún lugar, y que es una variable de tipo X».

En una declaración de función, se da un tipo (el valor de retorno), el nombre de la función, la lista de argumentos, y un punto y coma. Con esto el compilador ya tiene suficiente información para saber cómo será la función. Por inferencia, una declaración de variable consistirá en un tipo seguido por un nombre. Por ejemplo:

```
int a;
```

podría declarar la variable `a` como un entero usando la lógica usada anteriormente. Pero aquí está el conflicto: existe suficiente información en el código anterior como para que el compilador pueda crear espacio para un entero llamado `a` y es exactamente lo que ocurre. Para resolver el dilema, fue necesaria una palabra reservada en C y C++ para decir «Esto es sólo una declaración; esta variable estará definida en algún otro lado». La palabra reservada es `extern` que puede significar

que la definición es externa al fichero, o que la definición se encuentra después en este fichero.

Declarar una variable sin definirla implica usar la palabra reservada `extern` antes de una descripción de la variable, como por ejemplo:

```
extern int a;
```

`extern` también se puede aplicar a la declaración de funciones. Para `func1()` sería algo así:

```
extern int func1(int length, int width);
```

Esta sentencia es equivalente a las declaraciones anteriores para `func1()` . Como no hay cuerpo de función, el compilador debe tratarla como una declaración de función en lugar de como definición. La palabra reservada `extern` es bastante superflua y opcional para la declaración de funciones. Probablemente sea desafortunado que los diseñadores de C no obligaran al uso de `extern` para la declaración de funciones; hubiera sido más consistente y menos confuso (pero hubiera requerido teclear más, lo cual probablemente explica la decisión).

Aquí hay algunos ejemplos más de declaraciones:

```
//: C02:Declare.cpp
// Declaration & definition examples
extern int i; // Declaration without definition
extern float f(float); // Function declaration

float b;  // Declaration & definition
float f(float a) {  // Definition
  return a + 1.0;
}

int i; // Definition
int h(int x) { // Declaration & definition
  return x + 1;
}

int main() {
  b = 1.0;
  i = 2;
  f(b);
  h(i);
} ///:~
```

En la declaración de funciones, los identificadores de los argumentos son opcionales. En la definición son necesarios (los identificadores se requieren solamente en C, no en C++).

Incluir ficheros de cabecera

La mayoría de las librerías contienen un número importante de funciones y variables. Para ahorrar trabajo y asegurar la consistencia cuando se hacen declaraciones externas para estos elementos, C y C++ utilizan un artefacto llamado *fichero de ca-*

becera. Un fichero de cabecera es un fichero que contiene las declaraciones externas de una librería; convencionalmente tiene un nombre de fichero con extensión `.h`, como `headerfile.h` (no es difícil encontrar código más antiguo con extensiones diferentes, como `.hxx` o `.hpp`, pero es cada vez más raro).

El programador que crea la librería proporciona el fichero de cabecera. Para declarar las funciones y variables externas de la librería, el usuario simplemente incluye el fichero de cabecera. Para ello se utiliza la directiva de preprocesado `#include`. Eso le dice al preprocesador que abra el fichero de cabecera indicado e incluya el contenido en el lugar donde se encuentra la sentencia `#include`. Un `#include` puede indicar un fichero de dos maneras: mediante paréntesis angulares (< >) o comillas dobles.

Los ficheros entre paréntesis angulares, como:

```
#include <header>
```

hacen que el preprocesador busque el fichero como si fuera particular a un proyecto, aunque normalmente hay un camino de búsqueda que se especifica en el entorno o en la línea de comandos del compilador. El mecanismo para cambiar el camino de búsqueda (o ruta) varía entre maquinas, sistemas operativos, e implementaciones de C++ y puede que requiera un poco de investigación por parte del programador.

Los ficheros entre comillas dobles, como:

```
#include "header"
```

le dicen al preprocesador que busque el fichero en (de acuerdo a la especificación) «un medio de definición de implementación», que normalmente significa buscar el fichero de forma relativa al directorio actual. Si no lo encuentra, entonces la directiva se preprocesada como si tuviera paréntesis angulares en lugar de comillas.

Para incluir el fichero de cabecera `iostream`, hay que escribir:

```
#include <iostream>
```

El preprocesador encontrará el fichero de cabecera `iostream` (a menudo en un subdirectorio llamado «include») y lo incluirá.

Formato de inclusión del estándar C++

A medida que C++ evolucionaba, los diferentes fabricantes de compiladores elegían diferentes extensiones para los nombres de ficheros. Además, cada sistema operativo tiene sus propias restricciones para los nombres de ficheros, en particular la longitud. Estas características crearon problemas de portabilidad del código fuente. Para limar estos problemas, el estándar usa un formato que permite los nombres de ficheros más largos que los famosos ocho caracteres y permite eliminar la extensión. Por ejemplo en vez de escribir `iostream.h` en el estilo antiguo, que se asemejaría a algo así:

```
#include <iostream.h>
```

ahora se puede escribir:

```
#include <iostream>
```

El traductor puede implementar la sentencia del `include` de tal forma que se amolde a las necesidades de un compilador y sistema operativo particular, aunque sea necesario truncar el nombre y añadir una extensión. Evidentemente, también puede copiar las cabeceras que ofrece el fabricante de su compilador a otras sin extensiones si quiere usar este nuevo estilo antes de que su fabricante lo soporte.

Las librerías heredadas de C aún están disponibles con la extensión tradicional «`.h`». Sin embargo, se pueden usar con el estilo de inclusión más moderno colocando una «*c*» al nombre. Es decir:

```
#include <stdio.h>
#include <stdlib.h>
```

Se transformaría en:

```
#include <cstdio>
#include <cstdlib>
```

Y así para todas cabeceras del C Estándar. Eso proporciona al lector una distinción interesante entre el uso de librerías C versus C++.

El efecto del nuevo formato de include no es idéntico al antiguo: usar el «`.h`» da como resultado una versión más antigua, sin plantillas, y omitiendo el «`.h`» le ofrece la nueva versión con plantillas. Normalmente podría tener problemas si intenta mezclar las dos formas de inclusión en un mismo programa.

2.2.2. Enlazado

El enlazador (*linker*) agrupa los módulos objeto (que a menudo tienen extensiones como `.o` ó `.obj`), generados por el compilador, en un programa ejecutable que el sistema operativo puede cargar y ejecutar. Es la última fase del proceso de compilación.

Las características del enlazador varían de un sistema a otro. En general, simplemente se indican al enlazador los nombres de los módulos objeto, las librerías que se desean enlazar y el nombre del ejecutable de salida. Algunos sistemas requieren que sea el programador el que invoque al enlazador, aunque en la mayoría de los paquetes de C++ se llama al enlazador a través del compilador. En muchas situaciones, de manera transparente.

Algunos enlazadores antiguos no buscaban ficheros objeto más de una vez y buscaban en la lista que se les pasaba de izquierda a derecha. Esto significa que el orden de los ficheros objeto y las librerías puede ser importante. Si se encuentra con algún problema misterioso que no aparece hasta el proceso de enlazado, una posible razón es el orden en el que se indican los ficheros al enlazador.

2.2.3. Uso de librerías

Ahora que ya conoce la terminología básica, puede entender cómo utilizar una librería. Para usarla:

1. Se incluye el fichero de cabecera de la librería.
2. Se usan las funciones y las variables de la librería.
3. Se enlaza la librería junto con el programa ejecutable.

Estos pasos también se aplican cuando los módulos objeto no se combinan para formar una librería. Incluir el fichero cabecera y enlazar los módulos objeto es la base para la compilación separada en C y en C++.

Cómo busca el enlazador una librería

Cuando se hace una refencia externa a una función o una variable en C o C++, al enlazador, una vez encontrada esta referencia, puede hacer dos cosas. Si todavía no ha encontrado la definición de la función o variable, añade el identificador a su lista de «referencias no resueltas». Si el enlazador ya había encontrado la definición, se resuelve la referencia.

Si el enlazador no puede encontrar la definición en la lista de módulos objeto, busca en las librerías. Las librerías tienen algún tipo de indexación para que el enlazador no necesite buscar en todos los módulos objeto en la librería - solamente mira en el índice. Cuando el enlazador encuentra una definición en una librería, el módulo objeto entero, no sólo la definición de la función, se enlaza al programa ejecutable. Dese cuenta que no se enlaza la librería completa, tan solo el módulo objeto de la librería que contiene la definición que se necesita (de otra forma los programas se volverían innecesariamente largos). Si se desea minimizar el tamaño del programa ejecutable, se debería considerar poner una única función en cada fichero fuente cuando se construyan librerías propias. Esto requiere más trabajo de edición, [4] pero puede ser muy útil para el usuario.

Debido a que el enlazador busca los ficheros en el orden que se le dan, se puede prevenir el uso de una función de una librería insertando un fichero con su propia función, usando el mismo nombre de función, en la lista antes de que aparezca el nombre de la librería. Cuando el enlazador resuelva cualquier referencia a esa función encontrando la función antes de buscar en la librería, se utilizará su función en lugar de la que se encuentra en la librería. Eso también puede ser una fuente de errores, y es la clase de cosas que se puede evitar usando los espacios de nombres (*namespaces*) de C++.

Añadidos ocultos

Cuando se crea un programa ejecutable en C/C++, ciertos elementos se enlazan en secreto. Uno de estos elementos es el módulo de arranque, que contiene rutinas de inicialización que deben ejecutarse cada vez que arranca un programa C o C++. Estas rutinas preparan la pila e inicializan ciertas variables del programa.

El enlazador siempre busca la librería estándar para las versiones compiladas de cualquier función «estándar» llamada en el programa. Debido a que se busca siempre en la librería estándar, se puede usar cualquier cosa de esta librería simplemente añadiendo a su programa la cabecera apropiada; no necesita indicar dónde hay que buscar la librería estándar. Las funciones de flujo de entrada-salida (iostream), por ejemplo, están en la Librería Estándar de C++. Para usarla, sólo debe incluir el fichero de cabecera `<iostream>`.

[4] Yo le recomendaría usar Perl o Python para automatizar estas tareas como parte de su proceso de empaquetamiento de librerías (ver www.Perl.org ó www.Python.org).

Si se está usando una librería, se debe añadir explícitamente su nombre de ésta a la lista de ficheros manejados por el enlazador.

Uso de librerías C plano

Aunque esté escribiendo código en C++, nada le impide usar librerías de C. De hecho, toda la librería de C está incluida por defecto en el C++ Estándar. Hay una cantidad tremenda de trabajo ya realizado en esas librerías que le pueden ahorrar un montón de tiempo.

Este libro usará la librería Estándar de C++ cuando sea necesario (y por lo tanto la de C), pero sólo se utilizarán funciones de la librería *estándar*, para asegurar la portabilidad de los programas. En los pocos casos en los que las funciones no sean de C++ estándar, se intentará que sean funciones compatibles con POSIX. POSIX es un estándar basado en el esfuerzo por conseguir la estandarización de Unix, que incluye funciones que van más allá del ámbito de las librerías de C++. Normalmente puede esperar encontrar funciones POSIX en plataformas Unix (en particular, GNU/Linux), y a menudo en sistemas DOS/Windows. Por ejemplo, si está usando hilos (*threads*) será mejor usar la librería de hilos compatible con POSIX ya que su código será más fácil de entender, portar y mantener (y la librería de hilos usará los servicios que ofrece el sistema operativo, si es que están soportados).

2.3. Su primer programa en C++

Ahora ya tiene suficientes conocimientos para crear y compilar un programa. Este programa usará las clases de *flujo de entrada-salida* (`iostream`) del C++ estándar. `iostream` es capaz de leer y escribir en ficheros o en la entrada y salida estándar (que suele ser la consola, pero que puede ser redirigida a ficheros o dispositivos). En este programa simple, se usa un objeto `stream` (flujo) para imprimir un mensaje en pantalla.

2.3.1. Uso de las clases `iostream`

Para declarar las funciones y los datos externos que contenga la clase `iostream` hay que incluir el fichero de cabecera de la siguiente manera:

```
#include <iostream>
```

El primer programa usa el concepto de salida estándar, que significa «un lugar de propósito general, al que se le pueden enviar cosas». Verá otros ejemplos que utilizan la salida estándar de otras formas, pero aquí simplemente usaremos la consola. El paquete `iostream` define una variable (un objeto) llamado `cout` de forma automática que es capaz de enviar todo tipo de datos a la salida estándar.

Para enviar datos a la salida estándar, se usa el operador <<.Los programadores de C lo conocen como operador de «desplazamiento a la izquierda», que se explicará en el siguiente capítulo. Baste decir que el desplazamiento a la izquierda no tiene nada que ver con la salida. Sin embargo, C++ permite que los operadores sean *sobrecargados*. Cuando se sobrecarga un operador, se le da un nuevo significado siempre que dicho operador se use con un objeto de determinado tipo. Con los objetos de `iostream`, el operador << significa «enviar a». Por ejemplo:

```
cout << "howdy!";
```

envía la cadena «howdy!» al objeto llamado `cout` (que es un diminutivo de «console output» (salida por consola).

De momento ya hemos visto suficiente sobrecarga de operadores como para poder empezar. El Capítulo 12 cubre la sobrecarga de operadores con detalle.

2.3.2. Espacios de nombres

Como se menciona en el Capítulo 1, uno de los problemas del lenguaje C es que «nos quedamos sin nombres» para funciones e identificadores cuando los programas llegan a ser de cierto tamaño. Por supuesto que realmente no nos quedamos sin nombres; aunque se hace más difícil pensar en nombres nuevos después de un rato. Y todavía más importante, cuando un programa alcanza cierto tamaño es normal fragmentarlo en trozos más pequeños cada uno de los cuales es mantenido por diferentes personas o grupos. Como C sólo tiene un ruedo para lidiar con todos los identificadores y nombres de función, trae como consecuencia que todos los desarrolladores deben tener cuidado de no usar accidentalmente los mismos nombres en situaciones en las que pueden ponerse en conflicto. Esto se convierte en una pérdida de tiempo, se hace tedioso y en último término, es más caro.

El C++ Estándar tiene un mecanismo para impedir estas colisiones: la palabra reservada `namespace` (espacio de nombres). Cada conjunto de definiciones de una librería o programa se «envuelve» en un espacio de nombres, y si otra definición tiene el mismo nombre, pero está en otro espacio de nombres, entonces no se produce colisión.

El espacio de nombres es una herramienta útil y conveniente, pero su presencia implica que debe saber usarla antes de escribir un programa. Si simplemente escribe un fichero de cabecera y usa algunas funciones u objetos de esa cabecera, probablemente reciba extraños mensajes cuando compile el programa, debido a que el compilador no pueda encontrar las declaraciones de los elementos del fichero de cabecera. Después de ver este mensaje un par de veces se le hará familiar su significado (que es: *Usted ha incluido el fichero de cabecera pero todas las declaraciones están sin un espacio de nombres y no le dijo al compilador que quería usar las declaraciones en ese espacio de nombres*).

Hay una palabra reservada que le permite decir «quiero usar las declaraciones y/o definiciones de este espacio de nombres». Esa palabra reservada, bastante apropiada por cierto, es `using`. Todas las librerías de C++ Estándar están incluidas en un único espacio de nombres, que es `std` (por «standard»). Como este libro usa la librería estándar casi exclusivamente, verá la siguiente *directiva using* en casi todos los programas.

```
using namespace std;
```

Esto significa que quiere usar todos los elementos del espacio de nombres llamado `std`. Después de esta sentencia, ya no hay que preocuparse de si su componente o librería particular pertenece a un espacio de nombres, porque la directiva `using` hace que el espacio de nombres esté disponible para todo el fichero donde se escribió la directiva `using`.

Exponer todos los elementos de un espacio de nombres después de que alguien se ha molestado en ocultarlos, parece contraproducente, y de hecho, el lector deberá tener cuidado si considera hacerlo (como aprenderá más tarde en este libro). Sin

embargo, la directiva `using` expone solamente los nombres para el fichero actual, por lo que no es tan drástico como suena al principio. (pero pienselo dos veces antes de usarlo en un fichero cabecera, eso *es* temerario).

Existe una relación entre los espacios de nombres y el modo en que se incluyes los ficheros de cabecera. Antes de que se estandarizara la nueva forma de inclusión de los ficheros cabecera (sin el «`.h`» como en `<iostream>`), la manera típica de incluir un fichero de cabecera era con el «`.h`» como en `<iostream.h>`. En esa época los espacios de nombres tampoco eran parte del lenguaje, por lo que para mantener una compatibilidad hacia atrás con el código existente, si se escribía:

```
#include <iostream.h>
```

En realidad, significaba:

```
#include <iostream>
using namespace std;
```

Sin embargo en este libro se usará la forma estándar de inclusión (sin el «`.h`») y haciendo explícita la directiva `using`.

Por ahora, esto es todo lo que necesita saber sobre los espacios de nombres, pero el Capítulo 10 cubre esta materia en profundidad.

2.3.3. Fundamentos de la estructura de los programa

Un programa C o C++ es una colección de variables, definiciones de función, y llamada a funciones. Cuando el programa arranca, ejecuta el código de inicialización y llama a una función especial, «`main()`», que es donde debe colocarse el código principal del programa.

Como se mencionó anteriormente, una definición de función consiste en un valor de retorno (que se debe especificar en C++), un nombre de función, una lista de argumentos, y el código de la función entre llaves. Aquí hay un ejemplo de definición de función:

```
int funcion() {
   // óCdigo de la ófuncin íaqu (esto es un comentario)
}
```

La función de arriba tiene una lista vacía de argumentos y un cuerpo que contiene únicamente un comentario.

Puede haber varios pares de llaves en la definición de una función, pero siempre debe haber al menos dos que envuelvan todo el cuerpo de la función. Como `main()` es una función, debe seguir esas reglas. En C++, `main()` siempre devuelve un valor de tipo int (entero).

C y C++ son lenguajes de formato libre. Con un par de excepciones, el compilador ignora los espacios en blanco y los saltos de línea, por lo que hay que determinar el final de una sentencia. Las sentencias están delimitadas por punto y coma.

Los comentarios en C empiezan con `/*` y finalizan con `*/`. Pueden incluir saltos de línea. C++ permite este estilo de comentarios y añade la doble barra inclinada: `//`. La `//` empieza un comentario que finaliza con el salto de línea. Es más útil que

`/* */` y se usa ampliamente en este libro.

2.3.4. «Hello, World!»

Y por fin, el primer programa:

```
//: C02:Hello.cpp
// Saying Hello with C++
#include <iostream> // Stream declarations
using namespace std;

int main() {
  cout << "Hello, World! I am "
       << 8 << " Today!" << endl;
} ///:~
```

El objeto `cout` maneja una serie de argumentos por medio de los operadores `<<`, que imprime los argumentos de izquierda a derecha. La función especial `endl` provoca un salto de línea. Con los iostreams se puede encadenar una serie de argumentos como aquí, lo que hace que sea una clase fácil de usar.

En C, el texto que se encuentra entre comillas dobles se denomina «cadena» (*string*). Sin embargo, la librería Estándar de C++ tiene una poderosa clase llamada `string` para manipulación de texto, por lo que usaremos el término más preciso *array de caracteres* para el texto que se encuentre entre dobles comillas.

El compilador pide espacio de memoria para los arrays de caracteres y guarda el equivalente ASCII para cada carácter en este espacio. El compilador finaliza automáticamente este array de caracteres añadiendo el valor 0 para indicar el final.

Dentro del array de caracteres, se pueden insertar caracteres especiales usando las *secuencias de escape*. Consisten en una barra invertida (`\`) seguida de un código especial. por ejemplo `\n` significa salto de línea. El manual del compilador o la guía concreta de C ofrece una lista completa de secuencia; entre otras se incluye: `\t` (tabulador), `\\` (barra invertida), y `\b` (retroceso).

Tenga en cuenta que la sentencia puede continuar en otras líneas, y la sentencia completa termina con un punto y coma.

Los argumentos de tipo array de caracteres y los números constantes están mezclados en la sentencia `cout` anterior. Como el operador `<<` está sobrecargado con varios significados cuando se usa con `cout`, se pueden enviar distintos argumentos y `cout` se encargará de mostrarlos.

A lo largo de este libro notará que la primera línea de cada fichero es un comentario (empezando normalmente con `//`), seguido de dos puntos, y la última línea de cada listado de código acaba con un comentario seguido de «/-». Se trata de una una técnica que uso para extraer fácilmente información de los ficheros fuente (el programa que lo hace se puede encontrar en el Volumen 2 de este libro, en www.BruceEckel.com). La primera línea también tiene el nombre y localización del fichero, por lo que se puede localizar fácilmente en los ficheros de código fuente del libro (que también se puede descargar de www.BruceEckel.com).

2.3.5. Utilizar el compilador

Después de descargar y desempaquetar el código fuente del libro, busque el programa en el subdirectorio `C02`. Invoque el compilador con `Hello.cpp` como parámetro. La mayoría de los compiladores le abstraen de todo el proceso si el programa consta de un único fichero. Por ejemplo, para usar el compilador GNU C++ (que está disponible en Internet), escriba:

```
g++ Hello.cpp
```

Otros compiladores tendrán una sintaxis similar aunque tendrá que consultar la documentación para conocer los detalles particulares.

2.4. Más sobre iostreams

Hasta ahora sólo ha visto los aspectos más rudimentarios de las clases `iostream`. El formateo de salida que permiten los iostreams también incluyen características como el formateo de números en decimal, octal, y hexadecimal. Aquí tiene otro ejemplo del uso de los iostreams:

```
//: C02:Stream2.cpp
// More streams features
#include <iostream>
using namespace std;

int main() {
  // Specifying formats with manipulators:
  cout << "a number in decimal: "
       << dec << 15 << endl;
  cout << "in octal: " << oct << 15 << endl;
  cout << "in hex: " << hex << 15 << endl;
  cout << "a floating-point number: "
       << 3.14159 << endl;
  cout << "non-printing char (escape): "
       << char(27) << endl;
} ///:~
```

Este ejemplo muestra cómo la clase `iostreams` imprime números en decimal, octal, y hexadecimal usando *manipuladores* (los cuales no imprimen nada, pero cambian el estado del flujo de salida). El formato de los números en punto flotante lo determina automáticamente el compilador. Además, se puede enviar cualquier carácter a un objeto `stream` usando un molde (*cast*) a char (un char es un tipo de datos que manipula un sólo carácter). Este molde parece una llamada a función: `char()`, devuelve un valor ASCII. En el programa de arriba, el `char(27)` envía un «escape» a `cout`.

2.4.1. Concatenar vectores de caracteres

Una característica importante del preprocesador de C es la *concatenación de arrays de caracteres*. Esta característica se usa en algunos de los ejemplos de este libro. Si se colocan juntos dos arrays de caracteres entrecomillados, sin signos de puntuación entre ellos, el compilador los pegará en un único array de caracteres. Esto es particu-

larmente útil cuando los listados de código tienen restricciones de anchura.

```
//: C02:Concat.cpp
// Character array Concatenation
#include <iostream>
using namespace std;

int main() {
  cout << "This is far too long to put on a "
    "single line but it can be broken up with "
    "no ill effects\nas long as there is no "
    "punctuation separating adjacent character "
    "arrays.\n";
} ///:~
```

Al principio, el código de arriba puede parecer erróneo porque no está el ya familiar punto y coma al final de cada línea. Recuerde que C y C++ son lenguajes de formato libre, y aunque normalmente verá un punto y coma al final de cada línea, el requisito real es que haya un punto y coma al final de cada sentencia, por lo que es posible encontrar una sentencia que ocupe varias líneas.

2.4.2. Leer de la entrada

Las clases `iostream` proporcionan la habilidad de leer de la entrada. El objeto usado para la entrada estándar es `cin` (de «*console input*»). `cin` normalmente espera la entrada de la consola, pero esta entrada se puede redirigir desde otras fuentes. Un ejemplo de redirección se muestra más adelante en este capítulo.

El operador que usa `iostream` con el objeto `cin` es `>>`. Este operador espera como parámetro algún tipo de entrada. Por ejemplo, si introduce un parámetro de tipo entero, él espera un entero de la consola. Aquí hay un ejemplo:

```
//: C02:Numconv.cpp
// Converts decimal to octal and hex
#include <iostream>
using namespace std;

int main() {
  int number;
  cout << "Enter a decimal number: ";
  cin >> number;
  cout << "value in octal = 0"
       << oct << number << endl;
  cout << "value in hex = 0x"
       << hex << number << endl;
} ///:~
```

Este programa convierte un número introducido por el usuario en su representación octal y hexadecimal.

2.4.3. Llamar a otros programas

Mientras que el modo típico de usar un programa que lee de la entrada estándar y escribe en la salida estándar es dentro de un *shell script* Unix o en un fichero *batch* de DOS, cualquier programa se puede llamar desde dentro de un programa C o C++ usando la llamada a la función estándar `system()` que está declarada en el fichero de cabecera `<cstdlib>`:.

```
//: C02:CallHello.cpp
// Call another program
#include <cstdlib> // Declare "system()"
using namespace std;

int main() {
  system("Hello");
} ///:~
```

Para usar la función `system()`, hay que pasarle un array de caracteres con la línea de comandos que se quiere ejecutar en el prompt del sistema operativo. Puede incluir los parámetros que utilizaría en la línea de comandos, y el array de caracteres se puede fabricar en tiempo de ejecución (en vez de usar un array de caracteres estático como se mostraba arriba). El comando se ejecuta y el control vuelve al programa.

Este programa le muestra lo fácil que es usar C plano en C++; sólo incluya la cabecera y utilice la función. Esta compatibilidad ascendente entre el C y el C++ es una gran ventaja si está aprendiendo C++ y ya tenía conocimientos de C.

2.5. Introdución a las cadenas

Un array de caracteres puede ser bastante útil, aunque está bastante limitado. Simplemente son un grupo de caracteres en memoria, pero si quiere hacer algo útil, debe manejar todos los pequeños detalles. Por ejemplo, el tamaño de un array de caracteres es fijo en tiempo de compilación. Si tiene un array de caracteres y quiere añadirle más caracteres, tendrá que saber mucho sobre ellos (incluso manejo dinámico de memoria, copia de array de caracteres, y concatenación) antes de conseguir lo que desea. Esta es exactamente la clase de cosas que desearíamos que hiciera un objeto por nosotros.

La clase `string` (cadena) del C++ Estándar ha sido diseñada para que se encargue y oculte las manipulaciones de bajo nivel de los arrays de caracteres que antes tenía que realizar el programador de C. Estas manipulaciones han sido una fuente de constantes pérdidas de tiempo y errores desde los orígenes del lenguaje C. Aunque hay un capítulo entero dedicado a la clase `string` en el Volumen 2 de este libro, las cadenas son tan importantes y facilitan tanto la vida que las presentaré aquí para usarlas lo antes posible en el libro.

Para usar las cadenas debe incluir el fichero de cabecera `<string>`. La clase `string` se encuentra en el espacio de nombres `std` por lo que se necesita usar la directiva `using`. Gracias a la sobrecarga de operadores, la sintaxis del uso de las cadenas es muy intuitiva:

```
//: C02:HelloStrings.cpp
// The basics of the Standard C++ string class
#include <string>
```

```
#include <iostream>
using namespace std;

int main() {
  string s1, s2; // Empty strings
  string s3 = "Hello, World."; // Initialized
  string s4("I am"); // Also initialized
  s2 = "Today"; // Assigning to a string
  s1 = s3 + " " + s4; // Combining strings
  s1 += " 8 "; // Appending to a string
  cout << s1 + s2 + "!" << endl;
} ///:~
```

Las dos primeras cadenas, `s1` y `s2` empiezan estando vacías, mientras que `s3` y `s4` muestran dos formas de inicializar los objetos `string` con arrays de caracteres (puede inicializar objetos `string` igual de fácil con otros objetos `string`).

Se puede asignar a un objeto `string` usando =. Eso sustituye el contenido previo de la cadena con lo que se encuentra en el lado derecho de la asignación, y no hay que preocuparse de lo que ocurre con el contenido anterior porque se controla automáticamente. Para combinar las cadenas simplemente debe usar el operador de suma «+», que tambien le permite concatenar cadenas (`strings`) con arrays de caracteres. Si quiere añadir una cadena o un array de caracteres a otra cadena, puede usar el operador +=. Finalmente, dése cuenta que `iostream` sabe como tratar las cadenas, por lo que usted puede enviar una cadena (o una expresión que produzca un `string`, que es lo que sucede con `s1 + s2 + "!">`) directamente a `cout` para imprimirla.

2.6. Lectura y escritura de ficheros

En C, el proceso de abrir y manipular ficheros requería un gran conocimiento del lenguaje para prepararle para la complejidad de las operaciones. Sin embargo, la librería `iostream` de C++ proporciona una forma simple de manejar ficheros, y por eso se puede presentar mucho antes de lo que se haría en C.

Para poder abrir un fichero para leer y escribir, debe incluir la librería `fstream`. Aunque eso implica la inclusión automática de la librería `iostream`, es prudente incluir `iostream` si planea usar `cin`, `cout`, etc.

Para abrir un fichero para lectura, debe crear un objeto `ifstream` que se usará como `cin`. Para crear un fichero de escritura, se crea un objeto `ofstream` que se comporta como `cout`. Una vez que tiene abierto el fichero puede leer o escribir en él como si usara cualquier objeto `iostream`. Así de simple, que es el objetivo, por supuesto.

Una de funciones las más útiles de la librería `iostream` es `getline()`, que permite leer una línea (terminada en nueva línea) y guardarla en un objeto `string` [5]. El primer argumento es el objeto `ifstream` del que se va a leer la información y el segundo argumento es el objeto `string`. Cuando termina la llamada a la función, el objeto `string` contiene la línea capturada.

Aquí hay un ejemplo que copia el contenido de un fichero en otro.

[5] Actualmente existen variantes de `getline()`, que se discutirán profusamente en el capítulo de `iostreams` en el Volumen 2

```
//: C02:Scopy.cpp
// Copy one file to another, a line at a time
#include <string>
#include <fstream>
using namespace std;

int main() {
  ifstream in("Scopy.cpp"); // Open for reading
  ofstream out("Scopy2.cpp"); // Open for writing
  string s;
  while(getline(in, s)) // Discards newline char
    out << s << "\n"; // ... must add it back
} ///:~
```

Para abrir los ficheros, únicamente debe controlar los nombres de fichero que se usan en la creación de los objetos `ifstream` y `ofstream`.

Aquí se presenta un nuevo concepto: el bucle `while`. Aunque será explicado en detalle en el siguiente capítulo, la idea básica consiste en que la expresión entre paréntesis que sigue al `while` controla la ejecución de la sentencia siguiente (pueden ser múltiples sentencias encerradas entre llaves). Mientras la expresión entre paréntesis (en este caso `getline(in, s)` produzca un resultado «verdadero», las sentencias controladas por el `while` se ejecutarán. `getline()` devuelve un valor que se puede interprer como «verdadero» si se ha leido otra línea de forma satisfactoria, y «falso» cuando se llega al final de la entrada. Eso implica que el `while` anterior lee todas las líneas del fichero de entrada y las envía al fichero de salida.

`getline()` lee los caracteres de cada línea hasta que descubre un salto de línea (el carácter de terminación se puede cambiar pero eso no se verá hasta el capítulo sobre `iostreams` del Volumen 2). Sin embargo, descarta el carácter de nueva línea y no lo almacena en el objeto `string`. Por lo que si queremos copiar el fichero de forma idéntica al original, debemos añadir el carácter de nueva línea como se muestra arriba.

Otro ejemplo interesante es copiar el fichero entero en un único objeto `string`:

```
//: C02:FillString.cpp
// Read an entire file into a single string
#include <string>
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  ifstream in("FillString.cpp");
  string s, line;
  while(getline(in, line))
    s += line + "\n";
  cout << s;
} ///:~
```

Debido a la naturaleza dinámica de los `strings`, no hay que preocuparse de la cantidad de memoria que hay que reservar para el `string`. Simplemente hay que añadir cosas y el `string` irá expandiéndose para dar cabida a lo que le introduzca.

Una de las cosas agradables de poner el fichero entero en una cadena es que la clase `string` proporciona funciones para la búsqueda y manipulación que le permiten modificar el fichero como si fuera una simple línea. Sin embargo, tiene sus limitaciones. Por un lado, a menudo, es conveniente tratar un fichero como una colección de líneas en vez de un gran bloque de texto. Por ejemplo, si quiere añadir numeración de líneas es mucho más fácil si tiene un objeto `string` distinto para cada línea. Para realizarlo, necesitamos otro concepto.

2.7. Introducción a los vectores

Con cadenas, podemos rellenar un objeto `string` sin saber cuanta memoria se va a necesitar. El problema de introducir líneas de un fichero en objetos `string` es que se sabe cuántas cadenas habrá - solamente lo sabemos cuando ya hemos leido el fichero entero. Para resolver este problema necesitamos un nuevo tipo de datos que pueda crecer automáticamente para contener las cadenas que le vayamos introduciendo.

De hecho, ¿por qué limitarnos a manejar objetos `string`? Parece que este tipo de problema - no saber la cantidad de cosas a manejar mientras está escribiendo el problema - ocurre a menudo. Y este objeto «contenedor» podría resultar más útil si pudiera manejar *cualquier clase de objeto*. Afortunadamente, la Librería Estándar de C++ tiene una solución: las clases contenedor (*container*). Las clases contenedor son uno de los puntos fuertes del Estándar C++.

A menudo existe un poco de confusión entre los contenedores y los algoritmos en la librería Estándar de C++, y la STL. La *Standard Template Library* fue el nombre que usó Alex Stepanov (que en aquella época estaba trabajando en Hewlett-Packard) cuando presentó su librería al Comité del Estándar C++ en el encuentro en San Diego, California, en la primavera de 1994. El nombre sobrevivió, especialmente después de que HP decidiera dejarlo disponible para la descarga pública. Posteriormente el comité integró las STL en la Librería Estándar de C++ haciendo un gran número de cambios. El desarrollo de las STL continúa en Silicon Graphics (SGI; ver www.sgi.com/Technology/STL). Las SGI STL divergen de la Librería Estándar de C++ en muchos detalles sutiles. Aunque es una creencia ampliamente generalizada, el C++ Estándar no "incluye" las STL. Puede ser confuso debido a que los contenedores y los algoritmos en el C++ Estándar tienen la misma raíz (y a menudo el mismo nombre) que en el SGI STL. En este libro, intentaré decir «la librería Estándar de C++» o «Librería Estándar de contenedores», o algo similar y eludiré usar el término STL.

A pesar de que la implementación de los contenedores y algoritmos de la Librería Estándar de C++ usa algunos conceptos avanzados, que se cubren ampliamente en dos largos capítulos en el segundo volumen de este libro, esta librería también puede ser potente sin saber mucho sobre ella. Es tan útil que el más básico de los contenedores estándar, el `vector`, se introduce en este capítulo y se usará a lo largo de todo el libro. Verá que puede hacer muchas cosas con el `vector` y no saber cómo está implementado (de nuevo, uno de los objetivos de la POO). Los programas que usan `vector` en estos primeros capítulos del libro no son exactamente como los haría un programador experimentado, como comprobará en el volumen 2. Aún así, encontrará que en la mayoría de los casos el uso que se hace es adecuado.

La clase `vector` es una *plantilla*, lo que significa que se puede aplicar a tipos de datos diferentes. Es decir, se puede crear un `vector` de `figuras`, un `vector` de `gatos`, un `vector` de `strings`, etc. Básicamente, con una plantilla se puede crear un vector de «cualquier clase». Para decirle al compilador con qué clase trabajará

(en este caso que va a manejar el vector), hay que poner el nombre del tipo deseado entre «llaves angulares». Por lo que un `vector` de `string` se denota como `vector<string>`. Con eso, se crea un vector a medida que solamente contendrá objetos `string`, y recibirá un mensaje de error del compilador si intenta poner otra cosa en él.

Como el `vector` expresa el concepto de «contenedor», debe existir una manera de meter cosas en él y sacar cosas de él. Para añadir un nuevo elemento al final del vector, se una el método `push_back()`. Recuerde que, como es un método, hay que usar un '.' para invocarlo desde un objeto particular. La razón de que el nombre de la función parezca un poco verboso - `push_back()` en vez de algo más simple como `put` - es porque existen otros contenedores y otros métodos para poner nuevos elementos en los contenedores. Por ejemplo, hay un `insert()` para poner algo en medio de un contenedor. `vector` la soporta pero su uso es más complicado y no necesitamos explorarla hasta el segundo volumen del libro. También hay un `push_front()` (que no es parte de `vector`) para poner cosas al principio. Hay muchas más funciones miembro en `vector` y muchos más contenedores en la Librería Estándar, pero le sorprenderá ver la de cosas que se pueden hacer con sólo un par de características básicas.

Así que se pueden introducir elementos en un `vector` con `push_back()` pero ¿cómo puede sacar esos elementos? La solución es inteligente y elegante: se usa la sobrecarga de operadores para que el `vector` se parezca a un array. El array (que será descrito de forma más completa en el siguiente capítulo) es un tipo de datos que está disponible prácticamente en cualquier lenguaje de programación por lo que debería estar familiarizado con él. Los arrays son *agregados* lo que significa que consisten en un número de elementos agrupados. La característica distintiva de un array es que estos elementos tienen el mismo tamaño y están organizados uno junto a otro. Y todavía más importante, que se pueden seleccionar mediante un índice, lo que significa que puede decir: «Quiero el elemento número n» y el elemento será producido, normalmente de forma rápida. A pesar de que existen excepciones en los lenguajes de programación, normalmente se indica la «indexación» mediante corchetes, de tal forma que si se tiene un array `a` y quiere obtener el quinto elemento, sólo tiene que escribir `a[4]` (fíjese en que la indexación siempre empieza en cero).

Esta forma compacta y poderosa de notación indexada se ha incorporado al `vector` mediante la sobrecarga de operadores como el << y el >> de los `iostreams`. De nuevo, no hay que saber cómo se ha implementado la sobrecarga de operadores - lo dejamos para un capítulo posterior - pero es útil que sea consciente que hay algo de magia detrás de todo esto para conseguir que los corchetes funcionen con el `vector`.

Con todo esto en mente, ya puede ver un programa que usa la clase `vector`. Para usar un vector, hay que incluir el fichero de cabecera `<vector>`:

```
//: C02:Fillvector.cpp
// Copy an entire file into a vector of string
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

int main() {
  vector<string> v;
  ifstream in("Fillvector.cpp");
  string line;
```

```
  while(getline(in, line))
    v.push_back(line); // Add the line to the end
  // Add line numbers:
  for(int i = 0; i < v.size(); i++)
    cout << i << ": " << v[i] << endl;
} ///:~
```

Casi todo este programa es similar al anterior; se abre un fichero abierto y se leen las líneas en objetos `string` (uno cada vez). Sin embargo, estos objetos `string` se introducen al final del `vector` `v`. Una vez que el bucle `while` ha terminado, el fichero entero se encuentra en memoria dentro de `v`.

La siguiente sentencia en el programa es un bucle `for`. Es parecido a un bucle `while` aunque añade un control extra. Como en el bucle `while`, en el `for` hay una «expresión de control» dentro del paréntesis. Sin embargo, esta expresión está dividida en tres partes: una parte que inicializa, una que comprueba si hay que salir del bucle, y otra que cambia algo, normalmente da un paso en una secuencia de elementos. Este programa muestra el bucle `for` de la manera más habitual: la parte de inicialización `int i = 0` crea un entero `i` para usarlo como contador y le da el valor inicial de cero. La comprobación consiste en ver si `i` es menor que el número de elementos del `vector` `v`. (Esto se consigue usando la función miembro `size()` -tamaño- que hay que admitir que tiene un significado obvio) El último trozo, usa el operador de «autoincremento» para aumentar en uno el valor de `i`. Efectivamente, `i++` dice «coge el valor de `i` añádele uno y guarda el resultado en `i`». Conclusión: el efecto del bucle `for` es aumentar la variable `i` desde cero hasta el tamaño del `vector` menos uno. Por cada nuevo valor de `i` se ejecuta la sentencia del `cout`, que construye un linea con el valor de `i` (mágicamente convertida a un array de caracteres por `cout`), dos puntos, un espacio, la línea del fichero y el carácter de nueva línea que nos proporciona `endl`. Cuando lo compile y lo ejecute verá el efecto de numeración de líneas del fichero.

Debido a que el operador >> funciona con `iostreams`, se puede modificar fácilmente el programa anterior para que convierta la entrada en palabras separadas por espacios, en vez de líneas:

```
//: C02:GetWords.cpp
// Break a file into whitespace-separated words
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

int main() {
  vector<string> words;
  ifstream in("GetWords.cpp");
  string word;
  while(in >> word)
    words.push_back(word);
  for(int i = 0; i < words.size(); i++)
    cout << words[i] << endl;
} ///:~
```

La expresión:

```
while (in >> word)
```

es la que consigue que se lea una «palabra» cada vez, y cuando la expresión se evalúa como «falsa» significa que ha llegado al final del fichero. De acuerdo, delimitar una palabra mediante caracteres en blanco es un poco tosco, pero sirve como ejemplo sencillo. Más tarde, en este libro, verá ejemplos más sofisticados que le permiten dividir la entrada de la forma que quiera.

Para demostrar lo fácil que es usar un `vector` con cualquier tipo, aquí tiene un ejemplo que crea un vector de enteros:

```
//: C02:Intvector.cpp
// Creating a vector that holds integers
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<int> v;
  for(int i = 0; i < 10; i++)
    v.push_back(i);
  for(int i = 0; i < v.size(); i++)
    cout << v[i] << ", ";
  cout << endl;
  for(int i = 0; i < v.size(); i++)
    v[i] = v[i] * 10; // Assignment
  for(int i = 0; i < v.size(); i++)
    cout << v[i] << ", ";
  cout << endl;
} ///:~
```

Para crear un `vector` que maneje un tipo diferente basta con poner el tipo entre las llaves angulares (el argumento de las plantillas). Las plantillas y las librerías de plantillas pretenden ofrecer precisamente esta facilidad de uso.

Además este ejemplo demuestra otra característica esencial del `vector` en la expresión

```
v[i] = v[i] * 10;
```

Puede observar que el `vector` no está limitado a meter cosas y sacarlas. También puede *asignar* (es decir, cambiar) cualquier elemento del vector mediante el uso de los corchetes. Eso significa que el `vector` es un objeto útil, flexible y de propósito general para trabajar con colecciones de objetos, y haremos uso de él en los siguientes capítulos.

2.8. Resumen

Este capítulo pretende mostrarle lo fácil que puede llegar a ser la programación orientada a objetos - si alguien ha hecho el trabajo de definir los objetos por usted. En este caso, sólo hay que incluir el fichero de cabecera, crear los objetos y enviarles mensajes. Si los tipos que está usando están bien diseñados y son potentes, entonces

no tendrá mucho trabajo y su programa resultante también será potente.

En este proceso para mostrar la sencillez de la POO cuando se usan librerías de clases, este capítulo, también introduce algunos de los tipos de datos más básicos y útiles de la Librería Estándar de C++: La familia de los `iostreams` (en particular aquellos que leen y escriben en consola y ficheros), la clase `string`, y la plantilla `vector`. Ha visto lo sencillo que es usarlos y ahora es probable que se imagine la de cosas que se pueden hacer con ellos, pero hay muchas más cosas que son capaces de realizar [6]. A pesar de estar usando un pequeño subconjunto de la funcionalidad de estas herramientas en este principio del libro, supone un gran avance frente a los rudimentarios comienzos en el aprendizaje de un lenguaje de bajo nivel como C. Y aunque aprender los aspectos de bajo nivel de C es educativo también lleva tiempo. Al final usted es mucho más productivo si tiene objetos que manejen las características de bajo nivel. Después de todo, el principal objetivo de la POO es esconder los detalles para que usted pueda «pintar con una brocha más gorda».

Sin embargo, debido al alto nivel que la POO intenta tener, hay algunos aspectos fundamentales de C que no se pueden obviar, y de eso trata el siguiente capítulo.

2.9. Ejercicios

Las soluciones a los ejercicios se pueden encontrar en el documento electrónico titulado «The Thinking in C++ Annotated Solution Guide», disponible por poco dinero en www.BruceEckel.com.

1. Modifique `Hello.cpp` para que imprima su nombre y edad (o tamaño de pie, o la edad de su perro, si le gusta más). Compile y ejecute el programa.
2. Utilizando `Stream2.cpp` y `Numconv.cpp` como guías, cree un programa que le pida el radio de un círculo y le muestre el área del mismo. Puede usar el operador `*` para elevar el radio al cuadrado. No intente imprimir el valor en octal o en hexadecimal (sólo funciona con tipos enteros).
3. Cree un programa que abra un fichero y cuente las palabras (separadas por espacios en blanco) que contiene.
4. Cree un programa que cuente el número de ocurrencias de una palabra en concreto en un fichero (use el operador `==` de la clase `string` para encontrar la palabra)
5. Cambie `Fillvector.cpp` para que imprima las líneas al revés (de la última a la primera).
6. Cambie `Fillvector.cpp` para que concatene todos los elementos de la clase `vector` en un único `string` antes de imprimirlo, pero no añada numeración de líneas
7. Muestre un fichero línea a línea, esperando que el usuario pulse **Enter** después de cada línea.
8. Cree un `vector<float>` e introduzca en él 25 números en punto flotante usando un bucle `for`. Muestre el vector.

[6] Si está especialmente interesado en ver todas las cosas que se pueden hacer con los componentes de la Librería Estándar, vea el Volumen 2 de este libro en www.BruceEckel.com y también en www.dinkumware.com

9. Cree tres objetos `vector<float>` y rellene los dos primeros como en el ejercicio anterior. Escriba un bucle `for` que sume los elementos correspondientes y los añada al tercer vector. Muestre los tres vectores.

10. Cree un `vector<float>` e introduzca 25 números en él como en el ejercicio anterior. Eleve cada número al cuadrado y ponga su resultado en la misma posición del vector. Muestre el vector antes y después de la multiplicación.

3: C en C++

Como C++ está basado en C, debería estar familiarizado con la sintaxis de C para poder programar en C++, del mismo modo que debería tener una fluidez razonable en álgebra para poder hacer cálculos.

Si nunca antes ha visto C, este capítulo le dará una buena base sobre el estilo de C usado en C++. Si está familiarizado con el estilo de C descrito en la primera edición de Kernighan & Ritchie (también llamado K&R) encontrará algunas características nuevas o diferentes tanto en C++ como en el estándar C. Si está familiarizado con el estándar C debería echar un vistazo al capítulo en busca de las características particulares de C++. Note que hay algunas características fundamentales de C++ que se introducen aquí, que son ideas básicas parecidas a características de C o a menudo modificaciones en el modo en que C hace las cosas. Las características más sofisticadas de C++ se explicarán en capítulos posteriores

Este capítulo trata por encima las construcciones de C e introduce algunas construcciones básicas de C++, suponiendo que tiene alguna experiencia programando en otro lenguaje. En el CD-ROM que acompaña a este libro hay una introducción más suave a C, titulada *Thinking in C: Foundations for Java & C++* de Chuck Alison (publicada por MidView, Inc. y disponible también en www.MindView.net). Se trata de un seminario en CD-ROM cuyo objetivo es guiarle cuidadosamente a través de los fundamentos del lenguaje C. Se concentra en el conceptos necesarios para permitirle pasarse a C++ o a Java, en lugar de intentar convertirle en un experto en todos los oscuros recovecos de C (una de las razones para usar un lenguaje de alto nivel como C++ o Java es precisamente evitar muchos de estos recovecos). También contiene ejercicios y soluciones guiadas. Tenga presente que este capítulo va después del CD *Thinking in C*, el CD no reemplaza a este capítulo, sino que debería tomarse como una preparación para este capítulo y para el libro.

3.1. Creación de funciones

En el antiguo C (previo al estándar), se podía invocar una función con cualquier número y tipo de argumentos sin que el compilador se quejase. Todo parecía ir bien hasta que ejecutabas el programa. El programa acababa con resultados misteriosos (o peor, el programa fallaba) sin ninguna pista del motivo. La falta de ayuda acerca del paso de argumentos y los enigmáticos bugs que resultaban es, probablemente, la causa de que C se considerase «un lenguaje ensamblador de alto nivel». Los programadores de pre-Estándar C simplemente se adaptaron.

C y C++ Estándar usan una característica llamada *prototipado de funciones*. Con esta herramienta se han de describir los tipos de argumentos al declarar y definir una

función. Esta descripción es el «prototipo». Cuando la función es llamada, el compilador usa el prototipo para asegurar que los argumentos pasados son los apropiados, y que el valor retornado es tratado correctamente. Si el programador comete un error al llamar a la función, el compilador detecta el error.

Esencialmente, aprendió sobre prototipado de funciones (sin llamarlas de ese modo) en el capítulo previo, ya que la forma de declararlas en C++ requiere de un prototipado apropiado. En un prototipo de función, la lista de argumentos contiene los tipos de argumentos que se deben pasar a la función y (opcionalmente para la declaración), identificadores para los argumentos. El orden y tipo de los argumentos debe coincidir en la declaración, definición y llamada a la función. A continuación se muestra un ejemplo de un prototipo de función en una declaración:

```
int translate(float x, float y, float z);
```

No se puede usar la misma sintaxis para declarar los argumentos en el prototipo de una función que en las definiciones ordinarias de variables. Esto significa que no se puede escribir: `float x, y, z`. Se debe indicar el tipo de cada argumento. En una declaración de función, lo siguiente también es correcto:

```
int translate(float, float, float);
```

Ya que el compilador no hace más que chequear los tipos cuando se invoca la función, los identificadores se incluyen solamente para mejorar la claridad del código cuando alguien lo está leyendo.

En la definición de la función, los nombres son necesarios ya que los argumentos son referenciados dentro de la función:

```
int translate(float x, float  y, float z) {
    x = y = z;
    // ...
}
```

Esta regla sólo se aplica a C. En C++, un argumento puede no tener nombrado en la lista de argumentos de la definición de la función. Como no tiene nombre, no se puede utilizar en el cuerpo de la función, por supuesto. Los argumentos sin nombre se permiten para dar al programador una manera de «reservar espacio en la lista de argumentos». De cualquier modo, la persona que crea la función aún así debe llamar a la función con los parámetros apropiados. Sin embargo, la persona que crea la función puede utilizar el argumento en el futuro sin forzar una modificación en el código que llama a la función. Esta opción de ignorar un argumento en la lista también es posible si se indica el nombre, pero siempre aparecería un molesto mensaje de advertencia, informando que el valor no se utiliza, cada vez que se compila la función. La advertencia desaparece si se quita el nombre del argumento.

C y C++ tienen otras dos maneras de declarar una lista de argumentos. Si se tiene una lista de argumentos vacía, se puede declarar esta como `func()` en C++, lo que indica al compilador que hay exactamente cero argumentos. Hay que tener en cuenta que esto sólo significa una lista de argumentos vacía en C++. En C significa «un número indeterminado de argumentos» (lo que es un «agujero» en C ya que deshabilita la comprobación de tipos en ese caso). En ambos, C y C++, la declaración `func(void);` significa una lista de argumentos vacía. La palabra clave void significa «nada» en este caso (también puede significar «sin tipo» en el caso de los

punteros, como se verá mas adelante en este capítulo).

La otra opción para las listas de argumentos se produce cuando no se sabe cuantos argumentos o qué tipos tendrán los argumentos; esto se conoce como *lista de argumentos variable*. Esta «lista incierta de argumentos» se representada con puntos suspensivos (...). Definir una función con una lista de argumentos variable es significativamente más complicado que definir una función normal. Se puede utilizar una lista de argumentos variable para una función que tiene un grupo de argumentos fijos si (por alguna razón) se quiere deshabilitar la comprobación del prototipo de función. Por eso, se debe restringir el uso de listas de argumentos variables en C y evitarlas en C++ (en el cual, como aprenderá, hay alternativas mucho mejores). El manejo de listas de argumentos variables se describe en la sección de librerías de la documentación de su entorno C particular.

3.1.1. Valores de retorno de las funciones

Un prototipo de función C++ debe especificar el tipo de valor devuelto de la función (en C, si no se especifica será por defecto un int). La especificación del tipo de retorno precede al nombre de la función. Para especificar que no se devolverá valor alguno, se utiliza la palabra reservada void. Esto provocará un error si se intenta devolver un valor desde la función. A continuación hay algunos prototipos completos de funciones:

```
int f1(void); // Devuelve un entero, no tiene argumentos
int f2(); //  igual que f1() en C++ pero no en C Stantard
float f3(float, int, char, double); // Devuelve un float
void f4(void); // No toma argumentos, no devuelve nada
```

Para devolver un valor desde una función, se utiliza la sentencia `return`. Esta sentencia termina la función y salta hasta la sentencia que se halla justo después de la llamada a la función. Si `return` tiene un argumento, se convierte en el valor de retorno de la función. Si una función indica que retornara un tipo en particular, entonces cada sentencia `return` debe retornar un valor de ese tipo. Puede haber más de una sentencia `return` en una definición de función:

```
//: C03:Return.cpp
// Use of "return"
#include <iostream>
using namespace std;

char cfunc(int i) {
  if(i == 0)
    return 'a';
  if(i == 1)
    return 'g';
  if(i == 5)
    return 'z';
  return 'c';
}

int main() {
  cout << "type an integer: ";
  int val;
  cin >> val;
  cout << cfunc(val) << endl;
```

```
} ///:~
```

En `cfunc()`, el primer `if` que comprueba que la condición sea `true` sale de la función con la sentencia `return`. Fíjese que la declaración de la función no es necesaria puesto que la definición aparece antes de ser utilizada en `main()`, de modo que el compilador sabe de su existencia desde dicha definición.

3.1.2. Uso de funciones de librerías C

Todas las funciones en la librería local de funciones de C están disponibles cuando se programa en C++. Se debería buscar bien en la librería de funciones antes de definir una propia - hay muchas probabilidades de que alguien haya resuelto el problema antes, y probablemente haya dedicado más tiempo pensando y depurando.

Una advertencia, del mismo modo: muchos compiladores incluyen muchas funciones extra que hacen la vida mucho mas fácil y resultan tentadoras, pero no son parte de la Librería C Estándar. Si está seguro de que jamás deseará portar la aplicación a otra plataforma (¿y quién está seguro de eso?), adelante -utilice esas funciones y haga su vida más fácil. Si desea que la aplicación pueda ser portada, debería ceñirse únicamente al uso de funciones de la Librería Estándar. Si debe realizar actividades específicas de la plataforma, debería intentar aislar este código de tal modo que pueda cambiarse fácilmente al migrarlo a otra plataforma. En C++, las actividades de una plataforma específica a menudo se encapsulan en una clase, que es la solución ideal.

La fórmula para usar una librería de funciones es la siguiente: primero, encontrar la función en la referencia de programación (muchas referencias de programación ordenan las funciones por categoría además de alfabéticamente). La descripción de la función debería incluir una sección que demuestre la sintaxis del código. La parte superior de esta sección tiene al menos una línea `#include`, mostrando el fichero principal que contiene el prototipo de función. Debe copiar este `#include` en su fichero para que la función esté correctamente declarada. Ahora puede llamar la función de la misma manera que aparece en la sección de sintaxis. Si comete un error, el compilador lo descubrirá comparando la llamada a la función con el prototipo de la cabecera e informará de dicho error. El enlazador busca en la Librería Estándar por defecto, de modo que lo único que hay que hacer es: incluir el fichero de cabecera y llamar a la función.

3.1.3. Creación de librerías propias

Puede reunir funciones propias juntas en una librería. La mayoría de paquetes de programación vienen con un FIXME:bibliotecario que maneja grupos de módulos objeto. Cada FIXME:bibliotecario tiene sus propios comandos, pero la idea general es la siguiente: si se desea crear una librería, se debe hacer un fichero cabecera que contenga prototipos de todas las funciones de la librería. Hay que ubicar este fichero de cabecera en alguna parte de la ruta de búsqueda del preprocesador, ya sea en el directorio local (de modo que se podrá encontrar mediante `#include "header"`) o bien en el directorio `include` (por lo que se podrá encontrar mediante `#include <header>`). Luego se han de juntar todos los módulos objeto y pasarlos al FIXME:bibliotecario junto con un nombre para la librería recién construida (la mayoría de los bibliotecarios requieren una extensión común, como por ejemplo `.lib` o `.a`). Se ha de ubicar la librería completa donde residan todas las demás, de ma-

nera que el enlazador sabrá buscar esas funciones en dicha librería al ser invocadas. Pueden encontrar todos los detalles en su documentación particular, ya que pueden variar de un sistema a otro.

3.2. Control de flujo

Esta sección cubre las sentencias de control de flujo en C++. Debe familiarizarse con estas sentencias antes de que pueda leer o escribir código C o C++.

C++ usa todas las sentencias de control de ejecución de C. Esto incluye `if-else`, `do-while`, `for`, y una sentencia de selección llamada `switch`. C++ también admite el infame `goto`, el cual será evitado en este libro.

3.2.1. Verdadero y falso

Todas las sentencias condicionales utilizan la veracidad o la falsedad de una expresión condicional para determinar el camino de ejecución. Un ejemplo de expresión condicional es `A == B`. Esto utiliza el operador condicional == para saber si la variable `A` es equivalente a la variable `B`. La expresión produce un booleano `true` o `false` (estas son palabras reservadas sólo en C++; en C una expresión es verdadera(*true*) si se evalúa con un valor diferente de cero). Otros operadores condicionales son >, <, >=, etc. Las sentencias condicional se tratarán a fondo más adelante en este capítulo.

3.2.2. `if-else`

La sentencia `if-else` puede existir de dos formas: con o sin el `else`. Las dos formas son:

```
if (óexpresin)
    sentencia
```

ó

```
if (óexpresin)
    sentencia
else
    sentencia
```

La «expresión» se evalúa como `true` o `false`. La «sentencia» puede ser una simple acabada en un punto y coma, o bien una compuesta, lo que no es más que un grupo de sentencias simples encerradas entre llaves. Siempre que se utiliza la palabra «sentencia», implica que la sentencia es simple o compuesta. Tenga en cuenta que dicha sentencia puede ser incluso otro `if`, de modo que se pueden anidar.

```
//: C03:Ifthen.cpp
// Demonstration of if and if-else conditionals
#include <iostream>
using namespace std;

int main() {
```

```
  int i;
  cout << "type a number and 'Enter'" << endl;
  cin >> i;
  if(i > 5)
    cout << "It's greater than 5" << endl;
  else
    if(i < 5)
      cout << "It's less than 5 " << endl;
    else
      cout << "It's equal to 5 " << endl;

  cout << "type a number and 'Enter'" << endl;
  cin >> i;
  if(i < 10)
    if(i > 5)  // "if" is just another statement
      cout << "5 < i < 10" << endl;
    else
      cout << "i <= 5" << endl;
  else // Matches "if(i < 10)"
    cout << "i >= 10" << endl;
} ///:~
```

Por convenio se indenta el cuerpo de una sentencia de control de flujo, de modo que el lector puede determinar fácilmente donde comienza y dónde acaba [1].

3.2.3. `while`

En los bucles de control `while`, `do-while`, y `for`, una sentencia se repite hasta que la expresión de control sea `false`. La estructura de un bucle `while` es:

```
while(óexpresin) sentencia
```

La expresión se evalúa una vez al comienzo del bucle y cada vez antes de cada iteración de la sentencia.

Este ejemplo se mantiene en el cuerpo del bucle `while` hasta que introduzca el número secreto o presione Control-C.

```
//: C03:Guess.cpp
// Guess a number (demonstrates "while")
#include <iostream>
using namespace std;

int main() {
  int secret = 15;
  int guess = 0;
  // "!=" is the "not-equal" conditional:
  while(guess != secret) { // Compound statement
    cout << "guess the number: ";
    cin >> guess;
```

[1] Fíjese en que todas las convenciones parecen acabar estando de acuerdo en que hay que hacer algún tipo de indentación. La pelea entre los estilos de formateo de código no tiene fin. En el Apéndice A se explica el estilo de codificación que se usa en este libro.

```
  }
  cout << "You guessed it!" << endl;
} ///:~
```

La expresión condicional del `while` no está restringida a una simple prueba como en el ejemplo anterior; puede ser tan complicada como se desee siempre y cuando se produzca un resultado `true` o `false`. También puede encontrar código en el que el bucle no tiene cuerpo, sólo un simple punto y coma:

```
while(/* hacer muchas cosas */)
;
```

En estos casos, el programador ha escrito la expresión condicional no sólo para realizar la evaluación, sino también para hacer el trabajo.

3.2.4. `do-while`

El aspecto de `do-while` es

```
do
    sentencia
while(óexpresin);
```

El `do-while` es diferente del `while` ya que la sentencia siempre se ejecuta al menos una vez, aún si la expresión resulta `false` la primera vez. En un `while` normal, si la condición es falsa la primera vez, la sentencia no se ejecuta nunca.

Si se utiliza un `do-while` en `Guess.cpp`, la variable `guess` no necesitaría un valor ficticio inicial, ya que se inicializa por la sentencia `cin` antes de que la variable sea evaluada:

```
//: C03:Guess2.cpp
// The guess program using do-while
#include <iostream>
using namespace std;

int main() {
  int secret = 15;
  int guess; // No initialization needed here
  do {
    cout << "guess the number: ";
    cin >> guess; // Initialization happens
  }   while(guess != secret);
  cout << "You got it!" << endl;
} ///:~
```

Por alguna razón, la mayoría de los programadores tienden a evitar el `do-while` y se limitan a trabajar con `while`.

3.2.5. `for`

Un bucle `for` realiza una inicialización antes de la primera iteración. Luego ejecuta una evaluación condicional y, al final de cada iteración, efectúa algún tipo de «siguiente paso». La estructura del bucle `for` es:

```
for(óinitializacin; ócondicin; paso)
    sentencia
```

Cualquiera de las expresiones de «inicialización», «condición», o «paso» pueden estar vacías. El código de «inicialización» se ejecuta una única vez al principio. La expresión «condicional» se evalúa antes de cada iteración (si se evalúa a `false` desde el principio, el cuerpo del bucle nunca llega a ejecutarse). Al final de cada iteración del bucle, se ejecuta «paso».

Los bucles `for` se utilizan generalmente para tareas de «conteo»:

```
//: C03:Charlist.cpp
// Display all the ASCII characters
// Demonstrates "for"
#include <iostream>
using namespace std;

int main() {
  for(int i = 0; i < 128; i = i + 1)
    if (i != 26)  // ANSI Terminal Clear screen
      cout << " value: " << i
           << " character: "
           << char(i) // Type conversion
           << endl;
} ///:~
```

Puede ocurrir que la variable `i` sea definida en el punto en el que se utiliza, en vez de al principio del bloque delimitado por la apertura de la llave {. Esto difiere de los lenguajes procedurales tradicionales (incluyendo C), en los que se requiere que todas las variables se definan al principio del bloque. Esto se discutirá más adelante en este capítulo.

3.2.6. Las palabras reservadas `break` y `continue`

Dentro del cuerpo de cualquiera de las estructuras de bucle `while`, `do-while`, o `for`, se puede controlar el flujo del bucle utilizando `break` y `continue`. `break` interrumpe el bucle sin ejecutar el resto de las sentencias de esa iteración. `continue` detiene la ejecución de la iteración actual, vuelve al principio del bucle y comienza la siguiente iteración.

A modo de ejemplo de `break` y `continue`, este programa es un menu de sistema muy simple:

```
//: C03:Menu.cpp
// Simple menu program demonstrating
// the use of "break" and "continue"
#include <iostream>
using namespace std;
```

```
int main() {
  char c; // To hold response
  while(true) {
    cout << "MAIN MENU:" << endl;
    cout << "l: left, r: right, q: quit -> ";
    cin >> c;
    if(c == 'q')
      break; // Out of "while(1)"
    if(c == 'l') {
      cout << "LEFT MENU:" << endl;
      cout << "select a or b: ";
      cin >> c;
      if(c == 'a') {
        cout << "you chose 'a'" << endl;
        continue; // Back to main menu
      }
      if(c == 'b') {
        cout << "you chose 'b'" << endl;
        continue; // Back to main menu
      }
      else {
        cout << "you didn't choose a or b!"
             << endl;
        continue; // Back to main menu
      }
    }
    if(c == 'r') {
      cout << "RIGHT MENU:" << endl;
      cout << "select c or d: ";
      cin >> c;
      if(c == 'c') {
        cout << "you chose 'c'" << endl;
        continue; // Back to main menu
      }
      if(c == 'd') {
        cout << "you chose 'd'" << endl;
        continue; // Back to main menu
      }
      else {
        cout << "you didn't choose c or d!"
             << endl;
        continue; // Back to main menu
      }
    }
    cout << "you must type l or r or q!" << endl;
  }
  cout << "quitting menu..." << endl;
} ///:~
```

Si el usuario selecciona q en el menu principal, se utiliza la palabra reservada `break` para salir, de otro modo, el programa continúa ejecutándose indefinidamente. Después de cada selección de sub-menu, se usa la palabra reservada `continue` para volver atrás hasta el comienzo del bucle `while`.

La sentencia `while(true)` es el equivalente a decir «haz este bucle para siempre». La sentencia `break` permite romper este bucle infinito cuando el usuario teclea

q.

3.2.7. `switch`

Una sentencia `switch` selecciona un fragmento de código entre varios posibles en base al valor de una expresión entera. Su estructura es:

```
switch(selector) {
case valor-entero1 : sentencia; break;
case valor-entero2 : sentencia; break;
case valor-entero3 : sentencia; break;
case valor-entero4 : sentencia; break;
case valor-entero5 : sentencia; break;
    (...)
default: sentencia;
}
```

`selector` es una expresión que produce un valor entero. El `switch` compara el resultado de `selector` para cada valor entero. Si encuentra una coincidencia, se ejecutará la sentencia correspondiente (sea simple o compuesta). Si no se encuentra ninguna coincidencia se ejecutará la sentencia `default`.

Se puede observar en la definición anterior que cada `case` acaba con un `break`, lo que causa que la ejecución salte hasta el final del cuerpo del `switch` (la llave final que cierra el `switch`). Esta es la forma convencional de construir una sentencia `switch`, pero la palabra `break` es opcional. Si no se indica, el `case` que se ha cumplido «cae» al siguiente de la lista. Esto significa, que el código del siguiente `case`, se ejecutara hasta que se encuentre un `break`. Aunque normalmente no se desea este tipo de comportamiento, puede ser de ayuda para un programador experimentado.

La sentencia `switch` es una manera limpia de implementar una selección multimodo (por ejemplo, seleccionando de entre un número de paths de ejecución), pero requiere un selector que pueda evaluarse como un entero en el momento de la compilación. Si quisiera utilizar, por ejemplo, un objeto `string` como selector, no funcionará en una sentencia `switch`. Para un selector de tipo `string`, se debe utilizar una serie de sentencias `if` y comparar el `string` dentro de la condición.

El ejemplo del menu demostrado anteriormente proporciona un ejemplo particularmente interesante de un `switch`:

```
//: C03:Menu2.cpp
// A menu using a switch statement
#include <iostream>
using namespace std;

int main() {
  bool quit = false;  // Flag for quitting
  while(quit == false) {
    cout << "Select a, b, c or q to quit: ";
    char response;
    cin >> response;
    switch(response) {
      case 'a' : cout << "you chose 'a'" << endl;
                 break;
      case 'b' : cout << "you chose 'b'" << endl;
                 break;
```

```
      case 'c' : cout << "you chose 'c'" << endl;
                 break;
      case 'q' : cout << "quitting menu" << endl;
                 quit = true;
                 break;
      default  : cout << "Please use a,b,c or q!"
                 << endl;
    }
  }
} ///:~
```

El flag `quit` es un bool, abreviatura para «booleano», que es un tipo que sólo se encuentra en C++. Puede tener únicamente los valores `true` o `false`. Seleccionando q se asigna el valor `true` al flag «quit». La próxima vez que el selector sea evaluado, `quit == false` retornará `false` de modo que el cuerpo del bucle `while` no se ejecutará.

3.2.8. Uso y maluso de `goto`

La palabra clave `goto` está soportada en C++, dado que existe en C. El uso de `goto` a menudo es considerado como un estilo de programación pobre, y la mayor parte de las veces lo es. Siempre que se utilice `goto`, se debe revisar bien el código para ver si hay alguna otra manera de hacerlo. En raras ocasiones, `goto` puede resolver un problema que no puede ser resuelto de otra manera, pero, aún así, se debe considerar cuidadosamente. A continuación aparece un ejemplo que puede ser un candidato plausible:

```
//: C03:gotoKeyword.cpp
// The infamous goto is supported in C++
#include <iostream>
using namespace std;

int main() {
  long val = 0;
  for(int i = 1; i < 1000; i++) {
    for(int j = 1; j < 100; j += 10) {
      val = i * j;
      if(val > 47000)
        goto bottom;
        // Break would only go to the outer 'for'
    }
  }
  bottom: // A label
  cout << val << endl;
} ///:~
```

La alternativa sería dar valor a un booleano que sea evaluado en el `for` externo, y luego hacer un `break` desde el `for` interno. De todos modos, si hay demasiados niveles de `for` o `while` esto puede llegar a ser pesado.

3.2.9. Recursividad

La recursividad es una técnica de programación interesante y a veces útil, en donde se llama a la función desde el cuerpo de la propia función. Por supuesto, si eso es todo lo que hace, se estaría llamando a la función hasta que se acabase la memoria de ejecución, de modo que debe existir una manera de «escaparse» de la llamada recursiva. En el siguiente ejemplo, esta «escapada» se consigue simplemente indicando que la recursión sólo continuará hasta que `cat` exceda Z: [2]

```
//: C03:CatsInHats.cpp
// Simple demonstration of recursion
#include <iostream>
using namespace std;

void removeHat(char cat) {
  for(char c = 'A'; c < cat; c++)
    cout << "  ";
  if(cat <= 'Z') {
    cout << "cat " << cat << endl;
    removeHat(cat + 1); // Recursive call
  } else
    cout << "VOOM!!!" << endl;
}

int main() {
  removeHat('A');
} ///:~
```

En `removeHat()`, se puede ver que mientras `cat` sea menor que Z, `removeHat()` se llamará a sí misma, efectuando así la recursividad. Cada vez que se llama `removeHat()`, su argumento crece en una unidad más que el `cat` actual de modo que el argumento continúa aumentando.

La recursividad a menudo se utiliza cuando se evalúa algún tipo de problema arbitrariamente complejo, ya que no se restringe la solución a ningún tamaño particular - la función puede simplemente efectuar la recursividad hasta que se haya alcanzado el final del problema.

3.3. Introducción a los operadores

Se pueden ver los operadores como un tipo especial de función (aprenderá que en C++ la sobrecarga de operadores los trata precisamente de esa forma). Un operador recibe uno o más argumentos y produce un nuevo valor. Los argumentos se pasan de una manera diferente que en las llamadas a funciones normales, pero el efecto es el mismo.

Por su experiencia previa en programación, debe estar razonablemente cómodo con los operadores que se han utilizados. Los conceptos de adición (+), substracción y resta unaria (−), multiplicación (∗), división (/), y asignación (=) tienen todos el mismo significado en cualquier lenguaje de programación. El grupo completo de operadores se enumera más adelante en este capítulo.

[2] Gracias a Kris C. Matson por proponer este ejercicio.

3.3.1. Precedencia

La precedencia de operadores define el orden en el que se evalúa una expresión con varios operadores diferentes. C y C++ tienen reglas específicas para determinar el orden de evaluación. Lo más fácil de recordar es que la multiplicación y la división se ejecutan antes que la suma y la resta. Luego, si una expresión no es transparente al programador que la escribe, probablemente tampoco lo será para nadie que lea el código, de modo que se deben usar paréntesis para hacer explícito el orden de la evaluación. Por ejemplo:

```
A = X + Y - 2/2 + Z;
```

Tiene un significado muy distinto de la misma expresión pero con un configuración de paréntesis particular:

```
A = X + (Y - 2)/(2 + Z);
```

(Intente evaluar el resultado con X =1, Y = 2, y Z = 3.)

3.3.2. Auto incremento y decremento

C, y por tanto C++, está lleno de atajos. Los atajos pueden hacer el código mucho mas fácil de escribir, y a veces más difícil de leer. Quizás los diseñadores del lenguaje C pensaron que sería más fácil entender un trozo de código complicado si los ojos no tienen que leer una larga línea de letras.

Los operadores de auto-incremento y auto-decremento son de los mejores atajos. Se utilizan a menudo para modificar las variables que controlan el número de veces que se ejecuta un bucle.

El operador de auto-decremento es `--` que significa «decrementar de a una unidad». El operador de auto-incremento es `++` que significa «incrementar de a una unidad». Si es un entero, por ejemplo, la expresión `++A` es equivalente a `(A = A + 1)`. Los operadores de auto-incremento y auto-decremento producen el valor de la variable como resultado. Si el operador aparece antes de la variable (p.ej, `++A`), la operación se ejecuta primero y después se produce el valor resultante. Si el operador aparece a continuación de la variable (p.ej, `A++`), primero se produce el valor actual, y luego se realiza la operación. Por ejemplo:

```
//: C03:AutoIncrement.cpp
// Shows use of auto-increment
// and auto-decrement operators.
#include <iostream>
using namespace std;

int main() {
  int i = 0;
  int j = 0;
  cout << ++i << endl; // Pre-increment
  cout << j++ << endl; // Post-increment
  cout << --i << endl; // Pre-decrement
  cout << j-- << endl; // Post decrement
} ///:~
```

Si se ha estado preguntando acerca del nombre «C++», ahora lo entenderá. Significa «un paso más allá de C» [3]

3.4. Introducción a los tipos de datos

Los *tipos de datos* definen el modo en que se usa el espacio (memoria) en los programas. Especificando un tipo de datos, está indicando al compilador como crear un espacio de almacenamiento en particular, y también como manipular este espacio.

Los tipos de datos pueden estar predefinidos o abstractos. Un tipo de dato predefinido es intrínsecamente comprendido por el compilador. Estos tipos de datos son casi idénticos en C y C++. En contraste, un tipo de datos definido por el usuario es aquel que usted o cualquier otro programador crea como una clase. Estos se denominan comúnmente tipos de datos abstractos. El compilador sabe como manejar tipos predefinidos por si mismo; y «aprende» como manejar tipos de datos abstractos leyendo los ficheros de cabeceras que contienen las declaraciones de las clases (esto se verá con más detalle en los siguientes capítulos).

3.4.1. Tipos predefinidos básicos

La especificación del Estándar C para los tipos predefinidos (que hereda C++) no indica cuantos bits debe contener cada uno de ellos. En vez de eso, estipula el mínimo y máximo valor que cada tipo es capaz de almacenar. Cuando una máquina se basa en sistema binario, este valor máximo puede ser directamente traducido a un numero mínimo necesario de bits para alojar ese valor. De todos modos, si una maquina usa, por ejemplo, el código binario decimal (BCD) para representar los números, entonces el espacio requerido para alojar el máximo número para cada tipo de datos será diferente. El mínimo y máximo valor que se puede almacenar en los distintos tipos de datos se define en los ficheros de cabeceras del sistema `limits.h` y `float.h` (en C++ normalmente será `#include <climits>` y `<cfloat>`).

C y C++ tienen cuatro tipos predefinidos básicos, descritos aquí para máquinas basadas en sistema binario. Un char es para almacenar caracteres y utiliza un mínimo de 8 bits (un byte) de espacio, aunque puede ser mas largo. Un int almacena un número entero y utiliza un mínimo de dos bytes de espacio. Los tipos float y el double almacenan números con coma flotante, usualmente en formato IEEE. el float es para precisión simple y el double es para doble precisión.

Como se ha mencionado previamente, se pueden definir variables en cualquier sitio en un ámbito determinado, y puede definirlas e inicializarlas al mismo tiempo. A continuación se indica cómo definir variables utilizando los cuatro tipos básicos de datos:

```
//: C03:Basic.cpp
// Defining the four basic data
// types in C and C++

int main() {
  // Definition without initialization:
  char protein;
  int carbohydrates;
  float fiber;
```

[3] (N. de T.) ...aunque se evalúa como «C».

```
  double fat;
  // Simultaneous definition & initialization:
  char pizza = 'A', pop = 'Z';
  int dongdings = 100, twinkles = 150,
    heehos = 200;
  float chocolate = 3.14159;
  // Exponential notation:
  double fudge_ripple = 6e-4;
} ///:~
```

La primera parte del programa define variables de los cuatro tipos básicos sin inicializarlas. Si no se inicializa una variable, el Estándar dice que su contenido es indefinido (normalmente, esto significa que contienen basura). La segunda parte del programa define e inicializa variables al mismo tiempo (siempre es mejor, si es posible, dar un valor inicial en el momento de la definición). Note que el uso de notación exponencial en la contante 6e-4, significa «6 por 10 elevado a -4».

3.4.2. booleano, verdadero y falso

Antes de que bool se convirtiese en parte del Estándar C++, todos tendían a utilizar diferentes técnicas para producir comportamientos similares a los booleanos. Esto produjo problemas de portabilidad y podían acarrear errores sutiles.

El tipo bool del Estándar C++ puede tener dos estados expresados por las constantes predefinidas `true` (lo que lo convierte en el entero 1) y `false` (lo que lo convierte en el entero 0). Estos tres nombres son palabras reservadas. Además, algunos elementos del lenguaje han sido adaptados:

Elemento	**Uso con booleanos**
&& || !	Toman argumentos booleanos y producen valores bool
< > <= >= == !=	Producen resultados bool
`if`, `for`, `while`, `do`	Las expresiones condicionales se convierten en valores bool
?:	El primer operando se convierte a un valor bool

Cuadro 3.1: Expresiones que utilizan booleanos

Como hay mucho código existente que utiliza un int para representar una bandera, el compilador lo convertirá implícitamente de int a bool (los valores diferentes de cero producirán `true`, mientras que los valores cero, producirán `false`). Idealmente, el compilador le dará un aviso como una sugerencia para corregir la situación.

Un modismo que se considera «estilo de programación pobre» es el uso de ++ para asignar a una bandera el valor `true`. Esto aún se permite, pero está obsoleto, lo que implica que en el futuro será ilegal. El problema es que se está haciendo una conversión implícita de un bool a un int, incrementando el valor (quizá más allá del rango de valores booleanos cero y uno), y luego implícitamente convirtiéndolo otra vez a bool.

Los punteros (que se describen más adelante en este capitulo) también se convierten automáticamente a bool cuando es necesario.

3.4.3. Especificadores

Los especificadores modifican el significado de los tipos predefinidos básicos y los expanden a un conjunto más grande. Hay cuatro especificadores: `long`, `short`, `signed` y `unsigned`.

`long` y `short` modifican los valores máximos y mínimos que un tipo de datos puede almacenar. Un int plano debe tener al menos el tamaño de un short. La jerarquía de tamaños para tipos enteros es: short int, int, long int. Todos pueden ser del mismo tamaño, siempre y cuando satisfagan los requisitos de mínimo/máximo. En una maquina con una palabra de 64 bits, por defecto, todos los tipos de datos podrían ser de 64 bits.

La jerarquía de tamaño para los números en coma flotante es: float, double y long double. «long float» no es un tipo válido. No hay números en coma flotantes de tamaño `short`.

Los especificadores `signed` y `unsigned` indican al compilador cómo utilizar el bit del signo con los tipos enteros y los caracteres (los números de coma flotante siempre contienen un signo). Un número `unsigned` no guarda el valor del signo y por eso tiene un bit extra disponible, de modo que puede guardar el doble de números positivos que pueden guardarse en un número `signed`. `signed` se supone por defecto y sólo es necesario con char, char puede ser o no por defecto un `signed`. Especificando signed char, se está forzando el uso del bit del signo.

El siguiente ejemplo muestra el tamaño de los tipos de datos en bytes utilizando el operador `sizeof`, descripto más adelante en ese capítulo:

```
//: C03:Specify.cpp
// Demonstrates the use of specifiers
#include <iostream>
using namespace std;

int main() {
  char c;
  unsigned char cu;
  int i;
  unsigned int iu;
  short int is;
  short iis; // Same as short int
  unsigned short int isu;
  unsigned short iisu;
  long int il;
  long iil;  // Same as long int
  unsigned long int ilu;
  unsigned long iilu;
  float f;
  double d;
  long double ld;
  cout
    << "\n char= " << sizeof(c)
    << "\n unsigned char = " << sizeof(cu)
    << "\n int = " << sizeof(i)
    << "\n unsigned int = " << sizeof(iu)
    << "\n short = " << sizeof(is)
    << "\n unsigned short = " << sizeof(isu)
    << "\n long = " << sizeof(il)
    << "\n unsigned long = " << sizeof(ilu)
```

```
    << "\n float = " << sizeof(f)
    << "\n double = " << sizeof(d)
    << "\n long double = " << sizeof(ld)
    << endl;
} ///:~
```

Tenga en cuenta que es probable que los resultados que se consiguen ejecutando este programa sean diferentes de una maquina/sistema operativo/compilador a otro, ya que (como se mencionaba anteriormente) lo único que ha de ser consistente es que cada tipo diferente almacene los valores mínimos y máximos especificados en el Estándar.

Cuando se modifica un int con `short` o `long`, la palabra reservada int es opcional, como se muestra a continuación.

3.4.4. Introducción a punteros

Siempre que se ejecuta un programa, se carga primero (típicamente desde disco) a la memoria del ordenador. De este modo, todos los elementos del programa se ubican en algún lugar de la memoria. La memoria se representa normalmente como series secuenciales de posiciones de memoria; normalmente se hace referencia a estas localizaciones como bytes de ocho bits, pero realmente el tamaño de cada espacio depende de la arquitectura de cada máquina particular y se llamada normalmente tamaño de palabra de la máquina. Cada espacio se puede distinguir unívocamente de todos los demás espacios por su dirección. Para este tema en particular, se establecerá que todas las máquinas usan bytes que tienen direcciones secuenciales, comenzando en cero y subiendo hasta la cantidad de memoria que posea la máquina.

Como el programa reside en memoria mientras se está ejecutando, cada elemento de dicho programa tiene una dirección. Suponga que empezamos con un programa simple:

```
//: C03:YourPets1.cpp
#include <iostream>
using namespace std;

int dog, cat, bird, fish;

void f(int pet) {
  cout << "pet id number: " << pet << endl;
}

int main() {
  int i, j, k;
} ///:~
```

Cada uno de los elementos de este programa tiene una localización en memoria mientras el programa se está ejecutando. Incluso las funciones ocupan espacio. Como verá, se da por sentado que el tipo de un elemento y la forma en que se define determina normalmente el área de memoria en la que se ubica dicho elemento.

Hay un operador en C y C++ que permite averiguar la dirección de un elemento. Se trata del operador `&`. Sólo hay que anteponer el operador `&` delante del nom-

bre identificador y obtendrá la dirección de ese identificador. Se puede modificar `YourPets1.cpp` para mostrar las direcciones de todos sus elementos, del siguiente modo:

```
//: C03:YourPets2.cpp
#include <iostream>
using namespace std;

int dog, cat, bird, fish;

void f(int pet) {
  cout << "pet id number: " << pet << endl;
}

int main() {
  int i, j, k;
  cout << "f(): " << (long)&f << endl;
  cout << "dog: " << (long)&dog << endl;
  cout << "cat: " << (long)&cat << endl;
  cout << "bird: " << (long)&bird << endl;
  cout << "fish: " << (long)&fish << endl;
  cout << "i: " << (long)&i << endl;
  cout << "j: " << (long)&j << endl;
  cout << "k: " << (long)&k << endl;
} ///:~
```

El `(long)` es una molde. Indica «No tratar como su tipo normal, sino como un long». El molde no es esencial, pero si no existiese, las direcciones aparecerían en hexadecimal, de modo que el moldeado a long hace las cosas más legibles.

Los resultados de este programa variarán dependiendo del computador, del sistema operativo, y de muchos otros tipos de factores, pero siempre darán un resultado interesante. Para una única ejecución en mi computador, los resultados son como estos:

```
f(): 4198736
dog: 4323632
cat: 4323636
bird: 4323640
fish: 4323644
i: 6684160
j: 6684156
k: 6684152
```

Se puede apreciar como las variables que se han definido dentro de `main()` están en un área distinta que las variables definidas fuera de `main()`; entenderá el porque cuando se profundice más en el lenguaje. También, `f()` parece estar en su propia área; el código normalmente se separa del resto de los datos en memoria.

Otra cosa a tener en cuenta es que las variables definidas una a continuación de la otra parecen estar ubicadas de manera contigua en memoria. Están separadas por el número de bytes requeridos por su tipo de dato. En este programa el único tipo de dato utilizado es el int, y la variable `cat` está separada de `dog` por cuatro bytes, `bird` está separada por cuatro bytes de `cat`, etc. De modo que en el computador en que ha sido ejecutado el programa, un entero ocupa cuatro bytes.

¿Qué se puede hacer con las direcciones de memoria, además de este interesante experimento de mostrar cuanta memoria ocupan? Lo más importante que se puede hacer es guardar esas direcciones dentro de otras variables para su uso posterior. C y C++ tienen un tipo de variable especial para guardar una dirección. Esas variables se llaman *punteros*.

El operador que define un puntero es el mismo que se utiliza para la multiplicación: *. El compilador sabe que no es una multiplicación por el contexto en el que se usa, tal como podrá comprobar.

Cuando se define un puntero, se debe especificar el tipo de variable al que apunta. Se comienza dando el nombre de dicho tipo, después en lugar de escribir un identificador para la variable, usted dice «Espera, esto es un puntero» insertando un asterisco entre el tipo y el identificador. De modo que un puntero a int tiene este aspecto:

```
int* ip; // ip apunta a una variable int
```

La asociación del * con el tipo parece práctica y legible, pero puede ser un poco confusa. La tendencia podría ser decir «puntero-entero» como un si fuese un tipo simple. Sin embargo, con un int u otro tipo de datos básico, se puede decir:

```
int a, b, c;
```

así que con un puntero, diría:

```
int* ipa, ipb, ipc;
```

La sintaxis de C (y por herencia, la de C++) no permite expresiones tan cómodas. En las definiciones anteriores, sólo `ipa` es un puntero, pero `ipb` e `ipc` son ints normales (se puede decir que «`*` está mas unido al identificador»). Como consecuencia, los mejores resultados se pueden obtener utilizando sólo una definición por línea; y aún se conserva una sintaxis cómoda y sin la confusión:

```
int* ipa;
int* ipb;
int* ipc;
```

Ya que una pauta de programación de C++ es que siempre se debe inicializar una variable al definirla, realmente este modo funciona mejor. Por ejemplo, Las variables anteriores no se inicializan con ningún valor en particular; contienen basura. Es más fácil decir algo como:

```
int a = 47;
int* ipa = &a;
```

Ahora tanto `a` como `ipa` están inicializadas, y `ipa` contiene la dirección de `a`.

Una vez que se inicializa un puntero, lo más básico que se puede hacer con Él es utilizarlo para modificar el valor de lo que apunta. Para acceder a la variable a través del puntero, se *dereferencia* el puntero utilizando el mismo operador que se usó para definirlo, como sigue:

```
*ipa = 100;
```

Ahora `a` contiene el valor 100 en vez de 47.

Estas son las normas básicas de los punteros: se puede guardar una dirección, y se puede utilizar dicha dirección para modificar la variable original. Pero la pregunta aún permanece: ¿por qué se querría cambiar una variable utilizando otra variable como intermediario?

Para esta visión introductoria a los punteros, podemos dividir la respuesta en dos grandes categorías:

1. Para cambiar «objetos externos» desde dentro de una función. Esto es quizás el uso más básico de los punteros, y se examinará más adelante.

2. Para conseguir otras muchas técnicas de programación ingeniosas, sobre las que aprenderá en el resto del libro.

3.4.5. Modificar objetos externos

Normalmente, cuando se pasa un argumento a una función, se hace una copia de dicho argumento dentro de la función. Esto se llama *paso-por-valor*. Se puede ver el efecto de un paso-por-valor en el siguiente programa:

```
//: C03:PassByValue.cpp
#include <iostream>
using namespace std;

void f(int a) {
  cout << "a = " << a << endl;
  a = 5;
  cout << "a = " << a << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
  f(x);
  cout << "x = " << x << endl;
} ///:~
```

En `f()`, `a` es una variable local, de modo que existe únicamente mientras dura la llamada a la función `f()`. Como es un argumento de una función, el valor de `a` se inicializa mediante los argumentos que se pasan en la invocación de la función; en `main()` el argumento es `x`, que tiene un valor 47, de modo que el valor es copiado en `a` cuando se llama a `f()`.

Cuando ejecute el programa verá:

```
x = 47
a = 47
a = 5
x = 47
```

Por supuesto, inicialmente x es 47. Cuando se llama f(), se crea un espacio temporal para alojar la variable a durante la ejecución de la función, y el valor de x se copia a a, el cual es verificado mostrándolo por pantalla. Se puede cambiar el valor de a y demostrar que ha cambiado. Pero cuando f() termina, el espacio temporal que se había creado para a desaparece, y se puede observar que la única conexión que existía entre a y x ocurrió cuando el valor de x se copió en a.

Cuando está dentro de f(), x es el *objeto externo* (mi terminología), y cambiar el valor de la variable local no afecta al objeto externo, lo cual es bastante lógico, puesto que son dos ubicaciones separadas en la memoria. Pero ¿y si quiere modificar el objeto externo? Aquí es donde los punteros entran en acción. En cierto sentido, un puntero es un alias de otra variable. De modo que si a una función se le pasa un puntero en lugar de un valor ordinario, se está pasando de hecho un alias del objeto externo, dando la posibilidad a la función de que pueda modificar el objeto externo, tal como sigue:

```
//: C03:PassAddress.cpp
#include <iostream>
using namespace std;

void f(int* p) {
  cout << "p = " << p << endl;
  cout << "*p = " << *p << endl;
  *p = 5;
  cout << "p = " << p << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
  cout << "&x = " << &x << endl;
  f(&x);
  cout << "x = " << x << endl;
} ///:~
```

Ahora f() toma el puntero como un argumento y dereferencia el puntero durante la asignación, lo que modifica el objeto externo x. La salida es:

```
x = 47
&x = 0065FE00
p = 0065FE00
*p = 47
p = 0065FE00
x = 5
```

Tenga en cuenta que el valor contenido en p es el mismo que la dirección de x - el puntero p de hecho apunta a x. Si esto no es suficientemente convincente, cuando p es dereferenciado para asignarle el valor 5, se ve que el valor de x cambia a 5 también.

De ese modo, pasando un puntero a una función le permitirá a esa función modificar el objeto externo. Se verán muchos otros usos de los punteros más adelante, pero podría decirse que éste es el más básico y posiblemente el más común.

3.4.6. Introducción a las referencias de C++

Los punteros funcionan más o menos igual en C y en C++, pero C++ añade un modo adicional de pasar una dirección a una función. Se trata del *paso-por-referencia* y existe en otros muchos lenguajes, de modo que no es una invención de C++.

La primera impresión que dan las referencias es que no son necesarias, que se pueden escribir cualquier programa sin referencias. En general, eso es verdad, con la excepción de unos pocos casos importantes que se tratarán más adelante en el libro, pero la idea básica es la misma que la demostración anterior con el puntero: se puede pasar la dirección de un argumento utilizando una referencia. La diferencia entre referencias y punteros es que *invocar* a una función que recibe referencias es mas limpio, sintácticamente, que llamar a una función que recibe punteros (y es exactamente esa diferencia sintáctica la que hace a las referencias esenciales en ciertas situaciones). Si `PassAddress.cpp` se modifica para utilizar referencias, se puede ver la diferencia en la llamada a la función en `main()`:

```
//: C03:PassReference.cpp
#include <iostream>
using namespace std;

void f(int& r) {
  cout << "r = " << r << endl;
  cout << "&r = " << &r << endl;
  r = 5;
  cout << "r = " << r << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
  cout << "&x = " << &x << endl;
  f(x); // Looks like pass-by-value,
        // is actually pass by reference
  cout << "x = " << x << endl;
} ///:~
```

En la lista de argumentos de `f()`, en lugar de escribir int* para pasar un puntero, se escribe int& para pasar una referencia. Dentro de `f()`, si dice simplemente `r` (lo que produciría la dirección si `r` fuese un puntero) se obtiene *el valor en la variable que `r` está referenciando*. Si se asigna a `r`, en realidad se está asignado a la variable a la que que `r` referencia. De hecho, la única manera de obtener la dirección que contiene `r` es con el operador `&`.

En `main()`, se puede ver el efecto clave de las referencias en la sintaxis de la llamada a `f()`, que es simplemente `f(x)`. Aunque eso parece un paso-por-valor ordinario, el efecto de la referencia es que en realidad toma la dirección y la pasa, en lugar de hacer una copia del valor. La salida es:

```
x = 47
&x = 0065FE00
r = 47
&r = 0065FE00
r = 5
x = 5
```

De manera que se puede ver que un paso-por-referencia permite a una función

modificar el objeto externo, al igual que al pasar un puntero (también se puede observar que la referencia esconde el hecho de que se está pasando una dirección; esto se verá más adelante en el libro). Gracias a esta pequeña introducción se puede asumir que las referencias son sólo un modo sintácticamente distinto (a veces referido como «azúcar sintáctico») para conseguir lo mismo que los punteros: permitir a las funciones cambiar los objetos externos.

3.4.7. Punteros y Referencias como modificadores

Hasta ahora, se han visto los tipos básicos de datos char, int, float, y double, junto con los especificadores signed, unsigned, short, y long, que se pueden utilizar con los tipos básicos de datos en casi cualquier combinación. Ahora hemos añadido los punteros y las referencias, que son lo ortogonal a los tipos básicos de datos y los especificadores, de modo que las combinaciones posibles se acaban de triplicar:

```
//: C03:AllDefinitions.cpp
// All possible combinations of basic data types,
// specifiers, pointers and references
#include <iostream>
using namespace std;

void f1(char c, int i, float f, double d);
void f2(short int si, long int li, long double ld);
void f3(unsigned char uc, unsigned int ui,
  unsigned short int usi, unsigned long int uli);
void f4(char* cp, int* ip, float* fp, double* dp);
void f5(short int* sip, long int* lip,
  long double* ldp);
void f6(unsigned char* ucp, unsigned int* uip,
  unsigned short int* usip,
  unsigned long int* ulip);
void f7(char& cr, int& ir, float& fr, double& dr);
void f8(short int& sir, long int& lir,
  long double& ldr);
void f9(unsigned char& ucr, unsigned int& uir,
  unsigned short int& usir,
  unsigned long int& ulir);

int main() {} ///:~
```

Los punteros y las referencias entran en juego también cuando se pasan objetos dentro y fuera de las funciones; aprenderá sobre ello en un capítulo posterior.

Hay otro tipo que funciona con punteros: void. Si se establece que un puntero es un void*, significa que cualquier tipo de dirección se puede asignar a ese puntero (en cambio si tiene un int*, sólo puede asignar la dirección de una variable int a ese puntero). Por ejemplo:

```
//: C03:VoidPointer.cpp
int main() {
  void* vp;
  char c;
  int i;
  float f;
```

```
  double d;
  // The address of ANY type can be
  // assigned to a void pointer:
  vp = &c;
  vp = &i;
  vp = &f;
  vp = &d;
} ///:~
```

Una vez que se asigna a un void* se pierde cualquier información sobre el tipo de la variables. Esto significa que antes de que se pueda utilizar el puntero, se debe moldear al tipo correcto:

```
//: C03:CastFromVoidPointer.cpp
int main() {
  int i = 99;
  void* vp = &i;
  // Can't dereference a void pointer:
  // *vp = 3; // Compile-time error
  // Must cast back to int before dereferencing:
  *((int*)vp) = 3;
} ///:~
```

El molde `(int*)vp` toma el void* y le dice al compilador que lo trate como un int*, y de ese modo se puede dereferenciar correctamente. Puede observar que esta sintaxis es horrible, y lo es, pero es peor que eso - el void* introduce un agujero en el sistema de tipos del lenguaje. Eso significa, que permite, o incluso promueve, el tratamiento de un tipo como si fuera otro tipo. En el ejemplo anterior, se trata un int como un int mediante el moldeado de `vp` a int*, pero no hay nada que indique que no se lo puede moldear a char* o double*, lo que modificaría una cantidad diferente de espacio que ha sido asignada al int, lo que posiblemente provocará que el programa falle.. En general, los punteros void deberían ser evitados, y utilizados únicamente en raras ocasiones, que no se podrán considerar hasta bastante más adelante en el libro.

No se puede tener una referencia void, por razones que se explicarán en el capítulo 11.

3.5. Alcance

Las reglas de ámbitos dicen cuando es válida una variable, dónde se crea, y cuándo se destruye (es decir, sale de ámbito). El ámbito de una variable se extiende desde el punto donde se define hasta la primera llave que empareja con la llave de apertura antes de que la variable fuese definida. Eso quiere decir que un ámbito se define por su juego de llaves «más cercanas». Para ilustrarlo:

```
//: C03:Scope.cpp
// How variables are scoped
int main() {
  int scp1;
  // scp1 visible here
  {
```

```
    // scp1 still visible here
    //.....
    int scp2;
    // scp2 visible here
    //.....
    {
      // scp1 & scp2 still visible here
      //..
      int scp3;
      // scp1, scp2 & scp3 visible here
      // ...
    } // <-- scp3 destroyed here
    // scp3 not available here
    // scp1 & scp2 still visible here
    // ...
  } // <-- scp2 destroyed here
  // scp3 & scp2 not available here
  // scp1 still visible here
  //..
} // <-- scp1 destroyed here
///:~
```

El ejemplo anterior muestra cuándo las variables son visibles y cuando dejan de estar disponibles (es decir, cuando *salen del ámbito*). Una variable se puede utilizar sólo cuando se está dentro de su ámbito. Los ámbitos pueden estar anidados, indicados por parejas de llaves dentro de otras parejas de llaves. El anidado significa que se puede acceder a una variable en un ámbito que incluye el ámbito en el que se está. En el ejemplo anterior, la variable `scp1` está disponible dentro de todos los demás ámbitos, mientras que `scp3` sólo está disponible en el ámbito más interno.

3.5.1. Definición de variables «al vuelo»

Como se ha mencionado antes en este capítulo, hay una diferencia importante entre C y C++ al definir variables. Ambos lenguajes requieren que las variables estén definidas antes de utilizarse, pero C (y muchos otros lenguajes procedurales tradicionales) fuerzan a que se definan todas las variables al principio del bloque, de modo que cuando el compilador crea un bloque puede crear espacio para esas variables.

Cuando uno lee código C, normalmente lo primero que encuentra cuando empieza un ámbito, es un bloque de definiciones de variables. Declarar todas las variables al comienzo de un bloque requiere que el programador escriba de un modo particular debido a los detalles de implementación del lenguaje. La mayoría de las personas no conocen todas las variables que van a utilizar antes de escribir el código, de modo que siempre están volviendo al principio del bloque para insertar nuevas variables, lo cual resulta pesado y causa errores. Normalmente estas definiciones de variables no significan demasiado para el lector, y de hecho tienden a ser confusas porque aparecen separadas del contexto en el cual se utilizan.

C++ (pero no C) permite definir variables en cualquier sitio dentro de un ámbito, de modo que se puede definir una variable justo antes de usarla. Además, se puede inicializar la variable en el momento de la definición, lo que previene cierto tipo de errores. Definir las variables de este modo hace el código más fácil de escribir y reduce los errores que provoca estar forzado a volver atrás y adelante dentro de

un ámbito. Hace el código más fácil de entender porque es una variable definida en el contexto de su utilización. Esto es especialmente importante cuando se está definiendo e inicializando una variable al mismo tiempo - se puede ver el significado del valor de inicialización por el modo en el que se usa la variable.

También se pueden definir variables dentro de expresiones de control tales como los bucles `for` y `while`, dentro de las sentencias de condiciones `if`, y dentro de la sentencia de selección `switch`. A continuación hay un ejemplo que muestra la definición de variables al-vuelo:

```
//: C03:OnTheFly.cpp
// On-the-fly variable definitions
#include <iostream>
using namespace std;

int main() {
  //..
  { // Begin a new scope
    int q = 0; // C requires definitions here
    //..
    // Define at point of use:
    for(int i = 0; i < 100; i++) {
      q++; // q comes from a larger scope
      // Definition at the end of the scope:
      int p = 12;
    }
    int p = 1;  // A different p
  } // End scope containing q & outer p
  cout << "Type characters:" << endl;
  while(char c = cin.get() != 'q') {
    cout << c << " wasn't it" << endl;
    if(char x = c == 'a' || c == 'b')
      cout << "You typed a or b" << endl;
    else
      cout << "You typed " << x << endl;
  }
  cout << "Type A, B, or C" << endl;
  switch(int i = cin.get()) {
    case 'A': cout << "Snap" << endl; break;
    case 'B': cout << "Crackle" << endl; break;
    case 'C': cout << "Pop" << endl; break;
    default: cout << "Not A, B or C!" << endl;
  }
} ///:~
```

En el ámbito más interno, se define `p` antes de que acabe el ámbito, de modo que realmente es un gesto inútil (pero demuestra que se puede definir una variable en cualquier sitio). La variable `p` en el ámbito exterior está en la misma situación.

La definición de `i` en la expresión de control del bucle `for` es un ejemplo de que es posible definir una variable exactamente en el punto en el que se necesita (esto sólo se puede hacer en C++). El ámbito de `i` es el ámbito de la expresión controlada por el bucle `for`, de modo que se puede re-utilizar `i` en el siguiente bucle `for`. Se trata de un modismo conveniente y común en C++; `i` es el nombre habitual para el contador de un `for` y así no hay que inventar nombres nuevos.

A pesar de que el ejemplo también muestra variables definidas dentro de las

sentencias `while`, `if` y `switch`, este tipo de definiciones es menos común que las de expresiones `for`, quizás debido a que la sintaxis es más restrictiva. Por ejemplo, no se puede tener ningún paréntesis. Es decir, que no se puede indicar:

```
while((char c = cin.get()) != 'q')
```

Añadir los paréntesis extra parecería una acción inocente y útil, y debido a que no se pueden utilizar, los resultados no son los esperados. El problema ocurre porque `!=` tiene orden de precedencia mayor que `=`, de modo que el `char c` acaba conteniendo un bool convertido a char. Cuando se muestra, en muchos terminales se vería el carácter de la cara sonriente.

En general, se puede considerar la posibilidad de definir variables dentro de las sentencias `while`, `if` y `switch` por completitud, pero el único lugar donde se debería utilizar este tipo de definición de variables es en el bucle `for` (dónde usted las utilizará más a menudo).

3.6. Especificar la ubicación del espacio de almacenamiento

Al crear una variable, hay varias alternativas para especificar la vida de dicha variable, la forma en que se decide la ubicación para esa variable y cómo la tratará el compilador.

3.6.1. Variables globales

Las variables globales se definen fuera de todos los cuerpos de las funciones y están disponibles para todo el programa (incluso el código de otros ficheros). Las variables globales no están afectadas por ámbitos y están siempre disponibles (es decir, la vida de una variable global dura hasta la finalización del programa). Si la existencia de una variable global en un fichero se declara usando la palabra reservada `extern` en otro fichero, la información está disponible para su utilización en el segundo fichero. A continuación, un ejemplo del uso de variables globales:

```
//: C03:Global.cpp
//{L} Global2
// Demonstration of global variables
#include <iostream>
using namespace std;

int globe;
void func();
int main() {
  globe = 12;
  cout << globe << endl;
  func(); // Modifies globe
  cout << globe << endl;
} ///:~
```

Y el fichero que accede a `globe` como un `extern`:

```
//: C03:Global2.cpp {O}
// Accessing external global variables
extern int globe;
// (The linker resolves the reference)
void func() {
  globe = 47;
} ///:~
```

El espacio para la variable `globe` se crea mediante la definición en `Global.cpp`, y esa misma variable es accedida por el código de `Global2.cpp`. Ya que el código de `Global2.cpp` se compila separado del código de `Global.cpp`, se debe informar al compilador de que la variable existe en otro sitio mediante la declaración

```
extern int globe;
```

Cuando ejecute el programa, observará que la llamada `fun()` afecta efectivamente a la única instancia global de `globe`.

En `Global.cpp`, se puede ver el comentario con una marca especial (que es diseño mío):

```
//{L} Global2
```

Eso indica que para crear el programa final, el fichero objeto con el nombre `Global2` debe estar enlazado (no hay extensión ya que los nombres de las extensiones de los ficheros objeto difieren de un sistema a otro). En `Global2.cpp`, la primera línea tiene otra marca especial `{O}`, que significa «No intentar crear un ejecutable de este fichero, se compila para que pueda enlazarse con otro fichero». El programa `ExtractCode.cpp` en el Volumen 2 de este libro (que se puede descargar de www.BruceEckel.com) lee estas marcas y crea el `makefile` apropiado de modo que todo se compila correctamente (aprenderá sobre makefiles al final de este capítulo).

3.6.2. Variables locales

Las variables locales son las que se encuentran dentro de un ámbito; son «locales» a una función. A menudo se las llama variables automáticas porque aparecen automáticamente cuando se entra en un ámbito y desaparecen cuando el ámbito se acaba. La palabra reservada `auto` lo enfatiza, pero las variables locales son `auto` por defecto, de modo que nunca se necesita realmente declarar algo como `auto`.

Variables registro

Una variable registro es un tipo de variable local. La palabra reservada `register` indica al compilador «Haz que los accesos a esta variable sean lo más rápidos posible». Aumentar la velocidad de acceso depende de la implementación, pero, tal como sugiere el nombre, a menudo se hace situando la variable en un registro del microprocesador. No hay garantía alguna de que la variable pueda ser ubicada en un registro y tampoco de que la velocidad de acceso aumente. Es una ayuda para el compilador.

Hay restricciones a la hora de utilizar variables registro. No se puede consular o calcular la dirección de una variable registro. Una variable registro sólo se puede

declarar dentro de un bloque (no se pueden tener variables de registro globales o estáticas). De todos modos, se pueden utilizar como un argumento formal en una función (es decir, en la lista de argumentos).

En general, no se debería intentar influir sobre el optimizador del compilador, ya que probablemente él hará mejor el trabajo de lo que lo pueda hacer usted. Por eso, es mejor evitar el uso de la palabra reservada `register`.

3.6.3. Static

La palabra reservada `static` tiene varios significados. Normalmente, las variables definidas localmente a una función desaparecen al final del ámbito de ésta. Cuando se llama de nuevo a la función, el espacio de las variables se vuelve a pedir y las variables son re-inicializadas. Si se desea que el valor se conserve durante la vida de un programa, puede definir una variable local de una función como `static` y darle un valor inicial. La inicialización se realiza sólo la primera vez que se llama a la función, y la información se conserva entre invocaciones sucesivas de la función. De este modo, una función puede «recordar» cierta información entre una llamada y otra.

Puede surgir la duda de porqué no utilizar una variable global en este caso. El encanto de una variable `static` es que no está disponible fuera del ámbito de la función, de modo que no se puede modificar accidentalmente. Esto facilita la localización de errores.

A continuación, un ejemplo del uso de variables `static`:

```
//: C03:Static.cpp
// Using a static variable in a function
#include <iostream>
using namespace std;

void func() {
  static int i = 0;
  cout << "i = " << ++i << endl;
}

int main() {
  for(int x = 0; x < 10; x++)
    func();
} ///:~
```

Cada vez que se llama a `func()` dentro del bucle, se imprime un valor diferente. Si no se utilizara la palabra reservada `static`, el valor mostrado sería siempre `1`.

El segundo significado de `static` está relacionado con el primero en el sentido de que «no está disponible fuera de cierto ámbito». Cuando se aplica `static` al nombre de una función o de una variable que está fuera de todas las funciones, significa «Este nombre no está disponible fuera de este fichero». El nombre de la función o de la variable es local al fichero; decimos que tiene ámbito de fichero. Como demostración, al compilar y enlazar los dos ficheros siguientes aparece un error en el enlazado:

```
//: C03:FileStatic.cpp
// File scope demonstration. Compiling and
```

```
// linking this file with FileStatic2.cpp
// will cause a linker error

// File scope means only available in this file:
static int fs;

int main() {
  fs = 1;
} ///:~
```

Aunque la variable `fs` está destinada a existir como un `extern` en el siguiente fichero, el enlazador no la encontraría porque ha sido declarada `static` en `FileStatic.cpp`.

```
//: C03:FileStatic2.cpp {O}
// Trying to reference fs
extern int fs;
void func() {
  fs = 100;
} ///:~
```

El especificador `static` también se puede usar dentro de una clase. Esta explicación se dará más adelante en este libro, cuando aprenda a crear clases.

3.6.4. extern

La palabra reservada `extern` ya ha sido brevemente descripta. Le dice al compilador que una variable o una función existe, incluso si el compilado aún no la ha visto en el fichero que está siendo compilado en ese momento. Esta variable o función puede definirse en otro fichero o más abajo en el fichero actual. A modo de ejemplo:

```
//: C03:Forward.cpp
// Forward function & data declarations
#include <iostream>
using namespace std;

// This is not actually external, but the
// compiler must be told it exists somewhere:
extern int i;
extern void func();
int main() {
  i = 0;
  func();
}
int i; // The data definition
void func() {
  i++;
  cout << i;
} ///:~
```

Cuando el compilador encuentra la declaración `extern int i` sabe que la definición para `i` debe existir en algún sitio como una variable global. Cuando el compilador alcanza la definición de `i`, ninguna otra declaración es visible, de modo que sabe que ha encontrado la misma `i` declarada anteriormente en el fichero. Si se hubiera definido `i` como `static`, estaría indicando al compilador que `i` se define globalmente (por `extern`), pero también que tiene el ámbito de fichero (por `static`), de modo que el compilador generará un error.

Enlazado

Para comprender el comportamiento de los programas C y C++, es necesario saber sobre *enlazado*. En un programa en ejecución, un identificador se representa con espacio en memoria que aloja una variable o un cuerpo de función compilada. El enlazado describe este espacio tal como lo ve el enlazador. Hay dos formas de enlazado: *enlace interno* y *enlace externo*.

Enlace interno significa que el espacio se pide para representar el identificador sólo durante la compilación del fichero. Otros ficheros pueden utilizar el mismo nombre de identificador con un enlace interno, o para una variable global, y el enlazador no encontraría conflictos - se pide un espacio separado para cada identificador. El enlace interno se especifica mediante la palabra reservada `static` en C y C++.

Enlace externo significa que se pide sólo un espacio para representar el identificador para todos los ficheros que se estén compilando. El espacio se pide una vez, y el enlazador debe resolver todas las demás referencias a esa ubicación. Las variables globales y los nombres de función tienen enlace externo. Son accesibles desde otros ficheros declarándolas con la palabra reservada `extern`. Por defecto, las variables definidas fuera de todas las funciones (con la excepción de `const` en C++) y las definiciones de las funciones implican enlace externo. Se pueden forzar específicamente a tener enlace interno utilizando `static`. Se puede establecer explícitamente que un identificador tiene enlace externo definiéndolo como `extern`. No es necesario definir una variable o una función como `extern` en C, pero a veces es necesario para `const` en C++.

Las variables automáticas (locales) existen sólo temporalmente, en la pila, mientras se está ejecutando una función. El enlazador no entiende de variables automáticas, de modo que no tienen enlazado.

3.6.5. Constantes

En el antiguo C (pre-Estándar), si se deseaba crear una constante, se debía utilizar el preprocesador:

```
#define PI 3.14159
```

En cualquier sitio en el que utilizase PI, el preprocesador lo substituía por el valor 3.14159 (aún se puede utilizar este método en C y C++).

Cuando se utiliza el preprocesador para crear constantes, su control queda fuera del ámbito del compilador. No existe ninguna comprobación de tipo y no se puede obtener la dirección de PI (de modo que no se puede pasar un puntero o una referencia a PI). PI no puede ser una variable de un tipo definido por el usuario. El significado de PI dura desde el punto en que es definida, hasta el final del fichero; el preprocesador no entiende de ámbitos.

C++ introduce el concepto de constantes con nombre que es lo mismo que variable, excepto que su valor no puede cambiar. El modificador `const` le indica al compilador que el nombre representa una constante. Cualquier tipo de datos predefinido o definido por el usuario, puede ser definido como `const`. Si se define algo como `const` y luego se intenta modificar, el compilador generará un error.

Se debe especificar el tipo de un `const`, de este modo:

```
const int x = 10;
```

En C y C++ Estándar, se puede usar una constante en una lista de argumentos, incluso si el argumento que ocupa es un puntero o una referencia (p.e, se puede obtener la dirección de una constante). Las constantes tienen ámbito, al igual que una variable ordinaria, de modo que se puede «esconder» una constante dentro de una función y estar seguro de que ese nombre no afectará al resto del programa.

`const` ha sido tomado de C++ e incorporado al C Estándar pero un modo un poco distinto. En C, el compilador trata a `const` del mismo modo que a una variable que tuviera asociado una etiqueta que dice «No me cambies». Cuando se define un `const` en C, el compilador pide espacio para él, de modo que si se define más de un `const` con el mismo nombre en dos ficheros distintos (o se ubica la definición en un fichero de cabeceras), el enlazador generará mensajes de error sobre del conflicto. El concepto de `const` en C es diferente de su utilización en C++ (en resumen, es más bonito en C++).

Valores constantes

En C++, una constante debe tener siempre un valor inicial (En C, eso no es cierto). Los valores de las constantes para tipos predefinidos se expresan en decimal, octal, hexadecimal, o números con punto flotante (desgraciadamente, no se consideró que los binarios fuesen importantes), o como caracteres.

A falta de cualquier otra pista, el compilador assume que el valor de una constante es un número decimal. Los números 47, 0 y 1101 se tratan como números decimales.

Un valor constante con un cero al principio se trata como un número octal (base 8). Los números con base 8 pueden contener únicamente dígitos del 0 al 7; el compilador interpreta otros dígitos como un error. Un número octal legítimo es 017 (15 en base 10).

Un valor constante con `0x` al principio se trata como un número hexadecimal (base 16). Los números con base 16 pueden contener dígitos del 0 al 9 y letras de la a a la f o A a F. Un número hexadecimal legítimo es 0x1fe (510 en base 10).

Los números en punto flotante pueden contener comas decimales y potencias exponenciales (representadas mediante `e`, lo que significa «10 elevado a»). Tanto el punto decimal como la `e` son opcionales. Si se asigna una constante a una variable de punto flotante, el compilador tomará el valor de la constante y la convertirá a un número en punto flotante (este proceso es una forma de lo que se conoce como conversión implícita de tipo). De todos modos, es una buena idea el usar el punto decimal o una `e` para recordar al lector que está utilizando un número en punto flotante; algunos compiladores incluso necesitan esta pista.

Algunos valores válidos para una constante en punto flotante son: 1e4, 1.0001, 47.0, 0.0 y 1.159e-77. Se pueden añadir sufijos para forzar el tipo de número de punto flotante: `f` o `F` fuerza que sea float, `L` o `l` fuerza que sea un long double; de lo

contrario, el número será un double.

Las constantes de tipo char son caracteres entre comillas simples, tales como: `'A'`, `'o'`, `''`. Fíjese en que hay una gran diferencia entre el carácter `'o'` (ASCII 96) y el valor 0. Los caracteres especiales se representan con la «barra invertida»: `'\n'` (nueva línea), `'\t'` (tabulación), `'\\'` (barra invertida), `'\r'` (retorno de carro), `'\"'` (comilla doble), `'\''` (comilla simple), etc. Incluso se puede expresar constantes de tipo char en octal: `'\17'` o hexadecimal: `'\xff'`.

3.6.6. Volatile

Mientras que el calificador `const` indica al compilador «Esto nunca cambia» (lo que permite al compilador realizar optimizaciones extra), el calificador `volatile` dice al compilador «Nunca se sabe cuando cambiará esto», y evita que el compilador realice optimizaciones basadas en la estabilidad de esa variable. Se utiliza esta palabra reservada cuando se lee algún valor fuera del control del código, algo así como un registro en un hardware de comunicación. Una variable `volatile` se lee siempre que su valor es requerido, incluso si se ha leído en la línea anterior.

Un caso especial de espacio que está «fuera del control del código» es en un programa multi-hilo. Si está comprobando una bandera particular que puede ser modificada por otro hilo o proceso, esta bandera debería ser `volatile` de modo que el compilador no asuma que puede optimizar múltiples lecturas de la bandera.

Fíjese en que `volatile` puede no tener efecto cuando el compilador no está optimizando, pero puede prevenir errores críticos cuando se comienza a optimizar el código (que es cuando el compilador empezará a buscar lecturas redundantes).

Las palabras reservadas `const` y `volatile` se verán con más detalle en un capítulo posterior.

3.7. Los operadores y su uso

Esta sección cubre todos los operadores de C y C++.

Todos los operadores producen un valor a partir de sus operandos. Esta operación se efectúa sin modificar los operandos, excepto con los operadores de asignación, incremento y decremento. El hecho de modificar un operando se denomina *efecto colateral*. El uso más común de los operadores que modifican sus operandos es producir el efecto colateral, pero se debería tener en cuenta que el valor producido está disponible para su uso al igual que el de los operadores sin efectos colaterales.

3.7.1. Asignación

La asignación se realiza mediante el operador =. Eso significa «Toma el valor de la derecha (a menudo llamado *rvalue*) y cópialo en la variable de la izquierda (a menudo llamado *lvalue*).» Un *rvalue* es cualquier constante, variable o expresión que pueda producir un valor, pero un *lvalue* debe ser una variable con un nombre distintivo y único (esto quiere decir que debe haber un espacio físico dónde guardar la información). De hecho, se puede asignar el valor de una constante a una variable (`A = 4;`), pero no se puede asignar nada a una constante - es decir, una constante no puede ser un *lvalue* (no se puede escribir `4 = A;`).

3.7.2. Operadores matemáticos

Los operadores matemáticos básicos son los mismos que están disponibles en la mayoría de los lenguajes de programación: adición (+), substracción (−), división (/), multiplicación (*), y módulo (%; que produce el resto de una división entera). La división entera trunca el resultado (no lo redondea). El operador módulo no se puede utilizar con números con punto flotante.

C y C++ también utilizan notaciones abreviadas para efectuar una operación y una asignación al mismo tiempo. Esto se denota por un operador seguido de un signo igual, y se puede aplicar a todos los operadores del lenguaje (siempre que tenga sentido). Por ejemplo, para añadir `4` a la variable `x` y asignar `x` al resultado, se escribe: `x += 4;`.

Este ejemplo muestra el uso de los operadores matemáticos:

```
//: C03:Mathops.cpp
// Mathematical operators
#include <iostream>
using namespace std;

// A macro to display a string and a value.
#define PRINT(STR, VAR) \
  cout << STR " = " << VAR << endl

int main() {
  int i, j, k;
  float u, v, w;  // Applies to doubles, too
  cout << "enter an integer: ";
  cin >> j;
  cout << "enter another integer: ";
  cin >> k;
  PRINT("j",j);  PRINT("k",k);
  i = j + k; PRINT("j + k",i);
  i = j - k; PRINT("j - k",i);
  i = k / j; PRINT("k / j",i);
  i = k * j; PRINT("k * j",i);
  i = k % j; PRINT("k % j",i);
  // The following only works with integers:
  j %= k; PRINT("j %= k", j);
  cout << "Enter a floating-point number: ";
  cin >> v;
  cout << "Enter another floating-point number:";
  cin >> w;
  PRINT("v",v); PRINT("w",w);
  u = v + w; PRINT("v + w", u);
  u = v - w; PRINT("v - w", u);
  u = v * w; PRINT("v * w", u);
  u = v / w; PRINT("v / w", u);
  // The following works for ints, chars,
  // and doubles too:
  PRINT("u", u); PRINT("v", v);
  u += v; PRINT("u += v", u);
  u -= v; PRINT("u -= v", u);
  u *= v; PRINT("u *= v", u);
  u /= v; PRINT("u /= v", u);
} ///:~
```

Los *rvalues* de todas las asignaciones pueden ser, por supuesto, mucho mas complejos.

Introducción a las macros del preprocesador

Observe el uso de la macro `PRINT()` para ahorrar líneas (y errores de sintaxis!). Las macros de preprocesador se nombran tradicionalmente con todas sus letras en mayúsculas para que sea fácil distinguirlas - aprenderá más adelante que las macros pueden ser peligrosas (y también pueden ser muy útiles).

Los argumentos de de la lista entre paréntesis que sigue al nombre de la macro son sustituidos en todo el código que sigue al paréntesis de cierre. El preprocesador elimina el nombre `PRINT` y sustituye el código donde se invoca la macro, de modo que el compilador no puede generar ningún mensaje de error al utilizar el nombre de la macro, y no realiza ninguna comprobación de sintaxis sobre los argumentos (esto lo último puede ser beneficioso, como se muestra en las macros de depuración al final del capítulo).

3.7.3. Operadores relacionales

Los operadores relacionales establecen una relación entre el valor de los operandos. Producen un valor booleano (especificado con la palabra reservada bool en C++) `true` si la relación es verdadera, y `false` si la relación es falsa. Los operadores relacionales son: menor que (<), mayor que (>), menor o igual a (<=), mayor o igual a (>=), equivalente (==), y distinto (!=). Se pueden utilizar con todos los tipos de datos predefinidos en C y C++. Se pueden dar definiciones especiales para tipos definidos por el usuario en C++ (aprenderá más sobre el tema en el Capítulo 12, que cubre la sobrecarga de operadores).

3.7.4. Operadores lógicos

Los operadores lógicos *and* (`&&`) y *or* (`||`) producen `true` o `false` basándose en la relación lógica de sus argumentos. Recuerde que en C y C++, una condición es cierta si tiene un valor diferente de cero, y falsa si vale cero. Si se imprime un bool, por lo general verá un `1` para `true` y `0` para `false`.

Este ejemplo utiliza los operadores relacionales y lógicos:

```
//: C03:Boolean.cpp
// Relational and logical operators.
#include <iostream>
using namespace std;

int main() {
  int i,j;
  cout << "Enter an integer: ";
  cin >> i;
  cout << "Enter another integer: ";
  cin >> j;
  cout << "i > j is " << (i > j) << endl;
  cout << "i < j is " << (i < j) << endl;
  cout << "i >= j is " << (i >= j) << endl;
  cout << "i <= j is " << (i <= j) << endl;
  cout << "i == j is " << (i == j) << endl;
  cout << "i != j is " << (i != j) << endl;
```

```
  cout << "i && j is " << (i && j) << endl;
  cout << "i || j is " << (i || j) << endl;
  cout << " (i < 10) && (j < 10) is "
       << ((i < 10) && (j < 10))  << endl;
} ///:~
```

Se puede reemplazar la definición de int con float o double en el programa anterior. De todos modos, dese cuenta de que la comparación de un número en punto flotante con el valor cero es estricta; un número que es la fracción más pequeña diferente de otro número aún se considera «distinto de». Un número en punto flotante que es poca mayor que cero se considera verdadero.

3.7.5. Operadores para bits

Los operadores de bits permiten manipular bits individuales y dar como salida un número (ya que los valores con punto flotante utilizan un formato interno especial, los operadores de bitS sólo funcionan con tipos enteros: char, int y long). Los operadores de bitS efectúan álgebra booleana en los bits correspondientes de los argumentos para producir el resultado.

El operador *and* (`&`) para bits produce uno en la salida si ambos bits de entrada valen uno; de otro modo produce un cero. El operador *or* (`|`) para bits produce un uno en la salida si cualquiera de los dos valores de entrada vale uno, y produce un cero sólo si ambos valores de entrada son cero. El operador *or exclusivo* o *xor* (`^`) para bits produce uno en la salida si uno de los valores de entrada es uno, pero no ambos. El operador *not* (`~`) para bits (también llamado operador de *complemento a uno*) es un operador unario - toma un único argumento (todos los demás operadores son binarios). El operador *not* para bits produce el valor contrario a la entrada - uno si el bit de entrada es cero, y cero si el bit de entrada es uno.

Los operadores de bits pueden combinarse con el signo = para unir la operación y la asignación: `&=`, `|=`, y `^=` son todas operaciones legales (dado que `~` es un operador unario no puede combinarse con el signo =).

3.7.6. Operadores de desplazamiento

Los operadores de desplazamiento también manipulan bits. El operador de desplazamiento a izquierda (<<) produce el desplazamiento del operando que aparece a la izquierda del operador tantos bits a la izquierda como indique el número a la derecha del operador. El operador de desplazamiento a derecha (>>) produce el desplazamiento del operando de la izquierda hacia la derecha tantos bits como indique el número a la derecha del operador. Si el valor que sigue al operador de desplazamiento es mayor que el número de bits del lado izquierdo, el resultado es indefinido. Si el operando de la izquierda no tiene signo, el desplazamiento a derecha es un desplazamiento lógico de modo que los bits del principio se rellenan con ceros. Si el operando de la izquierda tiene signo, el desplazamiento derecho puede ser un desplazamiento lógico (es decir, significa que el comportamiento es indeterminado).

Los desplazamientos pueden combinarse con el signo igual (<<= y >>=). El *lvalue* se reemplaza por *lvalue* desplazado por el *rvalue*.

Lo que sigue a continuación es un ejemplo que demuestra el uso de todos los operadores que involucran bits. Primero, una función de propósito general que imprime un byte en formato binario, creada para que se pueda reutilizar fácilmente. El

fichero de cabecera declara la función:

```
//: C03:printBinary.h
// Display a byte in binary
void printBinary(const unsigned char val);
///:~
```

A continuación la implementación de la función:

```
//: C03:printBinary.cpp {O}
#include <iostream>
void printBinary(const unsigned char val) {
  for(int i = 7; i >= 0; i--)
    if(val & (1 << i))
      std::cout << "1";
    else
      std::cout << "0";
} ///:~
```

La función `printBinary()` toma un único byte y lo muestra bit a bit. La expresión:

```
(1 << i)
```

produce un uno en cada posición sucesiva de bit; en binario: `00000001`, `00000010`, etc. Si se hace *and* a este bit con `val` y el resultado es diferente de cero, significa que había un uno en esa posición de `val`.

Finalmente, se utiliza la función en el ejemplo que muestra los operadores de manipulación de bits:

```
//: C03:Bitwise.cpp
//{L} printBinary
// Demonstration of bit manipulation
#include "printBinary.h"
#include <iostream>
using namespace std;

// A macro to save typing:
#define PR(STR, EXPR) \
  cout << STR; printBinary(EXPR); cout << endl;

int main() {
  unsigned int getval;
  unsigned char a, b;
  cout << "Enter a number between 0 and 255: ";
  cin >> getval; a = getval;
  PR("a in binary: ", a);
  cout << "Enter a number between 0 and 255: ";
  cin >> getval; b = getval;
  PR("b in binary: ", b);
  PR("a | b = ", a | b);
  PR("a & b = ", a & b);
```

```
  PR("a ^ b = ", a ^ b);
  PR("~a = ", ~a);
  PR("~b = ", ~b);
  // An interesting bit pattern:
  unsigned char c = 0x5A;
  PR("c in binary: ", c);
  a |= c;
  PR("a |= c; a = ", a);
  b &= c;
  PR("b &= c; b = ", b);
  b ^= a;
  PR("b ^= a; b = ", b);
} ///:~
```

Una vez más, se usa una macro de preprocesador para ahorrar líneas. Imprime la cadena elegida, luego la representación binaria de una expresión, y luego un salto de línea.

En `main()`, las variables son unsigned. Esto es porque, en general, no se desean signos cuando se trabaja con bytes. Se debe utilizar un int en lugar de un char para `getval` porque de otro modo la sentencia `cin >>` trataría el primer dígito como un carácter. Asignando `getval` a `a` y `b`, se convierte el valor a un solo byte (truncándolo).

Los operadores << y >> proporcionan un comportamiento de desplazamiento de bits, pero cuando desplazan bits que están al final del número, estos bits se pierden (comúnmente se dice que se caen en el mítico *cubo de bits*, el lugar donde acaban los bits descartados, presumiblemente para que puedan ser utilizados...). Cuando se manipulan bits también se pueden realizar *rotaciones*; es decir, que los bits que salen de uno de los extremos se pueden insertar por el otro extremo, como si estuviesen rotando en un bucle. Aunque la mayoría de los procesadores de ordenadores ofrecen un comando de rotación a nivel máquina (se puede ver en el lenguaje ensamblador de ese procesador), no hay un soporte directo para *rotate* en C o C++. Se supone que a los diseñadores de C les pareció justificado el hecho de prescindir de *rotate* (en pro, como dijeron, de un lenguaje minimalista) ya que el programador se puede construir su propio comando *rotate*. Por ejemplo, a continuación hay funciones para realizar rotaciones a izquierda y derecha:

```
//: C03:Rotation.cpp {O}
// Perform left and right rotations

unsigned char rol(unsigned char val) {
  int highbit;
  if(val & 0x80) // 0x80 is the high bit only
    highbit = 1;
  else
    highbit = 0;
  // Left shift (bottom bit becomes 0):
  val <<= 1;
  // Rotate the high bit onto the bottom:
  val |= highbit;
  return val;
}

unsigned char ror(unsigned char val) {
  int lowbit;
```

```
  if(val & 1) // Check the low bit
    lowbit = 1;
  else
    lowbit = 0;
  val >>= 1; // Right shift by one position
  // Rotate the low bit onto the top:
  val |= (lowbit << 7);
  return val;
} ///:~
```

Al intentar utilizar estas funciones en `Bitwise.cpp`, advierta que las definiciones (o cuando menos las declaraciones) de `rol()` y `ror()` deben ser vistas por el compilador en `Bitwise.cpp` antes de que se puedan utilizar.

Las funciones de tratamiento de bits son por lo general extremadamente eficientes ya que traducen directamente las sentencias a lenguaje ensamblador. A veces una sentencia de C o C++ generará una única línea de código ensamblador.

3.7.7. Operadores unarios

El *not* no es el único operador de bits que toma sólo un argumento. Su compañero, el *not* lógico (`!`), toma un valor `true` y produce un valor `false`. El menos unario (`-`) y el más unario (`+`) son los mismos operadores que los binarios menos y más; el compilador deduce que uso se le pretende dar por el modo en el que se escribe la expresión. De hecho, la sentencia:

```
x = -a;
```

tiene un significado obvio. El compilador puede deducir:

```
x = a * -b;
```

pero el lector se puede confundir, de modo que es más seguro escribir:

```
x = a * (-b);
```

El menos unario produce el valor negativo. El más unario ofrece simetría con el menos unario, aunque en realidad no hace nada.

Los operadores de incremento y decremento (`++` y `--`) se comentaron ya en este capítulo. Son los únicos operadores, además de los que involucran asignación, que tienen efectos colaterales. Estos operadores incrementan o decrementan la variable en una unidad, aunque «unidad» puede tener diferentes significados dependiendo del tipo de dato - esto es especialmente importante en el caso de los punteros.

Los últimos operadores unarios son dirección-de (`&`), indirección (`*` y `->`), los operadores de moldeado en C y C++, y `new` y `delete` en C++. La dirección-de y la indirección se utilizan con los punteros, descriptos en este capítulo. El moldeado se describe mas adelante en este capítulo, y `new` y `delete` se introducen en el Capítulo 4.

3.7.8. El operador ternario

El `if-else` ternario es inusual porque tiene tres operandos. Realmente es un operador porque produce un valor, al contrario de la sentencia ordinaria `if-else`. Consta de tres expresiones: si la primera expresión (seguida de un `?`) se evalúa como cierto, se devuelve el resultado de evaluar la expresión que sigue al `?`. Si la primera expresión es falsa, se ejecuta la tercera expresión (que sigue a `:`) y su resultado se convierte en el valor producido por el operador.

El operador condicional se puede usar por sus efectos colaterales o por el valor que produce. A continuación, un fragmento de código que demuestra ambas cosas:

```
a = --b ? b : (b = -99);
```

Aquí, el condicional produce el *rvalue*. A `a` se le asigna el valor de `b` si el resultado de decrementar `b` es diferente de cero. Si `b` se queda a cero, `a` y `b` son ambas asignadas a -99. `b` siempre se decrementa, pero se asigna a -99 sólo si el decremento provoca que `b` valga 0. Se puede utilizar un sentencia similar sin el `a =` sólo por sus efectos colaterales:

```
--b ? b : (b = -99);
```

Aquí la segunda `b` es superflua, ya que no se utiliza el valor producido por el operador. Se requiere una expresión entre el `?` y `:`. En este caso, la expresión puede ser simplemente una constante, lo que haría que el código se ejecute un poco más rápido.

3.7.9. El operador coma

La coma no se limita a separar nombres de variables en definiciones múltiples, tales como

```
int i, j, k;
```

Por supuesto, también se usa en listas de argumentos de funciones. De todos modos, también se puede utilizar como un operador para separar expresiones - en este caso produce el valor de la última expresión. El resto de expresiones en la lista separada por comas se evalúa sólo por sus efectos colaterales. Este ejemplo incrementa una lista de variables y usa la última como el *rvalue*:

```
//: C03:CommaOperator.cpp
#include <iostream>
using namespace std;
int main() {
  int a = 0, b = 1, c = 2, d = 3, e = 4;
  a = (b++, c++, d++, e++);
  cout << "a = " << a << endl;
  // The parentheses are critical here. Without
  // them, the statement will evaluate to:
  (a = b++), c++, d++, e++;
  cout << "a = " << a << endl;
} ///:~
```

En general, es mejor evitar el uso de la coma para cualquier otra cosa que no sea separar, ya que la gente no está acostumbrada a verla como un operador.

3.7.10. Trampas habituales cuando se usan operadores

Como se ha ilustrado anteriormente, una de las trampas al usar operadores es tratar de trabajar sin paréntesis incluso cuando no se está seguro de la forma en la que se va a evaluar la expresión (consulte su propio manual de C para comprobar el orden de la evaluación de las expresiones).

Otro error extremadamente común se ve a continuación:

```
//: C03:Pitfall.cpp
// Operator mistakes

int main() {
  int a = 1, b = 1;
  while(a = b) {
    // ....
  }
} ///:~
```

La sentencia `a = b` siempre se va a evaluar como cierta cuando `b` es distinta de cero. La variable `a` obtiene el valor de `b`, y el valor de `b` también es producido por el operador =. En general, lo que se pretende es utilizar el operador de equivalencia (== dentro de una sentencia condicional, no la asignación. Esto le ocurre a muchos programadores (de todos modos, algunos compiladores advierten del problema, lo cual es una ayuda).

Un problema similar es usar los operadores *and* y *or* de bits en lugar de sus equivalentes lógicos. Los operadores *and* y *or* de bits usan uno de los caracteres (`&` o `|`), mientras que los operadores lógicos utilizan dos (`&&` y `||`). Al igual que con = y ==, es fácil escribir simplemente un carácter en vez de dos. Una forma muy fácil de recordarlo es que «los bits son mas pequeños, de modo que no necesitan tantos caracteres en sus operadores».

3.7.11. Operadores de moldeado

La palabra molde(*cast*) se usa en el sentido de "colocar dentro de un molde". El compilador cambiará automáticamente un tipo de dato a otro si tiene sentido. De hecho, si se asigna un valor entero a una variable de punto flotante, el compilador llamará secretamente a una función (o más probablemente, insertará código) para convertir el int a un float. El molde permite hacer este tipo de conversión explicita, o forzarla cuando normalmente no pasaría.

Para realizar un molde, se debe situar el tipo deseado (incluyendo todos los modificadores) dentro de paréntesis a la izquierda del valor. Este valor puede ser una variable, una constante, el valor producido por una expresión, o el valor devulto por una función. A continuación, un ejemplo:

```
//: C03:SimpleCast.cpp
int main() {
  int b = 200;
```

```
  unsigned long a = (unsigned long int)b;
} ///:~
```

El moldeado es poderoso, pero puede causar dolores de cabeza porque en algunas situaciones fuerza al compilador a tratar datos como si fuesen (por ejemplo) más largos de lo que realmente son, de modo que ocupará más espacio en memoria; lo que puede afectar a otros datos. Esto ocurre a menudo cuando se moldean punteros, no cuando se hacen moldes simples como los que ha visto anteriormente.

C++ tiene una sintaxis adicional para moldes, que sigue a la sintaxis de llamada a funciones. Esta sintaxis pone los paréntesis alrededor del argumento, como en una llamada a función, en lugar de a los lados del tipo:

```
//: C03:FunctionCallCast.cpp
int main() {
  float a = float(200);
  // This is equivalent to:
  float b = (float)200;
} ///:~
```

Por supuesto, en el caso anterior, en realidad no se necesitaría un molde; simplemente se puede decir `200.f` o `200.0f` (en efecto, eso es típicamente lo que el compilador hará para la expresión anterior). Los moldes normalmente se utilizan con variables, en lugar de con constantes.

3.7.12. Los moldes explícitos de C++

Los moldes se deben utilizar con cuidado, porque lo que está haciendo en realidad es decir al compilador «Olvida la comprobación de tipo - trátalo como si fuese de este otro tipo.» Esto significa, que está introduciendo un agujero en el sistema de tipos de C++ y evitando que el compilador informe de que está haciendo algo erróneo con un tipo. Lo que es peor, el compilador lo cree implícitamente y no realiza ninguna otra comprobación para buscar errores. Una vez ha comenzado a moldear, está expuesto a todo tipo de problemas. De hecho, cualquier programa que utilice muchos moldes se debe revisar con detenimiento, no importa cuanto haya dado por sentado que simplemente «debe» hacerse de esta manera. En general, los moldes deben ser pocos y aislados para solucionar problemas específicos.

Una vez se ha entendido esto y se presente un programa con errores, la primera impresión puede que sea mirar los moldes como si fuesen los culpables. Pero, ¿cómo encontrar los moldes estilo C? Son simplemente nombres de tipos entre paréntesis, y si se empieza a buscar estas cosas descubrirá que a menudo es difícil distinguirlos del resto del código.

El C++ Estándar incluye una sintaxis explícita de molde que se puede utilizar para reemplazar completamente los moldes del estilo antiguo de C (por supuesto, los moldes de estilo C no se pueden prohibir sin romper el código, pero los escritores de compiladores pueden advertir fácilmente acerca de los moldes antiguos). La sintaxis explícita de moldes está pensada para que sea fácil encontrarlos, tal como se puede observar por sus nombres:

Los primeros tres moldes explícitos se describirán completamente en las siguientes secciones, mientras que los últimos se explicarán después de que haya aprendido

`static_cast`	Para moldes que se *comportan bien* o *razonablemente bien*, incluyendo cosas que se podrían hacer sin un molde (como una conversión automática de tipo).
`const_cast`	Para moldear `const` y/o `volatile`
`reinterpret_cast`	Para moldear a un significado completamente diferente. La clave es que se necesitará volver a moldear al tipo original para poderlo usar con seguridad. El tipo al que moldee se usa típicamente sólo para jugar un poco o algún otro propósito misterioso. Éste es el más peligroso de todos los moldes.
`dynamic_cast`	Para realizar un *downcasting* seguro (este molde se describe en el Capítulo 15).

Cuadro 3.2: Moldes explícitos de C++

más en el Capítulo 15.

`static_cast`

El `static_cast` se utiliza para todas las conversiones que están bien definidas. Esto incluye conversiones «seguras» que el compilador permitiría sin utilizar un molde, y conversiones menos seguras que están sin embargo bien definidas. Los tipos de conversiones que cubre `static_cast` incluyen las conversiones típicas sin molde, conversiones de estrechamiento (pérdida de información), forzar una conversión de un void*, conversiones de tipo implícitas, y navegación estática de jerarquías de clases (ya que no se han visto aún clases ni herencias, este último apartado se pospone hasta el Capítulo 15):

```
//: C03:static_cast.cpp
void func(int) {}

int main() {
  int i = 0x7fff; // Max pos value = 32767
  long l;
  float f;
  // (1) Typical castless conversions:
  l = i;
  f = i;
  // Also works:
  l = static_cast<long>(i);
  f = static_cast<float>(i);

  // (2) Narrowing conversions:
  i = l; // May lose digits
  i = f; // May lose info
  // Says "I know," eliminates warnings:
  i = static_cast<int>(l);
  i = static_cast<int>(f);
  char c = static_cast<char>(i);
```

```
  // (3) Forcing a conversion from void* :
  void* vp = &i;
  // Old way produces a dangerous conversion:
  float* fp = (float*)vp;
  // The new way is equally dangerous:
  fp = static_cast<float*>(vp);

  // (4) Implicit type conversions, normally
  // performed by the compiler:
  double d = 0.0;
  int x = d; // Automatic type conversion
  x = static_cast<int>(d); // More explicit
  func(d); // Automatic type conversion
  func(static_cast<int>(d)); // More explicit
} ///:~
```

En la sección (FIXME:xref:1), se pueden ver tipos de conversiones que eran usuales en C, con o sin un molde. Promover un int a long o float no es un problema porque el último puede albergar siempre cualquier valor que un int pueda contener. Aunque es innecesario, se puede utilizar `static_cast` para remarcar estas promociones.

Se muestra en (2) como se convierte al revés. Aquí, se puede perder información porque un int no es tan «ancho» como un long o un float; no aloja números del mismo tamaño. De cualquier modo, este tipo de conversión se llama conversión de estrechamiento. El compilador no impedirá que ocurran, pero normalmente dará una advertencia. Se puede eliminar esta advertencia e indicar que realmente se pretendía esto utilizando un molde.

Tomar el valor de un void* no está permitido en C++ a menos que use un molde (al contrario de C), como se puede ver en (3). Esto es peligroso y requiere que los programadores sepan lo que están haciendo. El `static_cast`, al menos, es mas fácil de localizar que los moldes antiguos cuando se trata de cazar fallos.

La sección (FIXME:xref:4) del programa muestra las conversiones de tipo implícitas que normalmente se realizan de manera automática por el compilador. Son automáticas y no requieren molde, pero el utilizar `static_cast` acentúa dicha acción en caso de que se quiera reflejar claramente qué está ocurriendo, para poder localizarlo después.

const_cast

Si quiere convertir de un `const` a un no-`const` o de un `volatile` a un no-`volatile`, se utiliza `const_cast`. Es la única conversión permitida con `const_cast`; si está involucrada alguna conversión adicional se debe hacer utilizando una expresión separada o se obtendrá un error en tiempo de compilación.

```
//: C03:const_cast.cpp
int main() {
  const int i = 0;
  int* j = (int*)&i; // Deprecated form
  j  = const_cast<int*>(&i); // Preferred
  // Can't do simultaneous additional casting:
//! long* l = const_cast<long*>(&i); // Error
  volatile int k = 0;
  int* u = const_cast<int*>(&k);
```

```
} ///:~
```

Si toma la dirección de un objeto `const`, produce un puntero a `const`, éste no se puede asignar a un puntero que no sea `const` sin un molde. El molde al estilo antiguo lo puede hacer, pero el `const_cast` es el más apropiado en este caso. Lo mismo ocurre con `volatile`.

reinterpret_cast

Este es el menos seguro de los mecanismos de molde, y el más susceptible de crear fallos. Un `reinterpret_cast` supone que un objeto es un patrón de bits que se puede tratar (para algún oscuro propósito) como si fuese de un tipo totalmente distinto. Ese es el jugueteo de bits a bajo nivel por el cual C es famoso. Prácticamente siempre necesitará hacer `reinterpret_cast` para volver al tipo original (o de lo contrario tratar a la variable como su tipo original) antes de hacer nada más con ella.

```
//: C03:reinterpret_cast.cpp
#include <iostream>
using namespace std;
const int sz = 100;

struct X { int a[sz]; };

void print(X* x) {
  for(int i = 0; i < sz; i++)
    cout << x->a[i] << ' ';
  cout << endl << "--------------------" << endl;
}

int main() {
  X x;
  print(&x);
  int* xp = reinterpret_cast<int*>(&x);
  for(int* i = xp; i < xp + sz; i++)
    *i = 0;
  // Can't use xp as an X* at this point
  // unless you cast it back:
  print(reinterpret_cast<X*>(xp));
  // In this example, you can also just use
  // the original identifier:
  print(&x);
} ///:~
```

En este ejemplo, `struct X` contiene un array de int, pero cuando se crea uno en la pila como en `X x`, los valores de cada uno de los ints tienen basura (esto se demuestra utilizando la función `print()` para mostrar los contenidos de `struct`). Para inicializarlas, la dirección del X se toma y se moldea a un puntero int, que es luego iterado a través del array para inicializar cada int a cero. Fíjese como el límite superior de `i` se calcula «añadiendo» `sz` a `xp`; el compilador sabe que lo que usted quiere realmente son las direcciones de sz mayores que `xp` y él realiza el cálculo aritmético por usted. FIXME(Comprobar lo que dice este párrafo de acuerdo con el código)

La idea del uso de `reinterpret_cast` es que cuando se utiliza, lo que se obtiene es tan extraño que no se puede utilizar para los propósitos del tipo original, a menos que se vuelva a moldear. Aquí, vemos el molde otra vez a X* en la llamada a `print()`, pero por supuesto, dado que tiene el identificador original también se puede utilizar. Pero `xp` sólo es útil como un int*, lo que es verdaderamente una «reinterpretación» del X original.

Un `reinterpret_cast` a menudo indica una programación desaconsejada y/o no portable, pero está disponible si decide que lo necesita.

3.7.13. `sizeof` - un operador en si mismo

El operador `sizeof` es independiente porque satisface una necesidad inusual. `sizeof` proporciona información acerca de la cantidad de memoria ocupada por los elementos de datos. Como se ha indicado antes en este capítulo, `sizeof` indica el número de bytes utilizado por cualquier variable particular. También puede dar el tamaño de un tipo de datos (sin necesidad de un nombre de variable):

```
//: C03:sizeof.cpp
#include <iostream>
using namespace std;
int main() {
  cout << "sizeof(double) = " << sizeof(double);
  cout << ", sizeof(char) = " << sizeof(char);
} ///:~
```

Por definición, el `sizeof` de cualquier tipo de char (signed, unsigned o simple) es siempre uno, sin tener en cuenta que el almacenamiento subyacente para un char es realmente un byte. Para todos los demás tipos, el resultado es el tamaño en bytes.

Tenga en cuenta que `sizeof` es un operador, no una función. Si lo aplica a un tipo, se debe utilizar con la forma entre paréntesis mostrada anteriormente, pero si se aplica a una variable se puede utilizar sin paréntesis:

```
//: C03:sizeofOperator.cpp
int main() {
  int x;
  int i = sizeof x;
} ///:~
```

`sizeof` también puede informar de los tamaños de tipos definidos por el usuario. Se utilizará más adelante en el libro.

3.7.14. La palabra reservada `asm`

Este es un mecanismo de escape que permite escribir código ensamblador para el hardware dentro de un programa en C++. A menudo es capaz de referenciar variables C++ dentro del código ensamblador, lo que significa que se puede comunicar fácilmente con el código C++ y limitar el código ensamblador a lo necesario para ajustes eficientes o para utilizar instrucciones especiales del procesador. La sintaxis exacta que se debe usar cuando se escribe en lenguaje ensamblador es dependiente

del compilador y se puede encontrar en la documentación del compilador.

3.7.15. Operadores explícitos

Son palabras reservadas para los operadores lógicos y binarios. Los programadores de fuera de los USA sin teclados con caracteres tales como &, |, ^, y demás, estaban forzados a utilizar horribles *trígrafos*, que no sólo eran insoportable de escribir, además eran difíciles de leer. Esto se ha paliado en C++ con palabras reservadas adicionales:

Palabra reservada	**Significado**
`and`	&& («y» lógica)
`or`	|| («o» lógica)
`not`	! (negación lógica)
`not_eq`	!= (no-equivalencia lógica)
`bitand`	& (and para bits)
`and_eq`	&= (asignación-and para bits)
`bitor`	| (or para bits)
`or_eq`	!= (asignación-or para bits)
`xor`	^ («o» exclusiva para bits)
`xor_equ`	^= (asignación xor para bits)
`compl`	~ (complemento binario)

Cuadro 3.3: Nuevas palabras reservadas para operadores booleanos

Si el compilador obedece al Estándar C++, soportará estas palabras reservadas.

3.8. Creación de tipos compuestos

Los tipos de datos fundamentales y sus variantes son esenciales, pero más bien primitivos. C y C++ incorporan herramientas que permiten construir tipos de datos más sofisticados a partir de los tipos de datos fundamentales. Como se verá, el más importante de estos es `struct`, que es el fundamento para las `class` en C++. Sin embargo, la manera más simple de crear tipos más sofisticados es simplemente poniendo un alias a otro nombre mediante `typedef`.

3.8.1. Creación de alias usando `typedef`

Esta palabra reservada promete más de lo que da: `typedef` sugiere «definición de tipo» cuando «alias» habría sido probablemente una descripción más acertada, ya que eso es lo que hace realmente. La sintaxis es:

`typedef` descripción-de-tipo-existente nombre-alias

La gente a menudo utiliza `typedef` cuando los tipos de datos se vuelven complicados, simplemente para evitar escribir más de lo necesario. A continuación, una forma común de utilizar typedef:

```
typedef unsigned long ulong;
```

Ahora si pone ulong, el compilador sabe que se está refiriendo a unsigned long. Puede pensar que esto se puede lograr fácilmente utilizando sustitución en el pre-

procesador, pero hay situaciones en las cuales el compilador debe estar advertido de que está tratando un nombre como si fuese un tipo, y por eso `typedef` es esencial.

```
int* x, y;
```

Esto genera en realidad un int* que es `x`, y un int (no un int*) que es `y`. Esto significa que el * añade a la derecha, no a la izquierda. Pero, si utiliza un `typedef`:

```
typedef int* IntPtr;
IntPtr x, y;
```

Entonces ambos, `x` e `y` son del tipo int*.

Se puede discutir sobre ello y decir que es más explícito y por consiguiente mas legible evitar `typedef`s para los tipos primitivos, y de hecho los programas se vuelven difíciles de leer cuando se utilizan demasiados `typedef`s. De todos modos, los `typedef`s se vuelven especialmente importantes en C cuando se utilizan con `struct`.

3.8.2. Usar `struct` para combinar variables

Un `struct` es una manera de juntar un grupo de variables en una estructura. Cuando se crea un `struct`, se pueden crear varias instancias de este «nuevo» tipo de variable que ha inventado. Por ejemplo:

```
//: C03:SimpleStruct.cpp
struct Structure1 {
  char c;
  int i;
  float f;
  double d;
};

int main() {
  struct Structure1 s1, s2;
  s1.c = 'a'; // Select an element using a '.'
  s1.i = 1;
  s1.f = 3.14;
  s1.d = 0.00093;
  s2.c = 'a';
  s2.i = 1;
  s2.f = 3.14;
  s2.d = 0.00093;
} ///:~
```

La declaración de `struct` debe acabar con una llave. En `main()`, se crean dos instancias de Structure1: `s1` y `s2`. Cada una de ellas tiene su versión propia y separada de `c`, `I`, `f` y `d`. De modo que `s1` y `s2` representan bloques de variables completamente independientes. Para seleccionar uno de estos elementos dentro de `s1` o `s2`, se utiliza un ., sintaxis que se ha visto en el cápitulo previo cuando se utilizaban objetos `class` de C++ - ya que las clases surgían de `struct`s, de ahí proviene esta sintaxis.

Una cosa a tener en cuenta es la torpeza de usar Structure1 (como salta a la vista, eso sólo se requiere en C, y no en C++). En C, no se puede poner Structure1 cuando se definen variables, se debe poner struct Structure1. Aquí es donde `typedef` se vuelve especialmente útil en C:

```
//: C03:SimpleStruct2.cpp
// Using typedef with struct
typedef struct {
  char c;
  int i;
  float f;
  double d;
} Structure2;

int main() {
  Structure2 s1, s2;
  s1.c = 'a';
  s1.i = 1;
  s1.f = 3.14;
  s1.d = 0.00093;
  s2.c = 'a';
  s2.i = 1;
  s2.f = 3.14;
  s2.d = 0.00093;
} ///:~
```

Usando `typedef` de este modo, se puede simular (en C; intentar eliminar el `typedef` para C++) que Structure2 es un tipo predefinido, como int o float, cuando define `s1` y `s2` (pero se ha de tener en cuenta de que sólo tiene información - características - y no incluye comportamiento, que es lo que se obtiene con objetos reales en C++). Observe que el `struct` se ha declarado al principio, porque el objetivo es crear el `typedef`. Sin embargo, hay veces en las que sería necesario referirse a `struct` durante su definición. En esos casos, se puede repetir el nombre del `struct` como tal y como `typedef`.

```
//: C03:SelfReferential.cpp
// Allowing a struct to refer to itself

typedef struct SelfReferential {
  int i;
  SelfReferential* sr; // Head spinning yet?
} SelfReferential;

int main() {
  SelfReferential sr1, sr2;
  sr1.sr = &sr2;
  sr2.sr = &sr1;
  sr1.i = 47;
  sr2.i = 1024;
} ///:~
```

Si lo observa detenidamente, puede ver que `sr1` y `sr2` apuntan el uno al otro, guardando cada uno una parte de la información.

En realidad, el nombre `struct` no tiene que ser lo mismo que el nombre `typedef`, pero normalmente se hace de esta manera ya que tiende a simplificar las cosas.

Punteros y estructuras

En los ejemplos anteriores, todos los `structs` se manipulan como objetos. Sin embargo, como cualquier bloque de memoria, se puede obtener la dirección de un objeto `struct` (tal como se ha visto en `SelfReferential.cpp`). Para seleccionar los elementos de un objeto `struct` en particular, se utiliza un ., como se ha visto anteriormente. No obstante, si tiene un puntero a un objeto `struct`, debe seleccionar un elemento de dicho objeto utilizando un operador diferente: el ->. A continuación, un ejemplo:

```
//: C03:SimpleStruct3.cpp
// Using pointers to structs
typedef struct Structure3 {
  char c;
  int i;
  float f;
  double d;
} Structure3;

int main() {
  Structure3 s1, s2;
  Structure3* sp = &s1;
  sp->c = 'a';
  sp->i = 1;
  sp->f = 3.14;
  sp->d = 0.00093;
  sp = &s2; // Point to a different struct object
  sp->c = 'a';
  sp->i = 1;
  sp->f = 3.14;
  sp->d = 0.00093;
} ///:~
```

En `main()`, el puntero `sp` está apuntando inicialmente a `s1`, y los miembros de `s1` se inicializan seleccionándolos con el -> (y se utiliza este mismo operador para leerlos). Pero luego `sp` apunta a `s2`, y esas variables se inicializan del mismo modo. Como puede ver, otro beneficio en el uso de punteros es que pueden ser redirigidos dinámicamente para apuntar a objetos diferentes, eso proporciona más flexibilidad a sus programas, tal como verá.

De momento, es todo lo que debe saber sobre `struct`, pero se sentirá mucho más cómodo con ellos (y especialmente con sus sucesores mas potentes, las clases) a medida que progrese en este libro.

3.8.3. Programas más claros gracias a `enum`

Un tipo de datos enumerado es una manera de asociar nombres a números, y por consiguiente de ofrecer más significado a alguien que lea el código. La palabra reservada `enum` (de C) enumera automáticamente cualquier lista de identificadores que se le pase, asignándoles valores de 0, 1, 2, etc. Se pueden declarar variables `enum` (que se representan siempre como valores enteros). La declaración de un `enum` se

parece a la declaración de un `struct`.

Un tipo de datos enumerado es útil cuando se quiere poder seguir la pista de alguna característica:

```
//: C03:Enum.cpp
// Keeping track of shapes

enum ShapeType {
  circle,
  square,
  rectangle
};  // Must end with a semicolon like a struct

int main() {
  ShapeType shape = circle;
  // Activities here....
  // Now do something based on what the shape is:
  switch(shape) {
    case circle:  /* circle stuff */ break;
    case square:  /* square stuff */ break;
    case rectangle:  /* rectangle stuff */ break;
  }
} ///:~
```

`shape` es una variable del tipo de datos enumerado ShapeType, y su valor se compara con el valor en la enumeración. Ya que `shape` es realmente un int, puede albergar cualquier valor que corresponda a int (incluyendo un número negativo). También se puede comparar una variable int con un valor de una enumeración.

Se ha de tener en cuenta que el ejemplo anterior de intercambiar los tipos tiende a ser una manera problemática de programar. C++ tiene un modo mucho mejor de codificar este tipo de cosas, cuya explicación se pospondrá para mucho mas adelante en este libro.

Si el modo en que el compilador asigna los valores no es de su agrado, puede hacerlo manualmente, como sigue:

```
enum ShapeType {
  circle = 10, square = 20, rectangle = 50
};
```

Si da valores a algunos nombres y a otros no, el compilador utilizará el siguiente valor entero. Por ejemplo,

```
enum snap { crackle = 25, pop };
```

El compilador le da a `pop` el valor `26`.

Es fácil comprobar que el código es más legible cuando se utilizan tipos de datos enumerados. No obstante, en cierto grado esto sigue siendo un intento (en C) de lograr las cosas que se pueden lograr con una `class` en C++, y por eso verá que `enum` se utiliza menos en C++.

Comprobación de tipos para enumerados

Las enumeraciones en C son bastante primitivas, simplemente asocian valores enteros a nombres, pero no aportan comprobación de tipos. En C++, como era de esperar a estas alturas, el concepto de tipos es fundamental, y eso se cumple con las enumeraciones. Cuando crea una enumeración nombrada, crea efectivamente un nuevo tipo, tal como se hace con una clase: El nombre de la enumeración se convierte en una palabra reservada durante esa unidad de traducción.

Además, hay una comprobación de tipos mas estricta para la enumeración en C++ que en C. En particular, resulta evidente si tiene una instancia de la enumeración color llamada `a`. En C puede decir `a++`, pero en C++ no es posible. Eso se debe a que el incrementar una enumeración se realizan dos conversiones de tipo, una de ellas es legal en C++ y la otra no. Primero, el valor de la enumeración se convierte del tipo color a int, luego el valor se incrementa, y finalmente el int se vuelve a convertir a tipo color. En C++ esto no está permitido, porque color es un tipo diferente de int. Eso tiene sentido, porque ¿cómo saber si el incremento de `blue` siquiera estará en la lista de colores? Si quiere poder incrementar un color, debería ser una clase (con una operación de incremento) y no un `enum`, porque en la clase se puede hacer de modo que sea mucho más seguro. Siempre que escriba código que asuma una conversión implícita a un tipo `enum`, el compilador alertará de que se trata de una actividad inherentemente peligrosa.

Las uniones (descriptas a continuación) tienen una comprobación adicional de tipo similar en C++.

3.8.4. Cómo ahorrar memoria con `union`

A veces un programa manejará diferentes tipos de datos utilizando la misma variable. En esta situación, se tienen dos elecciones: se puede crear un `struct` que contenga todos los posibles tipos que se puedan necesitar almacenar, o se puede utilizar una `union`. Una `union` amontona toda la información en un único espacio; calcula la cantidad de espacio necesaria para el elemento más grande, y hace de ese sea el tamaño de la `union`. Utilice la `union` para ahorrar memoria.

Cuando se coloca un valor en una `union`, el valor siempre comienza en el mismo sitio al principio de la `union`, pero sólo utiliza el espacio necesario. Por eso, se crea una «super-variable» capaz de alojar cualquiera de las variables de la `union`. Las direcciones de todas las variables de la `union` son la misma (en una clase o `struct`, las direcciones son diferentes).

A continuación, un uso simple de una `union`. Intente eliminar varios elementos y observe qué efecto tiene en el tamaño de la `union`. Fíjese que no tiene sentido declarar más de una instancia de un sólo tipo de datos en una union (a menos que quiera darle un nombre distinto).

```
//: C03:Union.cpp
// The size and simple use of a union
#include <iostream>
using namespace std;

union Packed { // Declaration similar to a class
  char i;
  short j;
  int k;
  long l;
```

```
  float f;
  double d;
  // The union will be the size of a
  // double, since that's the largest element
};  // Semicolon ends a union, like a struct

int main() {
  cout << "sizeof(Packed) = "
       << sizeof(Packed) << endl;
  Packed x;
  x.i = 'c';
  cout << x.i << endl;
  x.d = 3.14159;
  cout << x.d << endl;
} ///:~
```

El compilador realiza la asignación apropiada para el miembro de la unión seleccionado.

Una vez que se realice una asignación, al compilador le da igual lo que se haga con la unión. En el ejemplo anterior, se puede asignar un valor en coma-flotante a `x`:

```
x.f = 2.222;
```

Y luego enviarlo a la salida como si fuese un int:

```
cout << x.i;
```

Eso produciría basura.

3.8.5. Arrays

Los vectores son un tipo compuesto porque permiten agrupar muchas variables, una a continuación de la otra, bajo un identificador único. Si dice:

```
int a[10];
```

Se crea espacio para 10 variables int colocadas una después de la otra, pero sin identificadores únicos para cada variable. En su lugar, todas están englobadas por el nombre `a`.

Para acceder a cualquiera de los *elementos del vector*, se utiliza la misma sintaxis de corchetes que se utiliza para definir el vector:

```
a[5] = 47;
```

Sin embargo, debe recordar que aunque el tamaño de `a` es `10`, se seleccionan los elementos del vector comenzando por cero (esto se llama a veces *indexado a cero*[4], de modo que sólo se pueden seleccionar los elementos del vector de 0 a 9, como sigue:

[4] (N. de T.) *zero indexing*

```
//: C03:Arrays.cpp
#include <iostream>
using namespace std;

int main() {
  int a[10];
  for(int i = 0; i < 10; i++) {
    a[i] = i * 10;
    cout << "a[" << i << "] = " << a[i] << endl;
  }
} ///:~
```

Los accesos a vectores son extremadamente rápidos, Sin embargo, si se indexa más allá del final del vector, no hay ninguna red de seguridad - se entrará en otras variables. La otra desventaja es que se debe definir el tamaño del vector en tiempo de compilación; si se quiere cambiar el tamaño en tiempo de ejecución no se puede hacer con la sintaxis anterior (C tiene una manera de crear un vector dinámicamente, pero es significativamente más sucia). El `vector` de C++ presentado en el capítulo anterior, proporciona un objeto parecido al vector que se redimensiona automáticamente , de modo que es una solución mucho mejor si el tamaño del vector no puede conocer en tiempo de compilación.

Se puede hacer un vector de cualquier tipo, incluso de `structs`:

```
//: C03:StructArray.cpp
// An array of struct

typedef struct {
  int i, j, k;
} ThreeDpoint;

int main() {
  ThreeDpoint p[10];
  for(int i = 0; i < 10; i++) {
    p[i].i = i + 1;
    p[i].j = i + 2;
    p[i].k = i + 3;
  }
} ///:~
```

Fíjese como el identificador de `struct i` es independiente del `i` del bucle `for`.

Para comprobar que cada elemento del vector es contiguo con el siguiente, puede imprimir la dirección de la siguiente manera:

```
//: C03:ArrayAddresses.cpp
#include <iostream>
using namespace std;

int main() {
  int a[10];
  cout << "sizeof(int) = "<< sizeof(int) << endl;
  for(int i = 0; i < 10; i++)
    cout << "&a[" << i << "] = "
```

```
          << (long)&a[i] << endl;
} ///:~
```

Cuando se ejecuta este programa, se ve que cada elemento está separado por el tamaño de un int del anterior. Esto significa, que están colocados uno a continuación del otro.

Punteros y arrays

El identificador de un vector es diferente de los identificadores de las variables comunes. Un identificador de un vector no es un *lvalue*; no se le puede asignar nada. En realidad es FIXME:gancho dentro de la sintaxis de corchetes, y cuando se usa el nombre de un vector, sin los corchetes, lo que se obtiene es la dirección inicial del vector:

```
//: C03:ArrayIdentifier.cpp
#include <iostream>
using namespace std;

int main() {
  int a[10];
  cout << "a = " << a << endl;
  cout << "&a[0] =" << &a[0] << endl;
} ///:~
```

Cuando se ejecuta este programa, se ve que las dos direcciones (que se imprimen en hexadecimal, ya que no se moldea a long) son las misma.

De modo que una manera de ver el identificador de un vector es como un puntero de sólo lectura al principio de éste. Y aunque no se pueda hacer que el identificador del vector apunte a cualquier otro sitio, se puede crear otro puntero y utilizarlo para moverse dentro del vector. De hecho, la sintaxis de corchetes también funciona con punteros convencionales:

```
//: C03:PointersAndBrackets.cpp
int main() {
  int a[10];
  int* ip = a;
  for(int i = 0; i < 10; i++)
    ip[i] = i * 10;
} ///:~
```

El hecho de que el nombre de un vector produzca su dirección de inicio resulta bastante importante cuando hay que pasar un vector a una función. Si declara un vector como un argumento de una función, lo que realmente está declarando es un puntero. De modo que en el siguiente ejemplo, `fun1()` y `func2()` tienen la misma lista de argumentos:

```
//: C03:ArrayArguments.cpp
#include <iostream>
#include <string>
using namespace std;
```

```
void func1(int a[], int size) {
  for(int i = 0; i < size; i++)
    a[i] = i * i - i;
}

void func2(int* a, int size) {
  for(int i = 0; i < size; i++)
    a[i] = i * i + i;
}

void print(int a[], string name, int size) {
  for(int i = 0; i < size; i++)
    cout << name << "[" << i << "] = "
         << a[i] << endl;
}

int main() {
  int a[5], b[5];
  // Probably garbage values:
  print(a, "a", 5);
  print(b, "b", 5);
  // Initialize the arrays:
  func1(a, 5);
  func1(b, 5);
  print(a, "a", 5);
  print(b, "b", 5);
  // Notice the arrays are always modified:
  func2(a, 5);
  func2(b, 5);
  print(a, "a", 5);
  print(b, "b", 5);
} ///:~
```

A pesar de que `func1()` y `func2()` declaran sus argumentos de distinta forma, el uso es el mismo dentro de la función. Hay otros hechos que revela este ejemplo: los vectores no se pueden pasados por valor[5], es decir, que nunca se puede obtener automáticamente una copia local del vector que se pasa a una función. Por eso, cuando se modifica un vector, siempre se está modificando el objeto externo. Eso puede resultar un poco confuso al principio, si lo que se espera es el paso-por-valor como en los argumentos ordinarios.

Fíjese que `print()` utiliza la sintaxis de corchetes para los argumentos de tipo vector. Aunque la sintaxis de puntero y la sintaxis de corchetes efectivamente es la mismo cuando se están pasando vectores como argumentos, la sintaxis de corchetes deja más clara al lector que se pretende enfatizar que dicho argumento es un vector.

Observe también que el argumento `size` se pasa en cada caso. La dirección no es suficiente información al pasar un vector; siempre se debe ser posible obtener el tamaño del vector dentro de la función, de manera que no se salga de los límites de

[5] A menos que tome la siguiente aproximación estricta: «todos los argumentos pasado en C/C++ son por valor, y el «valor» de un vector es el producido por su identificador: su dirección». Eso puede parecer correcto desde el punto de vista del lenguaje ensamblador, pero yo no creo que ayude cuando se trabaja con conceptos de alto nivel. La inclusión de referencias en C++ hace que el argumento «todo se pasa por valor» sea más confuso, hasta el punto de que siento que es más adecuado pensar en términos de «paso por valor» vs «paso por dirección».

dicho vector.

Los vectores pueden ser de cualquier tipo, incluyendo vectores de punteros. De hecho, cuando se quieren pasar argumentos de tipo línea de comandos dentro del programa, C y C++ tienen una lista de argumentos especial para `main()`, que tiene el siguiente aspecto:

```
int main(int argc, char* argv[]) { // ...
```

El primer argumento es el número de elementos en el vector, que es el segundo argumento. El segundo argumento es siempre un vector de char*, porque los argumentos se pasan desde la línea de comandos como vectores de caracteres (y recuerde, un vector sólo se puede pasar como un puntero). Cada bloque de caracteres delimitado por un espacio en blanco en la línea de comandos se aloja en un elemento separado en el vector. El siguiente programa imprime todos los argumentos de línea de comandos recorriendo el vector:

```
//: C03:CommandLineArgs.cpp
#include <iostream>
using namespace std;

int main(int argc, char* argv[]) {
  cout << "argc = " << argc << endl;
  for(int i = 0; i < argc; i++)
    cout << "argv[" << i << "] = "
         << argv[i] << endl;
} ///:~
```

Observe que `argv[0]` es la ruta y el nombre del programa en sí mismo. Eso permite al programa descubrir información de sí mismo. También añade un argumento más al vector de argumentos del programa, de modo que un error común al recoger argumentos de línea de comandos es tomar argv[0] como si fuera el primer argumento.

No es obligatorio utilizar `argc` y `argv` como identificadores de los parámetros de `main()`; estos identificadores son sólo convenciones (pero puede confundir al lector si no se respeta). También, hay un modo alternativo de declarar argv:

```
int main(int argc, char** argv) { // ...
```

Las dos formas son equivalentes, pero la versión utilizada en este libro es la más intuitiva al leer el código, ya que dice, directamente, «Esto es un vector de punteros a carácter».

Todo lo que se obtiene de la línea de comandos son vectores de caracteres; si quiere tratar un argumento como algún otro tipo, ha de convertirlos dentro del programa. Para facilitar la conversión a números, hay algunas funciones en la librería de C Estándar, declaradas en `<cstdlib>`. Las más fáciles de utilizar son `atoi()`, `atol()`, y `atof()` para convertir un vector de caracteres ASCII a int, long y double, respectivamente. A continuación, un ejemplo utilizando `atoi()` (las otras dos funciones se invocan del mismo modo):

```
//: C03:ArgsToInts.cpp
// Converting command-line arguments to ints
```

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main(int argc, char* argv[]) {
  for(int i = 1; i < argc; i++)
    cout << atoi(argv[i]) << endl;
} ///:~
```

En este programa, se puede poner cualquier número de argumentos en la línea de comandos. Fíjese que el bucle `for` comienza en el valor `1` para saltar el nombre del programa en `argv[0]`. También, si se pone un número decimal que contenga un punto decimal en la línea de comandos, `atoi()` sólo toma los dígitos hasta el punto decimal. Si pone valores no numéricos en la línea de comandos, `atoi()` los devuelve como ceros.

El formato de punto flotante

La función `printBinary()` presentada anteriormente en este capítulo es útil para indagar en la estructura interna de varios tipos de datos. El más interesante es el formato de punto-flotante que permite a C y C++ almacenar números que representan valores muy grandes y muy pequeños en un espacio limitado. Aunque los detalles no se pueden exponer completamente expuestos, los bits dentro de los floats y doubles están divididos en tres regiones: el exponente, la mantisa, y el bit de signo; así almacena los valores utilizando notación científica. El siguiente programa permite jugar con ello imprimiendo los patrones binarios de varios números en punto-flotante de modo que usted mismo pueda deducir el esquema del formato de punto flotante de su compilador (normalmente es el estándar IEEE para números en punto-flotante, pero su compilador puede no seguirlo):

```
//: C03:FloatingAsBinary.cpp
//{L} printBinary
//{T} 3.14159
#include "printBinary.h"
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char* argv[]) {
  if(argc != 2) {
    cout << "Must provide a number" << endl;
    exit(1);
  }
  double d = atof(argv[1]);
  unsigned char* cp =
    reinterpret_cast<unsigned char*>(&d);
  for(int i = sizeof(double)-1; i >= 0 ; i -= 2){
    printBinary(cp[i-1]);
    printBinary(cp[i]);
  }
} ///:~
```

Primero, el programa garantiza que se le haya pasado un argumento comprobando el valor de `argc`, que vale dos si hay un solo argumento (es uno si no hay argumentos, ya que el nombre del programa siempre es el primer elemento de `argv`). Si eso falla, imprime un mensaje e invoca la función `exit()` de la librería Estándar de C para finalizar el programa.

El programa toma el argumento de la línea de comandos y convierte los caracteres a double utilizando `atof()`. Luego el double se trata como un vector de bytes tomando la dirección y moldeándola a un unsigned char*. Para cada uno de estos bytes se llama a `printBinary()` para mostrarlos.

Este ejemplo se ha creado para imprimir los bytes en un orden tal que el bit de signo aparece al principio - en mi máquina. En otras máquinas puede ser diferente, por lo que puede querer re-organizar el modo en que se imprimen los bytes. También debería tener cuidado porque los formatos en punto-flotante no son tan triviales de entender; por ejemplo, el exponente y la mantisa no se alinean generalmente entre los límites de los bytes, en su lugar un número de bits se reserva para cada uno y se empaquetan en la memoria tan apretados como se pueda. Para ver lo que esta pasando, necesitaría averiguar el tamaño de cada parte del número (los bit de signo siempre son de un bit, pero los exponentes y las mantisas pueden ser de diferentes tamaños) e imprimir separados los bits de cada parte.

Aritmética de punteros

Si todo lo que se pudiese hacer con un puntero que apunta a un vector fuese tratarlo como si fuera un alias para ese vector, los punteros a vectores no tendrían mucho interés. Sin embargo, los punteros son mucho más flexibles que eso, ya que se pueden modificar para apuntar a cualquier otro sitio (pero recuerde, el identificador del vector no se puede modificar para apuntar a cualquier otro sitio).

La *aritmética de punteros* se refiere a la aplicación de alguno de los operadores aritméticos a los punteros. Las razón por la cual la aritmética de punteros es un tema separado de la aritmética ordinaria es que los punteros deben ajustarse a cláusulas especiales de modo que se comporten apropiadamente. Por ejemplo, un operador común para utilizar con punteros es ++, lo que "añade uno al puntero." Lo que de hecho significa esto es que el puntero se cambia para moverse al "siguiente valor," Lo que sea que ello signifique. A continuación, un ejemplo:

```
//: C03:PointerIncrement.cpp
#include <iostream>
using namespace std;

int main() {
  int i[10];
  double d[10];
  int* ip = i;
  double* dp = d;
  cout << "ip = " << (long)ip << endl;
  ip++;
  cout << "ip = " << (long)ip << endl;
  cout << "dp = " << (long)dp << endl;
  dp++;
  cout << "dp = " << (long)dp << endl;
} ///:~
```

Para una ejecución en mi máquina, la salida es:

```
ip = 6684124
ip = 6684128
dp = 6684044
dp = 6684052
```

Lo interesante aquí es que aunque la operación ++ parece la misma tanto para el int* como para el double*, se puede comprobar que el puntero de int* ha cambiado 4 bytes mientras que para el double* ha cambiado 8. No es coincidencia, que estos sean los tamaños de int y double en esta máquina. Y ese es el truco de la aritmética de punteros: el compilador calcula la cantidad apropiada para cambiar el puntero de modo que apunte al siguiente elemento en el vector (la aritmética de punteros sólo tiene sentido dentro de los vectores). Esto funciona incluso con vectores de `structs`:

```
//: C03:PointerIncrement2.cpp
#include <iostream>
using namespace std;

typedef struct {
  char c;
  short s;
  int i;
  long l;
  float f;
  double d;
  long double ld;
} Primitives;

int main() {
  Primitives p[10];
  Primitives* pp = p;
  cout << "sizeof(Primitives) = "
       << sizeof(Primitives) << endl;
  cout << "pp = " << (long)pp << endl;
  pp++;
  cout << "pp = " << (long)pp << endl;
} ///:~
```

La salida en esta máquina es:

```
sizeof(Primitives) = 40
pp = 6683764
pp = 6683804
```

Como puede ver, el compilador también hace lo adecuado para punteros a `structs` (y con `class` y `union`).

La aritmética de punteros también funciona con los operadores `--`, `+` y `-`, pero los dos últimos están limitados: no se puede sumar dos punteros, y si se restan punteros el resultado es el número de elementos entre los dos punteros. Sin embargo, se puede sumar o restar un valor entero y un puntero. A continuación, un ejemplo demostrando el uso de la aritmética de punteros:

```
//: C03:PointerArithmetic.cpp
#include <iostream>
using namespace std;

#define P(EX) cout << #EX << ": " << EX << endl;

int main() {
  int a[10];
  for(int i = 0; i < 10; i++)
    a[i] = i; // Give it index values
  int* ip = a;
  P(*ip);
  P(*++ip);
  P(*(ip + 5));
  int* ip2 = ip + 5;
  P(*ip2);
  P(*(ip2 - 4));
  P(*--ip2);
  P(ip2 - ip); // Yields number of elements
} ///:~
```

Comienza con otra macro, pero esta utiliza una característica del preprocesador llamada *stringizing* (implementada mediante el signo # antes de una expresión) que toma cualquier expresión y la convierte a un vector de caracteres. Esto es bastante conveniente, ya que permite imprimir la expresión seguida de dos puntos y del valor de la expresión. En `main()` puede ver lo útil que resulta este atajo.

Aunque tanto la versión prefijo como sufijo de ++ y −− son válidas para los punteros, en este ejemplo sólo se utilizan las versiones prefijo porque se aplican antes de referenciar el puntero en las expresiones anteriores, de modo que permite ver los efectos en las operaciones. Observe que se han sumado y restado valores enteros; si se combinasen de este modo dos punteros, el compilador no lo permitiría.

Aquí se ve la salida del programa anterior:

```
*ip: 0
*++ip: 1
*(ip + 5): 6
*ip2: 6
*(ip2 - 4): 2
*--ip2: 5
```

En todos los casos, el resultado de la aritmética de punteros es que el puntero se ajusta para apuntar al «sitio correcto», basándose en el tamaño del tipo de los elementos a los que está apuntado.

Si la aritmética de punteros le sobrepasa un poco al principio, no tiene porqué preocuparse. La mayoría de las veces sólo la necesitará para crear vectores e indexarlos con `[]`, y normalmente la aritmética de punteros más sofisticada que necesitará es ++ y −− . La aritmética de punteros generalmente está reservada para programas más complejos e ingeniosos, y muchos de los contenedores en la librería de Estándar C++ esconden muchos de estos inteligentes detalles, por lo que no tiene que preocuparse de ellos.

3.9. Consejos para depuración

En un entorno ideal, habrá un depurador excelente disponible que hará que el comportamiento de su programa sea transparente y podrá descubrir cualquier error rápidamente. Sin embargo, muchos depuradores tienen puntos débiles, y eso puede requerir tenga que añadir trozos de código a su programa que le ayuden a entender que está pasando. Además, puede que para la plataforma para la que esté desarrollando (por ejemplo en sistemas empotrados, con lo que yo tuve que tratar durante mis años de formación) no haya ningún depurador disponible, y quizá tenga una realimentación muy limitada (por ejemplo, un display de LEDs de una línea). En esos casos debe ser creativo a la hora de descubrir y representar información acerca de la ejecución de su programa. Esta sección sugiere algunas técnicas para conseguirlo.

3.9.1. Banderas para depuración

Si coloca el código de depuración mezclado con un programa, tendrá problemas. Empezará a tener demasiada información, que hará que los errores sean difíciles de aislar. Cuando cree que ha encontrado el error empieza a quitar el código de depuración, sólo para darse cuenta que necesita ponerlo de nuevo. Puede resolver estos problemas con dos tipos de banderas: banderas de depuración del preprocesador y banderas de depuración en ejecución.

Banderas de depuración para el preprocesador

Usando el preprocesador para definir (con `#define`) una o más banderas de depuración (preferiblemente en un fichero de cabecera), puede probar una bandera usando una sentencia `#ifdef` e incluir condicionalmente código de depuración. Cuando crea que la depuración ha terminado, simplemente utilice `#undef` la bandera y el código quedará eliminado automáticamente (y reducirá el tamaño y sobrecarga del fichero ejecutable).

Es mejor decidir los nombres de las banderas de depuración antes de empezar a contruir el proyecto para que los nombres sean consistentes. Las banderas del preprocesador tradicionalmente se distinguen de las variables porque se escriben todo en mayúsculas. Un nombre habitual es simplemente `DEBUG` (pero tenga cuidado de no usar `NDEBUG`, que está reservado en C). La secuencia de sentencias podrías ser:

```
#define DEBUG // Probably in a header file
//...
#ifdef DEBUG // Check to see if flag is defined
/* debugging code here */
#endif // DEBUG
```

La mayoría de las implementaciones de C y C++ también le permitirán definir y eliminar banderas (con `#define` y `#undef`) desde línea de comandos, y de ese modo puede recompilar código e insertar información de depuración con un único comando (preferiblemente con un `makefile`, una herramienta que será descrita en breve). Compruebe la documentación de su entorno si necesita más detalles.

Banderas para depuración en tiempo de ejecución

En algunas situaciones es más conveniente activar y desactivar las banderas de depuración durante la ejecución del programa, especialmente cuando el programa se ejecuta usando la línea de comandos. Con programas grandes resulta pesado re-

compilar sólo para insertar código de depuración.

Para activar y desactivar código de depuración dinámicamente cree banderas booleanas.

```
//: C03:DynamicDebugFlags.cpp
#include <iostream>
#include <string>
using namespace std;
// Debug flags aren't necessarily global:
bool debug = false;

int main(int argc, char* argv[]) {
  for(int i = 0; i < argc; i++)
    if(string(argv[i]) == "--debug=on")
      debug = true;
  bool go = true;
  while(go) {
    if(debug) {
      // Debugging code here
      cout << "Debugger is now on!" << endl;
    } else {
      cout << "Debugger is now off." << endl;
    }
    cout << "Turn debugger [on/off/quit]: ";
    string reply;
    cin >> reply;
    if(reply == "on") debug = true; // Turn it on
    if(reply == "off") debug = false; // Off
    if(reply == "quit") break; // Out of 'while'
  }
} ///:~
```

Este programa sigue permitiéndole activar y desactivar la bandera de depuración hasta que escriba **quit** para indicarle que quiere salir. Fíjese que es necesario escribir palabras completas, no solo letras (puede abreviarlo a letras si lo desea). Opcionalmente, también se puede usar un argumento en línea de comandos para comenzar la depuración - este argumento puede aparecer en cualquier parte de la línea de comando, ya que el código de activación en `main()` busca en todos los argumentos. La comprobación es bastante simple como se ve en la expresión:

```
string(argv[i])
```

Esto toma la cadena `argv[i]` y crea un `string`, el cual se puede comparar fácilmente con lo que haya a la derecha de ==. El programa anterior busca la cadena completa `--debug=on`. También puede buscar `--debug=` y entonces ver que hay después, para proporcionar más opciones. El Volumen 2 (disponible en www.BruceEckel.com) contiene un capítulo dedicado a la clase `string` Estándar de C++.

Aunque una bandera de depuración es uno de los relativamente pocos casos en los que tiene mucho sentido usar una variable global, no hay nada que diga que debe ser así. Fíjese en que la variable está escrita en minúsculas para recordar al lector que no es una bandera del preprocesador.

3.9.2. Convertir variables y expresiones en cadenas

Cuando se escribe código de depuración, resulta pesado escribir expresiones print que consisten en una cadena que contiene el nombre de una variable, seguido de el valor de la variable. Afortunadamente, el C estándar incluye el operador de FIXME *cadenización* #, que ya se usó antes en este mismo capítulo. Cuando se coloca un # antes de una argumentos en una macro, el preprocesador convierte ese argumentos en una cadena. Esto, combinado con el hecho de que las cadenas no indexadas colocadas una a continuación de la otra se concatenan, permite crear macros muy adecuadas para imprimir los valores de las variables durante la depuración:

```
#define PR(x) cout << #x " = " << x << "\n";
```

Si se imprime la variable `a` invocando `PR(a)`, tendrá el mismo efecto que este código:

```
cout << "a = " << a << "\n";
```

Este mismo proceso funciona con expresiones completas. El siguiente programa usa una macro para crear un atajo que imprime la expresión cadenizadas y después evalúa la expresión e imprime el resultado:

```
//: C03:StringizingExpressions.cpp
#include <iostream>
using namespace std;

#define P(A) cout << #A << ": " << (A) << endl;

int main() {
  int a = 1, b = 2, c = 3;
  P(a); P(b); P(c);
  P(a + b);
  P((c - a)/b);
} ///:~
```

Puede comprobar cómo una técnica como esta se puede convertir rápidamente en algo indispensable, especialmente si no tiene depurador (o debe usar múltiples entornos de desarrollo). También puede insertar un `#ifdef` para conseguir que `P-(A)` se defina como «nada» cuando quiera eliminar el código de depuración.

3.9.3. La macro C assert()

En el fichero de cabecera estándar `<cassert>` aparece `assert()`, que es una macro de depuración. Cuando se utiliza `assert()`, se le debe dar un argumento que es una expresión que usted está «aseverando». El preprocesador genera código que comprueba la aserción. Si la aserción no es cierta, el programa parará después de imprimir un mensaje de error informando que la aserción falló. Este es un ejemplo trivial:

```
//: C03:Assert.cpp
// Use of the assert() debugging macro
#include <cassert>  // Contains the macro
```

```
using namespace std;

int main() {
  int i = 100;
  assert(i != 100); // Fails
} ///:~
```

La macro original es C Estándar, así que está disponible también en el fichero de cabecera `assert.h`.

Cuando haya terminado la depuración, puede eliminar el código generado por la macro escribiendo la siguiente línea:

```
#define NDEBUG
```

en el programa, antes de la inclusión de `<cassert>`, o definiendo `NDEBUG` en la línea de comandos del compilador. `NDEBUG` es una bandera que se usa en `<cassert>` para cambiar el código generado por las macros.

Más adelante en este libro, verá algunas alternativas más sofisticadas a `assert()`.

3.10. Direcciones de función

Una vez que una función es compilada y cargada en la computadora para ser ejecutada, ocupa un trozo de memoria. Esta memoria, y por tanto esa función, tiene una dirección.

C nunca ha sido un lenguaje [FIXME] donde otros temen pisar. Puede usar direcciones de función con punteros igual que puede usar direcciones variables. La declaración y uso de punteros a función parece un poco opaca al principio, pero sigue el formato del resto del lenguaje.

3.10.1. Definición de un puntero a función

Para definir un puntero a una función que no tiene argumentos y no retorna nada, se dice:

```
void (*funcPtr)();
```

Cuando se observa una definición compleja como esta, el mejor método para entenderla es empezar en el medio e ir hacia afuera. «Empezar en el medio» significa empezar con el nombre de la variable, que es `funPtr`. «Ir hacia afuera» significa mirar al elemento inmediatamente a la derecha (nada en este caso; el paréntesis derecho marca el fin de ese elemento), después mire a la izquierda (un puntero denotado por el asterisco), después mirar de nuevo a la derecha (una lista de argumentos vacía que indica que no función no toma argumentos), después a la izquierda (void, que indica que la función no retorna nada). Este movimiento derecha-izquierda-derecha funciona con la mayoría de las declaraciones. [6]

[6] (N. del T.) Otra forma similar de entenderlo es dibujar mentalmente una espiral que empieza en el medio (el identificador) y se va abriendo.

Para repasar, «empezar en el medio» («`funcPtr` es un ...», va a la derecha (nada aquí - pare en el paréntesis derecho), va a la izquierda y encuentra el * («... puntero a ...»), va a la derecha y encuentra la lista de argumentos vacía («... función que no tiene argumentos ...») va a la izquierda y encuentra el void («`funcPtr` es un puntero a una función que no tiene argumentos y retorna void»).

Quizá se pregunte porqué `*funcPtr` necesita paréntesis. Si no los usara, el compilador podría ver:

```
void *funcPtr();
```

Lo que corresponde a la declaración de una función (que retorna un void*) en lugar de definir una variable. Se podría pensar que el compilador sería capaz distinguir una declaración de una definición por lo que se supone que es. El compilador necesita los paréntesis para «tener contra qué chocar» cuando vaya hacia la izquierda y encuentre el *, en lugar de continuar hacia la derecha y encontrar la lista de argumentos vacía.

3.10.2. Declaraciones y definiciones complicadas

Al margen, una vez que entienda cómo funciona la sintáxis de declaración de C y C++ podrá crear elementos más complicados. Por ejemplo:

```
//: V1C03:ComplicatedDefinitions.cpp

/* 1. */     void * (*(*fp1)(int))[10];

/* 2. */     float (*(*fp2)(int,int,float))(int);

/* 3. */     typedef double (*(*(*fp3)())[10])();
             fp3 a;

/* 4. */     int (*(*f4())[10])();

int main() {}
```

Estudie cada uno y use la regla derecha-izquierda para entenderlos. El número 1 dice «`fp1` es un puntero a una función que toma un entero como argumento y retorna un puntero a un array de 10 punteros void».

El 2 dice «`fp2` es un puntero a función que toma tres argumentos (int, int y float) de retorna un puntero a una función que toma un entero como argumento y retorna un float»

Si necesita crear muchas definiciones complicadas, debería usar `typedef`. El número 3 muestra cómo un `typedef` ahorra tener que escribir una descripción complicada cada vez. Dice «Un fp3 es un puntero a una función que no tiene argumentos y que retorna un puntero a un array de 10 punteros a funciones que no tienen argumentos y retornan doubles». Después dice «`a` es una variable de ese tipo fp3». `typedef` es útil para construir descripciones complicadas a partir de otras simples.

El 4 es una declaración de función en lugar de una definición de variable. Dice «`f4` es una función que retorna un puntero a un array de 10 punteros a funciones que retornan enteros».

Es poco habitual necesitar declaraciones y definiciones tan complicadas como éstas. Sin embargo, si se propone entenderlas, no le desconcertarán otras algo menos complicadas pero que si encontrará en la vida real.

3.10.3. Uso de un puntero a función

Una vez que se ha definido un puntero a función, debe asignarle la dirección de una función antes de poder usarlo. Del mismo modo que la dirección de un array `arr[10]` se obtiene con el nombre del array sin corchetes (`arr`), la dirección de una función `func()` se obtiene con el nombre de la función sin lista de argumentos (`func`). También puede usar una sintáxis más explícita: `&func()`. Para invocar la función, debe dereferenciar el puntero de la misma forma que lo ha declarado (recuerde que C y C++ siempre intentan hacer que las definiciones se parezcan al modo en que se usan). El siguiente ejemplo muestra cómo se define y usa un puntero a función:

```
//: C03:PointerToFunction.cpp
// Defining and using a pointer to a function
#include <iostream>
using namespace std;

void func() {
  cout << "func() called..." << endl;
}

int main() {
  void (*fp)();  // Define a function pointer
  fp = func;  // Initialize it
  (*fp)();    // Dereferencing calls the function
  void (*fp2)() = func;  // Define and initialize
  (*fp2)();
} ///:~
```

Una vez definido el puntero a función `fp`, se le asigna la dirección de una función `func()` usando `fp = func` (fíjese que la lista de argumentos no aparece junto al nombre de la función). El segundo caso muestra una definición e inicialización simultánea.

3.10.4. Arrays de punteros a funciones

Una de las construcciones más interesantes que puede crear es un array de punteros a funciones. Para elegir una función, sólo indexe el array y dereferencie el puntero. Esto permite implementar el concepto de *código dirigido por tabla*(*table-driven code*); en lugar de usar estructuras condicionales o sentencias case, se elige la función a ejecutar en base a una variable (o una combinación de variables). Este tipo de diseño puede ser útil si añade y elimina funciones de la tabla con frecuencia (o si quiere crear o cambiar una tabla dinámicamente).

El siguiente ejemplo crea algunas funciones falsas usando una macro de preprocesador, después crea un array de punteros a esas funciones usando inicialización automática. Como puede ver, es fácil añadir y eliminar funciones de la table (y por tanto, la funcionalidad del programa) cambiando una pequeña porción de código.

```
//: C03:FunctionTable.cpp
// Using an array of pointers to functions
#include <iostream>
using namespace std;

// A macro to define dummy functions:
#define DF(N) void N() { \
   cout << "function " #N " called..." << endl; }

DF(a); DF(b); DF(c); DF(d); DF(e); DF(f); DF(g);

void (*func_table[])() = { a, b, c, d, e, f, g };

int main() {
  while(1) {
    cout << "press a key from 'a' to 'g' "
      "or q to quit" << endl;
    char c, cr;
    cin.get(c); cin.get(cr); // second one for CR
    if ( c == 'q' )
      break; // ... out of while(1)
    if ( c < 'a' || c > 'g' )
      continue;
    (*func_table[c - 'a'])();
  }
} ///:~
```

A partir de este punto, debería ser capaz de imaginar cómo esta técnica podría resultarle útil cuando tenga que crear algún tipo de intérprete o programa para procesar listas.

3.11. Make: cómo hacer compilación separada

Cuando se usa *compilación separada* (dividiendo el código en varias unidades de traducción), aparece la *necesidad* de un medio para compilar automáticamente cada fichero y decirle al enlazador como montar todas las piezas - con las librerías apropiadas y el código de inicio - en un fichero ejecutable. La mayoría de los compiladores le permiten hacerlo desde una sólo instrucción desde línea de comandos. Por ejemplo, para el compilador de C++ de GNU se puede hacer:

```
$ g++ SourceFile1.cpp SourceFile2.cpp
```

En problema con este método es que el compilador compilará cada fichero individual tanto si el fichero *necesita* ser recompilado como sino. Cuando un proyecto tiene muchos ficheros, puede resultar prohibitivo recompilar todo cada vez que se cambia una línea en un fichero.

La solución a este problema, desarrollada en Unix pero disponible de alún modo en todos los sistemas es un programa llamado **make**. La utilidad **make** maneja todos los ficheros individuales de un proyecto siguiendo las instrucciones escritas en un fichero de texto llamado `makefile`. Cuando edite alguno de los ficheros del proyecto y ejecute **make**, el programa **make** seguirá las directrices del `makefile` para comparar las fechas de los ficheros fuente con las de los ficheros resultantes

correspondientes, y si una fichero fuente es más reciente que su fichero resultante, **make** recompila ese fichero fuente. **make** sólo recompila los ficheros fuente que han cambiado, y cualquier otro fichero que esté afectado por el fichero modificado. Usando **make** no tendrá que recompilar todos los ficheros de su proyecto cada vez que haga un cambio, ni tendrá que comprobar si todo se construye adecuadamente. El `makefile` contiene todas las instrucciones para montar el proyecto. Aprender a usar **make** le permitirá ahorrar mucho tiempo y frustraciones. También descubrirá que **make** es el método típico para instalar software nuevo en máquinas GNU o Unix[7] (aunque esos `makefiles` tienen a ser mucho más complicados que los que aparecen en este libro, y a menudo podrá generar automáticamente un `makefile` para su máquina particular como parte del proceso de instalación).

Como **make** está disponible de algún modo para prácticamente todos los compiladores de C++ (incluso si no lo está, puede usar **makes** libres con cualquier compilador), será la herramienta usada en este libro. Sin embargo, los fabricantes de compiladores crean también sus propias herramientas para construir proyectos. Estás herramientas preguntan qué ficheros hay en el proyecto y determinan las relaciones entre ellos. Estas herramientas utilizan algo similar a un `makefile`, normalmente llamado *fichero de proyecto*, pero el entorno de programación mantiene este fichero para que el programador no tenga que preocuparse de él. La configuración y uso de los ficheros de proyecto varía de un entorno de desarrollo a otro, de modo que tendrá que buscar la documentación apropiada en cada caso (aunque esas herramientas proporcionadas por el fabricante normalmente son tan simples de usar que es fácil aprender a usarlas jugando un poco con ellas - mi método educativo favorito).

Los `makefiles` que acompañan a este libro deberían funcionar bien incluso si también usa una herramienta específica para construcción de proyectos.

3.11.1. Las actividades de Make

Cuando escribe **make** (o cualquiera que sea el nombre del su programa **make**), **make** busca un fichero llamado `makefile` o `Makefile` en el directorio actual, que usted habrá creado para su proyecto. Este fichero contiene una lista de dependencias entre ficheros fuente, **make** comprueba las fechas de los ficheros. Si un fichero tiene una fecha más antigua que el fichero del que depende, **make** ejecuta la *regla* indicada después de la dependencia.

Todos los comentarios de los `makefiles` empiezan con un # y continúan hasta el fina de la línea.

Como un ejemplo sencillo, el `makefile` para una programa llamado «hello» podría contener:

```
# A comment
hello.exe: hello.cpp
        mycompiler hello.cpp
```

Esto dice que `hello.exe` (el objetivo) depende de `hello.cpp`. Cuando `hello.cpp` tiene una fecha más reciente que `hello.exe`, **make** ejecuta la «regla» **mycompiler hello.cpp**. Puede haber múltiples dependencias y múltiples reglas. Muchas implementaciones de **make** requieren que todas las reglas empiecen con un tabulador.

[7] (N. de T.) El método del que habla el autor se refiere normalmente a software instalado a partir de su código fuente. La instalación de paquetes binarios es mucho más simple y automatizada en la mayoría de las variantes actuales del sistema operativo GNU.

Para lo demás, por norma general los espacios en blanco se ignoran de modo que se pueden usar a efectos de legibilidad.

Las reglas no están restringidas a llamadas al compilador; puede llamar a cualquier programa que quiera. Creando grupos de reglas de dependencia, puede modificar sus ficheros fuentes, escribir `make` y estar seguro de que todos los fichero afectados serán re-construidos correctamente.

Macros

Un `makefile` puede contener *macros* (tenga en cuenta que estas macros no tienen nada que ver con las del preprocesador de C/C++). La macros permiten reemplazar cadenas de texto. Los `makefiles` del libro usan una macro para invocar el compilador de C++. Por ejemplo,

```
CPP = mycompiler
hello.exe: hello.cpp
        $(CPP) hello.cpp
```

El = se usa para indicar que `CPP` es una macro, y el $ y los paréntesis expanden la macro. En este caso, la expansión significa que la llamada a la macro `$(CPP)` será reemplazada con la cadena `mycompiler`. Con esta macro, si quiere utilizar un compilador diferente llamado **cpp**, sólo tiene que cambiar la macro a:

```
CPP = cpp
```

También puede añadir a la macro opciones del compilador, etc., o usar otras macros para añadir dichas opciones.

Reglas de sufijo

Es algo tedioso tener que decir a **make** que invoque al compilador para cada fichero `cpp` del proyecto, cuando se sabe que básicamente siempre es el mismo proceso. Como **make** está diseñado para ahorrar tiempo, también tiene un modo de abreviar acciones, siempre que dependan del sufijo de los ficheros. Estas abreviaturas se llaman *reglas de sufijo*. Una regla de sufijo es la la forma de indicar a **make** cómo convertir un fichero con cierta extensión (`.cpp` por ejemplo) en un fichero con otra extensión (`.obj` o `.exe`). Una vez que le haya indicado a **make** las reglas para producir un tipo de fichero a partir de otro, lo único que tiene que hacer es decirle a **make** cuales son las dependencias respecto a otros ficheros. Cuando **make** encuentra un fichero con una fecha previa a otro fichero del que depende, usa la regla para crear la versión actualizada del fichero objetivo.

La regla de sufijo le dice a **make** que no se necesitan reglas explícitas para construir cada cosa, en su lugar le explica cómo construir cosas en base a la extensión del fichero. En este caso dice «Para contruir un fichero con extensión `.exe` a partir de uno con extensión `.cpp`, invocar el siguiente comando». Así sería para ese ejemplo:

```
CPP = mycompiler
.SUFFIXES: .exe .cpp
.cpp.exe:
        $(CPP) $<
```

La directiva `.SUFFIXES` le dice a **make** que debe vigilar las extensiones que se

indican porque tiene un significado especial para este `makefile` en particular. Lo siguiente que aparece es la regla de sufijo `.cpp.exe`, que dice «cómo convertir cualquier fichero con extensión `.cpp` a uno con extensión `.exe`» (cuando el fichero `.cpp` es más reciente que el fichero `..exe`). Como antes, se usa la macro `$(CPP)`, pero aquí aparece algo nuevo: `$<`. Como empieza con un $ es que es una macro, pero esta es una de las macros especiales predefinidas por **make**. El `$<` se puede usar sólo en reglas de sufijo y significa «cualquier prerrequisito que dispare la regla» (a veces llamado *dependencia*), que en este caso se refiere al «fichero `.cpp` que necesita ser compilado».

Una ver que las reglas de sufijo se han fijado, puede indicar por ejemplo algo tan simple como **make Union.exe** y se aplicará la regla sufijo, incluso aunque no se mencione «Union» en ninguna parte del `makefile`.

Objetivos predeterminados

Después de las macros y las reglas de sufijo, **make** busca la primero «regla» del fichero, y la ejecuta, a menos que se especifica una regla diferente. Así que pare el siguiente `makefile`:

```
CPP = mycompiler
.SUFFIXES: .exe .cpp
.cpp.exe:
        $(CPP) $<
target1.exe:
target2.exe:
```

Si ejecuta simplemente **make**, se construirá `target1.exe` (usando la regla de sufijo predeterminada) porque ese es el primer objetivo que **make** va a encontrar. Para construir `target2.exe` se debe indicar explícitamente diciendo **make target2.exe**. Esto puede resultar tedioso de modo que normalmente se crea un objetivo «dummy» por defecto que depende del resto de objetivos, como éste:

```
CPP = mycompiler
.SUFFIXES: .exe .cpp
.cpp.exe:
        $(CPP) $<
all: target1.exe target2.exe
```

Aquí, `all` no existe y no hay ningún fichero llamada `all`, así que cada vez que ejecute **make**, el programa verá que `all` es el primer objetivo de la lista (y por tanto el objetivo por defecto), entonces comprobará que `all` no existe y analizará sus dependencias. Comprueba `target1.exe` y (usando la regla de sufijo) comprobará (1) que `target1.exe` existe y (2) que `target1.cpp` es más reciente que `target1.exe`, y si es así ejecutará la regla (si proporciona una regla explícita para un objetivo concreto, se usará esa regla en su lugar). Después pasa a analizar el siguiente fichero de la lista de objetivos por defecto. De este modo, breando una lista de objetivos por defecto (típicamente llamada `all` por convenio, aunque se puede tener cualquier nombre) puede conseguir que se construyan todos los ejecutables de su proyecto simplemente escribiendo **make**. Además, puede tener otras listas de objetivos para hacer otras cosas - por ejemplo, podría hacer que escribiendo **make debug** se reconstruyeran todos los ficheros pero incluyendo información de depuración.

3.11.2. Los Makefiles de este libro

Usando el programa `ExtractCode.cpp` del Volumen 2 de este libro, se han extraido automáticamente todos los listado de código de este libro a partir de la versión en texto ASCII y se han colocado en subdirectorios de acuerdo a sus capítulos. Además, `ExtractCode.cpp` crea varios `makefiles` en cada subdirectorio (con nombres diferentes) para que pueda ir a cualquier subdirectorio y escribir **make -f mycompiler.makefile** (sustituyendo «mycompiler» por el nombre de su compilador, la opción `-f` dice «utiliza lo siguiente como un `makefile`»). Finalmente, `ExtractCode.cpp` crea un `makefile` «maestro» en el directorio raíz donde se hayan extraido los fichero del libro, y este `makefile` descienda a cada subdirectorio y llama a **make** con el `makefile` apropiado. De este modo, se puede compilar todo el código de los listados del libro invocando un único comando **make**, y el proceso parará cada vez que su compilador no pueda manejar un fichero particular (tenga presente que un compilador conforme al Estándar C++ debería ser compatible con todos los ficheros de este libro). Como algunas implementaciones de **make** varían de un sistema a otro, en los `makefiles` generados se usan sólo las características más básicas y comunes.

3.11.3. Un ejemplo de Makefile

Tal como se mencionaba, la herramienta de extracción de código `ExtractCode.cpp` genera automáticamente `makefiles` para cada capítulo. Por eso, los `makefiles` de cada capítulo no aparecen en el libro (todos los `makefiles` están empaquetados con el código fuente, que se puede descargar de www.BruceEckel.com). Sin embargo, es útil ver un ejemplo de un `makefile`. Lo siguiente es una versión recortada de uno de esos `makefiles` generados automáticamente para este capítulo. Encontrará más de un `makefile` en cada subdirectorio (tienen nombres diferentes; puede invocar uno concreto con **make -f**. Éste es para GNU C++:

```
CPP = g++
OFLAG = -o
.SUFFIXES : .o .cpp .c
.cpp.o :
  $(CPP) $(CPPFLAGS) -c $<
.c.o :
  $(CPP) $(CPPFLAGS) -c $<

all: \
  Return \
  Declare \
  Ifthen \
  Guess \
  Guess2
# Rest of the files for this chapter not shown

Return: Return.o
  $(CPP) $(OFLAG)Return Return.o

Declare: Declare.o
  $(CPP) $(OFLAG)Declare Declare.o

Ifthen: Ifthen.o
  $(CPP) $(OFLAG)Ifthen Ifthen.o
```

```
Guess: Guess.o
  $(CPP) $(OFLAG)Guess Guess.o

Guess2: Guess2.o
  $(CPP) $(OFLAG)Guess2 Guess2.o

Return.o: Return.cpp
Declare.o: Declare.cpp
Ifthen.o: Ifthen.cpp
Guess.o: Guess.cpp
Guess2.o: Guess2.cpp
```

La macro CPP contiene el nombre del compilador. Para usar un compilador diferente, puede editar el `makefile` o cambiar el valor de la macro desde línea de comandos, algo como:

```
$ make CPP=cpp
```

Tenga en cuenta, sin embargo, que `ExtractCode.cpp` tiene un esquema automático para construir `makefiles` para compiladores adicionales.

La segunda macro `OFLAG` es la opción que se usa para indicar el nombre del fichero de salida. Aunque muchos compiladores asumen automáticamente que el fichero de salida tiene el mismo nombre base que el fichero de entrada, otros no (como los compiladores GNU/Unix, que por defecto crean un fichero llamado `a.out`).

Como ve, hay dos reglas de sufijo, una para ficheros `.cpp` y otra para fichero `.c` (en caso de que se necesite compilar algún fuente C). El objetivo por defecto es `all`, y cada línea de este objetivo está «continuada» usando la contrabarra, hasta `Guess2`, que el el último de la lista y por eso no tiene contrabarra. Hay muchos más fichero en este capítulo, pero (por brevedad) sólo se muestran algunos.

Las reglas de sufijo se ocupan de crear fichero objeto (con extensión `.o`) a partir de los fichero `.cpp`, pero en general se necesita escribir reglas explícitamente para crear el ejecutable, porque normalmente el ejecutable se crea enlazando muchos fichero objeto diferente y **make** no puede adivinar cuales son. También, en este caso (GNU/Unix) no se usan extensiones estándar para los ejecutables de modo que una regla de sufijo no sirve para esas situaciones. Por eso, verá que todas las reglas para construir el ejecutable final se indican explícitamente.

Este `makefile` toma el camino más seguro usando el mínimo de prestaciones de **make**; sólo usa los conceptos básicos de objetivos y dependencias, y también macros. De este modo está prácticamente asegurado que funcionará con la mayoría de las implementaciones de **make**. Eso implica que se producen fichero `makefile` más grandes, pero no es algo negativo ya que se generan automáticamente por `ExtractCode.cpp`.

Hay muchísimas otras prestaciones de **make** que no se usan en este libro, incluyendo las versiones más nuevas e inteligentes y las variaciones de **make** con atajos avanzados que permiten ahorrar mucho tiempo. La documentación propia de cada **make** particular describe en más profundidad sus características; puede aprender más sobre **make** en *Managing Projects with Make* de Oram y Taiboot (O'Reilly, 1993). También, si el fabricante de su compilador no proporciona un **make** o usa uno que no es estándar, puede encontrar GNU Make para prácticamente todas las plataformas que existen buscado en los archivos de GNU en internet (hay muchos).

3.12. Resumen

Este capítulo ha sido un repaso bastante intenso a través de todas las características fundamentales de la sintaxis de C++, la mayoría heredada de C (y ello redunda la compatibilidad hacia atrás FIXME:vaunted de C++ con C). Aunque algunas características de C++ se han presentado aquí, este repaso está pensado principalmente para personas con experiencia en programación, y simplemente necesitan una introducción a la sintaxis básica de C y C++. Incluso si usted ya es un programador de C, puede que haya visto una o dos cosas de C que no conocía, aparte de todo lo referente a C++ que probablemente sean nuevas. Sin embargo, si este capítulo le ha sobrepasado un poco, debería leer el curso en CD ROM *Thinking in C: Foundations for C++ and Java* que contiene lecturas, ejercicios, y soluciones guiadas), que viene con este libro, y también está disponible en www.BruceEckel.com.

3.13. Ejercicios

Las soluciones a los ejercicios se pueden encontrar en el documento electrónico titulado «The Thinking in C++ Annotated Solution Guide», disponible por poco dinero en www.BruceEckel.com.

1. Cree un fichero de cabecera (con extensión «.h»). En este fichero, declare un grupo de funciones variando las listas de argumentos y valores de retorno de entre los siguientes: `void`, `char`, `int` y `float`. Ahora cree un fichero `.cpp` que incluya su fichero de cabecera y haga definiciones para todas esas funciones. Cada definición simplemente debe imprimir en nombre de la función, la lista de argumentos y el tipo de retorno para que se sepa que ha sido llamada. Cree un segundo fichero `.cpp` que incluya el fichero de cabecera y defina una `int main()`, que contenga llamadas a todas sus funciones. Compile y ejecute su programa.
2. Escriba un programa que use dos bucles `for` anidados y el operador módulo (`%`) para detectar e imprimir números enteros (números enteros sólo divisibles entre si mismos y entre 1).
3. Escriba un programa que utilice un bucle `while` para leer palabras de la entrada estándar (`cin`) y las guarde en un string. Este es un bucle `while` «infinito», que debe romper (y salir del programa) usando la sentencia `break`. Por cada palabra que lea, evalúela primero usando una secuencia de sentencias `if` para «mapear» un valor entero de la palabra, y después use una sentencia `switch` que utilice ese valor entero como selector (esta secuencia de eventos no es un buen estilo de programación; solamente es un supuesto para que practique con el control de flujo). Dentro de cada `case`, imprima algo con sentido. Debe decidir cuales son las palabras interesantes y qué significan. También debe decidir qué palabra significa el fin del programa. Pruebe el programa redireccionando un fichero como entrada (si quiere ahorrarse tener que escribir, ese fichero puede ser el propio código fuente del programa).
4. Modifique `Menu.cpp` para usar sentencias `switch` en lugar de `if`.
5. Escriba un programa que evalúe las dos expresiones de la sección llamada «precedencia».
6. Modifique `YourPets2.cpp` para que use varios tipos de datos distintos (char, int, float, double, y sus variantes). Ejecute el programa y cree un mapa del esquema de memoria resultante. Si tiene acceso a más de un tipo de máquina,

sistema operativo, o compilador, intente este experimento con tantas variaciones como pueda manejar.

7. Cree dos funciones, una que tome un string* y una que tome un string&. Cada una de estas funciones debería modificar el objeto externo a su manera. En `main()`, cree e inicialice un objeto string, imprímalo, después páselo a cada una de las dos funciones, imprimiendo los resultados.
8. Escriba un programa que use todos los trígrafos para ver si su compilador los soporta.
9. Compile y ejecute `Static.cpp`. Elimine la palabra reservada `static` del código, compile y ejecútelo de nuevo, y explique lo que ocurre.
10. Intente compilar y enlazar `FileStatic.cpp` con `FileStatic2.cpp`. ¿Qué significan los mensajes de error que aparecen?
11. Modifique `Boolean.cpp` para que funcione con valores double en lugar de int.
12. Modifique `Boolean.cpp` y `Bitwise.cpp` de modo que usen los operadores explícitos (si su compilador es conforme al Estándar C++ los soportará).
13. Modifique `Bitwise.cpp` para usar las funciones de `Rotation.cpp`. Asegúrese de que muestra los resultados que deje claro qué ocurre durante las rotaciones.
14. Modifique `Ifthen.cpp` para usar el operador `if-else` ternario(`?:`).
15. Cree una `struct` que contenga dos objetos `string` y uno int. Use un `typedef` para el nombre de la `struct`. Cree una instancia de la `struct`, inicialice los tres valores de la instancia, y muestrelos en pantalla. Tome la dirección de su instancia y asígnela a un puntero a tipo de la `struct`. Usando el puntero, Cambie los tres valores de la instancia y muestrelos.
16. Cree un programa que use un enumerado de colores. Cree una variable de este tipo `enum` y, utilizando un bucle, muestre todos los números que corresponden a los nombres de los colores.
17. Experimente con `Union.cpp` eliminando varios elementos de la `union` para ver el efecto que causa en el tamaño de la `union` resultante. Intente asignar un elemento (por tanto un tipo) de la `union` y muéstrelo por medio de un elemento diferente (por tanto, un tipo diferente) para ver que ocurre.
18. Cree un programa que defina dos arrays de int, uno a continuación del otro. Indexe el primer array más allá de su tamaño para caer sobre el segundo, haga una asignación. Muestre el segundo array para ver los cambios que eso ha causado. Ahora intente definir una variable char entre las definiciones de los arrays, y repita el experimento. Quizá quiera crear una función para imprimir arrays y así simplificar el código.
19. Modifique `ArrayAddresses.cpp` para que funcione con los tipos de datos char, long int, float y double.
20. Aplique la técnica de `ArrayAddresses.cpp` para mostrar el tamaño de la `struct` y las direcciones de los elementos del array de `StructArray.cpp`.
21. Cree un array de objetos string y asigne una cadena a cada elemento. Muestre el array usando un bucle `for`.

22. Cree dos nuevos programas a partir de `ArgsToInts.cpp` que usen `atol()` y `atof()` respectivamente.

23. Modifique `PointerIncrement2.cpp` de modo que use una `union` en lugar de una `struct`.

24. Modifique `PointerArithmetic.cpp` para que funcione con long y long double.

25. Defina una variable float. Tome su dirección, moldee esa dirección a un unsigned char, y asígnela a un puntero unsigned char. Usando este puntero y `[]`, indexe la variable float y use la función `printBinary()` definida en este capítulo para mostrar un mapa de cada float (vaya desde 0 hasta `sizeof(float)`). Cambie el valor del float y compruebe si puede averiguar que hay en el float (el float contiene datos codificados).

26. Defina un array de int. Tome la dirección de comienzo de ese array y utilice `static_cast` para convertirlo a un void*. Escriba una función que tome un void*, un número (que indica el número de bytes), y un valor (indicando el valor que debería ser asignado a cada byte) como argumentos. La función debería asignar a cada byte en el rango especificado el valor dado como argumento. Pruebe la función con su array de int.

27. Cree un array `const` de double y un array `volatile` de double. Indexe cada array y utilice `const_cast` para moldear cada elemento de no-`const` y no-`volatile`, respectivamente, y asigne un valor a cada elemento.

28. Cree una función que tome un puntero a un array de double y un valor indicando el tamaño de ese array. La función debería mostrar cada valor del array. Ahora cree un array de double y inicialice cada elemento a cero, después utilice su función para mostrar el array. Después use `reinterpret_cast` para moldear la dirección de comienzo de su array a un unsigned char*, y ponga a 1 cada byte del array (aviso: necesitará usar `sizeof` para calcular el número de bytes que tiene un double). Ahora use su función de impresión de arrays para mostrar los resultados. ¿Por qué cree los elementos no tienen el valor 1.0?

29. (Reto) Modifique `FloatingAsBinary.cpp` para que muestra cada parte del double como un grupo separado de bits. Tendrá que reemplazar las llamadas a `printBinary()` con su propio código específico (que puede derivar de `printBinary()`) para hacerlo, y también tendrá que buscar y comprender el formato de punto flotante incluyendo el ordenamiento de bytes para su compilador (esta parte es el reto).

30. Cree un `makefile` que no sólo compile `YourPets1.cpp` y `YourPets2.cpp` (para cada compilador particular) sino que también ejecute ambos programas como parte del comportamiento del objetivo predeterminado. Asegúrese de usar las reglas de sufijo.

31. Modifique `StringizingExpressions.cpp` para que `P(A)` sea condicionalmente definida con `#ifdef` para permitir que el código de depuración sea eliminado automáticamente por medio de una bandera en línea de comandos. Necesitará consultar la documentación de su compilador para ver cómo definir y eliminar valores del preprocesador en el compilador de línea de comandos.

32. Defina una función que tome un argumento double y retorne un int. Cree e inicialice un puntero a esta función, e invoque la función por medio del puntero.

33. Declare un puntero a un función que toma un argumento int y retorna un puntero a una función que toma un argumento char y retorna un float.

34. Modifique `FunctionTable.cpp` para que cada función retorne un string (en lugar de mostrar un mensaje) de modo que este valor se imprima en `main()`.

35. Cree un `makefile` para uno de los ejercicios previos (a su elección) que le permita escribir **make** para construir una versión en producción del programa y **make debug** para construir una versión del programa que incluye información de depuración.

4: Abstracción de Datos

C++ es una herramienta de mejora de la productividad. ¿Por qué si no haría el esfuerzo (y es un esfuerzo, a pesar de lo fácil que intetemos hacer la transición)

de cambiar de algún lenguaje que ya conoce y con el cual ya es productivo a un nuevo lenguaje con el que será menos productivo durante un tiempo, hasta que se haga con él? Se debe a que está convencido de que conseguirá grandes ventajas usando esta nueva herramienta.

En términos de programación, productividad significa que menos personas, en menos tiempo, puedan realizar programas más complejos y significativos. Desde luego, hay otras cuestiones que nos deben importar a la hora de escoger un lenguaje de programación. Aspectos a tener en cuenta son la eficiencia (¿la naturaleza del lenguaje hace que nuestros programas sean lentos o demasiado grandes?), la seguridad (¿nos ayuda el lenguaje a asegurarnos de que nuestros programas hagan siempre lo que queremos? ¿maneja el lenguaje los errores apropiadamente?) y el mantenimiento (¿el lenguaje ayuda a crear código fácil de entender, modificar y extender?). Estos son, con certeza, factores importantes que se examinarán en este libro.

Pero la productividad real significa que un programa que para ser escrito, antes requería de tres personas trabajando una semana, ahora le lleve sólo un día o dos a una sola persona. Esto afecta a varios niveles de la esfera económica. A usted le agrada ver que es capaz de construir algo en menos tiempo, sus clientes (o jefe) están contentos porque los productos les llegan más rápido y utilizando menos mano de obra y finalmente los compradores se alegran porque pueden obtener productos más baratos. La única manera de obtener incrementos masivos en productividad es apoyándose en el código de otras personas; o sea, usando librerías.

Una librería es simplemente un montón de código que alguien ha escrito y empaquetado todo junto. Muchas veces, el paquete mínimo es tan sólo un archivo con una extensión especial como `lib` y uno o más archivos de cabecera que le dicen al compilador qué contiene la librería. El enlazador sabrá cómo buscar el archivo de la librería y extraer el código compilado correcto. Sin embargo, ésta es sólo una forma de entregar una librería. En plataformas que abarcan muchas arquitecturas, como GNU o Unix, el único modo sensato de entregar una libraría es con código fuente para que así pueda ser reconfigurado y reconstruido en el nuevo objetivo.

De esta forma, las librerías probablemente sean la forma más importante de progresar en términos de productividad y uno de los principales objetivos del diseño de C++ es hacer más fácil el uso de librerías. Esto implica entonces, que hay algo difícil al usar librerías en C. Entender este factor le dará una primera idea sobre el diseño de C++, y por lo tanto, de cómo usarlo.

4.1. Una librería pequeña al estilo C

Aunque muchas veces, una librería comienza como una colección de funciones, si ha usado alguna librería C de terceros habrá observado que la cosa no termina ahí porque hay más que comportamiento, acciones y funciones. También hay características (azul, libras, textura, luminiscencia), las cuales están representadas por datos. En C, cuando debemos representar características, es muy conveniente agruparlas todas juntas en una *estructura*, especialmente cuando queremos representar más de un tipo de cosa en el problema. Así, se puede trabajar con una variable de esta *estructuras* para representar cada cosa.

Por eso, la mayoría de las librerías en C están formadas por un conjunto de estructuras y funciones que actúan sobre las primeras. Como ejemplo de esta técnica, considere una herramienta de programación que se comporta como un array, pero cuyo tamaño se puede fijar en tiempo de ejecución, en el momento de su creación. La llamaremos CStash [1]. Aunque está escrito en C++, tiene el estilo clásico de una librería escrita en C:

```
//: C04:CLib.h
// Header file for a C-like library
// An array-like entity created at runtime

typedef struct CStashTag {
  int size;      // Size of each space
  int quantity;  // Number of storage spaces
  int next;      // Next empty space
  // Dynamically allocated array of bytes:
  unsigned char* storage;
} CStash;

void initialize(CStash* s, int size);
void cleanup(CStash* s);
int add(CStash* s, const void* element);
void* fetch(CStash* s, int index);
int count(CStash* s);
void inflate(CStash* s, int increase);
///:~
```

Normalmente se utiliza un «rótulo» como CStashTag en aquellas estructuras que necesitan referenciarse dentro de si mismas. Ese es el caso de una *lista enlazada* (cada elemento de la lista contiene un puntero al siguiente elemento) se necesita un puntero a la siguiente variable estructura, o sea, una manera de identificar el tipo de ese puntero dentro del cuerpo de la propia estructura. En la declaración de las estructuras de una librería escrita en C también es muy común ver el uso de `typedef` como el del ejemplo anterior. Esto permite al programador tratar las estructuras como un nuevo tipo de dato y así definir nuevas variables (de esa estructura) del siguiente modo:

```
CStash A, B, C;
```

El puntero `storage` es un unsigned char*. Un unsigned char es la menor pieza

[1] N de T:«Stash» se podría traducir como «Acumulador».

de datos que permite un compilador C, aunque en algunas máquinas puede ser de igual tamaño que la mayor. Aunque es dependiente de la implementación, por lo general un unsigned char tiene un tamaño de un byte. Dado que CStash está diseñado para almacenar cualquier tipo de estructura, el lector se puede preguntar si no sería más apropiado un puntero void *. Sin embargo, el objetivo no es tratar este puntero de almacenamiento como un bloque de datos de tipo desconocido, sino como un bloque de bytes contiguos.

El archivo de código fuente para la implementación (del que no se suele disponer si fuese una librería comercial —normalmente sólo dispondrá de un `.obj`, `.lib` o `.dll`, etc) tiene este aspecto:

```
//: C04:CLib.cpp {O}
// Implementation of example C-like library
// Declare structure and functions:
#include "CLib.h"
#include <iostream>
#include <cassert>
using namespace std;
// Quantity of elements to add
// when increasing storage:
const int increment = 100;

void initialize(CStash* s, int sz) {
  s->size = sz;
  s->quantity = 0;
  s->storage = 0;
  s->next = 0;
}

int add(CStash* s, const void* element) {
  if(s->next >= s->quantity) //Enough space left?
    inflate(s, increment);
  // Copy element into storage,
  // starting at next empty space:
  int startBytes = s->next * s->size;
  unsigned char* e = (unsigned char*)element;
  for(int i = 0; i < s->size; i++)
    s->storage[startBytes + i] = e[i];
  s->next++;
  return(s->next - 1); // Index number
}

void* fetch(CStash* s, int index) {
  // Check index boundaries:
  assert(0 <= index);
  if(index >= s->next)
    return 0; // To indicate the end
  // Produce pointer to desired element:
  return &(s->storage[index * s->size]);
}

int count(CStash* s) {
  return s->next;  // Elements in CStash
}

void inflate(CStash* s, int increase) {
  assert(increase > 0);
```

```
  int newQuantity = s->quantity + increase;
  int newBytes = newQuantity * s->size;
  int oldBytes = s->quantity * s->size;
  unsigned char* b = new unsigned char[newBytes];
  for(int i = 0; i < oldBytes; i++)
    b[i] = s->storage[i]; // Copy old to new
  delete [](s->storage); // Old storage
  s->storage = b; // Point to new memory
  s->quantity = newQuantity;
}

void cleanup(CStash* s) {
  if(s->storage != 0) {
   cout << "freeing storage" << endl;
   delete []s->storage;
  }
} ///:~
```

`initialize()` realiza las operaciones iniciales necesarias para la struct CStash, poniendo los valores apropiados en las variables internas. Inicialmente, el puntero `storage` tiene un cero dado que aún no se ha almacenado nada.

La función `add()` inserta un elemento en el siguiente lugar disponible de la CStash. Para lograrlo, primero verifica que haya suficiente espacio disponible. Si no lo hay, expande el espacio de almacenamiento (`storage`) usando la función `inflate()` que se describe después.

Como el compilador no conoce el tipo específico de la variable que está siendo almacenada (todo lo que obtiene la función es un void*), no se puede hacer una asignación simple, que sería lo más conveniente. En lugar de eso, la variable se copia byte a byte. La manera más directa de hacerlo es utilizando el indexado de arrays. Lo habitual es que en `storage` ya haya bytes almacenados, lo cual es indicado por el valor de `next`. Para obtener la posición de inserción correcta en el array, se multiplica `next` por el tamaño de cada elemento (en bytes) lo cual produce el valor de `startBytes`. Luego el argumento `element` se moldea a unsigned char* para que se pueda direccionar y copiar byte a byte en el espacio disponible de `storage`. Se incrementa `next` de modo que indique el siguiente lugar de almacenamiento disponible y el «índice» en el que ha almacenado el elemento para que el valor se puede recuperar utilizando el índice con `fetch()`.

`fetch()` verifica que el índice tenga un valor correcto y devuelve la dirección de la variable deseada, que se calcula en función del argumento `index`. Dado que `index` es un desplazamiento desde el principio en la CStash, se debe multiplicar por el tamaño en bytes que ocupa cada elemento para obtener dicho desplazamiento en bytes. Cuando utilizamos este desplazamiento como índice del array `storage` lo que obtenemos no es la dirección, sino el byte almacenado. Lo que hacemos entonces es utilizar el operador dirección-de `&`.

`count()` puede parecer un poco extraña a los programadores experimentados en C. Podría parecer demasiado complicada para una tarea que probablemente sea mucho más fácil de hacer a mano. Por ejemplo, si tenemos una CStash llamada `intStash`, es mucho más directo preguntar por la cantidad de elementos utilizando `intStash.next`, que llamar a una función (que implica sobrecarga), como `count(&intStash)`. Sin embargo, la cantidad de elementos se calcula en función tanto del puntero `next` como del tamaño en bytes de cada elemento de la CStash; por eso la interfaz de la función `count()` permite la flexibilidad necesaria para no tener

que preocuparnos por estas cosas. Pero, ¡ay!, la mayoría de los programadores no se preocuparán por descubrir lo que para nosotros es el «mejor» diseño para la librería. Probablemente lo que harán es mirar dentro de la estructura y obtener el valor de `next` directamente. Peor aún, podrían incluso cambiar el valor de `next` sin nuestro permiso. ¡Si hubiera alguna forma que permitiera al diseñador de la librería tener un mejor control sobre este tipo de cosas! (Sí, esto es un presagio).

4.1.1. Asignación dinámica de memoria

Nunca se puede saber la cantidad máxima de almacenamiento que se necesitará para una CStash, por eso la memoria a la que apuntan los elementos de `storage` se asigna desde el *montículo* (*heap*) [2]. El montículo es un gran bloque de memoria que se utiliza para asignar en pequeños trozos en tiempo de ejecución. Se usa el heap cuando no se conoce de antemano la cantidad de memoria que necesitará el programa que está escribiendo. Por ejemplo, eso ocurre en un programa en el que sólo en el momento de la ejecución se sabe si se necesia memoria para 200 variables Avión o para 20. En C Estándar, las funciones para asignación dinámica de memoria incluyen `malloc()`, `calloc()`, `realloc()` y `free()`. En lugar de llamadas a librerías, C++ cuenta con una técnica más sofisticada (y por lo tanto más fácil de usar) para tratar la memoria dinámica. Esta técnica está integrada en el lenguaje por medio de las palabras reservadas `new` y `delete`.

La función `inflate()` usa `new` para obtener más memoria para la CStash. En este caso el espacio de memoria sólo se amplia y nunca se reduce. `assert()` garantiza que no se pase un número negativo como argumento a `inflate()` como valor de incremento. La nueva cantidad de elmentos que se podrán almacenar (una vez se haya terminado `inflate()`) se determina en la variable `newQuantity` que se multiplica por el número de bytes que ocupa cada elemento, para obtener el nuevo número total de bytes de la asignación en la variable `newBytes`. Dado que se sabe cuántos bytes hay que copiar desde la ubicación anterior, `oldBytes` se calcula usando la cantidad antigua de bytes (`quantity`).

La petición de memoria ocurre realmente en la *expresión-new* que involucra la palabra reservada `new`:

```
new unsigned char[newBytes];
```

La forma general de una *expresión-new* es:

```
new Tipo;
```

donde Tipo describe el tipo de variable para la cual se solicita memoria en el *montículo*. Dado que en este caso, se desea asignar memoria para un array de unsigned char de `newBytes` elementos, eso es lo que aparece como Tipo. Del mismo modo, se puede asignar memoria para algo más simple como un int con la expresión:

```
new int;
```

y aunque esto se utiliza muy poco, demuestra que la sintaxis es consistente.

Una *expresión-new* devuelve un *puntero* a un objeto del tipo exacto que se le pidió.

[2] N. de T.: *heap* se suele traducir al castellano como «montón» o «montículo».

De modo que con `new Tipo` se obtendrá un puntero a un objeto de tipo Tipo, y con `new int` obtendrá un puntero a un int. Si quiere un nuevo array de unsigned char la expresión devolverá un puntero al primer elemento de dicho array. El compilador verificará que se asigne lo que devuelve la *expresión-new* a una variable puntero del tipo adecuado.

Por supuesto, es posible que al pedir memoria, la petición falle, por ejemplo, si no hay más memoria libre en el sistema. Como verá más adelante, C++ cuenta con mecanismos que entran en juego cuando la operación de asignación de memoria no se puede satisfacer.

Una vez que se ha obtenido un nuevo espacio de almacenamiento, los datos que estaban en el antiguo se deben copiar al nuevo. Esto se hace, nuevamente, en un bucle, utilizando la notación de índexado de arrays, copiando un byte en cada iteración del bucle. Una vez finalizada esta copia, ya no se necesitan los datos que están en el espacio de almacenamiento original por lo que se pueden liberar de la memoria para que otras partes del programa puedan usarlo cuando lo necesiten. La palabra reservada `delete` es el complemento de `new` y se debe utilizar sobre todas aquellas variables a las cuales se les haya asignado memoria con `new`. (Si se olvida de utilizar `delete` esa memoria queda in-utilizable. Si estas fugas de memoria (*memory leak*) son demasiado abundantes, la memoria disponible se acabará.) Existe una sintaxis especial cuando se libera un array. Es como si recordara al compilador que ese puntero no apunta sólo a un objeto, sino a un array de objetos; se deben poner un par de corchetes delante del puntero que se quiere liberar:

```
delete []myArray;
```

Una vez liberado el antiguo espacio de almacenamiento, se puede asignar el puntero del nuevo espacio de memoria al puntero `storage`, se actualiza `quantity` y con eso `inflate()` ha terminado su trabajo.

En este punto es bueno notar que el administrador de memoria del montículo> es bastante primitivo. Nos facilita trozos de memoria cuando se lo pedimos con `new` y los libera cuando invocamos a `delete`. Si un programa asigna y libera memoria muchas veces, terminaremos con un montículo *fragmentado*, es decir un montículo en el que si bien puede haber memoria libre utilizable, los trozos de memoria están divididos de tal modo que no exista un trozo que sea lo suficientemente grande para las necesidades concretas en un momento dado. Lamentablemente no existe una capacidad inherente del lenguaje para efectuar *defragmentaciones del montículo*. Un defragmentador del montículo complica las cosas dado que tiene que mover pedazos de memoria, y por lo tanto, hacer que los punteros dejen de apuntar a valores válidos. Algunos entornos operativos vienen con este tipo de facilidades pero obligan al programador a utilizar manejadores de memoria especiales en lugar de punteros (estos manipuladores se pueden convertir temporalmente en punteros una vez bloqueada la memoria para que el defragmentador del montículo no la modifique). También podemos construir nosotros mismos uno de estos artilugios, aunque no es una tarea sencilla.

Cuando creamos una variable en la pila en tiempo de compilación, el mismo compilador es quien se encarga de crearla y liberar la memoria ocupada por ella automáticamente. Conoce exactamente el tamaño y la duración de este tipo de variables dada por las reglas de ámbito. Sin embargo, en el caso de las variables almacenadas dinámicamente, el compilador no poseerá información ni del tamaño requerido por las mismas, ni de su duración. Esto significa que el compilador no puede encargarse de liberar automáticamente la memoria ocupada por este tipo de variables y de aquí

que el responsable de esta tarea sea el programador (o sea usted). Para esto se debe utilizar `delete`, lo cual le indica al administrador del montículo que ese espacio de memoria puede ser utilizado por próximas llamadas a `new`. En nuestra librería de ejemplo, el lugar lógico para esta tarea es la función `cleanup()` dado que allí es dónde se deben realizar todas las labores de finalización de uso del objeto.

Para probar la librería se crean dos Cstash, uno que almacene enteros y otro para cadenas de 80 caracteres:

```
//: C04:CLibTest.cpp
//{L} CLib
// Test the C-like library
#include "CLib.h"
#include <fstream>
#include <iostream>
#include <string>
#include <cassert>
using namespace std;

int main() {
  // Define variables at the beginning
  // of the block, as in C:
  CStash intStash, stringStash;
  int i;
  char* cp;
  ifstream in;
  string line;
  const int bufsize = 80;
  // Now remember to initialize the variables:
  initialize(&intStash, sizeof(int));
  for(i = 0; i < 100; i++)
    add(&intStash, &i);
  for(i = 0; i < count(&intStash); i++)
    cout << "fetch(&intStash, " << i << ") = "
         << *(int*)fetch(&intStash, i)
         << endl;
  // Holds 80-character strings:
  initialize(&stringStash, sizeof(char)*bufsize);
  in.open("CLibTest.cpp");
  assert(in);
  while(getline(in, line))
    add(&stringStash, line.c_str());
  i = 0;
  while((cp = (char*)fetch(&stringStash,i++))!=0)
    cout << "fetch(&stringStash, " << i << ") = "
         << cp << endl;
  cleanup(&intStash);
  cleanup(&stringStash);
} ///:~
```

Dado que debemos respetar la sintaxis de C, todas las variables se deben declarar al comienzo de `main()`. Obviamente, no nos podemos olvidar de inicializar todas las variables Cstash más adelante en el bloque main(), pero antes de usarlas, llamando a `initialize()`. Uno de los problemas con las librerías en C es que uno debe asegurarse de convencer al usuario de la importancia de las funciones de inicialización y destrucción. ¡Habrá muchos problemas si estas funciones se omiten! Lamen-

tablemente el usuario no siempre se preguntará si la inicialización y el limpiado de los objetos son obligatorios. Ellos le darán importancia a lo que *ellos* quieren hacer y no nos darán tanta importancia a nosotros (el programador de la librería) cuando les digamos «¡Hey! ¡espera un poco! ¡Debes hacer *esto* primero!». Otro problema que puede presentarse es el hecho de que algunos usuarios quieran inicializar los elementos (datos internos) de una estructura por su cuenta. En C no hay un mecanismo para prevenir este tipo de conductas (más presagios de los temás que vendrán...).

La `intStash` se va llenando con enteros mientras que el `stringStash` se va llenando con arrays de caracteres. Estos arrays de caracteres son producidos leyendo el archivo fuente `CLibTest.cpp` y almacenando las líneas de este archivo en el string `line`. Obtenemos la representación «puntero a carácter» de `line` con el método `c_str()`.

Una vez cargados los Stash ambos se muestran en pantalla. `intStash` se imprime usando un bucle `for` en el cual se usa `count()` para determinar la cantidad de elementos. El `stringStash` se muestra utilizando un bucle `while` dentro del cual se va llamando a `fetch()`. Cuando esta función devuelve cero se rompe el bucle ya que esto significará que se han sobrepasado los límites de la estructura.

El lector también pudo haber visto un molde adicional en la línea:

```
cp = (char*)fetch(&stringStash, i++)
```

Esto se debe a la comprobación estricta de tipos en C++, que no permite asignar un void * a una variable de cualquier tipo, mientras que C sí lo hubiera permitido.

4.1.2. Malas suposiciones

Antes de abordar los problemas generales de la creación de una librería C, discutiremos otro asunto importante que se debe tener claro. Fíjese que el archivo de cabecera `CLib.h` *debe* incluirse en cada archivo fuente que haga referencia al tipo CStash ya que el compilador no puede adivinar qué aspecto tiene la estructura. Sin embargo, *sí* puede adivinar el aspecto de una función. Aunque eso pueda parecer una ventaja, veremos que en realidad, es un grave problema de C.

Aunque siempre debería declarar las funciones incluyendo un archivo de cabecera, en C las declaraciones de funciones no son esenciales. En este lenguaje (pero no en C++), es posible llamar a una función que no ha sido declarada. Un buen compilador seguramente avisará de que deberíamos declarar la función antes de usarla, pero nos permitirá seguir dado que no es obligatorio hacerlo en C estándar. Esta es una práctica peligrosa ya que el compilador puede asumir que una función que ha sido llamada con un int como argumento, tenga un int como argumento cuando, en realidad, es un float. Como veremos, esto puede producir errores que pueden ser muy difíciles de depurar.

Se dice que cada archivo de implementación C (los archivos de extensión `.c`) es una unidad de traducción (*translation unit*). El compilador se ejecuta independientemente sobre cada unidad de traducción ocupándose, en ese momento, solamente en ese archivo. Por eso, la información que le demos al compilador por medio de los archivos de cabecera es muy importante dado que determina la forma enq que ese archivo se relaciona con las demás partes del programa. Por eso motivo, las declaraciones en los archivos de cabecera son particularmente importantes dado que, en cada lugar que se incluyen, el compilador sabrá exactamente qué hacer. Por ejemplo, si en un archivo de cabecera tenemos la declaración `void func(float)` , si

llamamos a `func()` con un int como argumento, el compilador sabrá que deberá convertir el int a float antes de pasarle el valor a la función (a esto se le llama *promoción* de tipos). Sin la declaración, el compilador asumiría que la función tiene la forma `func(int)`, no realizaría la promoción y pasaría, por lo tanto, datos incorrectos a la función.

Para cada unidad de traducción, el compilador crea un archivo objeto, de extensión `.o`, `.obj` o algo por el estilo. Estos archivos objeto, junto con algo de código de arranque se unens por el enlazador(*linker*) para crear el programa ejecutable. Todas las referencias externas se deben resolver en la fase de enlazado. En archivos como `CLibTest.cpp`, se declaran funciones como `initialize()` y `fetch()` (o sea, se le informa al compilador qué forma tienen estas funciones), pero no se definen. Están definidas en otro lugar, en este caso en el archivo `CLib.cpp`. De ese modo, las llamadas que se hacen en `CLibTest.cpp` a estas funciones son referencias externas. Cuando se unen los archivos objeto para formar el programa ejecutable, el enlazador debe, para cada referencia externa no resuelta, encontrar la dirección a la que hace referencia y reemplazar cada referencia externa con su dirección correspondiente.

Es importante señalar que en C, estas referencias externas que el enlazador busca son simples nombres de funciones, generalmente precedidos por un guión bajo. De esta forma, la única tarea del enlazador es hacer corresponder el nombre de la función que se llama, con el cuerpo (definición, código) de la función del archivo objeto, en el lugar exacto de la llamada a dicha función. Si, por ejemplo, accidentalmente hacemos una llamada a una función que el compilador interprete como `func(int)` y existe una definición de función para `func(float)` en algún archivo objeto, el enlazador verá `_func` en un lugar y `_func` en otro, por lo que *pensará* que todo está bien. En la llamada a `func()` se pasará un int en la pila pero el cuerpo de la función `func()` esperará que la pila tenga un float. Si la función sólo lee el valor de este dato y no lo escribe, la pila no sufrirá datos. De hecho, el supuesto float leído de la pila puede tener algo de sentido: la función seguirá funcionando aunque sobre basura, y es por eso que los fallos originadas por esta clase de errores son muy difíciles de encontrar.

4.2. ¿Qué tiene de malo?

Somos seres realmente destinados a la adaptación, incluso a las que quizá *no deberíamos* adaptarnos. El estilo de la librería CStash ha sido un modelo a seguir para los programadores en C durante mucho tiempo. Sin embargo, si nos ponemos a examinarla por un momento, nos daremos cuenta de que utilizar esta librería puede resultar incómodo. Cuando la usamos debemos, por ejemplo, pasar la dirección de la estructura a cada función de la librería. Por eso, cuando leemos el código, los mecanismos de la librería se mezclan con el significado de las llamadas a las funciones, lo cual dificulta la comprecsión del programa.

Sin embargo, uno de los mayores obstáculos al trabajar con librerías en C es el problema llamado *conflicto de nombres* (*name clashes*). C trabaja con un único espacio de nombres de funciones. Esto significa que, cuando el enlazador busca por el nombre de una función, lo hace en una única lista de nombres maestra. Además, cuando el compilador trabaja sobre una unidad de traducción, un nombre de función sólo puede hacer referencia a una única función con ese nombre.

Supongamos que compramos dos librerías de diferentes proveedores y que cada librería consta de una estructura que debe inicializar y destruir. Supongamos que cada proveedor ha decidido nombrar a dichas operaciones `initialize()` y `cleanup()`. ¿Cómo se comportaría el compilador si incluyéramos los archivos de cabecera

de ambas librerías en la misma unidad de traducción? Afortunadamente, el compilador C dará un mensaje de error diciéndonos que hay una incoherencia de tipos en las listas de argumentos de ambas declaraciones. No obstante, aunque no incluyamos los archivos de cabecera en la unidad de traducción igual tendremos problemas con el enlazador. Un buen enlazador detectará y avisará cuando se produzca uno de estos conflictos de nombres. Sin embargo, hay otros que simplemente tomarán el primer nombre de la función que encuentren, buscando en los archivos objeto en el orden en el que fueron pasados en la lista de enlazado. (Este comportamiento se puede considerar como una ventaja ya que permite reemplazar las funciones de las librerías ajenas con funciones propias.)

En cualquiera de los dos casos, llegamos a la conclusión de que en C es imposible usar dos bibliotecas en las cuales existan funciones con nombres idénticos. Para solucionar este problema, los proveedores de librerías en C ponen un prefijo único a todas las funciones de la librería. En nuestro ejemplo, las funciones `initialize()` y `cleanup()` habría que renombrarlas como `CStash_initialize()` y `CStash_cleanup()`. Esta es una técnica lógica: decoramos los nombres de las funciones con el nombre de la estructura sobre la cual trabajan.

Este es el momento de dirigir nuestros pasos a las primeras nociones de construcción de clases en C++. Como el lector ha de saber, las variables declaradas dentro de una estructura no tienen conflictos de nombres con las variables globales. ¿Por qué, entonces, no aprovechar esta característica de las variables para evitar los conflictos de nombres de funciones declarándolas dentro de la estructura sobre la cual operan? O sea, ¿por qué no hacer que las funciones sean también miembros de las estructuras?

4.3. El objeto básico

Nuestro primer paso será exactamente ese. Meter las funciones C++ dentro de las estructuras como «funciones miembro». Éste es el aspecto que tiene la estructura una vez realizados estos cambios de la versión C de la CStash a la versión en C++, a la que llamaremos Stash:

```
//: C04:CppLib.h
// C-like library converted to C++

struct Stash {
  int size;      // Size of each space
  int quantity;  // Number of storage spaces
  int next;      // Next empty space
   // Dynamically allocated array of bytes:
  unsigned char* storage;
  // Functions!
  void initialize(int size);
  void cleanup();
  int add(const void* element);
  void* fetch(int index);
  int count();
  void inflate(int increase);
}; ///:~
```

La primera diferencia que puede notarse es que no se usa `typedef`. A diferencia de C que requiere el uso de `typedef` para crear nuevos tipos de datos, el compila-

dor de C++ hará que el nombre de la estructura sea un nuevo tipo de dato automáticamente en el programa (tal como los nombres de tipos de datos int, char, float y double).

Todos los datos miembros de la estructura están declarados igual que antes; sin embargo, ahora las funciones están declaradas dentro del cuerpo de la `struct`. Más aún, fíjese que el primer argumento de todas las funciones ha sido eliminado. En C++, en lugar de forzar al usuario a que pase la dirección de la estructura sobre la que trabaja una función como primer argumento, el compilador hará este trabajo, secretamente. Ahora sólo debe preocuparse por los argumentos que le dan sentido a lo que la función *hace* y no de los mecanismos internos de la función.

Es importante darse cuenta de que el código generado por estas funciones es el mismo que el de las funciones de la librería al estilo C. El número de argumentos es el mismo (aunque no se le pase la dirección de la estructura como primer argumento, en realidad sí se hace) y sigue existiendo un único cuerpo (definición) de cada función. Esto último quiere decir que, aunque declare múltiples variables

```
Stash A, B, C;
```

no existirán múltiples definiciones de, por ejemplo, la función `add()`, una para cada variable.

De modo que el código generado es casi idéntico al que hubiese escrito para una versión en C de la librería, incluyendo la «decoración de nombres» ya mencionada para evitar los conflictos de nombres, nombrando a las funciones `Stash_initialize()`, `Stash_cleanup()` y demás. Cuando una función está dentro de una estructura, el compilador C++ hace lo mismo y por eso, una función llamada `initialize()` dentro de una estructura no estará en conflicto con otra función `initialize()` dentro de otra estructura o con una función `initialize()` global. De este modo, en general no tendrá que preocuparse por los conflictos de nombres de funciones - use el nombre sin decoración. Sin embargo, habrá situaciones en las que deseará especificar, por ejemplo, esta `initialize()` pertenece a la estructura Stash y no a ninguna otra. En particular, cuando defina la función, necesita especificar a qué estructura pertenece para lo cual, en C++ cuenta con el operador `::` llamado operador de resolución de ámbito (ya que ahora un nombre puede estar en diferentes ámbitos: el del ámbito global o dentro del ámbito de una estructura. Por ejemplo, si quiere referirse a una función `initialize()` que se encuentra dentro de la estructura Stash lo podrá hacer con la expresión `Stash::initialize(int size)`. A continuación podrá ver cómo se usa el operador de resolución de ámbito para definir funciones:

```
//: C04:CppLib.cpp {O}
// C library converted to C++
// Declare structure and functions:
#include "CppLib.h"
#include <iostream>
#include <cassert>
using namespace std;
// Quantity of elements to add
// when increasing storage:
const int increment = 100;

void Stash::initialize(int sz) {
  size = sz;
  quantity = 0;
```

```
  storage = 0;
  next = 0;
}

int Stash::add(const void* element) {
  if(next >= quantity) // Enough space left?
    inflate(increment);
  // Copy element into storage,
  // starting at next empty space:
  int startBytes = next * size;
  unsigned char* e = (unsigned char*)element;
  for(int i = 0; i < size; i++)
    storage[startBytes + i] = e[i];
  next++;
  return(next - 1); // Index number
}

void* Stash::fetch(int index) {
  // Check index boundaries:
  assert(0 <= index);
  if(index >= next)
    return 0; // To indicate the end
  // Produce pointer to desired element:
  return &(storage[index * size]);
}

int Stash::count() {
  return next; // Number of elements in CStash
}

void Stash::inflate(int increase) {
  assert(increase > 0);
  int newQuantity = quantity + increase;
  int newBytes = newQuantity * size;
  int oldBytes = quantity * size;
  unsigned char* b = new unsigned char[newBytes];
  for(int i = 0; i < oldBytes; i++)
    b[i] = storage[i]; // Copy old to new
  delete []storage; // Old storage
  storage = b; // Point to new memory
  quantity = newQuantity;
}

void Stash::cleanup() {
  if(storage != 0) {
    cout << "freeing storage" << endl;
    delete []storage;
  }
} ///:~
```

Hay muchas otras cosas que difieres entre C y C++. Para empezar, el compilador *requiere* que declare las funciones en los archivos de cabecera: en C++ no podrá llamar a una función sin haberla declarado antes y si no se cumple esta regla el compilador dará un error. Esta es una forma importante de asegurar que las llamadas a una función son consistentes entre el punto en que se llama y el punto en que se define. Al forzar a declarar una función antes de usarla, el compilador de C++ prác-

ticamente se asegura de que realizará esa declaración por medio de la inclusión de un fichero de cabecera. Además, si también incluye el mismo fichero de cabecera en el mismo lugar donde se defines las funciones, el compilador verificará que las declaraciones del archivo cabecera y las definiciones coinciden. Puede decirse entonces que, de algún modo, los ficheros de cabecera se vuelven un repositorio de validación de funciones y permiten asegurar que las funciones se usan de modo consistente en todas las unidades de traducción del proyecto.

Obviamente, las funciones globales se pueden seguir declarando a mano en aquellos lugares en las que se definen y usan (Sin embargo, esta práctica es tan tediosa que está en desuso.) De cualquier modo, las estructuras siempre se deben declarar antes de ser usadas y el mejor lugar para esto es un fichero de cabecera, exceptuando aquellas que queremos esconder intencionalmente en otro fichero.

Se puede ver que todas las funciones miembro (métodos) tienen casi la misma forma que sus versiones respectivas en C. Las únicas diferencias son su ámbito de resolución y el hecho de que el primer argumento ya no aparece explícito en el prototipo de la función. Por supuesto que sigue ahí ya que la función debe ser capaz de trabajar sobre una variable `struct` en particular. Sin embargo, fíjese también que, dentro del método, la selección de esta estructura en particular también ha desaparecido! Así, en lugar de decir `s->size = sz;` ahora dice `size = sz;` eliminando el tedioso `s->` que en realidad no aportaba nada al significado semántico de lo que estaba escribiendo. Aparentemente, el compilador de C++ está realizando estas tareas por el programador. De hecho, está tomando el primer argumento «secreto» (la dirección de la estructura que antes tenía que pasar a mano) y aplicándole el selector de miembro (->) siempre que escribe el nombre de uno de los datos miembro. Eso significa que, siempre y cuando esté dentro de la definición de una método de una estructura puede hacer referencia a cualquier otro miembro (incluyendo otro método) simplemente dando su nombre. El compilador buscará primero en los nombres locales de la estructura antes de buscar en versiones más globales de dichos nombres. El lector podrá descubrir que esta característica no sólo agiliza la escritura del código, sino que también hace la lectura del mismo mucho más sencilla.

Pero qué pasaría si, por alguna razón, *quisiera* hacer referencia a la dirección de memoria de la estructura. En la versión en C de la librería ésta se podía obtener fácilmente del primer argumento de cualquier función. En C++ la cosa es más consistente: existe la palabra reservada `this` que produce la dirección de la variable `struct` actual. Es el equivalente a la expresión `s` de la versión en C de la librería. De modo que, podremos volver al estilo de C escribiendo

```
this->size = Size;
```

El código generado por el compilador será exactamente el mismo por lo que no es necesario usar `this` en estos casos. Ocasionalmente, podrá ver por ahí código dónde la gente usa `this` en todos sitios sin agregar nada al significado del código (esta práctica es indicio de programadores inexpertos). Por lo general, `this` no se usa muy a menudo pero, cuando se necesite siempre estará allí (en ejemplos posteriores del libro verá más sobre su uso).

Queda aún un último tema que tocar. En C, se puede asignar un void * a cualquier otro puntero, algo como esto:

```
int i = 10;
void* vp = &i; // OK tanto en C como en C++
int* ip = vp; // óSlo aceptable en C
```

y no habrá ningún tipo de queja por parte de compilador. Sin embargo, en C++, lo anterior no está permitido. ¿Por qué? Porque C no es tan estricto con los tipos de datos y permite asignar un puntero sin un tipo específico a un puntero de un tipo bien determinado. No así C++, en el cual la verificación de tipos es crítica y el compilador se detendrá quejándose en cualquier conflicto de tipos. Esto siempre ha sido importante, pero es especialmente importante en C++ ya que dentro de las estructuras puede hacer métodos. Si en C++ estuviera permitido pasar punteros a estructuras con impunidad en cuanto a conflicto de tipos, ¡podría terminar llamando a un método de una estructura en la cual no existiera dicha función miembro! Una verdadera fórmula para el desastre. Así, mientras C++ sí deja asignar cualquier puntero a un void * (en realidad este es el propósito original del puntero a void: que sea suficientemente largo como para apuntar a cualquier tipo) no permite asignar un void * a cualquier otro tipo de puntero. Para ello se requiere un molde que le indique tanto al lector como al compilador que realmente quiere tratarlo como el puntero destino.

Y esto nos lleva a discutir un asunto interesante. Uno de los objetivos importantes de C++ es poder compilar la mayor cantidad posible de código C para así, permitir una fácil transición al nuevo lenguaje. Sin embargo, eso no significa, como se ha visto que cualquier segmento de código que sea válido en C, será permitido automáticamente en C++. Hay varias cosas que un compilador de C permite hacer que son potencialmente peligrosas y propensas a generar errores (verá ejemplos de a lo largo de libro). El compilador de C++ genera errores y avisos en este tipo de situaciones y como verá eso es más una ventaja que un obstáculo a pesar de su naturaleza restrictiva. ¡De hecho, existen muchas situaciones en las cuales tratará de detectar sin éxito un error en C y cuando recompiles el programa con un compilador de C++ éste avisa exactamente de la causa del problema!. En C, muy a menudo ocurre que para que un programa funcione correctamente, además de compilarlo, luego debe *hacer que ande*. ¡En C++, por el contrario, verá que muchas veces si un programa compila correctamente es probable que funcione bien! Esto se debe a que este último lenguaje es mucho más estricto respecto a la comprobación de tipos.

En el siguiente programa de prueba podrá apreciar cosas nuevas con respecto a cómo se utiliza la nueva versión de la Stash:

```
//: C04:CppLibTest.cpp
//{L} CppLib
// Test of C++ library
#include "CppLib.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
  Stash intStash;
  intStash.initialize(sizeof(int));
  for(int i = 0; i < 100; i++)
    intStash.add(&i);
  for(int j = 0; j < intStash.count(); j++)
    cout << "intStash.fetch(" << j << ") = "
         << *(int*)intStash.fetch(j)
         << endl;
  // Holds 80-character strings:
  Stash stringStash;
  const int bufsize = 80;
```

```
  stringStash.initialize(sizeof(char) * bufsize);
  ifstream in("CppLibTest.cpp");
  assure(in, "CppLibTest.cpp");
  string line;
  while(getline(in, line))
    stringStash.add(line.c_str());
  int k = 0;
  char* cp;
  while((cp =(char*)stringStash.fetch(k++)) != 0)
    cout << "stringStash.fetch(" << k << ") = "
         << cp << endl;
  intStash.cleanup();
  stringStash.cleanup();
} ///:~
```

Una de las cosas que el lector habrá podido observar en el código anterior es que las variables se definen «al vuelo», o sea (como se introdujo en el capítulo anterior) en cualquier parte de un bloque y no necesariamente -como en C- al comienzo de los mismos.

El código es bastante similar al visto en `CLibTest.cpp` con la diferencia de que, cuando se llama a un método, se utiliza el operador de selección de miembro `'.'` precedido por el nombre de la variable. Esta es una síntaxis conveniente ya que imita a la selección o acceso de un dato miembro de una estructura. La única diferencia es que, al ser un método, su llamada implica una lista de argumentos.

Tal y cómo se dijo antes, la llamada que el compilador hace genera *realmente* es mucho más parecida a la llamada a la función de la librería en C. Considere la decoración de nombres y el paso del puntero `this`: la llamada en C++ de `intStash.initialize(sizeof(int), 100)` se transformará en algo parecido a `Stash_initialize(&intStash, sizeof(int), 100)`. Si el lector se pregunta qué es lo que sucede realmente debajo del envoltorio, debería recordar que el compilador original de C++ cfront de AT&T producía código C como salida que luego debía ser compilada con un compilador de C para generar el ejecutable. Este método permitía a cfront ser rápidamente portable a cualquier máquina que soportara un compilador estándar de C y ayudó a la rápida difusión de C++. Dado que los compiladores antiguos de C++ tenían que generar código C, sabemos que existe una manera de representar síntaxis C++ en C (algunos compiladores de hoy en día aún permiten generar código C).

Comparando con `CLibTest.cpp` observará un cambio: la introducción del fichero de cabecera `require.h`. He creado este fichero de cabecera para realizar una comprobación de errores más sofisticada que la que proporciona `assert()`. Contiene varias funciones incluyendo la llamada en este último ejemplo, `assure()` que se usa sobre ficheros. Esta función verifica que un fichero se ha abierto exitosamente y en caso contrario reporta un aviso a la salida de error estándar (por lo que también necesita el nombre del fichero como segundo argumento) y sale del programa. Las funciones de `require.h` se usan a lo largo de este libro especialmente para asegurar que se ha indicado la cantidad correcta de argumentos en la línea de comandos y para verificar que los ficheros se abren correctamente. Las funciones de `require.h` reemplazan el código de detección de errores repetitivo y que muchas veces es causa de distracciones y más aún, proporcionan mensajes útiles para la detección de posibles errores. Estas funciones se explican detalladamente más adelante.

4.4. ¿Qué es un objeto?

Ahora que ya se ha visto y discutido un ejemplo incial es hora de retroceder para definir la terminología. El acto de introducir funciones en las estructuras es el eje central del cambio que C++ propone sobre C, e eso introduce una nueva forma de ver las estructuras: como conceptos. En C, una estructura (`struct`) es tan sólo una agrupación de datos: una manera de empaquetar datos para que se puedan tratar como un grupo. De esta forma, cuesta hacerse a la idea de que representan algo más que una mera conveniencia de programación. Las funciones que operan sobre esas estructuras están sueltas por ahí. Sin embargo, con las funciones dentro del mismo paquete que los datos, la estructura se convierte en una nueva criatura, capaz de representar las características (como hacen las `struct`s de C) *y* los comportamientos. El concepto de objeto, una entidad independiente y bien limitada que puede recordar *y* actuar, se sugiere a si mismo como definición.

En C++, un objeto es simplemente una variable, y la definición más purista es «una región de almacenamiento» (que es una forma más específica para decir «un objeto debe tener un único identificador» el cual, en el caso de C++, es una dirección única de memoria). Es un lugar en el cual se pueden almacenar datos y eso implica también operaciones que pueden actuar sobre esos datos.

Desafortunadamente no existe una consistencia completa entre los distintos lenguajes cuando se habla de estos términos, aunque son aceptados bastante bien. También se podrán encontrar discrepancias sobre lo que es un lenguaje orientado a objetos, aunque parece haber un consenso razonable hoy en día. Hay lenguajes *basados en objetos*, que cuentan con estructuras-con-funciones como las que ha visto aquí de C++. Sin embargo, esto es tan sólo una parte de lo que denomina un lenguaje *orientado a objetos*, y los lenguajes que solamente llegan a empaquetar las funciones dentro de las estructuras son lenguajes basados en objetos y no orientados a objetos.

4.5. Tipos abstractos de datos

La habilidad para empaquetar datos junto con funciones permite la creación de nuevos tipos de datos. Esto se llama a menudo *encapsulación* [3] Un tipo de dato existente puede contener varias piezas de datos empaquetadas juntas. Por ejemplo, un float tiene un exponente, una mantissa y un bit de signo. Le podemos pedir que haga varias cosas: sumarse a otro float o a un int, etc. Tiene características y comportamiento.

La definición de Stash crea un nuevo tipo de dato. Se le pueden agregar nuevos elementos (`add()`), sacar (`fetch()`) y agrandarlo (`inflate()`). Se puede crear uno escribiendo `Stash s;` igual que cuando se crea un float diciendo `float x;`. Un Stash también tiene características y un comportamiento bien determinado. Aunque actúe igual que un tipo de dato predefinido como float se dice que Stash es un *tipo abstracto de dato* tal vez porque permite abstraer un concepto desde el espacio de los problemas al espacio de la solución. Además, el compilador de C++ lo tratará exactamente como a un nuevo tipo de dato y si, por ejemplo, declara una función que acepta un Stash como argumento, el compilador se asegurará de que no se le pase otra cosa a la función. De modo que se realiza el mismo nivel de comprobación de tipos tanto para los tipos abstractos de datos (a veces también llamados *tipos definidos por el usuario*) como para los tipos predefinidos.

[3] Este término puede causar debates. Algunas personas lo utilizan tal y como está definido aquí, aunque otras lo usan para describir el *control de acceso*, término que se discutirá en el siguiente capítulo.

Sin embargo, notará inmediatamente una diferencia en la forma en que se realizan las operaciones sobre los objetos. Se hace `objeto.funciónMiembro(listaArgumentos)` o sea, «se llama a un método de un objeto». Pero en la jerga de la orientación a objetos, eso también se denomina «enviar un mensaje a un objeto». De modo que para una `Stash s`, en esta jerga la sentencia `s.add(&i)` le «envía un mensaje a `s`» diciéndole «añadete (`add()`) esto». De hecho, la programación orientada a objetos se puede resumir en la siguiente frase: *enviar mensajes a objetos*. Realmente, ¿eso es todo lo que se hace? crear un montón de objetos y enviarles mensajes. El truco, obviamente, es entender qué *son* en nuestro problema los objetos y los mensajes, pero una vez que se ha cumplido esa etapa, la implementación en C++ será sorprendentemente directa.

4.6. Detalles del objeto

Una pregunta que surge a menudo en seminarios es «¿Cómo de grande es un objeto y qué pinta tiene?» La respuesta es «más o menos lo que esperas de un `struct` en C». De hecho, el código que produce el compilador de C para un `struct` C (sin adornos C++) normalmente es *exactamente* el mismo que el producido por un compilador C++. Eso tranquiliza a aquellos programadores C que dependan de los detalles de tamaño y distribución de su código, y que por alguna razón accedan directamente a los bytes de la estructura en lugar de usar identificadores (confiar en un tamaño y distribución particular para una estructura no es portable).

El tamaño de una `struct` es la combinación de los tamaños de todos sus miembros. A veces cuando el compilador crea una `struct`, añade bytes extra para hacer que los límites encajen limpiamente - eso puede incrementar la eficiencia de la ejecución. En el Capítulo 14, verá cómo en algunos casos se añaden punteros «secretos» a la estructura, pero no tiene que preocuparse de eso ahora.

Puede determinar el tamaño de una `struct` usando el operador `sizeof`. Aquí tiene un pequeño ejemplo:

```
//: C04:Sizeof.cpp
// Sizes of structs
#include "CLib.h"
#include "CppLib.h"
#include <iostream>
using namespace std;

struct A {
  int i[100];
};

struct B {
  void f();
};

void B::f() {}

int main() {
  cout << "sizeof struct A = " << sizeof(A)
       << " bytes" << endl;
  cout << "sizeof struct B = " << sizeof(B)
       << " bytes" << endl;
  cout << "sizeof CStash in C = "
```

```
        << sizeof(CStash) << " bytes" << endl;
  cout << "sizeof Stash in C++ = "
        << sizeof(Stash) << " bytes" << endl;
} ///:~
```

En mi máquina (los resultados pueden variar) el primer resultado produce 200 porque cada int ocupa 2 bytes. La `struct` B es algo anómalo porque es una `struct` sin atributos. En C, eso es ilegal, pero en C++ necesitamos la posibilidad de crear una `struct` cuya única tarea es ofrecer un ámbito a nombres de funciones, por eso está permitido. Aún asi, el segundo resultado es un sorprendente valor distinto de cero. En versiones anteriores del lenguage, el tamaño era cero, pero aparecia una situación incómoda cuando se creaban estos objetos: tenían la misma dirección que el objeto creado antes que él, y eran indistinguibles. Una de las reglas fundamentales de los objetos es que cada objeto debe tener una dirección única, así que las estructuras sin atributos siempre tendrán tamaño mínimo distinto de cero.

Las dos últimas sentencias `sizeof` muestran que el tamaño de la estructura en C++ es el mismo que en la versión en C. C++ intenta no añadir ninguna sobrecarga innecesaria.

4.7. Conveciones para los ficheros de cabecera

Cuando se crea una `struct` que contiene funciones miembro, se está creando un nuevo tipo de dato. En general, se intenta que ese tipo sea fácilmente accesible. En resumen, se quiere que la interfaz (la declaración) esté separada de la implmentación (la definición de los métodos) de modo que la implementación pueda cambiar sin obligar a recompilar el sistema completo. Eso se consigue poniendo la declaración del nuevo tipo en un fichero de cabecera.

Cuando yo aprendí a programar en C, el fichero de cabecera era un misterio para mi. Muchos libros de C no hacen hincapié, y el compilador no obliga a hacer la declaración de las funciones, así que parecía algo opcional la mayor parte de las veces, excepto cuando se declaraban estrucutras. En C++ el uso de los ficheros de cabecera se vuelve claro como el cristal. Son prácticamente obligatorios para el desarrollo de programas sencillos, y en ellos podrá información muy específica: declaraciones. El fichero de cabecera informa al compilador de lo que hay disponible en la librería. Puede usar la librería incluso si sólo se dispone del fichero de cabecera y el fichero objeto o el fichero de librería; no necesita disponer del código fuente del fichero `cpp`. En el fichero de cabecera es donde se guarda la especificación de la interfaz.

Aunque el compilador no lo obliga, el mejor modo de construir grandes proyectos en C es usar librerías; colecciones de funciones asociadas en un mismo módulo objeto o librería, y usar un fichero de cabecera para colocar todas las declaraciones de las funciones. Es *de rigor* en C++, Podría meter cualquier función en una librería C, pero el tipo abstracto de dato C++ determina las funciones que están asociadas por medio del acceso común a los datos de una `struct`. Cualquier función miembro debe ser declarada en la declaración de la `struct`; no puede ponerse en otro lugar. El uso de librerías de funciones fue fomentado en C y institucionalizado en C++.

4.7.1. Importancia de los ficheros de cabecera

Cuando se usa función de una librería, C le permite la posibilidad de ignorar el fichero de cabecera y simplemente declarar la función a mano. En el pasado, la gente

hacía eso a veces para acelerar un poquito la compilación evitando la tarea de abrir e incluir el fichero (eso no supone ventaja alguna con los compiladores modernos). Por ejemplo, la siguiente es una declaración extremadamente vaga de la función `printf()` (de `<stdio.h>`):

```
printf(...);
```

Estos puntos suspensivos [4] especifican una *lista de argumentos variable* [5], que dice: la `printf()` tiene algunos argumentos, cada uno con su tipo, pero no se sabe cuales. Simplemente, coge los argumentos que veas y aceptalos. Usando este tipo de declaración, se suspenden todas las comprobaciones de errores en los argumentos.

Esta práctica puede causar problemas sutiles. Si declara funciones «a mano», en un fichero puede cometer un error. Dado que el compilador sólo verá las declaraciones hechas a mano en ese fichero, se adaptará al error. El programa enlazará correctamente, pero el uso de la función en ese fichero será defectuoso. Se trata de un error difícil de encontrar, y que se puede evitar fácilmente usando el fichero de cabecera correspondiente.

Si se colocan todas las declaraciones de funciones en un fichero de cabecera, y se incluye ese fichero allí donde se use la función se asegurará una declaración consistente a través del sistema completo. También se asegurará de que la declaración y la definición corresponden incluyendo el fichero de cabecera en el fichero de definición.

Si declara una `struct` en un fichero de cabecera en C++, *debe* incluir ese fichero allí donde se use una `struct` y también donde se definan los métodos de la `struct`. El compilador de C++ devolverá un mensaje de error si intenta llamar a una función, o llamar o definir un método, sin declararla primero. Imponiendo el uso apropiado de los ficheros de cabecera, el lenguaje asegura la consistencia de las librerías, y reduce el número de error forzando que se use la misma interface en todas partes.

El fichero de cabecera es un contrato entre el programador de la librería y el que la usa. El contrato describe las estructuras de datos, expone los argumentos y valores de retorno para las funciones. Dice, «Esto es lo que hace mi librería». El usuario necesita parte de esta información para desarrollar la aplicación, y el compilador necesita toda ella para generar el código correcto. El usuario de la `struct` simplemente incluye el fichero de cabecera, crea objetos (instancias) de esa `struct`, y enlaza con el módulo objeto o librería (es decir, el código compilado)

El compilador impone el contrato obligando a declarar todas las estruturas y funciones antes que puedan ser usadas y, en el caso de métodos, antes de ser definidos. De ese modo, se le obliga a poner las declaraciones en el fichero de cabecera e incluirlo en el fichero en el que se definen los métodos y en los ficheros en los que se usen. Como se incluye un único fichero que describe la librería para todo el sistema, el compilador puede asegurar la consistencia y evitar errores.

Hay ciertos asuntos a los que debe prestar atención para organizar su código apropiadamente y escribir ficheros de cabecera eficaces. La regla básica es «únicamente declaraciones», es decir, sólo información para el compiladore pero nada que requiera alojamiento en memoria ya sea generando código o creando variables. Esto es así porque el fichero de cabecera normalmente se incluye en varias unidades de

[4] (N. de T. *ellipsis*) en inglés)

[5] Para escribir una definición de función que toma una lista de argumentos realmente variable, debe usar *varargs*, aunque se debería evitar en C++. Puede encontar información detallada sobre el uso de *varargs* en un manual de C.

traducción en un mismo proyecto, y si el almacenamiento para un identificador se pide en más de un sitio, el enlazador indicará un error de definición múltiple (ésta es la *regla de definición única* de C++: Se puede declarar tantas veces como se quiera, pero sólo puede haber una definición real para cada cosa).

Esta norma no es completamente estricta. Si se define una variable que es «file static» (que tiene visibilidad sólo en un fichero) dentro de un fichero de cabecera, habrá múltiples instancias de ese dato a lo largo del proyecto, pero no causará un colisión en el enlazador [6]. Básicamente, debe evitar cualquier cosa en los ficheros de cabecera que pueda causar una ambigüedad en tiempo de enlazado.

4.7.2. El problema de la declaración múltiple

La segunda cuestión respecto a los ficheros de cabecera es ésta: cuando se pone una declaración de `struct` en un fichero de cabecera, es posible que el fichero sea incluido más de una vez en un programa complicado. Los iostreams son un buen ejemplo. Cada vez que una `struct` hace E/S debe incluir uno de los ficheros de cabecera iostream. Si el fichero `cpp` sobre el que se está trabajando utiliza más de un tipo de `struct` (típicamente incluyendo un fichero de cabecera para cada una), se está corriendo el riesgo de incluir el fichero `<isotream>` más de una vez y redeclarar los iostreams.

El compilador considera que la redeclaración de una estructura (eso es aplicable tando a las `struct` como a las `class`) es un error, dado que de otro modo, debería permitir el uso del mismo nombre para tipos diferentes. Para evitar este error cuando se incluyen múltiples ficheros de cabecera, es necesario dar algo de inteligencia a los ficheros de cabecera usando el preprocesador (los ficheros de cabecera estándares como `<iostream>` también tienen esta «inteligencia»).

Tanto C como C++ permiten redeclarar una función, siempre que las dos declaraciones coincidan, pero ni en ese caso se permite la redeclaración de una estructura. En C++ esta regla es especialmente importante porque si el compilador permitiera la redeclaración de una estructura y las dos declaraciones difirieran, ¿cuál debería usar?

El problema de la redeclaración se agrava un poco en C++ porque cada tipo de dato (estructura con funciones) generalmente tiene su propio fichero de cabecera, y hay que incluir un fichero de cabecera en otro si se quiere crear otro tipo de dato que use al primero. Es probable que en algún fichero `cpp` de su proyecto, que se incluyan varios ficheros que incluyan al mismo fichero de cabecera. Durante una compilación simple, el compilador puede ver el mismo fichero de cabecera varias veces. A menos que se haga algo al respecto, el compilador verá la redeclaración de la estructura e informará un error en tiempo de compilación. Para resolver el problema, necesitará saber un poco más acerca del preprocesador.

4.7.3. Las directivas del preprocesador #define, #ifndef y #endif

La directiva de preprocesador `#define` se puede usar para crear banderas en tiempo de compilación. Tiene dos opciones: puede simplemente indicar al preprocesador que la bandera está definida, sin especificar un valor:

[6] Sin embargo, en C++ estándar «file static» es una característica obsoleta.

```
#define FLAG
```

o puede darle un valor (que es la manera habitual en C para definir una constante):

```
#define PI 3.14159
```

En cualquier caso, ahora el preprocesador puede comprobar si la etiqueta ha sido definida:

```
#ifdef FLAG
```

Esto producirá un resultado verdadero, y el código que sigue al `#ifdef` se incluirá en el paquete que se envía al compilador. Esta inclusión acaba cuando el preprocesador encuentra la sentencia:

```
#endif
```

o

```
#endif // FLAG
```

Cualquier cosa después de `#endif` en la misma línea que no sea un comentario es ilegal, incluso aunque algunos compiladores lo acepten. Los pares `#ifdef`/`#endif` se pueden anidar.

El complementario de `#define` es `#undef` (abreviación de «un-define» que hará que una sentencia `#ifdef` que use la misma variable produzca un resultado falso. `#undef` también causará que el preprocesador deje de usar una macro. El complementario de `#ifdef` es `#ifndef`, que producirá verdadero si la etiqueta no ha sido definida (éste es el que usaremos en los ficheros de cabecera).

Hay otras características útiles en el preprocesador de C. Consulte la documentación de su preprocesador para ver todas ellas.

4.7.4. Un estándar para los ficheros de cabecera

En cada fichero de cabecera que contiene una estructura, primero debería comprobar si ese fichero ya ha sido incluido en este fichero `cpp` particular. Hágalo comprobando una bandera del preprocesador. Si la bandera no está definida, el fichero no se ha incluido aún, y se debería definir la bandera (de modo que la estructura no se pueda redeclarar) y declarar la estructura. Si la bandera estaba definida entonces el tipo ya ha sido declarado de modo que debería ignorar el código que la declara. Así es como debería ser un fichero de cabecera:

```
#ifndef HEADER_FLAG
#define HEADER_FLAG
// Escriba la ódeclaracin íaqu...
#endif // HEADER_FLAG
```

Como puede ver, la primera vez que se incluye el fichero de cabecera, los contenidos del fichero (incluyendo la declaración del tipo) son incluidos por el preprocesador. Las demás veces que se incluya -en una única unidad de programación- la declaración del tipo será ignorada. El nombre `HEADER_FLAG` puede ser cualquier nombre único, pero un estándar fiable a seguir es poner el nombre del fichero de cabecera en mayúsculas y reemplazar los puntos por guiones bajos (sin embargo, el guión bajo al comienzo está reservado para nombres del sistema). Este es un ejemplo:

```
//: C04:Simple.h
// Simple header that prevents re-definition
#ifndef SIMPLE_H
#define SIMPLE_H

struct Simple {
  int i,j,k;
  initialize() { i = j = k = 0; }
};
#endif // SIMPLE_H ///:~
```

Aunque el `SIMPLE_H` después de `#endif` está comentado y es ignorado por el preprocesador, es útil para documentación.

Estas sentencias del preprocesador que impiden inclusiones múltiples se denominan a menudo *guardas de inclusión* (*include guards*)

4.7.5. Espacios de nombres en los ficheros de cabecera

Notará que las *directivas using* están presentes en casi todos los ficheros `cpp` de esto libro, normalmente en la forma:

```
    using namespace std;
```

Como `std` es el espacio de nombres que encierra la librería Estándar C++ al completo, esta directiva `using` en particular permite que se puedan usar los nombres de la librería Estándar C++. Sin embargo, casi nunca verá una directiva `using` en un fichero de cabecera (al menos, no fuera de un bloque). La razón es que la directiva `using` elimina la protección de ese espacio de nombres en particular, y el efecto dura hasta que termina la unidad de compilación actual. Si pone una directiva `using` (fuera de un bloque) en un fichero de cabecera, significa que esta perdida de «protección del espacio de nombres» ocurrirá con cualquier fichero que incluya este fichero de cabecera, lo que a menudo significa otros ficheros de cabecera, es muy fácil acabar «desactivando» los espacios de nombres en todos sitios, y por tanto, neutralizando los efectos beneficiosos de los espacios de nombres.

En resumen: no ponga directivas `using` en ficheros de cabecera.

4.7.6. Uso de los ficheros de cabecera en proyectos

Cuando se construye un proyecto en C++, normalmente lo creará poniendo juntos un montón de tipos diferentes (estructuras de datos con funciones asociadas). Normalmente pondrá la declaración para cada tipo o grupo de tipos asociados en un fichero de cabecera separado, entonces definirá las funciones para ese tipo en una

unidad de traducción. Cuando use ese tipo, deberá incluir el fichero de cabecera para efectuar las declaraciones apropiadamente.

A veces ese patrón se seguirá en este libro, pero más a menudo los ejemplos serán muy pequeños, así que todo - la declaración de las estructuras, la definición de las funciones, y la función `main()` - pueden aparecer en un único fichero. Sin embargo, tenga presente que debería usar ficheros separados y ficheros de cabecera para aplicaciones reales.

4.8. Estructuras anidadas

La conveniencia de coger nombres de funciones y datos fuera del espacio de nombre global es aplicable a las estructuras. Puede anidar una estructura dentro de otra estructura, y por tanto guardar juntos elementos asociados. La sintaxis de declaración es la que podría esperarse, tal como puede ver en la siguiente estructura, que implementa una pila como una lista enlazada simple de modo que «nunca» se queda sin memoria.

```
//: C04:Stack.h
// Nested struct in linked list
#ifndef STACK_H
#define STACK_H

struct Stack {
  struct Link {
    void* data;
    Link* next;
    void initialize(void* dat, Link* nxt);
  }* head;
  void initialize();
  void push(void* dat);
  void* peek();
  void* pop();
  void cleanup();
};
#endif // STACK_H ///:~
```

La `struck` anidada se llama Link, y contiene un puntero al siguiente Link en la lista y un puntero al dato almacenado en el Link. Si el siguiente puntero es cero, significa que es el último elemento de la lista.

Fíjese que el puntero `head` está definido a la derecha después de la declaración de la `struct` Link, es lugar de una definición separada `Link* head`. Se trata de una sintaxis que viene de C, pero que hace hincapié en la importancia del punto y coma después de la declaración de la estructura; el punto y coma indica el fin de una lista de definiciones separadas por comas de este tipo de estructura (Normalmente la lista está vacía.)

La estructura anidada tiene su propia función `initialize()`, como todas las estructuras hasta el momento, para asegurar una inicialización adecuada. `Stack` tiene tanto función `initialice()` como `cleanup()`, además de `push()`, que toma un puntero a los datos que se desean almacenar (asume que ha sido alojado en el montículo), y `pop()`, que devuelve el puntero `data` de la cima de la `Stack` y elimina el elemento de la cima. (El que hace `pop()` de un elemento se convierte en

responsable de la destrucción del objeto apuntado por `data`.) La función `peak()` también devuelve un puntero `data` a la cima de la pila, pero deja el elemento en la `Stack`.

Aquí se muestran las definiciones de los métodos:

```
//: C04:Stack.cpp {O}
// Linked list with nesting
#include "Stack.h"
#include "../require.h"
using namespace std;

void
Stack::Link::initialize(void* dat, Link* nxt) {
  data = dat;
  next = nxt;
}

void Stack::initialize() { head = 0; }

void Stack::push(void* dat) {
  Link* newLink = new Link;
  newLink->initialize(dat, head);
  head = newLink;
}

void* Stack::peek() {
  require(head != 0, "Stack empty");
  return head->data;
}

void* Stack::pop() {
  if(head == 0) return 0;
  void* result = head->data;
  Link* oldHead = head;
  head = head->next;
  delete oldHead;
  return result;
}

void Stack::cleanup() {
  require(head == 0, "Stack not empty");
} ///:~
```

La primera definición es particularmente interesante porque muestra cómo se define un miembro de una estructura anidada. Simplemente se usa un nivel adicional de resolución de ámbito para especificar el nombre de la `struct` interna. `Stack::Link::initialize()` toma dos argumentos y los asigna a sus atributos.

`Stack::initialize()` asgina cero a `head`, de modo que el objeto sabe que tiene una lista vacía.

`Stack::push()` toma el argumento, que es un puntero a la variable a la que se quiere seguir la pista, y la apila en la `Stack`. Primero, usa `new` para pedir alojamiento para el `Link` que se insertará en la cima. Entonces llama a la función `initialize()` para asignar los valores apropiados a los miembres del `Link`. Fijese que el siguiente puntero se asigna al `head` actual; entonces `head` se asigna al nuevo puntero `Link`.

Esto apila eficazmente el `Link` en la cima de la lista.

`Stack::pop()` captura el puntero `data` en la cima actual de la `Stack`; entonces mueve el puntero `head` hacia abajo y borra la anterior cima de la `Stack`, finalmente devuelve el puntero capturado. Cuando `pop()` elemina el último elemento, `head` vuelve a ser cero, indicando que la `Stack` está vacía.

`Stack::cleanup()` realmente no hace ninguna limpieza. En su lugar, establece una política firme que dice «el programador cliente que use este objeto `Stack` es responsable de des-apilar todos los elementos y borrarlos». `require()` se usa para indicar que ha ocurrido un error de programación si la `Stack` no está vacía.

¿Por qué no puede el destructor de `Stack` responsabilizarse de todos los objetos que el programador cliente no des-apiló? El problema es que la `Stack` está usando punteros void, y tal como se verá en el Capítulo 13 usar `delete` para un void* no libera correctamente. El asunto de «quién es el responsable de la memoria» no siempre es sencillo, tal como veremos en próximos capítulos.

Un ejemplo para probar la `Stack`:

```
//: C04:StackTest.cpp
//{L} Stack
//{T} StackTest.cpp
// Test of nested linked list
#include "Stack.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // File name is argument
  ifstream in(argv[1]);
  assure(in, argv[1]);
  Stack textlines;
  textlines.initialize();
  string line;
  // Read file and store lines in the Stack:
  while(getline(in, line))
    textlines.push(new string(line));
  // Pop the lines from the Stack and print them:
  string* s;
  while((s = (string*)textlines.pop()) != 0) {
    cout << *s << endl;
    delete s;
  }
  textlines.cleanup();
} ///:~
```

Es similar al ejemplo anterior, pero en este se apilan líneas de un fichero (como punteros a cadena) en la `Stack` y después los des-apila, lo que provoca que el fichero sea imprimido en orden inverso. Fíjese que `pop()` devuelve un void* que debe ser moldeado a string* antes de poderse usar. Para imprimir una cadena, el puntero es dereferenciado.

Como `textlines` se llena, el contenido de `line` se «clona» para cada `push()` creando un `new string(line)`. El valor devuelto por la expresión `new` es un puntero al nuevo string que fue creado y al que se ha copiado la información de la `line`. Si se hubiera pasado directamente la dirección de `line` a `push()`, la `Stack` se llenaría con direcciones idénticas, todas apuntando a `line`. Más adelante en ese libro aprenderá más sobre este proceso de «clonación».

El nombre del fichero se toma de línea de comando. Para garantizar que hay suficientes argumentos en la línea de comando, se usa una segunda función del fichero de cabecera `require.h`: `requireArgs()` que compara `argc` con el número de argumentos deseado e imprime un mensaje de error y termina el programa si no hay suficientes argumentos.

4.8.1. Resolución de ámbito global

El operador de resolución de ámbito puede ayudar en situaciones en las que el nombre elegido por el compilador (el nombre «más cercano») no es el que se quiere. Por ejemplo, suponga que tiene una estructura con un identificador local `a`, y quiere seleccionar un identificador global `a` desde dentro de un método. El compilador, por defecto, elegirá el local, de modo que es necesario decirle que haga otra cosa. Cuando se quiere especificar un nombre global usando la resolución de ámbito, debe usar el operador sin poner nada delante de él. A continuación aparece un ejemplo que muestra la resolución de ámbito global tanto para una variable como para una función:

```
//: C04:Scoperes.cpp
// Global scope resolution
int a;
void f() {}

struct S {
  int a;
  void f();
};

void S::f() {
  ::f();  // Would be recursive otherwise!
  ::a++;  // Select the global a
  a--;    // The a at struct scope
}
int main() { S s; f(); } ///:~
```

Sin resolución de ámbito en `S::f()`, el compilador elegiría por defecto las versiones miembro para `f()` y `a`.

4.9. Resumen

En este capítulo, ha aprendido lo fundamental de C++: que puede poner funciones dentro de las estructuras. Este nuevo tipo de estructura se llama *tipo abstracto de dato*, y las variables que se crean usando esta estructura se llaman *objetos*, o *instancias*, de ese tipo. Invocar un método de una objeto se denomina *enviar un mensaje* al objeto. La actividad principal en la programación orientada a objetos es el envío de

mensajes a objetos.

Aunque empaquetar datos y funciones juntos es un benificio significativo para la organización del código y hace la librería sea más fácil de usar porque previene conflictos de nombres ocultando los nombres, hay mucho más que se puede hacer para tener programación más segura en C++. En el próximo capítulo, aprenderá cómo proteger algunos miembros de una `struct` para que sólo el programador pueda manipularlos. Esto establece un límite claro entre lo que puede cambiar el usuario de la estructura y lo que sólo el programador puede cambiar.

4.10. Ejercicios

Las soluciones a los ejercicios se pueden encontrar en el documento electrónico titulado «The Thinking in C++ Annotated Solution Guide», disponible por poco dinero en www.BruceEckel.com.

1. En la librería C estándar, la función `puts()` imprime un array de caracteres a la consola (de modo que puede escribir `puts("Hola")`). Escriba un program C que use `puts()` pero que no incluya `<stdio.h>` o de lo contrario declare la función. Compile ese programa con su compilador de C. (algunos compiladores de C++ no son programas distintos de sus compiladores de C, es ese caso puede que necesite averiguar que opción de línea de comando fuerza una compilación C.) Ahora compílelo con el compilador C++ y preste atención a la diferencia.
2. Cree una declaración de `struct` con un único método, entonces cree una definición para ese método. Cree un objeto de su nuevo tipo de dato, e invoque el método.
3. Cambie su solución al Ejercicio 2 para que la `struct` sea declarada en un fichero de cabecera convenientemente «guardado», con la definición en un fichero `cpp` y el `main()` en otro.
4. Cree una `struct` con un único atributo de tipo entero, y dos funciones globales, cada una de las cuales acepta un puntero a ese `struct`. La primera función tiene un segundo argumento de tipo entero y asigna al entero de l `struct` el valor del argumento, la segunda muestra el entero de la `struct`. Prueba las funciones.
5. Repita el Ejercicio 4 pero mueva las función de modo que sean métodos de la `struct`, y pruebe de nuevo.
6. Cree una clase que (de forma redundante) efectúe la selección de atributos y una llamada a método usando la palabra reservada `this` (que indica a la dirección del objeto actual)
7. Cree una `Stach` que mantenga doubles. Rellénela con 25 valores double, después muéstrelos en consola.
8. Repita el Ejercicio 7 con `Stack`.
9. Cree un fichero que contenga una función `f()` que acepte un argumento entero y lo imprima en consola usando la función `printf()` de `<stdio>` escribiendo: `printf("%d\n", i)` donde `i` es el entero que desea imprimir. Cree un fichero separado que contenga `main()`, y este fichero declare `f()` pero aceptando un argumento float. Invoque `f()` desde `main()`. Intente compilar y enlazar el programa con el compilador C++ y vea qué ocurre. Ahora compile y

enlace el programa usando el compilador C, y vea que ocurre cuando se ejecuta. Explique el comportamiento.

10. Averigüe cómo generar lenguaje ensamblador con su compilador C y C++. Escriba una función en C y una `struct` con un único miembro en C++. Genere la salida en lenguaje ensamblador para cada una de ellas y encuentre los nombres de ambas funciones, de modo que pueda ver qué tipo de «decoración» aplica el compilador a dichos nombres.

11. Escriba un programa con código condicionalmente-compilado en `main()`, para que cuando se defina un valor del preprocesador, se muestre un mensaje, pero cuando no se defina, se imprima otra mensaje distinto. Compile este experimentando con un `#define` en el programa, después averigüe la forma de indicar al compilador definiciones de preprocesador en la línea de comandos y experimente con ello.

12. Escriba un programa que use `assert()` con un argumento que siempre sea falso (cero) y vea que ocurre cuando lo ejecuta. Ahora compílelo con `#define NDEBUG` y ejecútelo de nuevo para ver la diferencia.

13. Cree un tipo abstracto de dato que represente un cinta de vídeo en una tienda de alquiler. Considere todos los datos y operaciones que serían necesarias para que el tipo `Video` funcione con el sistema de gestión de la tienda. Incluya un método `print()` que muestre información sobre el `Video`

14. Cree un objeto `Pila` que almacene objetos `Video` del Ejercicio 13. Cree varios objetos `Video`, guárdelos en la `Stack` y entonces muéstrelos usando `Video::print()`.

15. Escriba un programa que muestre todos los tamaños de los tipos de datos fundamentales de su computadora usando `sizeof`.

16. Modifique `Stash` para usar `vector<char>` como estructura de datos subyacente.

17. Cree dinámicamente espacio de almacenamiento para los siguiente tipos usando `new`: int, long, un array de 100 char, un array de 100 float. Muestre sus direcciones y libérelos usando `delete`.

18. Escriba una función que tome un argumento char*. Usando `new`, pida alojamiento dinámico para un array de char con un tamaño igual al argumento pasado a la función. Usando indexación de array, copie los caracteres del argumento al array dinámico (no olvide el terminador nulo) y devuelva el puntero a la copia. En su `main()`, pruebe la función pasando una cadena estática entre comillas, después tome el resultado y páselo de nuevo a la función. Muestre ambas cadenas y punteros para poder ver que tienen distinta ubicación. Mediante `delete` libere todo el almacenamiento dinámico.

19. Haga un ejemplo de estructura declarada con otra estructura dentro (un estructura anidada). Declare atributos en ambas `structs`, y declare y defina métodos en ambas `structs`. Escriba un `main()` que pruebe los nuevos tipos.

20. ¿Cómo de grande es una estructura? Escriba un trozo de código que muestre el tamaño de varias estructuras. Cree estructuras que tengan sólo atributos y otras que tengan atributos y métodos. Después cree una estructura que no tenga ningún miembro. Muestre los tamaños de todas ellas. Explique el motivo del tamaño de la estructura que no tiene ningún miembro.

21. C++ crea automáticamente el equivalente de `typedef` para `structs`, tal como ha visto en este capítulo. También lo hace para las enumeraciones y las uniones. Escriba un pequeño programa que lo demuestre.

22. Cree una `Stack` que maneje `Stash`es. Cada `Stash` mantendrá cinco líneas procedentes de un fichero. Cree las `Stash` usando `new`. Lea un fichero en su `Stack`, después muéstrelo en su forma original extrayéndolo de la `Stack`.

23. Modifique el Ejercicio 22 de modo que cree una estructura que encapsule la `Stack` y las `Stash`. El usuario sólo debería añadir y pedir líneas a través de sus métodos, pero debajo de la cubierta la estructura usa una `Stack`(pila) de `Stash`es.

24. Cree una `struct` que mantenga un int y un puntero a otra instancia de la misma `struct`. Escriba una función que acepte como parámetro la dirección de una de estas `struct` y un int indicando la longitud de la lista que se desea crear. Esta función creará una cadena completa de estas `struct` (una lista enlazada), empezando por el argumento (la cabeza de la lista), con cada una apuntando a la siguiente. Cree las nuevas `struct` usando `new`, y ponga la posición (que número de objeto es) en el int. En la última `struct` de la lista, ponga un valor cero en el puntero para indicar que es el último. Escriba una segunda función que acepte la cabeza de la lista y la recorra hasta el final, mostrando los valores del puntero y del int para cada una.

25. Repita el ejercicio 24, pero poniendo las funciones dentro de una `struct` en lugar de usar `struct` y funciones «crudas».

5: Ocultar la implementación

Una librería C típica contiene una estructura y una serie de funciones que actúan sobre esa estructura. Hasta ahora hemos visto cómo C++ toma funciones *conceptualmente* asociadas y las asocia *literalmente* poniendo la declaración de la función dentro del dominio de la estructura, cambiando la forma en que se invoca a las funciones desde las estructuras, eliminando el paso de la dirección de la estructura como primer parámetro, y añadiendo un nuevo tipo al programa (de ese modo no es necesario crear un `typedef` para la estructura).

Todo esto son mejoras, le ayuda a organizar su código haciéndolo más fácil de escribir y leer. Sin embargo, hay otros aspectos importantes a la hora de hacer que las librerías sean más sencillas en C++, especialmente los aspectos de seguridad y control. Este capítulo se centra en el tema de la frontera de las estructuras.

5.1. Establecer los límites

En toda relación es importante tener fronteras que todas las partes respeten. Cuando crea una librería, establece una relación con el *programador cliente* que la usa para crear un programa u otra librería.

En una estructura de C, como casi todo en C, no hay reglas. Los programadores cliente pueden hacer lo que quieran con esa estructura, y no hay forma de forzar un comportamiento particular. Por ejemplo, aunque vio en el capítulo anterior la importancia de las funciones llamadas `initialize()` y `cleanup()`, el programador cliente tiene la opción de no llamarlas. (Veremos una forma mejor de hacerlo en el capítulo siguiente.) Incluso si realmente prefiere que el programador cliente no manipule directamente algunos miembros de su estructura, en C no hay forma de evitarlo. Todo está expuesto al todo el mundo.

Hay dos razones para controlar el acceso a los miembros. La primera es no dejar que el programador cliente ponga las manos sobre herramientas que no debería tocar, herramientas que son necesarias para los entresijos del tipo definido, pero no parte del interfaz que el programador cliente necesita para resolver sus problemas particulares. Esto es realmente una ventaja para los programadores cliente porque así pueden ver lo que es realmente importante para ellos e ignorar el resto.

La segunda razón para el control de acceso es permitir al diseñador de la librería cambiar su funcionamiento interno sin preocuparse de como afectara al programador cliente. En el ejemplo `Stack` del capítulo anterior, podría querer solicitar espacio de almacenamiento en grandes trozos, para conseguir mayor velocidad, en vez de crear un nuevo espacio cada vez que un elemento es añadido. Si la interfaz y la implementación están claramente separadas y protegidas, puede hacerlo y forzar al

programador cliente sólo a enlazar de nuevo sus programas.

5.2. Control de acceso en C++

C++ introduce tres nuevas palabras clave para establecer las fronteras de una estructura: `public`, `private` y `protected`. Su uso y significado es bastante claro. Los *especificadores de acceso* se usan solo en la declaración de las estructuras, y cambian las fronteras para todas las declaraciones que los siguen. Cuando use un especificador de acceso, debe ir seguido de «:»

`public` significa que todas las declaraciones de miembros que siguen estarán accesibles para cualquiera. Los miembros `public`son como miembros de una estructura. Por ejemplo, las siguientes declaraciones de estructuras son idénticas:

```
//: C05:Public.cpp
// Public is just like C's struct

struct A {
  int i;
  char j;
  float f;
  void func();
};

void A::func() {}

struct B {
public:
  int i;
  char j;
  float f;
  void func();
};

void B::func() {}

int main() {
  A a; B b;
  a.i = b.i = 1;
  a.j = b.j = 'c';
  a.f = b.f = 3.14159;
  a.func();
  b.func();
} ///:~
```

La palabra clave `private`, por otro lado, significa que nadie podrá acceder a ese miembro excepto usted, el creador del tipo, dentro de los métodos de ese tipo. `private` es una pared entre usted y el programador cliente; si alguien intenta acceder a un miembro `private`, obtendrá un error en tiempo de compilación. En `struct B` en el ejemplo anterior, podría querer hacer partes de la representación (es decir, los atributos) ocultos, accesibles solo a usted:

```
//: C05:Private.cpp
// Setting the boundary
```

```
struct B {
private:
  char j;
  float f;
public:
  int i;
  void func();
};

void B::func() {
  i = 0;
  j = '0';
  f = 0.0;
};

int main() {
  B b;
  b.i = 1;    // OK, public
//!  b.j = '1';  // Illegal, private
//!  b.f = 1.0;  // Illegal, private
} ///:~
```

Aunque `func()` puede acceder a cualquier miembro de `B` (pues `func()` en un miembro de `B`, garantizando así automáticamente el acceso), una función global ordinaria como `main()` no puede. Por supuesto tampoco miembros de otras estructuras. Solo las funciones que pertenezcan a la declaración de la estructura (el «contrato») tendrán acceso a miembros `private`.

No hay un orden fijo para los especificadores de acceso, y pueden aparecer más de una vez. Afectan a todos los miembros declarados después de ellos hasta el siguiente especificador.

5.2.1. `protected`

Es el último que nos queda por ver, `protected` actúa como `private`, con una excepción de la que hablaremos más tarde: estructuras heredadas (que no pueden acceder a lo miembros privados) si tienen acceso a los miembros `protected`. Todo esto se verá más claramente en el capítulo 14 cuando veamos la herencia. Con lo que sabe hasta ahora puede considerar `protected` igual que `private`.

5.3. Amigos (friends)

¿Que pasa si explícitamente se quiere dar acceso a una función que no es miembro de la estructura? Esto se consigue declarando la función como `friend` dentro de la declaración de la estructura. Es importante que la declaración de una función `friend` se haga dentro de la declaración de la estructura pues usted (y el compilador) necesita ver la declaración de la estructura y todas las reglas sobre el tamaño y comportamiento de ese tipo de dato. Y una regla muy importante en toda relación es, «¿Quién puede acceder a mi parte privada?»

La clase controla que código tiene acceso a sus miembros. No hay ninguna manera mágica de «colarse» desde el exterior si no eres `friend`; no puedes declarar

una nueva clase y decir, «Hola, soy `friend` de `Bob`» y esperar ver los miembros `private` y `protected` de `Bob`.

Puede declarar una función global como `friend`, también puede declarar un método de otra estructura, o incluso una estructura completa, como `friend`. Aquí hay un ejemplo:

```
//: C05:Friend.cpp
// Friend allows special access

// Declaration (incomplete type specification):
struct X;

struct Y {
  void f(X*);
};

struct X { // Definition
private:
  int i;
public:
  void initialize();
  friend void g(X*, int); // Global friend
  friend void Y::f(X*);  // Struct member friend
  friend struct Z; // Entire struct is a friend
  friend void h();
};

void X::initialize() {
  i = 0;
}

void g(X* x, int i) {
  x->i = i;
}

void Y::f(X* x) {
  x->i = 47;
}

struct Z {
private:
  int j;
public:
  void initialize();
  void g(X* x);
};

void Z::initialize() {
  j = 99;
}

void Z::g(X* x) {
  x->i += j;
}

void h() {
  X x;
```

```
  x.i = 100; // Direct data manipulation
}

int main() {
  X x;
  Z z;
  z.g(&x);
} ///:~
```

`struct Y` tiene un método `f()` que modifica un objeto de tipo `X`. Aquí hay un poco de lío pues en C++ el compilador necesita que usted declare todo antes de poder hacer referencia a ello, así `struct Y` debe estar declarado antes de que su método `Y::f(X*)` pueda ser declarado como `friend` en `struct X`. Pero para declarar `Y::f(X*)`, `struct X` debe estar declarada antes!

Aquí vemos la solución. Dese cuenta de que `Y::f(X*)` toma como argumento la dirección de un objeto de tipo `X`. Esto es fundamental pues el compilador siempre sabe cómo pasar una dirección, que es de un tamaño fijo sin importar el tipo, aunque no tenga información del tamaño real. Si intenta pasar el objeto completo, el compilador necesita ver la definición completa de `X`, para saber el tamaño de lo que quiere pasar y cómo pasarlo, antes de que le permita declarar una función como `Y::g(X)`.

Pasando la dirección de un `X`, el compilador le permite hacer una *identificación de tipo incompleta* de `X` antes de declarar `Y::f(X*)`. Esto se consigue con la declaración:

```
struct X;
```

Esta declaración simplemente le dice al compilador que hay una estructura con ese nombre, así que es correcto referirse a ella siempre que sólo se necesite el nombre.

Ahora, en `struct X`, la función `Y::f(X*)` puede ser declarada como `friend` sin problemas. Si intenta declararla antes de que el compilador haya visto la especificación completa de `Y`, habría dado un error. Esto es una restricción para asegurar consistencia y eliminar errores.

Fíjese en las otras dos funciones `friend`. La primera declara una función global ordinaria `g()` como `friend`. Pero `g()` no ha sido declarada antes como global!. Se puede usar `friend` de esta forma para declarar la función y darle el estado de `friend` simultáneamente. Esto se extiende a estructuras completas:

```
friend struct Z;
```

es una especificación incompleta del tipo `Z`, y da a toda la estructura el estado de `friend`.

5.3.1. Amigas anidadas

Hacer una estructura anidada no le da acceso a los miembros privados. Para conseguir esto, se debe: primero, declarar (sin definir) la estructura anidada, después declararla como `friend`, y finalmente definir la estructura. La definición de la estructura debe estar separada de su declaración como `friend`, si no el compilador la vería como no miembro. Aquí hay un ejemplo:

```
//: C05:NestFriend.cpp
// Nested friends
#include <iostream>
#include <cstring> // memset()
using namespace std;
const int sz = 20;

struct Holder {
private:
  int a[sz];
public:
  void initialize();
  struct Pointer;
  friend struct Pointer;
  struct Pointer {
  private:
    Holder* h;
    int* p;
  public:
    void initialize(Holder* h);
    // Move around in the array:
    void next();
    void previous();
    void top();
    void end();
    // Access values:
    int read();
    void set(int i);
  };
};

void Holder::initialize() {
  memset(a, 0, sz * sizeof(int));
}

void Holder::Pointer::initialize(Holder* rv) {
  h = rv;
  p = rv->a;
}

void Holder::Pointer::next() {
  if(p < &(h->a[sz - 1])) p++;
}

void Holder::Pointer::previous() {
  if(p > &(h->a[0])) p--;
}

void Holder::Pointer::top() {
  p = &(h->a[0]);
}

void Holder::Pointer::end() {
  p = &(h->a[sz - 1]);
}

int Holder::Pointer::read() {
  return *p;
```

```
}

void Holder::Pointer::set(int i) {
  *p = i;
}

int main() {
  Holder h;
  Holder::Pointer hp, hp2;
  int i;

  h.initialize();
  hp.initialize(&h);
  hp2.initialize(&h);
  for(i = 0; i < sz; i++) {
    hp.set(i);
    hp.next();
  }
  hp.top();
  hp2.end();
  for(i = 0; i < sz; i++) {
    cout << "hp = " << hp.read()
         << ", hp2 = " << hp2.read() << endl;
    hp.next();
    hp2.previous();
  }
} ///:~
```

Una vez que `Pointer` está declarado, se le da acceso a los miembros privados de `Holder` con la sentencia:

```
friend Pointer;
```

La estructura `Holder` contiene un array de enteros y `Pointer` le permite acceder a ellos. Como `Pointer` está fuertemente asociada con `Holder`, es comprensible que sea una estructura miembro de `Holder`. Pero como `Pointer` es una clase separada de `Holder`, puede crear más de una instancia en el `main()` y usarlas para seleccionar diferentes partes del array. `Pointer` es una estructura en vez de un puntero de C, así que puede garantizar que siempre apuntará dentro de `Holder`.

La función de la librería estándar de C `memset()` (en <`cstring`>) se usa en el programa por conveniencia. Hace que toda la memoria a partir de una determinada dirección (el primer argumento) se cargue con un valor particular (el segundo argumento) para `n` bytes a partir de la dirección donde se empezó (`n` es el tercer argumento). Por supuesto, se podría haber usado un bucle para hacer lo mismo, pero `memset()` está disponible, bien probada (así que es más factible que produzca menos errores), y probablemente es más eficiente.

5.3.2. ¿Es eso puro?

La definición de la clase le da la pista, mirando la clase se puede saber qué funciones tienen permiso para modificar su parte privada. Si una función es `friend`, significa que no es miembro, pero que de todos modos se le quiere dar permiso para modificar la parte privada, y debe estar especificado en la definición de la clase para

que todo el mundo pueda ver que esa es una de las funciones privilegiadas.

C++ es un lenguaje orientado a objetos híbrido, no es puro, y `friend` fue añadido para solucionar algunos problemas que se presentaban en la práctica. Es bueno apuntar que esto hace al lenguaje menos «puro», pues C++ fue diseñado para ser pragmático, no para aspirar a un ideal abstracto.

5.4. Capa de objetos

En el capítulo 4 se dijo que una `struct` escrita para un compilador C y más tarde compilada en uno de C++ no cambiaría. Se refería básicamente a la estructura interna del objeto que surge de la `struct`, es decir, la posición relativa en memoria donde se guardan los valores de las diferentes variables. Si el compilador C++ cambiase esta estructura interna, entonces el código escrito en C que hiciese uso del conocimiento de las posiciones de las variables fallaría.

Cuando se empiezan a usar los especificadores de acceso, se cambia al universo del C++, y las cosas cambian un poco. Dentro de un «bloque de acceso» (un grupo de declaraciones delimitado por especificadores de acceso), se garantiza que las variables se encontraran contiguas, como en C. Sin embargo, los bloques de acceso pueden no aparecer en el objeto en el mismo orden en que se declaran. Aunque el compilador normalmente colocará los bloques como los definió, no hay reglas sobre esto, pues una arquitectura hardware especifica y/o un sistema operativo puede tener soporte especifico para `private` y `protected` que puede requerir que estos bloques se coloquen en lugares específicos de la memoria. La especificación del lenguaje no quiere impedir este tipo de ventajas.

Los especificadores de acceso son parte de la estructura y no afectan a los objetos creados desde ésta. Toda la información de accesos desaparece antes de que el programa se ejecute; en general ocurre durante la compilación. En un programa en ejecución, los objetos son «zonas de almacenamiento» y nada más. Si realmente quiere, puede romper todas las reglas y acceder a la memoria directamente, como en C. C++ no está diseñado para prohibir hacer cosas salvajes. Solo le proporciona una alternativa mucho más fácil, y deseable.

En general, no es una buena idea hacer uso de nada que dependa de la implementación cuando se escribe un programa. Cuando necesite hacerlo, encapsúlelo en una estructura, así en caso de tener que portarlo se podrá concentrar en ella.

5.5. La clase

El control de acceso se suele llamar también *ocultación de la implementación*. Incluir funciones dentro de las estructuras (a menudo llamado encapsulación [1]) produce tipos de dato con características y comportamiento, pero el control de acceso pone fronteras en esos tipos, por dos razones importantes. La primera es para establecer lo que el programador cliente puede y no puede hacer. Puede construir los mecanismos internos de la estructura sin preocuparse de que el programador cliente pueda pensar que son parte de la interfaz que debe usar.

Esto nos lleva directamente a la segunda razón, que es separar la interfaz de la implementación. Si la estructura se usa en una serie de programas, y el programador cliente no puede hacer más que mandar mensajes a la interfaz pública, usted puede cambiar cualquier cosa privada sin que se deba modificar código cliente.

[1] Como se dijo anteriormente, a veces el control de acceso se llama también encapsulación

La encapsulación y el control de acceso, juntos, crean algo más que una estructura de C. Estamos ahora en el mundo de la programación orientada a objetos, donde una estructura describe una clase de objetos como describiría una clase de peces o pájaros: Cualquier objeto que pertenezca a esa clase compartirá esas características y comportamiento. En esto se ha convertido la declaración de una estructura, en una descripción de la forma en la que los objetos de este tipo serán y actuarán.

En el lenguaje OOP original, Simula-67, la palabra clave `class` fue usada para describir un nuevo tipo de dato. Aparentemente esto inspiro a Stroustrup a elegir esa misma palabra en C++, para enfatizar que este era el punto clave de todo el lenguaje: la creación de nuevos tipos de dato que son más que solo estructuras de C con funciones. Esto parece suficiente justificación para una nueva palabra clave.

De todas formas, el uso de `class` en C++ es casi innecesario. Es idéntico a `struct` en todos los aspectos excepto en uno: `class` pone por defecto `private`, mientras que `struct` lo hace a `public`. Estas son dos formas de decir lo mismo:

```
//: C05:Class.cpp
// Similarity of struct and class

struct A {
private:
  int i, j, k;
public:
  int f();
  void g();
};

int A::f() {
  return i + j + k;
}

void A::g() {
  i = j = k = 0;
}

// Identical results are produced with:

class B {
  int i, j, k;
public:
  int f();
  void g();
};

int B::f() {
  return i + j + k;
}

void B::g() {
  i = j = k = 0;
}

int main() {
  A a;
  B b;
  a.f(); a.g();
  b.f(); b.g();
```

```
} ///:~
```

La clase (`class`) en un concepto OOP fundamental en C++. Es una de la palabras clave que no se pondrán en negrita en este libro - es incomodo pues se repite mucho. El cambio a clases es tan importante que sospecho que Stroustrup hubiese preferido eliminar completamente `struct`, pero la necesidad de compatibilidad con C no lo hubiese permitido.

Mucha gente prefiere crear clases a la manera `struct` en vez de a la mánera `class`, pues sustituye el «por-defecto-`private`» de `class` empezando con los elementos `public`:

```
class X {
public:
    void miembro_de_interfaz();
private:
    void miembro_privado();
    int representacion_interna;
};
```

El porqué de esto es que tiene más sentido ver primero lo que más interesa, el programador cliente puede ignorar todo lo que dice `private`. De hecho, la única razón de que todos los miembros deban ser declarados en la clase es que el compilador sepa como de grande son los objetos y pueda colocarlos correctamente, garantizando así la consistencia.

De todas formas, los ejemplos en este libro pondrán los miembros privados primero, así:

```
class X {
    void private_function();
    int internal_representation;
public:
    void interface_function();
};
```

Alguna gente incluso decora sus nombres privados

```
class Y {
public:
    void f();
private:
    int mX;  // "Self-decorated" name
};
```

Como `mX` esta ya oculto para `Y`, la *m* (de «miembro») es innecesaria. De todas formas, en proyectos con muchas variables globales (algo que debe evitar a toda costa, aunque a veces inevitable en proyectos existentes), es de ayuda poder distinguir variables globales de atributos en la definición de los métodos.

5.5.1. Modificaciones en `Stash` para usar control de acceso

Tiene sentido coger el ejemplo del capítulo 4 y modificarlo para usar clases y control de acceso. Dese cuenta de cómo la parte de la interfaz a usar en la programación cliente está claramente diferenciada, así no hay posibilidad de que el programador cliente manipule accidentalmente parte de la clase que no debería.

```
//: C05:Stash.h
// Converted to use access control
#ifndef STASH_H
#define STASH_H

class Stash {
  int size;      // Size of each space
  int quantity;  // Number of storage spaces
  int next;      // Next empty space
  // Dynamically allocated array of bytes:
  unsigned char* storage;
  void inflate(int increase);
public:
  void initialize(int size);
  void cleanup();
  int add(void* element);
  void* fetch(int index);
  int count();
};
#endif // STASH_H ///:~
```

La función `inflate()` se ha hecho `private` porque solo es usada por la función `add()` y por tanto es parte de la implementación interna, no de la interfaz. Esto significa que, más tarde, puede cambiar la implementación interna para usar un sistema de gestión de memoria diferente.

Aparte del nombre del archivo include, la cabecera de antes es lo único que ha sido cambiado para este ejemplo. El fichero de implementación y de prueba son los mismos.

5.5.2. Modificar `Stack` para usar control de acceso

Como un segundo ejemplo, aquí está `Stack` convertido en clase. Ahora la estructura anidada es `private`, lo que es bueno pues asegura que el programador cliente no tendrá que fijarse ni depender de la representación interna de `Stack`:

```
//: C05:Stack2.h
// Nested structs via linked list
#ifndef STACK2_H
#define STACK2_H

class Stack {
  struct Link {
    void* data;
    Link* next;
    void initialize(void* dat, Link* nxt);
```

```
  }* head;
public:
  void initialize();
  void push(void* dat);
  void* peek();
  void* pop();
  void cleanup();
};
#endif // STACK2_H ///:~
```

Como antes, la implementación no cambia por lo que no la repetimos aquí. El programa de prueba es también idéntico. La única cosa que ha cambiado es la robustez del interfaz de la clase. El valor real del control de acceso es prevenirle de traspasar las fronteras durante el desarrollo. De hecho, el compilador es el único que conoce los niveles de protección de los miembros de la clase. No hay información sobre el control de acceso añadida en el nombre del miembro que llega al enlazador. Todas las comprobaciones sobre protección son hechas por el compilador; han desaparecido al llegar a la ejecución.

Dese cuenta de que la interfaz presentada al programador cliente es ahora realmente la de una pila. Sucede que esta implementada como una lista enlazada, pero usted puede cambiar esto sin afectar a la forma en que los programas cliente interactúan con ella, o (más importante aun) sin afectar a una sola linea de su código.

5.6. Manejo de clases

El control de acceso en C++ le permite separar la interfaz de la implementación, pero la ocultación de la implementación es solo parcial. El compilador debe ver aún la declaración de todas las partes del objeto para poder crearlo y manipularlo correctamente. Podría imaginar un lenguaje de programación que requiriese solo la interfaz publica del objeto y permitiese que la implementación privada permaneciese oculta, pero C++ realiza comparación de tipos estáticamente (en tiempo de compilación) tanto como es posible. Esto significa que se dará cuenta lo antes posible de si hay un error. También significa que su programa será más eficiente. De todas formas, la inclusión de la implementación privada tiene dos efectos: la implementación es visible aunque no se pueda acceder a ella fácilmente, y puede causar recompilaciones innecesarias.

5.6.1. Ocultar la implementación

Algunos proyectos no pueden permitirse tener visible su implementación al publico. Puede dejar a la vista información estratégica en un fichero de cabecera de una librería que la compañía no quiere dejar disponible a los competidores. Puede estar trabajando en un sistema donde la seguridad sea clave - un algoritmo de encriptación, por ejemplo - y no quiere dejar ninguna pista en un archivo de cabecera que pueda ayudar a la gente a romper el código. O puede que su librería se encuentre en un ambiente «hostil», donde el programador accederá a los componentes privados de todas formas, usando punteros y conversiones. En todas estas situaciones, es de gran valor tener la estructura real compilada dentro de un fichero de implementación mejor que a la vista en un archivo de cabecera.

5.6.2. Reducir la recompilación

Su entorno de programación provocará una recompilación de un fichero si este se modifica, o si se modifica otro fichero del que depende, es decir, un archivo de cabecera que se haya incluido. Esto significa que cada vez que se haga un cambio en una clase, ya sea a la interfaz pública o a las declaraciones de los miembros privados, se provocará una recompilación de todo lo que incluya ese archivo de cabecera. Este efecto se conoce usualmente como *el problema de la clase-base frágil*. Para un proyecto grande en sus comienzos esto puede ser un gran problema pues la implementación suele cambiar a menudo; si el proyecto es muy grande, el tiempo de las compilaciones puede llegar a ser un gran problema.

La técnica para resolver esto se llama a veces *clases manejador* o el «gato de Chesire» [2] - toda la información sobre la implementación desaparece excepto por un puntero, la "sonrisa". El puntero apunta a una estructura cuya definición se encuentra en el fichero de implementación junto con todas las definiciones de las funciones miembro. Así, siempre que la interfaz no se cambie, el archivo de cabecera permanece inalterado. La implementación puede cambiar a su gusto, y sólo el fichero de implementación deberá ser recompilado y reenlazado con el proyecto.

Aquí hay un ejemplo que demuestra como usar esta técnica. El archivo de cabecera contiene solo la interfaz publica y un puntero de una clase especificada de forma incompleta:

```
//: C05:Handle.h
// Handle classes
#ifndef HANDLE_H
#define HANDLE_H

class Handle {
  struct Cheshire; // Class declaration only
  Cheshire* smile;
public:
  void initialize();
  void cleanup();
  int read();
  void change(int);
};
#endif // HANDLE_H ///:~
```

Esto es todo lo que el programador cliente puede ver. La linea

```
struct Cheshire;
```

es una *especificación de tipo incompleta* o una *declaración de clase* (una *definición de clase* debe incluir el cuerpo de la clase). Le dice al compilador que *Chesire* es el nombre de una estructura, pero no detalles sobre ella. Esta es información suficiente para crear un puntero a la estructura; no puede crear un objeto hasta que el cuerpo de la estructura quede definido. En esta técnica, el cuerpo de la estructura está escondido en el fichero de implementación:

[2] Este nombre se le atribuye a John Carolan, uno de los pioneros del C++, y por supuesto, Lewis Carroll. Esta técnica se puede ver también como una forma del tipo de diseño «puente», descrito en el segundo volumen.

```
//: C05:Handle.cpp {O}
// Handle implementation
#include "Handle.h"
#include "../require.h"

// Define Handle's implementation:
struct Handle::Cheshire {
  int i;
};

void Handle::initialize() {
  smile = new Cheshire;
  smile->i = 0;
}

void Handle::cleanup() {
  delete smile;
}

int Handle::read() {
  return smile->i;
}

void Handle::change(int x) {
  smile->i = x;
} ///:~
```

Chesire es una estructura anidada, así que se debe ser definido con resolución de ámbito:

```
struct Handle::Cheshire {
```

En `Handle::initialize()`, se solicita espacio de almacenamiento para una estructura `Chesire`, y en `Handle::cleanup()` se libera ese espacio. Este espacio se usa para almacenar todos los datos que estarían normalmente en la sección privada de la clase. Cuando compile `Handle.cpp`, esta definición de la estructura estará escondida en el fichero objeto donde nadie puede verla. Si cambia los elementos de *Chesire*, el único archivo que debe ser recompilado es `Handle.cpp` pues el archivo de cabecera permanece inalterado.

El uso de `Handle` es como el uso de cualquier clase: incluir la cabecera, crear objetos, y mandar mensajes.

```
//: C05:UseHandle.cpp
//{L} Handle
// Use the Handle class
#include "Handle.h"

int main() {
  Handle u;
  u.initialize();
  u.read();
  u.change(1);
  u.cleanup();
} ///:~
```

La única cosa a la que el programador cliente puede acceder es a la interfaz publica, así que mientras la implementación sea lo único que cambie, el fichero anterior no necesita recompilarse. Así, aunque esto no es ocultación de implementación perfecta, es una gran mejora.

5.7. Resumen

El control de acceso en C++ ofrece un gran control al creador de la clase. Los usuarios de la clase pueden ver claramente lo que pueden usar y qué puede ignorar. Más importante aún es la posibilidad de asegurar que ningún programador cliente depende de ninguna parte de la implementación interna de la clase. Si sabe esto como creador de la clase, puede cambiar la implementación subyacente con la seguridad de que ningún programador cliente se verá afectado por los cambios, pues no pueden acceder a esa parte de la clase.

Cuando tenga la posibilidad de cambiar la implementación subyacente, no solo podrá mejorar su diseño más tarde, también tiene la libertad de cometer errores. No importa con qué cuidado planee su diseño, cometerá errores. Sabiendo que es relativamente seguro que cometerá esos errores, experimentará más, aprenderá más rápido, y acabará su proyecto antes.

La interfaz pública de una clase es lo que *realmente ve* el programador cliente, así que es la parte de la clase más importante durante el análisis y diseño. Pero incluso esto le deja algo de libertad para el cambio. Si no consigue la interfaz correcta a la primera, puede añadir más funciones, mientras no quite ninguna que el programador cliente ya haya usado en su código.

5.8. Ejercicios

Las soluciones a los ejercicios se pueden encontrar en el documento electrónico titulado «The Thinking in C++ Annotated Solution Guide», disponible por poco dinero en www.BruceEckel.com.

1. Cree una clase con atributos y métodos `public`, `private` y `protected`. Cree un objeto de esta clase y vea qué mensajes de compilación obtiene cuando intenta acceder a los diferentes miembros de la clase.

2. Escriba una estructura llamada `Lib` que contenga tres objetos `string a, b` y `c`. En `main()` cree un objeto `Lib` llamado `x` y asígnelo a `x.a, x.b` y `x.c`. Imprima por pantalla sus valores. Ahora reemplace `a, b` y `c` con un array de cadenas `s[3]`. Dese cuenta de que su función `main()` deja de funcionar como resultado del cambio. Ahora cree una clase, llámela `Libc` con tres cadenas como datos miembro privados `a, b` y `c`, y métodos `seta(),geta(),setb(),getb(),setc()` y `getc()` para establecer y recuperar los distintos valores. Escriba una función `main()` como antes. Ahora cambie las cadenas privadas `a, b` y `c` por un array de cadenas privado `s[3]`. Vea que ahora `main()` sigue funcionando.

3. Cree una clase y una función `friend` global que manipule los datos privados de la clase.

4. Escriba dos clases, cada una de ellas con un método que reciba como argumento un puntero a un objeto de la otra clase. Cree instancias de ambas clases en `main()` y llame a los métodos antes mencionados de cada clase.

5. Cree tres clases. La primera contiene miembros privados, y declara como `friend` a toda la segunda estructura y a una función miembro de la tercera. En `main()` demuestre que todo esto funciona correctamente.

6. Cree una clase `Hen`. Dentro de ésta, inserte una clase `Nest`. Y dentro de ésta una clase `Egg`. Cada clase debe tener un método `display()`. En `main()`, cree una instancia de cada clase y llame a la función `display()` de cada una.

7. Modifique el ejercicio 6 para que `Nest` y `Egg` contengan datos privados. De acceso mediante `friend` para que las clases puedan acceder a los contenidos privados de las clases que contienen.

8. Cree una clase con atributos diseminados por numerosas secciones `public`, `private` y `protected`. Añada el método `ShowMap()` que imprima por pantalla los nombres de cada uno de esos atributos y su dirección de memoria. Si es posible, compile y ejecute este programa con más de un compilador y/o ordenador y/o sistema operativo para ver si existen diferencias en las posiciones en memoria.

9. Copie la implementación y ficheros de prueba de `Stash` del capítulo 4 para así poder compilar y probar el `Stash.h` de este capítulo.

10. Ponga objetos de la clase `Hern` definidos en el ejercicio 6 en un `Stash`. Apunte a ellos e imprímalos (si no lo ha hecho aún necesitará una función `Hen::print()`).

11. Copie los ficheros de implementación y la prueba de `Stack` del capítulo 4 y compile y pruebe el `Stack2.h` de este capítulo.

12. Ponga objetos de la clase `Hen` del ejercicio 6 dentro de `Stack`. Apunte a ellos e imprímalos (si no lo ha hecho aún, necesitara añadir un `Hen::print()`).

13. Modifique `Chesire` en `Handle.cpp`, y verifique que su entorno de desarrollo recompila y reemplaza sólo este fichero, pero no recompila `UseHandle.cpp`.

14. Cree una clase `StackOfInt` (una pila que guarda enteros) usando la técnica «Gato de Chesire» que esconda la estructura de datos de bajo nivel que usa para guardar los elementos, en una clase llamada `StackImp`. Implemente dos versiones de `StackImp`: una que use un array de longitud fija de enteros, y otra que use un `vector<int>`. Ponga un tamaño máximo para la pila preestablecido, así no se tendrá que preocupar de expandir el array en la primera versión. Fíjese que la clase `StackOfInt.h` no tiene que cambiar con `StackImp`.

6: Inicialización y limpieza

El capitulo 4 constituye una mejora significativa en el uso de librerías tomando los diversos componentes de una librería C típica y encapsulándolos en una estructura (un tipo abstracto de dato, llamado *clase* a partir de ahora).

Esto no sólo permite disponer de un único punto de entrada en un componente de librería, también oculta los nombres de las funciones con el nombre de la clase. Esto le da al diseñador de la clase la posibilidad de establecer límites claros que determinan qué cosas puede hacer el programador cliente y qué queda fuera de sus límites. Eso significa que los mecanismos internos de las operaciones sobre los tipos de datos están bajo el control y la discreción del diseñador de la clase, y deja claro a qué miembros puede y debe prestar atención el programador cliente.

Juntos, la encapsulación y el control de acceso representan un paso significativo para aumentar la sencillez de uso de las librerías. El concepto de «nuevo tipo de dato» que ofrecen es mejor en algunos sentidos que los tipos de datos que incorpora C. El compilador C++ ahora puede ofrecer garantías de comprobación de tipos para esos tipos de datos y así asegura un nivel de seguridad cuando se usan esos tipos de datos.

A parte de la seguridad, el compilador puede hacer mucho más por nosotros de lo que ofrece C. En éste y en próximos capítulos verá posibilidades adicionales que se han incluido en C++ y que hacen que los errores en sus programas casi salten del programa y le agarren, a veces antes incluso de compilar el programa, pero normalmente en forma de advertencias y errores en el proceso de compilación. Por este motivo, pronto se acostumbrará a la extraña situación en que un programa C++ que compila, funciona a la primera.

Dos de esas cuestiones de seguridad son la inicialización y la limpieza. Gran parte de los errores de C se deben a que el programador olvida inicializar o liberar una variable. Esto sucede especialmente con las librerías C, cuando el programador cliente no sabe como inicializar una estructura, o incluso si debe hacerlo. (A menudo las librerías no incluyen una función de inicialización, de modo que el programador cliente se ve forzado a inicializar la estructura a mano). La limpieza es un problema especial porque los programadores C se olvidan de las variables una vez que han terminado, de modo que omiten cualquier limpieza que pudiera ser necesaria en alguna estructura de la librería.

En C++. el concepto de inicialización y limpieza es esencial para facilitar el uso de las librerías y eliminar muchos de los errores sutiles que ocurren cuando el programador cliente olvida cumplir con sus actividades. Este capítulo examina las posibilidades de C++ que ayudan a garantizar una inicialización y limpieza apropiadas.

6.1. Inicialización garantizada por el constructor

Tanto la clase `Stash` como la `Stack` definidas previamente tienen una función llamada `initialize()`. que como indica su nombre se debería llamar antes de usar el objeto. Desafortunadamente, esto significa que el programador cliente debe asegurar una inicialización apropiada. Los programadores cliente son propensos a olvidar detalles como la inicialización cuando tienen prisa por hacer que la librería resuelva sus problemas. En C++, la inicialización en demasiado importante como para dejársela al programador cliente. El diseñador de la clase puede garantizar la inicialización de cada objeto facilitando una función especial llamada *constructor*. Si una clase tiene un constructor, el compilador hará que se llame automáticamente al constructor en el momento de la creación del objeto, antes de que el programador cliente pueda llegar a tocar el objeto. La invocación del constructor no es una opción para el programador cliente; es realizada por el compilador en el punto en el que se define el objeto.

El siguiente reto es cómo llamar a esta función. Hay dos cuestiones. La primera es que no debería ser ningún nombre que pueda querer usar para un miembro de la clase. La segunda es que dado que el compilador es el responsable de la invocación del constructor, siempre debe saber qué función llamar. La solución elegida por Stroustrup parece ser la más sencilla y lógica: el nombre del constructor es el mismo que el de la clase. Eso hace que tenga sentido que esa función sea invocada automáticamente en la inicialización.

Aquí se muestra un clase sencilla con un constructor:

```
class X {
  int i;
public:
  X();  // Constructor
};
```

Ahora, se define un objeto,

```
void f() {
  X a;
  // ...
}
```

Lo mismo pasa si `a` fuese un entero: se pide alojamiento para el objeto. Pero cuando el programa llega al punto de ejecución en el que se define `a`, se invoca el constructor automáticamente. Es decir, el compilador inserta la llamada a `X::X()` para el objeto `a` en el punto de la definición. Como cualquier método, el primer argumento (secreto) para el constructor es el puntero `this` - la dirección del objeto al que corresponde ese método. En el caso del constructor, sin embargo, `this` apunta a un bloque de memoria no inicializado, y el trabajo del constructor es inicializar esa memoria de forma adecuada.

Como cualquier función, el constructor puede tomar argumentos que permitan especificar cómo ha de crearse el objeto, dados unos valores de inicialización. Los argumentos del constructor son una especie de garantía de que todas las partes del objeto se inicializan con valores apropiados. Por ejemplo, si una clase `Tree`[1] tiene un constructor que toma como argumento un único entero que indica la altura del

[1] árbol

árbol, entonces debe crear un objeto árbol como éste:

```
Tree t(12)   // árbol de 12 metros
```

Si `Tree(int)` es el único constructor, el compilador no le permitirá crear un objeto de otro modo. (En el próximo capítulo veremos cómo crear múltiples constructores y diferentes maneras para invocarlos.)

Y realmente un constructor no es más que eso; es una función con un nombre especial que se invoca automáticamente por el compilador para cada objeto en el momento de su creación. A pesar de su simplicidad, tiene un valor excepcional porque evita una gran cantidad de problemas y hace que el código sea más fácil de escribir y leer. En el fragmento de código anterior, por ejemplo, no hay una llamada explícita a ninguna función `initilize()` que, conceptualmente es una función separada de la definición. En C++, la definición e inicialización son conceptos unificados - no se puede tener el uno si el otro.

Constructor y destructor son tipos de funciones muy inusuales: no tienen valor de retorno. Esto es distinto de tener valor de retorno void, que indicaría que la función no retorna nada pero teniendo la posibilidad de hacer otra cosa. Constructores y destructores no retornan nada y no hay otra posibilidad. El acto de traer un objeto al programa, o sacarlo de él es algo especial, como el nacimiento o la muerte, y el compilador siempre hace que la función se llame a si misma, para asegurarse de que ocurre realmente. Si hubiera un valor de retorno, y usted pudiera elegir uno propio, el compilador no tendría forma de saber qué hacer con el valor retornado, o el programador cliente tendría que disponer de una invocación explícita del constructor o destructor, lo que eliminaría la seguridad.

6.2. Limpieza garantizada por el destructor

Como un programador C, a menudo pensará sobre lo importante de la inicialización, pero rara vez piensa en la limpieza. Después de todo, ¿qué hay que limpiar de un int? Simplemente, olvidarlo. Sin embargo, con las librerías, «dejarlo pasar» en un objeto cuando ya no lo necesita no es seguro. Qué ocurre si ese objeto modifica algo en el hardware, o escribe algo en pantalla, o tiene asociado espacio en el montículo(heap). Si simplemente pasa de él, su objeto nunca logrará salir de este mundo. En C++, la limpieza es tan importante como la inicialización y por eso está garantizada por el destructor.

La sintaxis del destructor es similar a la del constructor: se usa el nombre de la clase como nombre para la función. Sin embargo, el destructor se distingue del constructor porque va precedido de una virgulilla (~). Además, el destructor nunca tiene argumentos porque la destrucción nunca necesita ninguna opción. Aquí hay una declaración de un destructor:

```
class Y {
public:
  ~Y();
};
```

El destructor se invoca automáticamente por el compilador cuando el objeto sale del ámbito. Puede ver dónde se invoca al constructor por el punto de la definición del objeto, pero la única evidencia de que el destructor fue invocado es la llave de cierre del ámbito al que pertenece el objeto. El constructor se invoca incluso aunque

utilice `goto` para saltar fuera del del ámbito (`goto` sigue existiendo en C++ por compatibilidad con C.) Debería notar que un `goto` no-local, implementado con las funciones `setjmp` y `longjmp()` de la librería estándar de C, evitan que el destructor sea invocado. (Eso es la especificación, incluso si su compilador no lo implementa de esa manera. Confiar un una característica que no está en la especificación significa que su código no será portable).

A continuación, un ejemplo que demuestra las características de constructores y destructores que se han mostrado hasta el momento.

```
//: C06:Constructor1.cpp
// Constructors & destructors
#include <iostream>
using namespace std;

class Tree {
  int height;
public:
  Tree(int initialHeight);  // Constructor
  ~Tree();  // Destructor
  void grow(int years);
  void printsize();
};

Tree::Tree(int initialHeight) {
  height = initialHeight;
}

Tree::~Tree() {
  cout << "inside Tree destructor" << endl;
  printsize();
}

void Tree::grow(int years) {
  height += years;
}

void Tree::printsize() {
  cout << "Tree height is " << height << endl;
}

int main() {
  cout << "before opening brace" << endl;
  {
    Tree t(12);
    cout << "after Tree creation" << endl;
    t.printsize();
    t.grow(4);
    cout << "before closing brace" << endl;
  }
  cout << "after closing brace" << endl;
} ///:~
```

Y esta sería la salida del programa anterior:

```
antes de la llave de aperturaé
```

```
despus de la ócreacin de Tree
la altura del árbol es 12
antes de la llave de cierre
dentro del destructor de Tree
la altura del árbol es 16é
despus de la llave de cierre
```

Puede ver que el destructor se llama automáticamente al acabar el ámbito (llave de cierre) en el que está definido el objeto.

6.3. Eliminación del bloque de definiciones

En C, siempre se definen todas las variables al principio de cada bloque, justo después de la llave de apertura. Ése es un requisito habitual en los lenguajes de programación, y la razón que se da a menudo es que se considera «buenas prácticas de programación». En este tema, yo tengo mis sospechas. Eso siempre me pareció un inconveniente, como programador, volver al principio del bloque cada vez que necesitaba definir una nueva variable. También encuentro más legible el código cuando la definición de la variable está certa del punto donde se usa.

Quizá esos argumentos son estilísticos. En C++, sin embargo, existe un problema significativo si se fuerza a definir todos los objetos al comienzo un ámbito. Si existe un constructor, debe invocarse cuando el objeto se crea. Sin embargo, si el constructor toma uno o más argumentos, ¿cómo saber que se dispone de la información de inicialización al comienzo del ámbito? Generalmente no se dispone de esa información. Dado que C no tiene el concepto de privado, la separación entre definición e inicialización no es un problema. Además, C++ garantiza que cuando se crea un objeto, es inicializado simultáneamente. Esto asegura que no se tendrán objetos no inicializados ejecutándose en el sistema. C no tiene cuidado, de hecho, C promueve esta práctica ya que obliga a que se definan las variables al comienzo de un bloque, antes de disponer de la información de inicialización necesaria [2].

En general, C++ no permite crear un objeto antes de tener la información de inicialización para el constructor. Por eso, el lenguaje no sería factible si tuviera que definir variables al comienzo de un bloque. De hecho, el estilo del lenguaje parece promover la definición de un objeto tan cerca como sea posible del punto en el que se usa. En C++, cualquier regla que se aplica a un «objeto» automáticamente también se refiere a un objeto de un tipo básico. Esto significa que cualquier clase de objeto o variable de un tipo básico también se puede definir en cualquier punto del bloque. Eso también significa que puede esperar hasta disponer de la información para una variable antes de definirla, de modo que siempre puede definir e inicializar al mismo tiempo:

```
//: C06:DefineInitialize.cpp
// Defining variables anywhere
#include "../require.h"
#include <iostream>
#include <string>
using namespace std;

class G {
  int i;
public:
  G(int ii);
```

[2] C99, la versión actual del Estándar de C, permite definir variables en cualquier punto del bloque, como C++

```
};

G::G(int ii) { i = ii; }

int main() {
  cout << "initialization value? ";
  int retval = 0;
  cin >> retval;
  require(retval != 0);
  int y = retval + 3;
  G g(y);
} ///:~
```

Puede ver que se ejecuta parte del código, entonces se define `>retval`, que se usa para capturar datos de la consola, y entonces se definen `y` y `g`. C, al contrario, no permite definir una variable en ningún sitio que no sea el comienzo de un bloque.

En general, debería definir las variables tan cerca como sea posible del punto en que se usa, e inicializarlas siempre cuando se definen. (Ésta es una sugerencia de estilo para tipos básicos, en los que la inicialización es opcional.) Es una cuestión de seguridad. Reduciendo la duración de disponibilidad al bloque, se reduce la posibilidad de que sea usada inapropiadamente en otra parte del bloque. En resumen, la legibilidad mejora porque el lector no teiene que volver al inicio del bloque para ver el tipo de una variable.

6.3.1. Bucles `for`

En C++, a menudo verá bucles `for` con el contador definido dentro de la propia expresión.

```
for (int j = 0; j < 100; j++) {
    cout << "j = " << j << endl;
}
for (int i = 0; i < 100; i++)
    cout  << "i = " << i << endl;
```

Las sentencias anteriores son casos especiales importantes, que provocan confusión en los programadores novatos de C++.

Las variables `i` y `j` están definidas directamente dentro la expresión `for` (algo que no se puede hacer en C). Esas variables están disponibles para usarlas en el bucle. Es una sintaxis muy conveniente porque el contexto disipa cualquier duda sobre el proposito de `i` y `j`, asi que no necesita utilizar nombres extraños como `contador_bucle_i` para quede más claro.

Sin embargo, podría resultar confuso si espera que la vida de las variables `i` y `j` continúe después del bucle - algo que no ocurre[3]

El capítulo 3 indica que las sentencias `while` y `switch` también permiten la definición de objetos en sus expresiones de control, aunque ese uso es menos importante que con el bucle `for`.

[3] Un reciente borrador del estándar C++ dice que la vida de la variable se extiende hasta el final del ámbito que encierra el bucle `for`. Algunos compiladores lo implementan, pero eso no es correcto de modo que su código sólo será portable si limita el ámbito al bucle `for`.

Hay que tener cuidado con las variables locales que ocultan las variables del ámbito superior. En general, usar el mismo nombre para una variable anidada y una variable que es global en ese ámbito es confuso y propenso a errores[4]

Creo que los bloques pequeños son un indicador de un buen diseño. Si una sola función requiere varias páginas, quizá está intentando demasiadas cosas en esa función. Funciones de granularidad más fina no sólo son más útiles, también facilitan la localización de errores.

6.3.2. Alojamiento de memoria

Ahora una variable se puede definir en cualquier parte del bloque, podría parecer que el alojamiento para una variable no se puede llevar a cabo hasta el momento en que se define. En realidad, lo más probable es que el compilador siga la práctica de pedir todo el alojamiento para el bloque en la llave de apertura del bloque. No importa porque, como programador, no puede acceder al espacio asociado (es decir, el objeto) hasta que ha sido definido[5]. Aunque el espacio se pida al comienzo del bloque, la llamada al constructor no ocurre hasta el punto en el que se define el objeto ya que el identificador no está disponible hasta entonces. El compilador incluso comprueba que no ponga la definición del objeto (y por tanto la llamada al constructor) en un punto que dependa de una sentencia condicional, como en una sentencia `switch` o algún lugar que pueda saltar un `goto`. Descomentar las sentencias del siguiente código generará un error o aviso.

```
//: C06:Nojump.cpp
// Can't jump past constructors

class X {
public:
  X();
};

X::X() {}

void f(int i) {
  if(i < 10) {
   //! goto jump1; // Error: goto bypasses init
  }
  X x1;  // Constructor called here
 jump1:
  switch(i) {
    case 1 :
      X x2;  // Constructor called here
      break;
  //! case 2 : // Error: case bypasses init
      X x3;  // Constructor called here
      break;
  }
}

int main() {
  f(9);
  f(11);
```

[4] El lenguaje Java considera esto una idea tan mala que lo considera un error.
[5] De acuerdo, probablemente podría trucarlo usando punteros, pero sería muy, muy malo

```
}///:~
```

En el código anterior, tanto el `goto` como el `switch` pueden saltar la sentencia en la que se invoca un constructor. Ese objeto corresponde al ámbito incluso si no se invoca el constructor, de modo que el compilador dará un mensaje de error. Esto garantiza de nuevo que un objeto no se puede crear si no se inicializa.

Todo el espacio de almacenamiento necesario se asigna en la pila, por supuesto. Ese espacio lo faciliza el compilador moviendo el puntero de pila «hacia abajo» (dependiendo de la máquina implica incrementar o decrementar el valor del puntero de pila). Los objetos también se pueden alojar en el montículo usando `new`, algo que se verá en el capítulo 13. (FIXME:Ref C13)

6.4. Stash con constructores y destructores

Los ejemplos de los capítulos anteriores tienen funciones que tienen correspondencia directa con constructores y destructores: `initialize()` y `cleanup()`. Éste es el fichero de cabecera de `Stash`, utilizando constructor y destructor:

```
//: C06:Stash2.h
// With constructors & destructors
#ifndef STASH2_H
#define STASH2_H

class Stash {
  int size;      // Size of each space
  int quantity;  // Number of storage spaces
  int next;      // Next empty space
  // Dynamically allocated array of bytes:
  unsigned char* storage;
  void inflate(int increase);
public:
  Stash(int size);
  ~Stash();
  int add(void* element);
  void* fetch(int index);
  int count();
};
#endif // STASH2_H ///:~
```

Las únicas definiciones de métodos que han cambiado son `initialize()` y `cleanup()`, que han sido reemplazadas con un constructor y un destructor.

```
//: C06:Stash2.cpp {O}
// Constructors & destructors
#include "Stash2.h"
#include "../require.h"
#include <iostream>
#include <cassert>
using namespace std;
const int increment = 100;

Stash::Stash(int sz) {
```

```
  size = sz;
  quantity = 0;
  storage = 0;
  next = 0;
}

int Stash::add(void* element) {
  if(next >= quantity) // Enough space left?
    inflate(increment);
  // Copy element into storage,
  // starting at next empty space:
  int startBytes = next * size;
  unsigned char* e = (unsigned char*)element;
  for(int i = 0; i < size; i++)
    storage[startBytes + i] = e[i];
  next++;
  return(next - 1); // Index number
}

void* Stash::fetch(int index) {
  require(0 <= index, "Stash::fetch (-)index");
  if(index >= next)
    return 0; // To indicate the end
  // Produce pointer to desired element:
  return &(storage[index * size]);
}

int Stash::count() {
  return next; // Number of elements in CStash
}

void Stash::inflate(int increase) {
  require(increase > 0,
    "Stash::inflate zero or negative increase");
  int newQuantity = quantity + increase;
  int newBytes = newQuantity * size;
  int oldBytes = quantity * size;
  unsigned char* b = new unsigned char[newBytes];
  for(int i = 0; i < oldBytes; i++)
    b[i] = storage[i]; // Copy old to new
  delete [](storage); // Old storage
  storage = b; // Point to new memory
  quantity = newQuantity;
}

Stash::~Stash() {
  if(storage != 0) {
   cout << "freeing storage" << endl;
   delete []storage;
  }
} ///:~
```

Puede ver que las funciones de `require.h` se usan para vigilar errores del programador, en lugar de `assert()`. La salida de un `assert()` fallido no es tan útil como las funciones de `require.h` (que se verán más adelante en el libro).

Dado que `inflate()` es privado, el único modo en que `require()` podría fallar sería si uno de los otros miembros pasara accidentalmente un valor incorrecto a `inflate()`. Si está seguro de que eso no puede pasar, debería considerar eliminar el `require()`, pero debería tener en mente que hasta que la clase sea estable, siempre existe la posibilidad de que el código nuevo añadido a la clase podría provocar errores. El coste de `require()` es bajo (y podría ser eliminado automáticamente por el preprocesador) mientras que la robustez del código es alta.

Fijese cómo en el siguiente programa de prueba la definición de los objetos `Stash` aparece justo antes de necesitarse, y cómo la inicialización aparece como parte de la definición, en la lista de argumentos del constructor.

```
//: C06:Stash2Test.cpp
//{L} Stash2
// Constructors & destructors
#include "Stash2.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
  Stash intStash(sizeof(int));
  for(int i = 0; i < 100; i++)
    intStash.add(&i);
  for(int j = 0; j < intStash.count(); j++)
    cout << "intStash.fetch(" << j << ") = "
         << *(int*)intStash.fetch(j)
         << endl;
  const int bufsize = 80;
  Stash stringStash(sizeof(char) * bufsize);
  ifstream in("Stash2Test.cpp");
  assure(in, " Stash2Test.cpp");
  string line;
  while(getline(in, line))
    stringStash.add((char*)line.c_str());
  int k = 0;
  char* cp;
  while((cp = (char*)stringStash.fetch(k++))!=0)
    cout << "stringStash.fetch(" << k << ") = "
         << cp << endl;
} ///:~
```

También observe que se han eliminado llamadas a `cleanup()`, pero los destructores se llaman automáticamente cuando `intStash` y `stringStash` salen del ámbito.

Una cosa de la que debe ser consciente en los ejemplos con `Stash`: Tengo mucho cuidado usando sólo tipos básicos; es decir, aquellos sin destructores. Si intenta copiar objetos dentro de `Stash`, aparecerán todo tipo de problemas y no funcionará bien. En realidad la Librería Estándar de C++ puede hacer copias correctas de objetos en sus contenedores, pero es un proceso bastante sucio y complicado. En el siguiente ejemplo de `Stack`, verá que se utilizan punteros para esquivar esta cuestión, y en un capítulo posterior `Stash` también se convertirá para que use punteros.

6.5. Stack con constructores y destructores

Reimplementar la lista enlazada (dentro de `Stack`) con constructores y destructores muestra claramente cómo costructores y destructores utilizan `new` y `delete`. Éste es el fichero de cabecera modficado:

```
//: C06:Stack3.h
// With constructors/destructors
#ifndef STACK3_H
#define STACK3_H

class Stack {
  struct Link {
    void* data;
    Link* next;
    Link(void* dat, Link* nxt);
    ~Link();
  }* head;
public:
  Stack();
  ~Stack();
  void push(void* dat);
  void* peek();
  void* pop();
};
#endif // STACK3_H ///:~
```

No sólo hace que `Stack` tenga un constructor y destructor, también aparece la clase anidada `Link`.

```
//: C06:Stack3.cpp {O}
// Constructors/destructors
#include "Stack3.h"
#include "../require.h"
using namespace std;

Stack::Link::Link(void* dat, Link* nxt) {
  data = dat;
  next = nxt;
}

Stack::Link::~Link() { }

Stack::Stack() { head = 0; }

void Stack::push(void* dat) {
  head = new Link(dat,head);
}

void* Stack::peek() {
  require(head != 0, "Stack empty");
  return head->data;
}

void* Stack::pop() {
  if(head == 0) return 0;
```

```
  void* result = head->data;
  Link* oldHead = head;
  head = head->next;
  delete oldHead;
  return result;
}

Stack::~Stack() {
  require(head == 0, "Stack not empty");
} ///:~
```

El constructor `Link:Link()` simplemente inicializa los punteros `data` y `next`, así que en `Stack::push()`, la línea:

```
head = new Link(dat,head);
```

no sólo aloja un nuevo enlace (usando creación dinámica de objetos con la sentencia `new`, vista en el capítulo 4), también inicializa los punteros para ese enlace.

Puede que le asombre que el destructor de `Link` no haga nada - en concreto, ¿por qué no elimina el puntero `data`? Hay dos problemas. En el capítulo 4, en el que apareció `Stack`, se decía que no puede eliminar un puntero void si está apuntado a un objeto (una afirmación que se demostrará en el capítulo 13). Pero además, si el destructor de `Link` eliminara el puntero `data`, `pop()` retornaría un puntero a un objeto borrado, que definitivamente supone un error. A veces esto se considera como una cuestión de *propiedad*: `Link` y por consiguiente `Stack` sólo contienen los punteros, pero no son responsables de su limpieza. Eso significa que debe tener mucho cuidado para saber quién es el responsable. Por ejemplo, si no invoca `pop()` y elimina todos los punteros de `Stack()`, no se limpiarán automáticamente por el destructor de `Stack`. Esto puede ser una cuestión engorrosa y llevar a fugas de memoria, de modo que saber quién es el responsable de la limpieza de un objeto puede suponer la diferencia entre un programa correcto y uno erroneo - es decir, porqué `Stack-::~Stack()` imprime un mensaje de error si el objeto `Stack` no está vacío en el momento su destrucción.

Dado que el alojamiento y limpieza de objetos `Link` está oculto dentro de `Stack` - es parte de la implementación subyacente - no verá este suceso en el programa de prueba, aunque será el responsable de eliminar los punteros que devuelva `pop()`:

```
//: C06:Stack3Test.cpp
//{L} Stack3
//{T} Stack3Test.cpp
// Constructors/destructors
#include "Stack3.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // File name is argument
  ifstream in(argv[1]);
  assure(in, argv[1]);
  Stack textlines;
```

```
  string line;
  // Read file and store lines in the stack:
  while(getline(in, line))
    textlines.push(new string(line));
  // Pop the lines from the stack and print them:
  string* s;
  while((s = (string*)textlines.pop()) != 0) {
    cout << *s << endl;
    delete s;
  }
} ///:~
```

En este caso, todas las líneas de `textlines` son desapiladas y eliminadas, pero si no fuese así, obtendría un mensaje de `require()` que indica que hubo una fuga de memoria.

6.6. Inicialización de tipos agregados

Un agregado es justo lo que parece: un grupo de cosas agrupados juntos. Esta definición incluye agregados de tipos mixtos, como estructuras o clases. Un array es un agregado de un único tipo.

Inicializar agregados puede ser tedioso y propenso a errores. La inicialización de agregados en C++ lo hace mucho más seguro. Cuando crea un objeto agregado, todo lo que tiene que hacer es una asignación, y la inicialización la hará el compilador. Esta asignación tiene varias modalidades, dependiendo del tipo de agregado del que se trate, pero en cualquier caso los elementos en la asignación deben estar rodeadas de llaves. Para arrays de tipos básicos es bastante simple:

```
int a[5] = { 1, 2, 3, 4, 5};
```

Si intenta escribir más valores que elementos tiene el array, el compilador dará un mensaje de error. Pero, ¿qué ocurre si escribe menos valores? Por ejemplo:

```
int b[6] = {0};
```

Aquí, el compilador usará el primer valor para el primer elemento del array, y después usará ceros para todos los elementos para los que no se tiene un valor. Fíjese en que este comportamiento en la inicialización no ocurre si define un array sin una lista de valores de inicialización. Así que la expresión anterior es una forma resumida de inicializar a cero un array sin usar un bucle `for`, y sin ninguna posibilidad de un «error por uno» (Dependiendo del compilador, también puede ser más eficiente que un bucle `for`).

Un segundo método para los arrays es el conteo automático, en el cual se permite que el compilador determine el tamaño del array basándose en el número de valores de inicialización.

```
int c[] = { 1, 2, 3, 4 };
```

Ahora, si decide añadir otro elemento al array, simplemente debe añadir otro valor. Si puede hacer que su código necesite modificaciones en un único sítio, reducirá

la posibilidad de introducir errores durante la modificación. Pero, ¿cómo determinar el tamaño del array? La expresión `sizeof c / sizeof *c` (el tamaño del array completo dividido entre el tamaño del primer elemento) es un truco que hace que no sea necesario cambiarlo si cambia el tamaño del array [6]:

```
for(int i = 0; i < sizeof c / sizeof *c; i++)
  c[i]++;
```

Dado que las estructuras también son agregados, se pueden inicializar de un modo similar. Como en una estructura estilo-C todos sus miembros son públicos, se pueden asignar directamente:

```
struct X {
  int i;
  float f;
  char c;
};

X x1 = { 1, 2.2, 'c'};
```

Si tiene una array de esos objetos, puede inicializarlos usando un conjunto anidado de llaves para cada elemento:

```
X x2[3] = { {1, 1.1, 'a'}, {2, 2.2, 'b'} };
```

Aquí, el tercer objeto se inicializó a cero.

Si alguno de los atributos es privado (algo que ocurre típicamente en el caso de clases bien diseñadas en C++), o incluso si todos son públicos pero hay un constructor, las cosas son distintas. En el ejemplo anterior, los valores se han asignado directamente a los elementos del agregado, pero los constructores son una manera de forzar que la inicialización ocurra por medio de una interfaz formal. Aquí, los constructores deben ser invocados para realizar la inicialización. De modo, que si tiene un constructor parecido a éste,

```
struct Y {
  float f;
  int i;
  Y(int a);
};
```

Debe indicar la llamada al constructor. La mejor aproximación es una explícita como la siguiente:

```
Y y1[] = { Y(1), Y(2), Y(3) };
```

Obtendrá tres objetos y tres llamadas al constructor. Siempre que tenga un constructor, si es una estructura con todos sus miembros públicos o una clase con atributos privados, toda la inicialización debe ocurrir a través del constructor, incluso si está usando la inicialización de agregados.

[6] En el segundo volumen de este libro (disponible libremente en www.BruceEckel.com), verá una forma más corta de calcular el tamaño de un array usando plantillas.

Se muestra un segundo ejemplo con un constructor con múltiples argumentos.

```
//: C06:Multiarg.cpp
// Multiple constructor arguments
// with aggregate initialization
#include <iostream>
using namespace std;

class Z {
  int i, j;
public:
  Z(int ii, int jj);
  void print();
};

Z::Z(int ii, int jj) {
  i = ii;
  j = jj;
}

void Z::print() {
  cout << "i = " << i << ", j = " << j << endl;
}

int main() {
  Z zz[] = { Z(1,2), Z(3,4), Z(5,6), Z(7,8) };
  for(int i = 0; i < sizeof zz / sizeof *zz; i++)
    zz[i].print();
} ///:~
```

Fíjese en cómo se invoca un constructor explícito para cada objeto de un array.

6.7. Constructores por defecto

Un *constructor por defecto* es uno que puede ser invocado sin argumentos. Un constructor por defecto se usa para crear un «objeto vainilla»[7] pero también es importante cuando el compilador debe crear un objeto pero no se dan detalles. Por ejemplo, si se toma la `struct Y` definida previamente y se usa en una definición como ésta,

```
Y y2[2] = { Y(1) };
```

el compilador se quejará porque no puede encontrar un constructor por defecto. El segundo objeto del array se creará sin argumentos, y es ahí donde el compilador busca un constructor por defecto. De hecho, si simplemente define un array de objetos `Y`,

```
Y y3[7];
```

el compilador se quejará porque debería haber un constructor para inicializar cada objeto del array.

[7] N.de.T: Para los anglosajones *Vainilla* es el sabor más «sencillo», sin adornos ni sofisticaciones.

El mismo problema ocurre si crea un objeto individual como éste:

```
Y y4;
```

Recuerde, si tiene un constructor, el compilador asegura que siempre ocurrirá la construcción, sin tener en cuenta la situación.

El constructor por defecto es tan importante que si (y sólo si) una estructura (`struct` o `clase`) no tiene constructor, el compilador creará uno automáticamente. Por ello, lo siguiente funciona:

```
//: C06:AutoDefaultConstructor.cpp
// Automatically-generated default constructor

class V {
  int i;  // private
}; // No constructor

int main() {
  V v, v2[10];
} ///:~
```

Si se han definido constructores, pero no hay constructor por defecto, las instancias anteriores de `V` provocarán errores durante la compilación.

Podría pensarse que el constructor sintetizado por el compilador debería hacer alguna inicialización inteligente, como poner a cero la memoria del objeto. Pero no lo hace - añadiría una sobrecarga que quedaría fuera del control del programador. Si quiere que la memoria sea inicializada a cero, debería hacerlo escribiendo un constructor por defecto explícito.

Aunque el compilador creará un constructor por defecto, el comportamiento de ese constructor raramente hará lo que se espera. Debería considerar esta característica como una red de seguridad, pero que debe usarse con moderación. En general, debería definir sus constructores explicitamente y no permitir que el compilador lo haga por usted.

6.8. Resumen

Los mecanismos aparentemente elaborados proporcionados por C++ deberían darle una idea de la importancia crítica que tiene en el lenguaje la inicialización y limpieza. Como Stroustrup fue quien diseño C++, una de las primeras observaciones que hizo sobre la productividad de C fue que una parte importante de los problemas de programación se deben a la inicialización inapropiada de las variables. Este tipo de errores son difíciles de encontrar, y otro tanto se puede decir de una limpieza inapropiada. Dado que constructores y destructores le permiten garantizar una inicialización y limpieza apropiada (el compilador no permitirá que un objeto sea creado o destruido sin la invocación del constructor y destructor correspondiente), conseguirá control y seguridad.

La inicialización de agregados está incluida de un modo similar - previene de errores de inicialización típicos con agregados de tipos básicos y hace que el código sea más corto.

La seguridad durante la codificación es una cuestión importante en C++. La inicialización y la limpieza son una parte importante, pero también verá otras cuestiones de seguridad más adelante en este libro.

6.9. Ejercicios

Las soluciones a los ejercicios se pueden encontrar en el documento electrónico titulado «The Thinking in C++ Annotated Solution Guide», disponible por poco dinero en www.BruceEckel.com.

1. Escriba una clase simple llamada `Simple` con un constructor que imprima algo indicando que se ha invocado. En `main()` creae un objeto de esa clase.
2. Añada un destructor al Ejercicio 1 que imprima un mensaje indicado que se ha llamado.
3. Modifique el Ejercicio 2 de modo que la clase contenga un miembro int. Modifique el constructor para que tome un argumento int que se almacene en el atributo. Tanto el constructor como el destructor deberán imprimir el valor del entero como parte se su mensaje, de modo que se pueda ver cómo se crean y destruyen los objetos.
4. Demuestre que los destructores se invocan incluso cuando se utiliza `goto` para salir de un bucle.
5. Escriba dos bucles `for` que impriman los valores de 0 a 10. En el primero, defina el contador del bucle antes del bucle, y en el segundo, defina el contador en la expresión de control del `for`. En la segunda parte del ejercicio, modifique el identificador del segundo bucle para que tenga el mismo nombre del el contador del primero y vea que hace el compilador.
6. Modifique los ficheros `Handle.h`, `Handle.cpp`, y `UseHandle.cpp` del capítulo 5 para que usen constructores y destructores.
7. Use inicialización de agregados para crear un array de double en el que se indique el tamaño del array pero no se den suficientes elementos. Imprima el array usando `sizeof` para determinar el tamaño del array. Ahora cree un array de double usando inicialización de agregados y conteo automático. Imprima el array.
8. Utilice inicialización de agregados para crear un array de objetos string. Cree una `Stack` para guardar esas cadenas y recorra el array, apilando cada cadena en la pila. Finalmente, extraiga las cadenas de la pila e imprima cada una de ellas.
9. Demuestre el conteo automático e inicialización de agregados con un array de objetos de la clase creada en el Ejercicio 3. Añada un método a la clase que imprima un mensaje. Calcule el tamaño del array y recórralo, llamando al nuevo método.
10. Cree una clase sin ningún constructor, y demuestre que puede crear objetos con el constructor por defecto. Ahora cree un constructor explícito (que tenga un argumento) para la clase, e intente compilar de nuevo. Explique lo que ocurre.

7: Sobrecarga de funciones y argumentos por defecto

Una de las características más importantes en cualquier lenguaje de programación es la utilización adecuada de los nombres.

Cuando crea un objeto (una variable) le está asignando un nombre a una región de memoria. Una función es un nombre para una acción. El hecho de poner nombres adecuados a la hora de describir un sistema hace que un programa sea más fácil de entender y modificar. Es muy parecido a la prosa escrita, el objetivo es comunicarse con los lectores.

Cuando se trata de representar sutilezas del lenguaje humano en un lenguaje de programación aparecen los problemas. A menudo, la misma palabra expresa diversos significados dependiendo del contexto. Una palabra tiene múltiples significados, es decir, está sobrecargada (polisemia). Esto es muy útil, especialmente cuando las diferencias son obvias. Puede decir «lave la camiseta, lave el coche.» Sería estúpido forzar la expresión anterior para convertirla en «lavar_camiseta la camiseta, lavar_coche el coche» pues el oyente no tiene que hacer ninguna distinción sobre la acción realizada. Los lenguajes humanos son muy redundantes, así que incluso si pierde algunas palabras, todavía puede determinar el significado. Los identificadores únicos no son necesarios, pues se puede deducir el significado a partir del contexto.

Sin embargo, la mayoría de los lenguajes de programación requieren que se utilice un identificador único para cada función. Si tiene tres tipos diferentes de datos que desee imprimir en la salida: int, char y float, generalmente tiene que crear tres funciones diferentes, como por ejemplo `print_int()`, `print_char()` y `print_float()`. Esto constituye un trabajo extra tanto para el programador, al escribir el programa, como para el lector que trate de entenderlo.

En C++ hay otro factor que fuerza la sobrecarga de los nombres de función: el constructor. Como el nombre del constructor está predeterminado por el nombre de la clase, podría parecer que sólo puede haber un constructor. Pero, ¿qué ocurre si desea crear un objeto de diferentes maneras? Por ejemplo, suponga que escribe una clase que puede inicializarse de una manera estándar o leyendo información de un fichero. Necesita dos constructores, uno que no tiene argumentos (el constructor por defecto) y otro que tiene un argumento de tipo string, que es el nombre del fichero que inicializa el objeto. Ambos son constructores, así pues deben tener el mismo nombre: el nombre de la clase. Así, la sobrecarga de funciones es esencial para permitir el mismo nombre de función (el constructor en este caso) se utilice con diferentes argumentos.

Aunque la sobrecarga de funciones es algo imprescindible para los constructo-

res, es también de utilidad general para cualquier función, incluso aquellas que no son métodos. Además, la sobrecarga de funciones significa que si tiene dos librerías que contienen funciones con el mismo nombre, no entrarán en conflicto siempre y cuando las listas de argumentos sean diferentes. A lo largo del capítulo se mostrarán todos los detalles.

El tema de este capítulo es la elección adecuada de los nombres de la funciones. La sobrecarga de funciones permite utilizar el mismo nombre para funciones diferentes, pero hay otra forma más adecuada de llamar a una función. ¿Qué ocurriría si le gustara llamar a la misma función de formas diferentes? Cuando las funciones tienen una larga lista de argumentos, puede resultar tediosa la escritura (y confusa la lectura) de las llamadas a la función cuando la mayoría de los argumentos son lo mismos para todas las llamadas. Una característica de C++ comúnmente utilizada se llama *argumento por defecto*. Un argumento por defecto es aquel que el compilador inserta en caso de que no se especifique cuando se llama a la función. Así, las llamadas `f("hello")`, `f("hi", 1)` y `f("howdy", 2, 'c')` pueden ser llamadas a la misma función. También podrían ser llamadas a tres funciones sobrecargadas, pero cuando las listas de argumentos son tan similares, querrá que tengan un comportamiento similar, que le lleva a tener una única función.

La sobrecarga de funciones y los argumentos por defecto no son muy complicados. En el momento en que termine este capítulo, sabrá cuándo utilizarlos y entenderá los mecanismos internos que el compilador utiliza en tiempo de compilación y enlace.

7.1. Más decoración de nombres

En el Capítulo 4 se presentó el concepto de *decoración de nombres*. En el código:

```
void f();
class X { void f(); };
```

La función `f()` dentro del ámbito de la clase `X` no entra en conflicto con la versión global de `f()`. El compilador resuelve los ámbitos generando diferentes nombres internos tanto para la versión global de `f()` como para `X::f()`. En el Capítulo 4 se sugirió que los nombres son simplemente el nombre de la clase junto con el nombre de la función. Un ejemplo podría ser que el compilador utilizara como nombres `_f` y `_X_f`. Sin embargo ahora se ve que la decoración del nombre de la función involucra algo más que el nombre de la clase.

He aquí el porqué. Suponga que quiere sobrecargar dos funciones

```
void print(char);
void print(float);
```

No importa si son globales o están dentro de una clase. El compilador no puede generar identificadores internos únicos si sólo utiliza el ámbito de las funciones. Terminaría con `_print` en ambos casos. La idea de una función sobrecargada es que se utilice el mismo nombre de función, pero diferente lista de argumentos. Así pues, para que la sobrecarga funcione el compilador ha de decorar el nombre de la función con los nombres de los tipos de los argumentos. Las funciones planteadas más arriba, definidas como globales, producen nombres internos que podrían parecerse a algo así como `_print_char` y `_print_float`. Nótese que como no hay

ningún estándar de decoración, podrá obtener resultados diferentes de un compilador a otro. (Puede ver lo que saldría diciéndole al compilador que genere código fuente en ensamblador). Esto, por supuesto, causa problemas si desea comprar unas librerías compiladas por un compilador y enlazador particulares, aunque si la decoración de nombres fuera estándar, habría otros obstáculos debido a las diferencias de generación de código máquina entre compiladores.

Esto es todo lo que hay para la sobrecarga de funciones: puede utilizar el mismo nombre de función siempre y cuando la lista de argumentos sea diferente. El compilador utiliza el nombre, el ámbito y la lista de argumentos para generar un nombre interno que el enlazador pueda utilizar.

7.1.1. Sobrecarga en el valor de retorno

Es muy común la pregunta «¿Por qué solamente el ámbito y la lista de argumentos? ¿Por qué no también el valor de retorno?». A primera vista parece que tendría sentido utilizar también el valor de retorno para la decoración del nombre interno. De esta manera, también podría sobrecargar con los valores de retorno:

```
void f();
int f();
```

Esto funciona bien cuando el compilador puede determinar sin ambigüedades a qué tipo de valor de retorno se refiere, como en `int x = f();`. No obstante, en C se puede llamar a una función y hacer caso omiso del valor de retorno (esto es, puede querer llamar a la función debido a sus *efectos laterales*). ¿Cómo puede el compilador distinguir a qué función se refiere en este caso? Peor es la dificultad que tiene el lector del código fuente para dilucidar a qué función se refiere. La sobrecarga mediante el valor de retorno solamente es demasiado sutil, por lo que C++ no lo permite.

7.1.2. Enlace con FIXME:tipos seguros

Existe un beneficio añadido a la decoración de nombres. En C hay un problema particularmente fastidioso cuando un programador cliente declara mal una función o, aún peor, se llama a una función sin haber sido previamente declarada, y el compilador infiere la declaración de la función mediante la forma en que se llama. Algunas veces la declaración de la función es correcta, pero cuando no lo es, suele resultar en un fallo difícil de encontrar.

A causa de que en C++ se *deben* declarar todas las funciones antes de llamarlas, las probabilidades de que ocurra lo anteriormente expuesto se reducen drásticamente. El compilador de C++ rechaza declarar una función automáticamente, así que es probable que tenga que incluir la cabecera apropiada. Sin embargo, si por alguna razón se las apaña para declarar mal una función, o declararla a mano o incluir una cabecera incorrecta (quizá una que sea antigua), la decoración de nombres proporciona una seguridad que a menudo se denomina como *enlace con tipos seguros*.

Considere el siguiente escenario. En un fichero está la definición de una función:

```
//: C07:Def.cpp {O}
// Function definition
void f(int) {}
///:~
```

En el segundo fichero, la función está mal declarada y en `main` se le llama:

```
//: C07:Use.cpp
//{L} Def
// Function misdeclaration
void f(char);

int main() {
//!  f(1); // Causes a linker error
} ///:~
```

Incluso aunque pueda ver que la función es realmente `f(int)`, el compilador no lo sabe porque se le dijo, a través de una declaración explícita, que la función es `f(char)`. Así pues, la compilación tiene éxito. En C, el enlazador podría tener también éxito, pero *no* en C++. Como el compilador decora los nombres, la definición se convierte en algo así como `f_int`, mientras que se trata de utilizar `f_char`. Cuando el enlazador intenta resolver la referencia a `f_char`, sólo puede encontrar `f_int`, y da un mensaje de error. Éste es el enlace de tipos seguro. Aunque el problema no ocurre muy a menudo, cuando ocurre puede ser increíblemente difícil de encontrar, especialmente en proyectos grandes. Éste método puede utilizarse para encontrar un error en C simplemente intentando compilarlo en C++.

7.2. Ejemplo de sobrecarga

Ahora puede modificar ejemplos anteriores para utilizar la sobrecarga de funciones. Como ya se dijo, el lugar inmediatamente más útil para la sobrecarga es en los constructores. Puede comprobarlo en la siguiente versión de la clase `Stash`:

```
//: C07:Stash3.h
// Function overloading
#ifndef STASH3_H
#define STASH3_H

class Stash {
  int size;      // Size of each space
  int quantity;  // Number of storage spaces
  int next;      // Next empty space
  // Dynamically allocated array of bytes:
  unsigned char* storage;
  void inflate(int increase);
public:
  Stash(int size); // Zero quantity
  Stash(int size, int initQuantity);
  ~Stash();
  int add(void* element);
  void* fetch(int index);
  int count();
};
#endif // STASH3_H ///:~
```

El primer constructor de `Stash` es el mismo que antes, pero el segundo tiene un argumento `Quantity` que indica el número inicial de espacios de memoria que podrán ser asignados. En la definición, puede observar que el valor interno de `quantity` se pone a cero, al igual que el puntero `storage`. En el segundo constructor, la llamada a `inflate(initQuantity)` incrementa `quantity` al tamaño asignado:

```
//: C07:Stash3.cpp {O}
// Function overloading
#include "Stash3.h"
#include "../require.h"
#include <iostream>
#include <cassert>
using namespace std;
const int increment = 100;

Stash::Stash(int sz) {
  size = sz;
  quantity = 0;
  next = 0;
  storage = 0;
}

Stash::Stash(int sz, int initQuantity) {
  size = sz;
  quantity = 0;
  next = 0;
  storage = 0;
  inflate(initQuantity);
}

Stash::~Stash() {
  if(storage != 0) {
    cout << "freeing storage" << endl;
    delete []storage;
  }
}

int Stash::add(void* element) {
  if(next >= quantity) // Enough space left?
    inflate(increment);
  // Copy element into storage,
  // starting at next empty space:
  int startBytes = next * size;
  unsigned char* e = (unsigned char*)element;
  for(int i = 0; i < size; i++)
    storage[startBytes + i] = e[i];
  next++;
  return(next - 1); // Index number
}

void* Stash::fetch(int index) {
  require(0 <= index, "Stash::fetch (-)index");
  if(index >= next)
    return 0; // To indicate the end
  // Produce pointer to desired element:
  return &(storage[index * size]);
}
```

```
int Stash::count() {
  return next; // Number of elements in CStash
}

void Stash::inflate(int increase) {
  assert(increase >= 0);
  if(increase == 0) return;
  int newQuantity = quantity + increase;
  int newBytes = newQuantity * size;
  int oldBytes = quantity * size;
  unsigned char* b = new unsigned char[newBytes];
  for(int i = 0; i < oldBytes; i++)
    b[i] = storage[i]; // Copy old to new
  delete [](storage); // Release old storage
  storage = b; // Point to new memory
  quantity = newQuantity; // Adjust the size
} ///:~
```

Cuando utiliza el primer constructor no se asigna memoria alguna para `storage`. La asignación ocurre la primera vez que trata de añadir (con `add()`) un objeto y en cualquier momento en el que el bloque de memoria actual se exceda en `add()`.

Ambos constructores se prueban en este programa de ejemplo:

```
//: C07:Stash3Test.cpp
//{L} Stash3
// Function overloading
#include "Stash3.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
  Stash intStash(sizeof(int));
  for(int i = 0; i < 100; i++)
    intStash.add(&i);
  for(int j = 0; j < intStash.count(); j++)
    cout << "intStash.fetch(" << j << ") = "
         << *(int*)intStash.fetch(j)
         << endl;
  const int bufsize = 80;
  Stash stringStash(sizeof(char) * bufsize, 100);
  ifstream in("Stash3Test.cpp");
  assure(in, "Stash3Test.cpp");
  string line;
  while(getline(in, line))
    stringStash.add((char*)line.c_str());
  int k = 0;
  char* cp;
  while((cp = (char*)stringStash.fetch(k++))!=0)
    cout << "stringStash.fetch(" << k << ") = "
         << cp << endl;
} ///:~
```

La llamada al constructor para la variable `stringStash` utiliza un segundo argumento; se presume que conoce algo especial sobre el problema específico que usted está resolviendo que le permite elegir un tamaño inicial para el `Stash`.

7.3. Uniones

Como ya ha visto, la única diferencia en C++ entre `struct` y `class` es que `struct` pone todo por defecto a `public` y la clase pone todo por defecto a `private`. Una `struct` también puede tener constructores y destructores, como cabía esperar. Pero resulta que el tipo `union` también puede tener constructores, destructores, métodos e incluso controles de acceso. Puede ver de nuevo la utilización y las ventajas de la sobrecarga de funciones en el siguiente ejemplo:

```
//: C07:UnionClass.cpp
// Unions with constructors and member functions
#include<iostream>
using namespace std;

union U {
private: // Access control too!
  int i;
  float f;
public:
  U(int a);
  U(float b);
  ~U();
  int read_int();
  float read_float();
};

U::U(int a) { i = a; }

U::U(float b) { f = b;}

U::~U() { cout << "U::~U()\n"; }

int U::read_int() { return i; }

float U::read_float() { return f; }

int main() {
  U X(12), Y(1.9F);
  cout << X.read_int() << endl;
  cout << Y.read_float() << endl;
} ///:~
```

Podría pensar que en el código anterior la única diferencia entre una unión y una clase es la forma en que los datos se almacenan en memoria (es decir, el `int` y el `float` están superpuestos). Sin embargo una unión no se puede utilizar como clase base durante la herencia, lo cual limita bastante desde el punto de vista del diseño orientado a objetos (veremos la herencia en el Capítulo 14).

Aunque los métodos civilizan ligeramente el tratamiento de uniones, sigue sin haber manera alguna de prevenir que el programador cliente seleccione el tipo de

elemento equivocado una vez que la unión se ha inicializado. En el ejemplo anterior, podría escribir `X.read_float()` incluso aunque sea inapropiado. Sin embargo, una unión «segura» se puede encapsular en una clase. En el siguiente ejemplo, vea cómo la enumeración clarifica el código, y cómo la sobrecarga viene como anillo al dedo con los constructores:

```
//: C07:SuperVar.cpp
// A super-variable
#include <iostream>
using namespace std;

class SuperVar {
  enum {
    character,
    integer,
    floating_point
  } vartype;  // Define one
  union {  // Anonymous union
    char c;
    int i;
    float f;
  };
public:
  SuperVar(char ch);
  SuperVar(int ii);
  SuperVar(float ff);
  void print();
};

SuperVar::SuperVar(char ch) {
  vartype = character;
  c = ch;
}

SuperVar::SuperVar(int ii) {
  vartype = integer;
  i = ii;
}

SuperVar::SuperVar(float ff) {
  vartype = floating_point;
  f = ff;
}

void SuperVar::print() {
  switch (vartype) {
    case character:
      cout << "character: " << c << endl;
      break;
    case integer:
      cout << "integer: " << i << endl;
      break;
    case floating_point:
      cout << "float: " << f << endl;
      break;
  }
}
```

```
int main() {
  SuperVar A('c'), B(12), C(1.44F);
  A.print();
  B.print();
  C.print();
} ///:~
```

En ese ejemplo la enumeración no tiene nombre de tipo (es una enumeración sin etiqueta). Esto es aceptable si va a definir inmediatamente un ejemplar de la enumeración, tal como se hace aquí. No hay necesidad de indicar el nombre del tipo de la enumeración en el futuro, por lo que aquí el nombre de tipo es opcional.

La unión no tiene nombre de tipo ni nombre de variable. Esto se denomina *unión anónima*, y crea espacio para la unión pero no requiere acceder a los elementos de la unión con el nombre de la variable y el operador punto. Por ejemplo, si su unión anónima es:

```
//: C07:AnonymousUnion.cpp
int main() {
  union {
    int i;
    float f;
  };
  // Access members without using qualifiers:
  i = 12;
  f = 1.22;
} ///:~
```

Note que accede a los miembros de una unión anónima igual que si fueran variables normales. La única diferencia es que ambas variables ocupan el mismo espacio de memoria. Si la unión anónima está en el ámbito del fichero (fuera de todas las funciones y clases), entonces se ha de declarar estática para que tenga enlace interno.

Aunque ahora `SuperVar` es segura, su utilidad es un poco dudosa porque la razón de utilizar una unión principalmente es la de ahorrar memoria y la adición de `vartype` hace que ocupe bastante espacio en la unión (relativamente), por lo que la ventaja del ahorro desaparece. Hay un par de alternativas para que este esquema funcione. Si `vartype` controlara más de una unión (en el caso de que fueran del mismo tipo) entonces sólo necesitaría uno para el grupo y no ocuparía más memoria. Una aproximación más útil es tener `#ifdef`s alrededor del código de `vartype`, el cual puede entonces garantizar que las cosas se utilizan correctamente durante el desarrollo y las pruebas. Si el código ha de entregarse, antes puede eliminar las sobrecargas de tiempo y memoria.

7.4. Argumentos por defecto

En `Stash3.h`, examine los dos constructores para `Stash`. No parecen muy diferentes, ¿verdad?. De hecho el primer constructor parece ser un caso especial del segundo pero con `size` inicializado a cero. Es un poco una pérdida de tiempo y esfuerzo crear y mantener dos versiones diferentes de una función similar.

C++ proporciona un remedio mediante los *argumentos por defecto*. Un argumento por defecto es una valor que se da en la declaración para que el compilador lo inserte

automáticamente en el caso de que no se proporcione en la llamada a la función. En el ejemplo de `Stash`, se puede reemplazar las dos funciones:

```
Stash(int size); // Zero quantity
Stash(int size, int initQuantity);
```

por ésta otra:

```
Stash(int size, int initQuantity = 0);
```

La definición de `Stash(int)` simplemente se quita; todo lo necesario está ahora en la definición de `Stash(int, int)`.

Ahora, las definiciones de los dos objetos

```
 Stash A(100), B(100, 0);
```

producirán exactamente los mismos resultados. En ambos casos se llama al mismo constructor, aunque el compilador substituye el segundo argumento de `A` automáticamente cuando ve que que el primer argumento es un entero y no hay un segundo argumento. El compilador ha detectado un argumento por defecto, así que sabe que todavía puede llamar a la función si substituye este segundo argumento, que es lo que usted le ha dicho que haga al no poner ese argumento.

Los argumentos por defecto, al igual que la sobrecarga de funciones, son muy convenientes. Ambas características le permiten utilizar un único nombre para una función en situaciones diferentes. La diferencia está en que el compilador substituye los argumentos por defecto cuando no se ponen. El ejemplo anterior en un buen ejemplo para utilizar argumentos por defecto en vez de la sobrecarga de funciones; de otra modo se encuentra con dos o más funciones que tienen signaturas y comportamientos similares. Si las funciones tienen comportamientos muy diferentes, normalmente no tiene sentido utilizar argumentos por defecto (de hecho, debería preguntarse si dos funciones con comportamientos muy diferentes deberían llamarse igual).

Hay dos reglas que se deben tener en cuenta cuando se utilizan argumentos por defecto. La primera es que sólo los últimos pueden ser por defecto, es decir, no puede poner un argumento por defecto seguido de otro que no lo es. La segunda es que una vez se empieza a utilizar los argumentos por defecto al realizar una llamada a una función, el resto de argumentos también serán por defecto (esto sigue a la primera regla).

Los argumentos por defecto sólo se colocan en la declaración de la función (normalmente en el fichero de cabecera). El compilador debe conocer el valor por defecto antes de utilizarlo. Hay gente que pone los valores por defecto comentados en la definición por motivos de documentación.

```
void fn(int x /* = 0 */) { // ...
```

7.4.1. Argumentos de relleno

Los argumentos de una función pueden declararse sin identificadores. Cuando esto se hace con argumentos por defecto, puede parecer gracioso. Puede encontrarse

con

```
void f(int x, int = 0, float = 1.1);
```

En C++, la definición de la función tampoco necesita identificadores:

```
void f(int x, int, float flt) { /* ... */ }
```

En el cuerpo de la función, se puede hacer referencia a `x` y a `flt`, pero no al argumento de en medio puesto que no tiene nombre. A pesar de esto, las llamadas a función deben proporcionar un valor para este argumento de relleno: `f(1)` ó `f(1, 2, 3,0)`. Esta sintaxis permite poner el argumento como un argumento de relleno sin utilizarlo. La idea es que podría querer cambiar la definición de la función para utilizar el argumento de relleno más tarde, sin cambiar todo el código en que ya se invoca la función. Por supuesto, puede obtener el mismo resultado utilizando un argumento con nombre, pero en ese caso está definiendo el argumento para el cuerpo de la función sin que éste lo utilice, y la mayoría de los compiladores darán un mensaje de aviso, dando por hecho que usted ha cometido un error. Si deja el argumento sin nombre intencionadamente, evitará la advertencia.

Más importante, si empieza utilizando un argumento que más tarde decide dejar de utilizar, puede quitarlo sin generar avisos ni fastidiar al código cliente que esté utilizando la versión anterior de la función.

7.5. Elección entre sobrecarga y argumentos por defecto

Tanto la sobrecarga de funciones como los argumentos por defecto resultan útiles para ponerle nombre a las funciones. Sin embargo, a veces puede resultar confuso saber qué técnica utilizar. Por ejemplo, estudie la siguiente herramienta que está diseñada para tratar automáticamente bloques de memoria:

```
//: C07:Mem.h
#ifndef MEM_H
#define MEM_H
typedef unsigned char byte;

class Mem {
  byte* mem;
  int size;
  void ensureMinSize(int minSize);
public:
  Mem();
  Mem(int sz);
  ~Mem();
  int msize();
  byte* pointer();
  byte* pointer(int minSize);
};
#endif // MEM_H ///:~
```

El objeto `Mem` contiene un bloque de octetos y se asegura de que tiene suficiente memoria. El constructor por defecto no reserva memoria pero el segundo constructor se asegura de que hay *sz* octetos de memoria en el objeto `Mem`. El destructor libera la memoria, `msize()` le dice cuántos octetos hay actualmente en `Mem` y `pointer()` retorna un puntero al principio de la memoria reservada (`Mem` es una herramienta a bastante bajo nivel). Hay una versión sobrecargada de `pointer()` que los programadores clientes pueden utilizar para obtener un puntero que apunta a un bloque de memoria con al menos el tamaño `minSize`, y el método lo asegura.

El constructor y el método `pointer()` utilizan el método privado `ensureMinSize()` para incrementar el tamaño del bloque de memoria (note que no es seguro mantener el valor de retorno de `pointer()` si se cambia el tamaño del bloque de memoria).

He aquí la implementación de la clase:

```
//: C07:Mem.cpp {O}
#include "Mem.h"
#include <cstring>
using namespace std;

Mem::Mem() { mem = 0; size = 0; }

Mem::Mem(int sz) {
  mem = 0;
  size = 0;
  ensureMinSize(sz);
}

Mem::~Mem() { delete []mem; }

int Mem::msize() { return size; }

void Mem::ensureMinSize(int minSize) {
  if(size < minSize) {
    byte* newmem = new byte[minSize];
    memset(newmem + size, 0, minSize - size);
    memcpy(newmem, mem, size);
    delete []mem;
    mem = newmem;
    size = minSize;
  }
}

byte* Mem::pointer() { return mem; }

byte* Mem::pointer(int minSize) {
  ensureMinSize(minSize);
  return mem;
} ///:~
```

Puede observar que `ensureMinSize()` es la única función responsable de reservar memoria y que la utilizan tanto el segundo constructor como la segunda versión sobrecargada de `pointer()`. Dentro de `ensureSize()` no se hace nada si el tamaño es lo suficientemente grande. Si se ha de reservar más memoria para que el bloque sea más grande (que es el mismo caso cuando el bloque tiene tamaño cero

después del constructor por defecto), la nueva porción de más se pone a cero utilizando la función de la librería estándar de C `memset()`, que fue presentada en el Capítulo 5. La siguiente llamada es a la función de la librería estándar de C `memcpy()`, que en este caso copia los octetos existentes de `mem` a `newmem` (normalmente de una manera eficaz). Finalmente, se libera la memoria antigua y se asignan a los atributos apropiados la nueva memoria y su tamaño.

La clase `Mem` se ha diseñado para su utilización como herramienta dentro de otras clases para simplificar su gestión de la memoria (también se podría utilizar para ocultar un sistema de gestión de memoria más avanzada proporcionado, por ejemplo, por el el sistema operativo). Esta clase se comprueba aquí con una simple clase de tipo string:

```
//: C07:MemTest.cpp
// Testing the Mem class
//{L} Mem
#include "Mem.h"
#include <cstring>
#include <iostream>
using namespace std;

class MyString {
  Mem* buf;
public:
  MyString();
  MyString(char* str);
  ~MyString();
  void concat(char* str);
  void print(ostream& os);
};

MyString::MyString() {  buf = 0; }

MyString::MyString(char* str) {
  buf = new Mem(strlen(str) + 1);
  strcpy((char*)buf->pointer(), str);
}

void MyString::concat(char* str) {
  if(!buf) buf = new Mem;
  strcat((char*)buf->pointer(
    buf->msize() + strlen(str) + 1), str);
}

void MyString::print(ostream& os) {
  if(!buf) return;
  os << buf->pointer() << endl;
}

MyString::~MyString() { delete buf; }

int main() {
  MyString s("My test string");
  s.print(cout);
  s.concat(" some additional stuff");
  s.print(cout);
  MyString s2;
  s2.concat("Using default constructor");
```

```
  s2.print(cout);
} ///:~
```

Todo lo que puede hacer con esta clase es crear un `MyString`, concatenar texto e imprimir a un `ostream`. La clase sólo contiene un puntero a un `Mem`, pero note la diferencia entre el constructor por defecto, que pone el puntero a cero, y el segundo constructor, que crea un `Mem` y copia los datos dentro del mismo. La ventaja del constructor por defecto es que puede crear, por ejemplo, un array grande de objetos `MyString` vacíos con pocos recursos, pues el tamaño de cada objeto es sólo un puntero y la única sobrecarga en el rendimiento del constructor por defecto es el de asignarlo a cero. El coste de un `MyString` sólo empieza a aumentar cuando concatena datos; en ese momento el objeto `Mem` se crea si no ha sido creado todavía. Sin embargo, si utiliza el constructor por defecto y nunca concatena ningún dato, la llamada al destructor todavía es segura porque cuando se llama a `delete` con un puntero a cero, el compilador no hace nada para no causar problemas.

Si mira los dos constructores, en principio, podría parecer que son candidatos para utilizar argumentos por defecto. Sin embargo, si elimina el constructor por defecto y escribe el constructor que queda con un argumento por defecto:

```
MyString(char* str = "");
```

todo funcionará correctamente, pero perderá la eficacia anterior pues siempre se creará el objeto `Mem`. Para volver a tener la misma eficacia de antes, ha de modificar el constructor:

```
MyString::MyString(char* str) {
  if (!*str) { // Apunta a un string ívaco
    buf = 0;
    return;
  }
  buf = new Mem(strlen(str) + 1);
  strcpy((char*)buf->pointer(), str);
}
```

Esto significa, en efecto, que el valor por defecto es un caso que ha de tratarse separadamente de un valor que no lo es. Aunque parece algo inocente con un pequeño constructor como éste, en general esta práctica puede causar problemas. Si tiene que tratar por separado el valor por defecto en vez de tratarlo como un valor ordinario, debería ser una pista para que al final se implementen dos funciones diferentes dentro de una función: una versión para el caso normal y otra para el caso por defecto. Podría partirlo en dos cuerpos de función diferentes y dejar que el compilador elija. Esto resulta en un ligero (pero normalmente invisible) incremento de la eficacia porque el argumento extra no se pasa y por tanto el código extra debido a la condición condición no se ejecuta. Más importante es que está manteniendo el código *en* dos funciones separadas en vez de combinarlas en una utilizando argumentos por defecto, lo que resultará en un mantenimiento más sencillo, sobre todo si las funciones son largas.

Por otro lado, considere la clase `Mem`. Si mira las definiciones de los dos constructores y las dos funciones `pointer()`, puede ver que la utilización de argumentos por defecto en ambos casos no causará que los métodos cambien. Así, la clase podría ser fácilmente:

```
//: C07:Mem2.h
#ifndef MEM2_H
#define MEM2_H
typedef unsigned char byte;

class Mem {
  byte* mem;
  int size;
  void ensureMinSize(int minSize);
public:
  Mem(int sz = 0);
  ~Mem();
  int msize();
  byte* pointer(int minSize = 0);
};
#endif // MEM2_H ///:~
```

Note que la llamada a `ensureMinSize(0)` siempre será bastante eficiente.

Aunque ambos casos se basan en decisiones por motivos de eficacia, debe tener cuidado para no caer en la trampa de pensar sólo en la eficacia (siempre fascinante). Lo más importante en el diseño de una clase es la interfaz de la clase (sus miembros públicos, que son las que el programador cliente tiene a su disposición). Si se implementa una clase fácil de utilizar y reutilizar, entonces ha tenido éxito; siempre puede realizar ajustes para mejorar la eficacia en caso necesario, pero el efecto de una clase mal diseñada porque el programador está obsesionado con la eficacia puede resultar grave. Su primera preocupación debería ser que la interfaz tenga sentido para aquéllos que la utilicen y para los que lean el código. Note que en `MemTest.cpp` el uso de `MyString` no cambia independientemente de si se utiliza el constructor por defecto o si la eficacia es buena o mala.

7.6. Resumen

Como norma, no debería utilizar argumentos por defecto si hay que incluir una condición en el código. En vez de eso debería partir la función en dos o más funciones sobrecargadas si puede. Un argumento por defecto debería ser un valor que normalmente pondría ahí. Es el valor que es más probable que ocurra, para que los programadores clientes puedan hacer caso omiso de él o sólo lo pongan cuando no quieran utilizar el valor por defecto.

El argumento por defecto se incluye para hacer más fáciles las llamadas a función, especialmente cuando esas funciones tiene muchos argumentos con valores típicos. No sólo es mucho más sencillo escribir las llamadas, sino que además son más sencillas de leer, especialmente si el creador de la clase ordena los argumentos de tal manera que aquéllos que menos cambian se ponen al final del todo.

Una utilización especialmente importante de los argumentos por defecto es cuando empieza con una función con un conjunto de argumentos, y después de utilizarla por un tiempo se da cuenta que necesita añadir más argumentos. Si pone los nuevos argumentos como por defecto, se asegura de que no se rompe el código cliente que utiliza la interfaz anterior.

7.7. Ejercicios

Las soluciones a los ejercicios se pueden encontrar en el documento electrónico titulado «The Thinking in C++ Annotated Solution Guide», disponible por poco dinero en www.BruceEckel.com.

1. Cree una clase `Text` que contenga un objeto string para que guarde el texto de un fichero. Póngale dos constructores: un constructor por defecto y un constructor que tome un argumento de tipo string que sea el nombre del fichero que se vaya a abrir. Cuando se utilice el segundo constructor, abra el fichero y ponga su contenido en el atributo string. Añada un método llamado `contents()` que retorne el string para que, por ejemplo, se pueda imprimir. En `main()` abra un fichero utilizando `Text` e imprima el contenido en pantalla.

2. Cree una clase `Message` con un constructor que tome un sólo string con un valor por defecto. Cree un atributo privado string y asigne en el constructor el argumento string al atributo string. Cree dos métodos sobrecargados llamados `print()`: uno que no tome argumentos y que imprima simplemente el mensaje guardado en el objeto, y el otro que tome un argumento string, que imprima el mensaje interno además del argumento. ¿Tiene sentido utilizar esta aproximación en vez de la utilizada por el constructor?

3. Descubra cómo generar código ensamblador con su compilador y haga experimentos para deducir el esquema de decoración de nombres.

4. Cree una clase que contenga cuatro métodos con 0, 1, 2 y 3 argumentos de tipo `int` respectivamente. Cree un `main()` que haga un objeto de su clase y llame a cada método. Ahora modifique la clase para que tenga sólo un método con todos los argumentos por defecto. ¿Eso cambia su `main()`?

5. Cree una función con dos argumentos y llámela desde `main()`. Ahora haga que uno de los argumentos sea un argumento de relleno (sin identificador) y compruebe si necesita hacer cambios en `main()`.

6. Modifique `Stash3.h` y `Stash3.cpp` para que el constructor utilice argumentos por defecto. Pruebe el constructor haciendo dos versiones diferentes de un objeto `Stash`.

7. Cree una nueva versión de la clase `Stack` (del Capítulo 6) que contenga el constructor por defecto al igual que antes, y un segundo constructor que tome como argumentos un array de punteros a objetos y el tamaño del array. Este constructor debería recorrer el array y poner cada puntero en la pila (`Stack`). Pruebe su clase con un array de string's.

8. Modifique `SuperVar` para que haya `#ifdef`'s que engloben el código de `vartype` tal como se describe en la sección sobre enumeraciones. Cambie `vartype` como una enumeración pública (sin ejemplares) y modifique `print()` para que requiera un argumento de tipo `vartype` que le indique qué tiene qué hacer.

9. Implemente `Mem2.h` y asegúrese de que la clase modificada todavía funciona con `MemTest.cpp`.

10. Utilice la clase `Mem` para implementar `Stash`. Note que debido a que la implementación es privada y por tanto oculta al programador cliente, no necesita modificar el código de prueba.

11. Añada un método *bool moved()* en la clase `Mem` que tome el resultado de una llamada a `pointer()` y le diga si el puntero ha cambiado (debido a una reasignación). Escriba una función `main()` que pruebe su método `moved()`. ¿Tiene más sentido utilizar algo como `moved()` o simplemente invocar `pointer()` cada vez que necesite acceder a la memoria de `Mem`?

8: Constantes

El concepto de constante (expresión con la palabra reservada `const`) se creó para permitir a los programadores marcar la diferencia entre lo que puede cambiar y lo que no. Esto facilita el control y la seguridad en un proyecto de programación.

Desde su origen, `const` ha sido utilizada para diferentes propósitos. Mientras tanto FIXME:it trickled back en el lenguaje C en el que su significado cambió. Todo esto puede parecer un poco confuso al principio, y en este capítulo aprenderá cuándo, porqué y cómo usar la palabra reservada `const`. Hacia el final se expone una disertación sobre *volatile*, que es familia de `const` (ambos se refieren a los cambios) y su sintaxis es idéntica.

El primer motivo para la creación de `const` parece que fue eliminar el uso de la directiva del preprocesador `#define` para sustitución de valores. Desde entonces se usa para punteros, argumentos de funciones, tipos de retorno, objetos y funciones miembro. Todos ellos tienen pequeñas diferencias pero su significado es conceptualmente compatible. Se tratarán en las siguientes secciones de este capítulo.

8.1. Sustitución de valores

Cuando se programa en C, se usa libremente el preprocesador para crear macros y sustituir valores. El preprocesador simplemente hace un reemplazo textual y no realiza ninguna comprobación de tipo. Por ello, la sustitución de valores introduce pequeños problemas que se pueden evitar usando valores constantes.

El uso más frecuente del preprocesador es la sustitución de valores por nombres, en C es algo como:

```
#define BUFSIZE 100
```

`BUFSIZE` es un nombre que sólo existe durante el preprocesado. Por tanto, no ocupa memoria y se puede colocar en un fichero de cabecera para ofrecer un valor único a todas las unidades que lo utilicen. Es muy importante para el mantenimiento del código el uso de sustitución de valores en lugar de los también llamados «números mágicos». Si usa números mágicos en su código. no solamente impedirá al lector conocer su procedencia o significado si no que complicará innecesariamente la edición del código si necesita cambiar dicho valor.

La mayor parte del tiempo, `BUFSIZE` se comportará como un valor ordinario, pero no siempre. No tiene información de tipo. Eso puede esconder errores difíciles de localizar. C++ utiliza `const` para eliminar estos problemas llevando la sustitución

de valores al terreno del compilador. Ahora, puede escribir:

```
const int bufsize = 100;
```

Puede colocar `bufsize` en cualquier lugar donde se necesite conocer el valor en tiempo de compilación. El compilador utiliza `bufsize` para hacer *propagación de constantes*[1], que significa que el compilador reduce una expresión constante complicada a un valor simple realizando los cálculos necesarios en tiempo de compilación. Esto es especialmente importante en las definiciones de vectores:

```
char buf[bufsize];
```

Puede usar `const` con todos los tipos básicos(char, int, float y double) y sus variantes (así como clases y todo lo que verá después en este capítulo). Debido a los problemas que introduce el preprocesador deberá utilizar siempre `const` en lugar de `#define` para la sustitución de valores.

8.1.1. `const` en archivos de cabecera

Para poder usar `const` en lugar de `#define`, debe ser posible colocar las definiciones `const` en los archivos de cabecera como se hacía con los `#define`. De este modo, puede colocar la definición de una constante en un único lugar y distribuirla incluyendo el archivo de cabecera en las unidades del programa que la necesiten. Una constante en C++ utiliza *enlazado interno*, es decir, es visible sólo desde el archivo donde se define y no puede verse en tiempo de enlazado por otros módulos. Deberá asignar siempre un valor a las constantes cuando las defina, excepto cuando explícitamente use la declaración `extern`:

```
extern const int bufsize;
```

Normalmente el compilador de C++ evita la asignación de memoria para las constantes, pero en su lugar ocupa una entrada en la tabla de símbolos. Cuando se utiliza `extern` con una constante, se fuerza el alojamiento en memoria (esto también ocurre en otros casos, como cuando se solicita la dirección de una constante). El uso de la memoria debe hacerse porque `extern` dice «usa enlazado externo», es decir, que varios módulos deben ser capaces de hacer referencia al elemento, algo que requiere su almacenamiento en memoria.

Por lo general, cuando `extern` no forma parte de la definición, no se pide memoria. Cuando la constante se utiliza simplemente se incorpora en tiempo de compilación.

El objetivo de no almacenar en memoria las constantes tampoco se cumple con estructuras complicadas. Cuando el compilador se ve obligado a pedir memoria no puede realizar *propagación de constantes* (ya que el compilador no tiene forma de conocer con seguridad que valor debe almacenar; si lo conociese, no necesitaría pedir memoria).

Como el compilador no siempre puede impedir el almacenamiento para una constante, las definiciones de constantes utilizan enlace interno, es decir, se enlazan sólo con el módulo en que se definen. En caso contrario, los errores de enlace podrían ocurrir con las expresiones constantes complicadas ya que causarían petición

[1] N. del T.: del inglés *constant folding*

de almacenamiento en diferentes módulos. Entonces, el enlazador vería la misma definición en múltiples archivos objeto, lo que causaría un error en el enlace. Como las constantes utilizan enlace interno, el enlazador no intenta enlazar esas definiciones a través de los módulos, y así no hay colisiones. Con los tipos básicos, que son los se ven involucrados en la mayoría de los casos, el compilador siempre realiza propagación de constantes.

8.1.2. constantes seguras

El uso de las constantes no está limitado a la sustitución de los `#define` por expresiones constantes. Si inicializa una variable con un valor que se produce en tiempo de ejecución y sabe que no cambiará durante la vida de la variable, es una buena práctica de programación hacerla constante para que de ese modo el compilador produzca un mensaje de error si accidentalmente alguien intenta modificar dicha variable. Aquí hay un ejemplo:

```
//: C08:Safecons.cpp
// Using const for safety
#include <iostream>
using namespace std;

const int i = 100;  // Typical constant
const int j = i + 10; // Value from const expr
long address = (long)&j; // Forces storage
char buf[j + 10]; // Still a const expression

int main() {
  cout << "type a character & CR:";
  const char c = cin.get(); // Can't change
  const char c2 = c + 'a';
  cout << c2;
  // ...
} ///:~
```

Puede ver que `i` es una constante en tiempo de compilación, pero `j` se calcula a partir de `i`. Sin embargo, como `i` es una constante, el valor calculado para `j` es una expresión constante y es en si mismo otra constante en tiempo de compilación. En la siguiente línea se necesita la dirección de `j` y por lo tanto el compilador se ve obligado a pedir almacenamiento para `j`. Ni siquiera eso impide el uso de `j` para determinar el tamaño de `buf` porque el compilador sabe que `j` es una constante y que su valor es válido aunque se asigne almacenamiento, ya que eso se hace para mantener el valor en algún punto en el programa.

En `main()`, aparece un tipo diferente de constante en el identificador `c`, porque el valor no puede ser conocido en tiempo de compilación. Eso significa que se requiere almacenamiento, y por eso el compilador no intenta mantener nada en la tabla de símbolos (el mismo comportamiento que en C). La inicialización debe ocurrir, aún así, en el punto de la definición, y una vez que ocurre la inicialización, el valor ya no puede ser cambiado. Puede ver que `c2` se calcula a partir de `c` y además las reglas de ámbito funcionan para las constantes igual que para cualquier otro tipo, otra ventaja respecto al uso de `#define`.

En la práctica, si piensa que una variable no debería cambiar, debería hacer que fuese una constante. Esto no sólo da seguridad contra cambios inadvertidos, también

permite al compilador generar código más eficiente ahorrando espacio de almacenamiento y lecturas de memoria en la ejecución del programa.

8.1.3. Vectores

Es posible usar constantes para los vectores, pero prácticamente está dando por hecho que el compilador no será lo suficientemente sofisticado para mantener un vector en la tabla de símbolos, así que le asignará espacio de almacenamiento. En estas situaciones, `const` significa «un conjunto de datos en memoria que no pueden modificarse». En cualquier caso, sus valores no puede usarse en tiempo de compilación porque el compilador no conoce en ese momento los contenidos de las variables que tienen espacio asignado. En el código siguiente puede ver algunas declaraciones incorrectas.

```
//: C08:Constag.cpp
// Constants and aggregates
const int i[] = { 1, 2, 3, 4 };
//! float f[i[3]]; // Illegal
struct S { int i, j; };
const S s[] = { { 1, 2 }, { 3, 4 } };
//! double d[s[1].j]; // Illegal
int main() {} ///:~
```

En la definición de un vector, el compilador debe ser capaz de generar código que mueva el puntero de pila para dar cabida al vector. En las definiciones incorrectas anteriores, el compilador se queja porque no puede encontrar una expresión constante en la definición del tamaño del vector.

8.1.4. Diferencias con C

Las constantes se introdujeron en las primeras versiones de C++ mientras la especificación del estándar C estaba siendo terminada. Aunque el comité a cargo de C decidió entonces incluir `const` en C, por alguna razón, vino a significar para ellos «una variable ordinaria que no puede cambiarse». En C, una constante siempre ocupa espacio de almacenamiento y su ámbito es global. El compilador C no puede tratar `const` como una constante en tiempo de compilación. En C, si escribe:

```
const int bufsize = 100;
char buf[bufsize];
```

aparecerá un error, aunque parezca algo razonable. `bufsize` está guardado en algún sitio y el compilador no conoce su valor en tiempo de compilación. Opcionalmente puede escribir:

```
const int bufsize;
```

en C, pero no en C++, y el compilador C lo acepta como una declaración que indica que se almacenará en alguna parte. Como C utiliza enlace externo para las constantes, esa semántica tiene sentido. C++ utiliza normalmente enlace interno, así que, si quiere hacer lo mismo en C++, debe indicar expresamente que se use enlace externo usando `extern`.

```
extern const int bufsize;  // es ódeclaracin, no ódefinicin
```

Esta declaración también es válida en C.

En C++, `const` no implica necesariamente almacenamiento. En C, las constantes siempre necesitan almacenamiento. El hecho de que se necesite almacenamiento o no depende de cómo se use la constante. En general, si una constante se usa simplemente para reemplazar un número por un nombre (como hace `#define`), entonces no requiere almacenamiento. Si es así (algo que depende de la complejidad del tipo de dato y de lo sofisticación del compilador) los valores pueden expandirse en el código para conseguir mayor eficiencia después de la comprobación de los tipos, no como con `#define`. Si de todas formas, se necesita la dirección de una constante (aún desconocida, para pasarla a una función como argumento por referencia) o se declara como `extern`, entonces se requiere asignar almacenamiento para la constante.

En C++, una constante que esté definida fuera de todas las funciones tiene ámbito de archivo (es decir, es inaccesible fuera del archivo). Esto significa que usa enlace interno. Esto es diferente para el resto de identificadores en C++ (y que las constantes en C) que utilizan siempre enlace externo. Por eso, si declara una constante con el mismo nombre en dos archivos diferentes y no toma sus direcciones ni los define como `extern`, el compilador C++ ideal no asignará almacenamiento para la constante, simplemente la expandirá en el código. Como las constantes tienen implícito el ámbito a su archivo, puede ponerlas en un archivo de cabecera de C++ sin que origine conflictos en el enlace.

Dado que las constante en C++ utilizan por defecto enlace interno, no puede definir una constante en un archivo y utilizarla desde otro. Para conseguir enlace externo para la constante y así poder usarla desde otro archivo, debe definirla explícitamente como `extern`, algo así:

```
extern const int x = 1;  // ódefinicin, no ódeclaracin
```

Señalar que dado un identificador, si se dice que es `extern`, se fuerza el almacenamiento para la constante (aunque el compilador tenga la opción de hacer la expansión en ese punto). La inicialización establece que la sentencia es una definición, no una declaración. La declaración:

```
extern const int x;
```

en C++ significa que la definición existe en algún sitio (mientras que en C no tiene porqué ocurrir así). Ahora puede ver porqué C++ requiere que las definiciones de constantes incluyan la inicialización: la inicialización diferencia una declaración de una definición (en C siempre es una definición, aunque no esté inicializada). Con una declaración `const extern`, el compilador no hace expansión de la constante porque no conoce su valor.

La aproximación de C a las constantes es poco útil, y si quiere usar un valor simbólico en una expresión constante (que deba evaluarse en tiempo de compilación) casi está obligado a usar `#define`.

8.2. Punteros

Los punteros pueden ser constantes. El compilador pondrá más esfuerzo aún para evitar el almacenamiento y hacer expansión de constantes cuando se trata de punteros constantes, pero estas características parecen menos útiles en este caso. Lo más importante es que el compilador le avisará si intenta cambiar un puntero constante, lo que representa un buen elemento de seguridad. Cuando se usa `const` con punteros tiene dos opciones: se pueden aplicar a lo que apunta el puntero o a la propia dirección almacenada en el puntero. La sintaxis es un poco confusa al principio pero se vuelve cómodo con la práctica.

8.2.1. Puntero a constante

El truco con la definición de un puntero, al igual que con una definición complicada, es leerla empezando por el identificador e ir analizando la definición hacia afuera. El especificador `const` está ligado a la cosa «más cercana». Así que si se quiere impedir cambios en el elemento apuntado, escribe una definición parecida a esta:

```
const int* u;
```

Empezando por el identificador, se lee «`u` es un puntero, que apunta a un entero constante». En este caso no se requiere inicialización porque está diciendo que `u` puede apuntar a cualquier cosa (es decir, no es constante), pero la cosa a la que apunta no puede cambiar.

Ahora viene la parte confusa. Podría pensar que hacer el puntero inalterable en si mismo, es decir, impedir cualquier cambio en la dirección que contiene `u`, es tan simple como mover la palabra `const` al otro lado de la palabra int:

```
int const* v;
```

y pensar que esto debería leerse «`v` es un puntero constante a un entero». Sin embargo, la forma de leerlo es «`v` es un puntero ordinario a un entero que es constante». Es decir, la palabra `const` se refiere de nuevo al entero y el efecto es el mismo que en la definición previa. El hecho de que estas definiciones sean equivalentes es confuso, para evitar esta confusión por parte del lector del código, debería ceñirse a la primera forma.

8.2.2. Puntero constante

Para conseguir que el puntero sea inalterable, debe colocar el especificador `const` a la derecha del *:

```
int d = 1;
int * const w = &d;
```

Ahora, se lee «`w` es un puntero constate, y apunta a un entero». Como el puntero en sí es ahora una constante, el compilador obliga a darle un valor inicial que no podrá alterarse durante la vida del puntero. En cualquier caso, puede cambiar el valor de lo que apunta el puntero con algo como:

```
*w = 2;
```

También puede hacer un puntero constante a un elemento constante usando una de las formas siguientes:

```
int d = 1;
const int* const x = &d;  // (1)
int const* const x2 = &d; // (2)
```

Ahora ni el puntero ni el elemento al que apunta pueden modificarse.

Algunos argumentan que la segunda forma es más consistente porque el `const` se coloca siempre a la derecha de lo que afecta. Debe decidir que forma resulta más clara para su estilo de codificación particular.

Algunas líneas de un archivo susceptible de ser compilado.

```
//: C08:ConstPointers.cpp
const int* u;
int const* v;
int d = 1;
int* const w = &d;
const int* const x = &d;  // (1)
int const* const x2 = &d; // (2)
int main() {} ///:~
```

Formato

Este libro sigue la norma de poner sólo una definición de puntero por línea, e inicializar cada puntero en el punto de definición siempre que sea posible. Por eso, el estilo es colocar el * al lado del tipo:

```
int* u = &i;
```

como si int* fuese un tipo de dato básico. Esto hace que el código sea más fácil de leer, pero desafortunadamente, esta no es la forma en que funciona. El «*» se refiere al identificador no al tipo. Se puede colocar en cualquier sitio entre el nombre del tipo y el identificador. De modo que puede hacer esto:

```
int * u = &i, v = 0;
```

donde se crea un `int* u` y después un `int v` (que no es puntero). Como esto puede parecer confuso a los lectores, es mejor utilizar el estilo mostrado en este libro.

8.2.3. Asignación y comprobación de tipos

C++ es muy exigente en lo referente a la comprobación de tipos y esto se extiende a la asignación de punteros. Puede asignar la dirección de una variable no constante a un puntero constante porque simplemente está prometiendo no cambiar algo que puede cambiarse. De todos modos, no puede asignar la dirección de una variable constante a un puntero no constante porque entonces está diciendo que podría

modificar la variable a través del puntero. Por supuesto, siempre puede usar «un molde» para forzar la asignación, pero eso es siempre una mala práctica de programación ya que rompe la consistencia de la variable además del grado de seguridad que ofrece el especificador `const`. Por ejemplo:

```
//: C08:PointerAssignment.cpp
int d = 1;
const int e = 2;
int* u = &d; // OK -- d not const
//! int* v = &e; // Illegal -- e const
int* w = (int*)&e; // Legal but bad practice
int main() {} ///:~
```

Aunque C++ ayuda a evitar errores, no le protege de usted mismo si se empeña en romper los mecanismos de seguridad.

Literales de cadena

C++ no es tan estricto con los literales en lo referente a constantes. Puede escribir:

```
char * cp = "howdy";
```

y el compilador lo aceptará sin objeción. Técnicamente esto supone un error porque el literal de cadena («`howdy`» en este caso) se crea por el compilador como un vector de caracteres constante, y el resultado del vector de caracteres entrecomillado es la dirección de memoria del primer elemento. Si se modifica uno de los caracteres del vector en tiempo de ejecución es un error, aunque no todos los compiladores lo imponen correctamente.

Así que los literales de cadena son arrays de caracteres constantes. Por supuesto, el compilador le permite tratarlos como no constantes porque existe mucho código C que depende de ello. De todas formas, si intenta cambiar los valores de un literal, el resultado no está definido, y probablemente funcione en muchos computadores.

Si quiere poder modificar una cadena, debe ponerla en un vector:

```
char cp[] = "howdy";
```

Como los compiladores a menudo no imponen la diferencia no tiene porqué recordar que debe usar esta la última forma y la cuestión pasa a ser algo bastante sutil.

8.3. Argumentos de funciones y valores de retorno

El uso del especificador `const` con argumentos de funciones y valores de retorno es otro lugar donde el concepto de constante puede resultar confuso. Si está pasando variables por valor, utilizar `const` no tiene significado para el cliente (significa que el argumento que se pasa no puede modificarse en la función). Si está devolviendo una variable de un tipo derivado y utiliza el especificador `const`, significa que el valor de retorno no puede modificarse. Si pasa o devuelve direcciones, `const` impide que el destinatario de la dirección pueda modificarse.

8.3.1. Paso por valor constante

Puede indicar que los argumentos de funciones son constantes cuando se pasa por valor como:

```
void f1(const int i) {
    i++; // ilegal
}
```

pero, ¿qué significa esto? Está impidiendo que el valor de la variable original pueda ser cambiado en la función `f1()`. De todos formas, como el argumento se pasa por valor, es sabido que inmediatamente se hace una copia de la variable original, así que dicha restricción se cumple implícitamente sin necesidad de usar el especificador `const`.

Dentro de la función, `const` si toma un significado: El argumento no se puede cambiar. Así que, en realidad, es una herramienta para el programador de la función, no para el que la usa.

Para evitar la confusión del usuario de la función, puede hacer que el argumento sea constante dentro de la función en lugar de en la lista de argumentos. Podría hacerlo con un puntero, pero la sintaxis más adecuada para lograrlo es la referencia, algo que se tratará en profundidad en el capitulo 11[FIXME:XREF].

Brevemente, una referencia es como un puntero constante que se dereferencia automáticamente, así que es como tener un alias de la variable. Para crear una referencia, debe usar el símbolo & en la definición. De ese modo se tiene una definición libre de confusiones.

```
void f2(int ic) {
    const int &i = ic;
    i++;  // ilegal (error de ócompilacin)
}
```

De nuevo, aparece un mensaje de error, pero esta vez el especificador `const` no forma parte de la cabecera de la función, solo tiene sentido en la implementación de la función y por la tanto es invisible para el cliente.

8.3.2. Retorno por valor constante

Algo similar ocurre con los valores de retorno. Si dice que el valor de retorno de una función es constante:

```
const int g();
```

está diciendo que el valor de la variable original (en el ámbito de la función) no se modificará. Y de nuevo, como lo está devolviendo por valor, es la copia lo que se retorna, de modo que el valor original nunca se podrá modificar.

En principio, esto puede hacer suponer que el especificador `const` tiene poco significado. Puede ver la aparente falta de sentido de devolver constantes por valor en este ejemplo:

```
//: C08:Constval.cpp
```

```
// Returning consts by value
// has no meaning for built-in types

int f3() { return 1; }
const int f4() { return 1; }

int main() {
  const int j = f3(); // Works fine
  int k = f4(); // But this works fine too!
} ///:~
```

Para los tipos básicos, no importa si el retorno es constante, así que debería evitar la confusión para el programador cliente y no utilizar `const` cuando se devuelven variables de tipos básicos por valor.

Devolver por valor como constante se vuelve importante cuando se trata con tipos definidos por el programador. Si una función devuelve un objeto por valor como constante, el valor de retorno de la función no puede ser un recipiente [2]

Por ejemplo:

```
//: C08:ConstReturnValues.cpp
// Constant return by value
// Result cannot be used as an lvalue

class X {
  int i;
public:
  X(int ii = 0);
  void modify();
};

X::X(int ii) { i = ii; }

void X::modify() { i++; }

X f5() {
  return X();
}

const X f6() {
  return X();
}

void f7(X& x) { // Pass by non-const reference
  x.modify();
}

int main() {
  f5() = X(1); // OK -- non-const return value
  f5().modify(); // OK
//!  f7(f5()); // Causes warning or error
// Causes compile-time errors:
//!  f6() = X(1);
//!  f6().modify();
```

[2] N. del T.: «recipiente» corresponde con el término *lvalue* que se refiere a una variable que puede ser modificada o a la que se le puede asignar un valor.

```
//!  f7(f6());
} ///:~
```

`f5()` devuelve un objeto de clase `X` no constante, mientras que `f6()` devuelve un objeto de clase `X` pero constante. Solo el valor de retorno por valor no constante se puede usar como recipiente.

Por eso, es importante usar `const` cuando se devuelve un objeto por valor si quiere impedir que se use como recipiente.

La razón por la que `const` no tiene sentido cuando se usa para devolver por valor variables de tipos del lenguaje es que el compilador impide automáticamente el uso de dichos tipos como recipiente, ya que devuelve un valor, no una variable. Solo cuando se devuelven objetos por valor de tipos definidos por el programador esta funcionalidad toma sentido.

La función `f7()` toma como argumento una referencia no constante (la referencia es una forma adicional para manejar direcciones en C++ y se trata en el [FIX-ME:XREF:capitulo 11]). Es parecido a tomar un puntero no constante, aunque la sintaxis es diferente. La razón por la que no compila es por la creación de un temporario.

Temporarios

A veces, durante la evaluación de una expresión, el compilador debe crear objetos temporales (temporarios). Son objetos como cualquier otro: requieren espacio de almacenamiento y se deben construir y destruir. La diferencia es que nunca se ven, el compilador es el responsable de decidir si se necesitan y los detalles de su existencia. Una particularidad importante de los temporarios es que siempre son constantes. Como normalmente no manejará objetos temporarios, hacer algo que cambie un temporario es casi seguro un error porque no será capaz de usar esa información. Para evitar esto, el compilador crea todos los temporarios como objetos constantes, de modo que le avisará si intenta modificarlos.

En el ejemplo anterior, `f5()` devuelve un objeto no constante. Pero en la expresión:

```
f7(f5());
```

el compilador debe crear un temporario para albergar el valor de retorno de `f5()` para que pueda ser pasado a `f7()`. Esto funcionaría bien si `f7()` tomara su argumento por valor; entonces el temporario se copiaría en `f7()` y no importaría lo que se pase al temporario X.

Sin embargo, `f7()` toma su argumento por referencia, lo que significa que toma la dirección del temporario X. Como `f7()` no toma su argumento por referencia constante, tiene permiso para modificar el objeto temporario. Pero el compilador sabe que el temporario desaparecerá en cuanto se complete la evaluación de la expresión, y por eso cualquier modificación hecha en el temporario se perderá. Haciendo que los objetos temporarios sean constantes automáticamente, la situación causa un error de compilación de modo que evitará cometer un error muy difícil de localizar.

En cualquier caso, tenga presente que las expresiones siguientes son correctas:

```
f5() = X(1);
f5().modify();
```

Aunque son aceptables para el compilador, en realidad son problemáticas. `f5()` devuelve un objeto de clase `X`, y para que el compilador pueda satisfacer las expresiones anteriores debe crear un temporario para albergar el valor de retorno. De modo que en ambas expresiones el objeto temporario se modifica y tan pronto como la expresión es evaluada el temporario se elimina. Como resultado, las modificaciones se pierden, así que probablemente este código es erróneo, aunque el compilador no diga nada al respecto. Las expresiones como éstas son suficientemente simples como para detectar el problema, pero cuando las cosas son más complejas los errores son más difíciles de localizar.

La forma de preservar la constancia de los objetos se muestra más adelante en este capítulo.

8.3.3. Paso y retorno de direcciones

Si pasa o retorna una dirección (ya sea un puntero o una referencia), el programador cliente puede recoger y modificar el valor al que apunta. Si hace que el puntero o referencia sea constante, impedirá que esto suceda, lo que puede ahorrarle problemas. De hecho, cada vez que se pasa una dirección como parámetro a una función, debería hacerla constante siempre que sea posible. Si no lo hace, está excluyendo la posibilidad de usar la función con constantes.

La opción de devolver un puntero o referencia constante depende de lo que quiera permitir hacer al programador cliente. Aquí se muestra un ejemplo que demuestra el uso de punteros constantes como argumentos de funciones y valores de retorno.

```
//: C08:ConstPointer.cpp
// Constant pointer arg/return

void t(int*) {}

void u(const int* cip) {
//!  *cip = 2; // Illegal -- modifies value
  int i = *cip; // OK -- copies value
//!  int* ip2 = cip; // Illegal: non-const
}

const char* v() {
  // Returns address of static character array:
  return "result of function v()";
}

const int* const w() {
  static int i;
  return &i;
}

int main() {
  int x = 0;
  int* ip = &x;
  const int* cip = &x;
  t(ip);  // OK
//!  t(cip); // Not OK
  u(ip);  // OK
  u(cip); // Also OK
//!  char* cp = v(); // Not OK
```

```
  const char* ccp = v(); // OK
//!  int* ip2 = w(); // Not OK
  const int* const ccip = w(); // OK
  const int* cip2 = w(); // OK
//!  *w() = 1; // Not OK
} ///:~
```

La función `t()` toma un puntero no-constante ordinario como argumento, y `u()` toma un puntero constante. En el cuerpo de `u()` puede ver un intento de modificar el valor de un puntero constante, algo incorrecto, pero puede copiar su valor en una variable no constante. El compilador también impide crear un puntero no constante y almacenar en él la dirección contenida en un puntero constante.

Las funciones `v()` y `w()` prueban las semánticas de retorno de valores. `v()` devuelve un const char* que se crea a partir de un literal de cadena. Esta sentencia en realidad genera la dirección del literal una vez que el compilador lo crea y almacena en área de almacenamiento estática. Como se ha dicho antes, técnicamente este vector de caracteres es una constante, como bien indica el tipo de retorno de `v()`.

El valor de retorno de `w()` requiere que tanto el puntero como lo que apunta sean constantes. Como en `v()`, el valor devuelto por `w()` es valido una vez terminada la función solo porque es estático. Nunca debe devolver un puntero a una variable local pues se almacenan en la pila y al terminar la función los datos de la pila desaparecen. Lo que si puede hacer es devolver punteros que apuntan a datos almacenados en el montón (*heap*), pues siguen siendo validos después de terminar la función.

En `main()` se prueban las funciones con varios argumentos. Puede ver que `t()` aceptará como argumento un puntero ordinario, pero si intenta pasarle un puntero a una constante, no hay garantía de que no vaya a modificarse el valor de la variable apuntada; por ello el compilador lo indica con un mensaje de error. `u()` toma un puntero a constante, así que puede aceptar los dos tipos de argumentos. Por eso una función que acepta un puntero a constante es más general que una que acepta un puntero ordinario.

Como es lógico, el valor de retorno de `v()` sólo se puede asignar a un puntero a constante. También era de esperar que el compilador rehuse asignar el valor devuelto por `w()` a un puntero ordinario, y que sí acepte un const int* const, pero podría sorprender un poco que también acepta un const int*, que no es exactamente el tipo de retorno declarado en la función. De nuevo, como el valor (que es la dirección contenida en el puntero) se copia, el requisito de que la variable original permanezca inalterable se cumple automáticamente. Por eso, el segundo `const` en la declaración const int* const sólo se aplica cuando lo use como recipiente, en cuyo caso el compilador lo impediría.

Criterio de paso de argumentos

En C es muy común el paso por valor, y cuando se quiere pasar una dirección la única posibilidad es usar un puntero[3]. Sin embargo, ninguno de estos modos es el preferido en C++. En su lugar, la primera opción cuando se pasa un parámetro es hacerlo por referencia o mejor aún, por referencia constante. Para el cliente de la función, la sintaxis es idéntica que en el paso por valor, de ese modo no hay confusión posible con los punteros, no hay que pensar en términos de punteros. Para

[3] Algunos autores dicen que todo en C se pasa por valor, ya que cuando se pasa un puntero se hace también una copia (de modo que el puntero se pasa por valor). En cualquier caso, hacer esta precisión puede, en realidad, confundir la cuestión.

el creador de una función, pasar una dirección es siempre más eficiente que pasar un objeto completo, y si pasa por referencia constante significa que la función no podrá cambiar lo almacenado en esa dirección, así que el efecto desde el punto de vista del programador cliente es lo mismo que el paso por valor (sin embargo es más eficiente).

A causa de la sintaxis de las referencias (para el cliente es igual que el paso por valor) es posible pasar un objeto temporario a una función que toma una referencia constante, mientras que nunca puede pasarse un objeto temporario a una función que toma un puntero (con un puntero, la dirección debe darse explícitamente). Así que con el paso por referencia se produce una nueva situación que nunca ocurre en C: un temporario, que es siempre constante, puede pasar su dirección a una función (una función puede tomar por argumento la dirección de un temporario). Esto es así porque, para permitir que los temporarios se pasen por referencia, el argumento debe ser una referencia constante. El siguiente ejemplo lo demuestra:

```
//: C08:ConstTemporary.cpp
// Temporaries are const

class X {};

X f() { return X(); } // Return by value

void g1(X&) {} // Pass by non-const reference
void g2(const X&) {} // Pass by const reference

int main() {
  // Error: const temporary created by f():
//!  g1(f());
  // OK: g2 takes a const reference:
  g2(f());
} ///:~
```

`f()` retorna un objeto de la clase `X` por valor. Esto significa que cuando tome el valor de retorno y lo pase inmediatamente a otra función como en las llamadas a `g1()` y `g2()`, se crea un temporario y los temporarios son siempre constantes. Por eso, la llamada a `g1()` es un error pues `g1()` no acepta una referencia constante, mientras que la llamada a `g2()` sí es correcta.

8.4. Clases

Esta sección muestra la forma en la que se puede usar el especificador `const` con las clases. Puede ser interesante crear una constante local a una clase para usarla en expresiones constantes que serán evaluadas en tiempo de compilación. Sin embargo, el significado del especificador `const` es diferente para las clases [4], de modo que debe comprender las opciones adecuadas para crear miembros constantes en una clase.

También se puede hacer que un objeto completo sea constante (y como se ha visto, el compilador siempre hace constantes los objetos temporarios). Pero preservar la consistencia de un objeto constante es más complicado. El compilador puede asegurar la consistencia de las variables de los tipos del lenguaje pero no puede vigilar la

[4] N. del T.: Esto se conoce como polisemia del lenguaje

complejidad de una clase. Para garantizar dicha consistencia se emplean las funciones miembro constantes; que son las únicas que un objeto constante puede invocar.

8.4.1. `const` en las clases

Uno de los lugares donde interesa usar `const` es para expresiones constantes dentro de las clases. El ejemplo típico es cuando se define un vector en una clase y se quiere usar `const` en lugar de `#define` para establecer el tamaño del vector y para usarlo al calcular datos concernientes al vector. El tamaño del vector es algo que desea mantener oculto en la clase, así que si usa un nombre como `size`, por ejemplo, se podría usar el mismo nombre en otra clase sin que ocurra un conflicto. El preprocesador trata todos los `#define` de forma global a partir del punto donde se definen, algo que `const` permite corregir de forma adecuada consiguiendo el efecto deseado.

Se podría pensar que la elección lógica es colocar una constante dentro de la clase. Esto no produce el resultado esperado. Dentro de una clase `const` recupera un poco su significado en C. Asigna espacio de almacenamiento para cada variable y representa un valor que es inicializado y ya no se puede cambiar. El uso de una constante dentro de una clase significa «Esto es constante durante la vida del objeto». Por otra parte, en cada objeto la constante puede contener un valor diferente.

Por eso, cuando crea una constante ordinaria (no estática) dentro de una clase, no puede darle un valor inicial. Esta inicialización debe ocurrir en el constructor. Como la constante se debe inicializar en el punto en que se crea, en el cuerpo del constructor la constante debe estar ya inicializada. De otro modo, le quedaría la opción de esperar hasta algún punto posterior en el constructor, lo que significaría que la constante no tendría valor por un momento. Y nada impediría cambiar el valor de la constante en varios sitios del constructor.

La lista de inicialización del constructor.

Un punto especial de inicialización es la llamada «lista de inicialización del constructor» y fue pensada en un principio para su uso en herencia (tratada en el [FIX-ME:XREF:capítulo 14]). La lista de inicialización del constructor (que como su nombre indica, sólo aparece en la definición del constructor) es una lista de llamadas a constructores que aparece después de la lista de argumentos del constructor y antes de abrir la llave del cuerpo del constructor. Se hace así para recordarle que las inicialización de la lista sucede antes de ejecutarse el constructor. Ese es el lugar donde poner las inicializaciones de todas las constantes de la clase. El modo apropiado para colocar las constantes en una clase se muestra a continuación:

```
//: C08:ConstInitialization.cpp
// Initializing const in classes
#include <iostream>
using namespace std;

class Fred {
  const int size;
public:
  Fred(int sz);
  void print();
};

Fred::Fred(int sz) : size(sz) {}
```

```
void Fred::print() { cout << size << endl; }

int main() {
  Fred a(1), b(2), c(3);
  a.print(), b.print(), c.print();
} ///:~
```

El aspecto de la lista de inicialización del constructor mostrada arriba puede crear confusión al principio porque no es usual tratar los tipos del lenguaje como si tuvieran constructores.

Constructores para los tipos del lenguaje

Durante el desarrollo del lenguaje se puso más esfuerzo en hacer que los tipos definidos por el programador se pareciesen a los tipos del lenguaje, pero a veces, cuando se vio útil se hizo que los tipos predefinidos (*built-in* se pareciesen a los definidos por el programador. En la lista de inicialización del constructor, puede tratar a los tipos del lenguaje como si tuvieran un constructor, como aquí:

```
//: C08:BuiltInTypeConstructors.cpp
#include <iostream>
using namespace std;

class B {
  int i;
public:
  B(int ii);
  void print();
};

B::B(int ii) : i(ii) {}
void B::print() { cout << i << endl; }

int main() {
  B a(1), b(2);
  float pi(3.14159);
  a.print(); b.print();
  cout << pi << endl;
} ///:~
```

Esto es especialmente crítico cuando se inicializan atributos constantes porque se deben inicializar antes de entrar en el cuerpo de la función. Tiene sentido extender este «constructor» para los tipos del lenguaje (que simplemente significan asignación) al caso general que es por lo que la definición float funciona en el código anterior. A menudo es útil encapsular un tipo del lenguaje en una clase para garantizar la inicialización con el constructor. Por ejemplo, aquí hay una clase `Integer`:

```
//: C08:EncapsulatingTypes.cpp
#include <iostream>
using namespace std;

class Integer {
  int i;
public:
```

```
  Integer(int ii = 0);
  void print();
};

Integer::Integer(int ii) : i(ii) {}
void Integer::print() { cout << i << ' '; }

int main() {
  Integer i[100];
  for(int j = 0; j < 100; j++)
    i[j].print();
} ///:~
```

El vector de enteros declarado en `main()` se inicializa automáticamente a cero. Esta inicialización no es necesariamente más costosa que un bucle `for` o `memset()`. Muchos compiladores lo optimizan fácilmente como un proceso muy rápido.

8.4.2. Constantes en tiempo de compilación dentro de clases

El uso anterior de `const` es interesante y probablemente útil en muchos casos, pero no resuelve el programa original de «cómo hacer una constante en tiempo de compilación dentro de una clase». La respuesta requiere del uso de un especificador adicional que se explicará completamente en el [FIXME:capítulo 10]: `static`. El especificador `static`, en esta situación significa «hay sólo una instancia a pesar de que se creen varios objetos de la clase» que es precisamente lo que se necesita: un atributo de clase que es constante, y que no cambia de un objeto a otro de la misma clase. Por eso, una `static const` de un tipo básico se puede tratar como una constante en tiempo de compilación.

Hay un característica de `static const` cuando se usa dentro de clases que es un tanto inusual: se debe indicar el valor inicial en el punto en que se define. Esto sólo ocurre con `static const` y no funciona en otras situaciones porque todos lo otros atributos deben inicializarse en el constructor o en otros métodos.

A continuación aparece un ejemplo que muestra la creación y uso de una `static const` llamada `size` en una clase que representa una pila de punteros a cadenas[5].

```
//: C08:StringStack.cpp
// Using static const to create a
// compile-time constant inside a class
#include <string>
#include <iostream>
using namespace std;

class StringStack {
  static const int size = 100;
  const string* stack[size];
  int index;
public:
  StringStack();
  void push(const string* s);
  const string* pop();
```

[5] Al termino de este libro, no todos los compiladores permiten esta característica.

```
};

StringStack::StringStack() : index(0) {
  memset(stack, 0, size * sizeof(string*));
}

void StringStack::push(const string* s) {
  if(index < size)
    stack[index++] = s;
}

const string* StringStack::pop() {
  if(index > 0) {
    const string* rv = stack[--index];
    stack[index] = 0;
    return rv;
  }
  return 0;
}

string iceCream[] = {
  "pralines & cream",
  "fudge ripple",
  "jamocha almond fudge",
  "wild mountain blackberry",
  "raspberry sorbet",
  "lemon swirl",
  "rocky road",
  "deep chocolate fudge"
};

const int iCsz =
  sizeof iceCream / sizeof *iceCream;

int main() {
  StringStack ss;
  for(int i = 0; i < iCsz; i++)
    ss.push(&iceCream[i]);
  const string* cp;
  while((cp = ss.pop()) != 0)
    cout << *cp << endl;
} ///:~
```

Como `size` se usa para determinar el tamaño del vector `stack`, es adecuado usar una constante en tiempo de compilación, pero que queda oculta dentro de la clase.

Fíjese en que `push()` toma un const string* como argumento, `pop()` retorna un const string* y `StringStack` contiene const string*. Si no fuera así, no podría usar una `StringStack` para contener los punteros de `icecream`. En cualquier caso, también impide hacer algo que cambie los objetos contenidos en `StringStack`. Por supuesto, no todos los contenedores están diseñados con esta restricción.

El enumerado en codigo antiguo

En versiones antiguas de C++ el tipo `static const` no se permitía dentro de las clases. Esto hacía que `const` no pudiese usarse para expresiones constantes dentro

de clases. Pero muchos programadores lo conseguían con una solución típica (normalmente conocida como «*enum hack*») que consiste en usar un enum sin etiqueta y sin instancias. Una enumeración debe tener establecidos sus valores en tiempo de compilación, es local a una clase y sus valores están disponibles para expresiones constantes. Por eso, es habitual ver código como:

```
//: C08:EnumHack.cpp
#include <iostream>
using namespace std;

class Bunch {
  enum { size = 1000 };
  int i[size];
};

int main() {
  cout << "sizeof(Bunch) = " << sizeof(Bunch)
       << ", sizeof(i[1000]) = "
       << sizeof(int[1000]) << endl;
} ///:~
```

Este uso de enum garantiza que no se ocupa almacenamiento en el objeto, y que todos los símbolos definidos en la enumeración se evalúan en tiempo de compilación. Además se puede establecer explícitamente el valor de los símbolos:

```
enum { one = 1, two = 2, three };
```

utilizando tipos enum enteros, el compilador continuará contando a partir del último valor, así que el símbolo `three` tendrá un valor 3.

En el ejemplo StringStack anterior, la línea:

```
static const int size = 100;
```

podriá sustituirse por:

```
enum { size = 100 };
```

Aunque es fácil ver esta técnica en código correcto, el uso de `static const` fue añadido al lenguaje precisamente para resolver este problema. En todo caso, no existe ninguna razón abrumadora por la que deba usar `static const` en lugar de enum, y en este libro se utiliza enum porque hay más compiladores que le dan soporte en el momento en el momento en que se escribió este libro.

8.4.3. Objetos y métodos constantes

Las funciones miembro (métodos) se pueden hacer constantes. ¿Qué significa eso? Para entenderlo, primero debe comprender el concepto de objeto constante.

Un objeto constante se define del mismo modo para un tipo definido por el usuario que para un tipo del lenguaje. Por ejemplo:

```
const int i = 1;
const blob b(2);
```

Aquí, `b` es un objeto constante de tipo blob, su constructor se llama con un `2` como argumento. Para que el compilador imponga que el objeto sea constante, debe asegurar que el objeto no tiene atributos que vayan a cambiar durante el tiempo de vida del objeto. Puede asegurar fácilmente que los atributos no públicos no sean modificables, pero. ¿Cómo puede saber que métodos cambiarán los atributos y cuáles son seguros para un objeto constante?

Si declara un método como constante, le está diciendo que la función puede ser invocada por un objeto constante. Un método que no se declara constante se trata como uno que puede modificar los atributos del objeto, y el compilador no permitirá que un objeto constante lo utilice.

Pero la cosa no acaba ahí. Sólo porque un método afirme ser `const` no garantiza que actuará del modo correcto, de modo que el compilador fuerza que en la definición del método se reitere el especificador `const` (la palabra `const` se convierte en parte del nombre de la función, así que tanto el compilador como el enlazador comprobarán que no se viole la constancia). De este modo, si durante la definición de la función se modifica algún miembro o se llama algún método no constante, el compilador emitirá un mensaje de error. Por eso, está garantizado que los miembros que declare `const` se comportarán del modo esperado.

Para comprender la sintaxis para declarar métodos constantes, primero debe recordar que colocar `const` delante de la declaración del método indica que el valor de retorno es constante, así que no produce el efecto deseado. Lo que hay que hacer es colocar el especificador `const` *después* de la lista de argumentos. Por ejemplo:

```
//: C08:ConstMember.cpp
class X {
  int i;
public:
  X(int ii);
  int f() const;
};

X::X(int ii) : i(ii) {}
int X::f() const { return i; }

int main() {
  X x1(10);
  const X x2(20);
  x1.f();
  x2.f();
} ///:~
```

La palabra `const` debe incluirse tanto en la declaración como en la definición del método o de otro modo el compilador asumirá que es un método diferente. Como `f()` es un método constante, si intenta modificar `i` de alguna forma o llamar a otro método que no sea constante, el compilador informará de un error.

Puede ver que un miembro constante puede llamarse tanto desde objetos constantes como desde no constantes de forma segura. Por ello, debe saber que esa es la forma más general para un método (a causa de esto, el hecho de que los métodos

no sean `const` por defecto resulta desafortunado). Un método que no modifica ningún atributo se debería escribir como constante y así se podría usar desde objetos constantes.

Aquí se muestra un ejemplo que compara métodos const y métodos ordinarios:

```
//: C08:Quoter.cpp
// Random quote selection
#include <iostream>
#include <cstdlib> // Random number generator
#include <ctime> // To seed random generator
using namespace std;

class Quoter {
  int lastquote;
public:
  Quoter();
  int lastQuote() const;
  const char* quote();
};

Quoter::Quoter(){
  lastquote = -1;
  srand(time(0)); // Seed random number generator
}

int Quoter::lastQuote() const {
  return lastquote;
}

const char* Quoter::quote() {
  static const char* quotes[] = {
    "Are we having fun yet?",
    "Doctors always know best",
    "Is it ... Atomic?",
    "Fear is obscene",
    "There is no scientific evidence "
    "to support the idea "
    "that life is serious",
    "Things that make us happy, make us wise",
  };
  const int qsize = sizeof quotes/sizeof *quotes;
  int qnum = rand() % qsize;
  while(lastquote >= 0 && qnum == lastquote)
    qnum = rand() % qsize;
  return quotes[lastquote = qnum];
}

int main() {
  Quoter q;
  const Quoter cq;
  cq.lastQuote(); // OK
//!  cq.quote(); // Not OK; non const function
  for(int i = 0; i < 20; i++)
    cout << q.quote() << endl;
} ///:~
```

Ni los constructores ni los destructores pueden ser métodos constantes porque prácticamente siempre realizan alguna modificación en el objeto durante la inicialización o la terminación. El miembro `quote()` tampoco puede ser constante porque modifica el atributo `lastquote` (ver la sentencia de retorno). Por otra parte `lastQuote()` no hace modificaciones y por eso puede ser `const` y puede ser llamado de forma segura por el objeto constante `cq`.

mutable: constancia binaria vs. lógica

¿Qué ocurre si quiere crear un método constante, pero necesita cambiar algún atributo del objeto? Esto se aplica a veces a la diferencia entre constante binaria (*bitwise*) y constante lógica (llamado también constante *memberwise*). Constante binaria significa que todos los bits del objeto son permanentes, así que la imagen binaria del objeto nunca cambia. Constante lógica significa que, aunque el objeto completo es conceptualmente constante puede haber cambios a nivel de miembro. Si se informa al compilador que un objeto es constante, cuidará celosamente el objeto para asegurar constancia binaria. Para conseguir constancia lógica, hay dos formas de cambiar los atributos con un método constante.

La primera solución es la tradicional y se llama constancia *casting away*. Esto se hace de un modo bastante raro. Se toma `this` (la palabra que inidica la dirección del objeto actual) y se moldea el puntero a un puntero a objeto de la clase actual. Parece que `this` ya es un puntero válido. Sin embargo, dentro de un método constante, `this` es en realidad un puntero constante, así que moldeándolo a un puntero ordinario se elimina la constancia del objeto para esta operación. Aquí hay un ejemplo:

```
//: C08:Castaway.cpp
// "Casting away" constness

class Y {
  int i;
public:
  Y();
  void f() const;
};

Y::Y() { i = 0; }

void Y::f() const {
//!  i++; // Error -- const member function
  ((Y*)this)->i++; // OK: cast away const-ness
  // Better: use C++ explicit cast syntax:
  (const_cast<Y*>(this))->i++;
}

int main() {
  const Y yy;
  yy.f(); // Actually changes it!
} ///:~
```

Esta aproximación funciona y puede verse en código correcto, pero no es la técnica ideal. El problema es que esta falta de constancia está oculta en la definición de un método y no hay ningún indicio en la interfaz de la clase que haga sospechar que ese dato se modifica a menos que puede accederse al código fuente (buscando el molde). Para poner todo al descubierto se debe usar la palabra `mutable` en la de-

claración de la clase para indicar que un atributo determinado se puede cambiar aún perteneciendo a un objeto constante.

```
//: C08:Mutable.cpp
// The "mutable" keyword

class Z {
  int i;
  mutable int j;
public:
  Z();
  void f() const;
};

Z::Z() : i(0), j(0) {}

void Z::f() const {
//! i++; // Error -- const member function
    j++; // OK: mutable
}

int main() {
  const Z zz;
  zz.f(); // Actually changes it!
} ///:~
```

De este modo el usuario de la clase puede ver en la declaración qué miembros tienen posibilidad de ser modificados por un método.

ROMability

Si un objeto se define como constante es un candidato para ser almacenado en memoria de sólo lectura (ROM), que a menudo es una consideración importante en programación de sistemas empotrados. Para conseguirlo no es suficiente con que el objeto sea constante, los requisitos son mucha más estrictos. Por supuesto, el objeto debe ser una constante binaria. Eso es fácil de comprobar si la constancia lógica se implementa mediante el uso de `mutable`, pero probablemente el compilador no podrá detectarlo si se utiliza la técnica del moldeado dentro de un método constante. Además:

- La clase o estructura no puede tener constructores o destructor definidos por el usuario.
- No pueden ser clases base (capitulo 14) u objetos miembro con constructores o destructor definidos por el usuario.

El efecto de una operación de escritura en una parte del objeto constante de un tipo ROMable no está definido. Aunque un objeto pueda ser colocado en ROM de forma conveniente, no todos lo requieren.

8.5. Volatile

La sintaxis de `volatile` es idéntica a la de `const`, pero `volatile` significa «este dato puede cambiar sin que el compilador sea informado de ello». De algún

modo, el entorno modifica el dato (posiblemente mediante multitarea, multihilo o interrupciones), y `volatile` indica la compilador que no haga suposiciones sobre el dato, especialmente durante la optimización.

Si el compilador dice, «yo guardé este dato en un registro anteriormente, y no he tocado ese registro», normalmente no necesitará leer el dato de nuevo desde memoria. Pero si esa variable es `volatile`, el compilador no debe hacer esa suposición porque el dato puede haber cambiado a causa de otro proceso, y debe releer el dato en vez de optimizar el código (dicha optimización consiste en eliminar la lectura redundante que se hace normalmente).

Pueden crearse objetos `volatile` usando la misma sintaxis que se usa para crear objetos constantes. También puede crearse objetos `volatile` constantes que no pueden cambiarse por el programador cliente pero se pueden modificar por una entidad ajena al programa. Aquí se muestra un ejemplo que representa una clase asociada con algún elemento físico de comunicación.

```
//: C08:Volatile.cpp
// The volatile keyword

class Comm {
  const volatile unsigned char byte;
  volatile unsigned char flag;
  enum { bufsize = 100 };
  unsigned char buf[bufsize];
  int index;
public:
  Comm();
  void isr() volatile;
  char read(int index) const;
};

Comm::Comm() : index(0), byte(0), flag(0) {}

// Only a demo; won't actually work
// as an interrupt service routine:
void Comm::isr() volatile {
  flag = 0;
  buf[index++] = byte;
  // Wrap to beginning of buffer:
  if(index >= bufsize) index = 0;
}

char Comm::read(int index) const {
  if(index < 0 || index >= bufsize)
    return 0;
  return buf[index];
}

int main() {
  volatile Comm Port;
  Port.isr(); // OK
//!  Port.read(0); // Error, read() not volatile
} ///:~
```

Como ocurre con `const`, se puede usar `volatile` para los atributos de la clase,

los métodos y para los objetos en sí mismos. Sólo puede llamar a métodos `volatile` desde objetos `volatile`.

La razón por la que `isr()` no se puede usar como una rutina de servicio de interrupción (ISR) es que en un método, la dirección del objeto actual (`this`) debe pasarse secretamente, y una ISR no requiere argumentos. Para resolver este problema se puede hacer que el método `isr()` sea un método de clase (`static`), un asunto que se trata en el [FIXME:capitulo 10].

La sintaxis de `volatile` es idéntica a la de `const`, así que por eso se suelen tratar juntos. Cuando se usan combinados se conocen como cuantificador *c-v* (const-volatile).

8.6. Resumen

La palabra `const` permite la posibilidad de definir objetos, argumentos de función, valores de retorno y métodos como constantes y elimina el uso del preprocesador para la sustitución de valores sin perder ninguna de sus ventajas. Todo ello ofrece una forma adicional de comprobación de tipos y seguridad en la programación. El uso de la llamada «constancia exacta» (*const correctness*) es decir, el uso de `const` en todo lugar donde sea posible, puede ser un salvavidas para muchos proyectos.

Aunque ignore a `const` y continue usando el estilo tradicional de C, `const` existe para ayudarle. El [FIXME:capitulo 11] utiliza las referencias extensamente, y se verá más sobre la importancia del uso de `const` en los argumentos de funciones.

8.7. Ejercicios

Las soluciones a los ejercicios se pueden encontrar en el documento electrónico titulado «The Thinking in C++ Annotated Solution Guide», disponible por poco dinero en www.BruceEckel.com.

1. Cree 3 valores enteros constantes, después súmelos todos para producir un valor que determine el tamaño en la definición de un vector. Intente compilar el mismo código en C y vea qué sucede (generalmente se puede forzar al compilador de C++ para que funcione como un compilador de C utilizando alguna opción de linea de comandos).
2. Probar que los compiladores de C y C++ realmente tratan las constantes de modo diferente. Cree una constante global y úsela en una expresión global constante, compile dicho código en C y C++.
3. Cree definiciones constantes para todos los tipos del lenguaje y sus variantes. Úselos en expresiones con otras constantes para hacer definiciones de constantes nuevas. Compruebe que compilan correctamente.
4. Cree una definición de constante en un archivo de cabecera, incluya dicho archivo en dos archivos `.cpp`, compílelos y enlácelos con el compilador de C++. No deberían ocurrir errores. Ahora intente el mismo experimento con el compilador de C.
5. Cree una constante cuyo valor se determine en tiempo de ejecución leyendo la hora en que comienza la ejecución del programa (puede usar `<ctime>`). Después, en el programa, intente leer un segundo valor de hora, almacenarlo en la constante y vea qué sucede.

6. Cree un vector de caracteres constante, después intente cambiar uno de los caracteres.

7. Cree una declaración de constante `extern` en un fichero y ponga un `main()` en el que se imprima el valor de dicha constante. Cree una definición de constante `extern` en un segundo fichero, compile y enlace los dos ficheros.

8. Defina dos punteros a const long utilizando las dos formas de definición. Apunte con uno de ellos a un vector de long. Demuestre que se puede incrementar o decrementar el puntero, pero no se puede cambiar el valor de lo que apunta.

9. Defina un puntero constante a double, y apunte con él a un vector de double. Demuestre que se puede cambiar lo que apunta el puntero pero no se puede incrementar ni decrementar el puntero.

10. Defina un puntero constante a objeto constante. Pruebe que solamente se puede leer el valor de lo que apunta el puntero, pero no se puede cambiar el puntero ni lo que apunta.

11. Elimine el comentario de la linea errónea en `PointerAssignemt.cpp` para ver qué mensaje de error muestra el compilador.

12. Cree un literal de cadena y un puntero que apunte al comienzo del literal. Ahora, use el puntero para modificar los elementos del vector, ¿Informa el compilador de algún error? ¿Debería? Si no lo hace, ¿Porqué piensa que puede ser?

13. Cree una función que tome un argumento por valor como constante, después intente cambiar el argumento en el cuerpo de la función.

14. Cree una función que tome un float por valor. Dentro de la función vincule el argumento a un const float& y use dicha referencia para asegurar que el argumento no sea modificado

15. Modifique `ConstReturnValues.cpp` eliminando los comentarios en las líneas erróneas una cada vez para ver qué mensajes de error muestra el compilador.

16. Modifique `ConsPointer.cpp` eliminando los comentarios en las líneas erróneas para ver qué mensajes de error muestra el compilador.

17. Haga una nueva versión de `ConstPointer.cpp` llamada `ConstReference.cpp` que demuestre el funcionamiento con referencias en lugar de con punteros. (quizá necesite consultar el [FIXME:capítulo 11]).

18. Modifique `ConstTemporary.cpp` eliminando el comentario en la línea errónea para ver el mensaje de error que muestra el compilador.

19. Cree una clase que contenga un float constante y otro no constante. Inicialícelos usando la lista de inicialización del constructor.

20. Cree una clase llamada `MyString` que contenga una cadena y tenga un constructor que inicialice la cadena y un método `print()`. Modifique `StringStack.cpp` para que maneje objetos `MyString` y `main()` para que los imprima.

21. Cree una clase que contenga un atributo constante que se inicialice en la lista de inicialización del constructor y una enumeración no etiquetada que se use para determinar el tamaño de un vector.

22. Elimine el especificador `const` en la definición del método de `ConstMember.cpp`, pero deje el de la declaración para ver qué mensaje de error muestra el compilador.

23. Cree una clase con un método constante y otro ordinario. Cree un objeto constante y otro no constante de esa clase e intente invocar ambos métodos desde ambos objetos.

24. Cree una clase con un método constante y otro ordinario. Intente llamar al método ordinario desde el método constante para ver qué mensaje de error muestra el compilador.

25. Elimine el comentario de la línea errónea en `mutable.cpp` para ver el mensaje de error que muestra el compilador.

26. Modifique `Quoter.cpp` haciendo que `quote()` sea un método constante y `lastquote` sea `mutable`.

27. Cree una clase con un atributo `volatile`. Cree métodos `volatile` y no `volatile` que modifiquen el atributo `volatile` y vea qué dice el compilador. Cree objetos `volatile` y no `volatile` de esa clase e intente llamar a ambos métodos para comprobar si funciona correctamente y ver qué mensajes de error muestra el compilador en caso contrario.

28. Cree una clase llamada `bird` que pueda ejecutar `fly()` y una clase `rock` que no pueda. Crear un objeto `rock`, tome su dirección y asigne a un void*. Ahora tome el void*, asígnelo a un bird* (debe usar un molde) y llame a `fly()` a través de dicho puntero. ¿Esto es posible porque la característica de C que permite asignar a un void* (sin un molde) es un agujero del lenguaje, que no debería propagarse a C++?

9: Funciones `inline`

Una de las características más importantes que C++ hereda de C es la eficiencia. Si la eficiencia de C++ fuese dramáticamente menor que la de C, podría haber un contingente significativo de programadores que no podrían justificar su uso.

En C, una de las maneras de preservar la eficiencia es mediante el uso de macros, lo que permite hacer lo que parece una llamada a una función sin la sobrecarga habitual de la llamada a función. La macro está implementada con el preprocesador en vez del propio compilador, y el preprocesador reemplaza todas las llamadas a macros directamente con el código de la macro, de manera que no hay que complicarse pasando argumentos, escribiendo código de ensamblador para `CALL`, retornando argumentos ni implementando código ensamblador para el `RETURN`. Todo el trabajo lo realizar el preprocesador, de manera que se tiene la coherencia y legibilidad de una llamada a una función pero sin ningún coste.

Hay dos problemas respecto al uso del preprocesador con macros en C++. La primera también existe en C: una macro parece una llamada a función, pero no siempre actúa como tal. Esto puede acarrear dificultades para encontrar errores. El segundo problema es específico de C++: el preprocesador no tiene permisos para acceder a la información de los miembros de una clase. Esto significa que las macros de preprocesador no pueden usarse como métodos de una clase.

Para mantener la eficiencia del uso del preprocesador con macros pero añadiendo la seguridad y la semántica de ámbito de verdaderas funciones en las clases. C++ tiene las funciones `inline`. En este capítulo veremos los problemas del uso de las maros de preprocesador en C++, cómo se resuelven estos problemas con funciones `inline`, y las directrices e incursiones en la forma en que trabajan las funciones inline.

9.1. Los peligros del preprocesador

La clave de los problemas con las macros de preprocesador radica en que puedes caer en el error de pensar que el comportamiento del preprocesador es igual que el del compilador. Por supuesto, la intención era que una macro se parezca y actúe como una llamada a una función, por eso es bastante fácil caer en este error. Las dificultades comienzan cuando las diferencias aparecen subyacentes.

Consideremos un ejemplo sencillo:

```
#define F (x) (x + 1)
```

Ahora, si hacemos una llamada a `F` de esta manera:

```
F(1)
```

El preprocesador la expande de manera inesperada:

```
(x) (x + 1)(1)
```

El problema se debe al espacio entre `F` y su paréntesis de apertura en la definición de la macro. Cuando el espacio es eliminado en el código de la macro, puedes llamar a la función incluso incluyendo el espacio.

```
F (1)
```

Y se expandirá de manera correcta a lo siguiente:

```
    (1 + 1)
```

El ejemplo anterior es un poco trivial y el problema es demasiado evidente. Las dificultades reales ocurren cuando se usan expresiones como argumentos en llamadas a macros.

Hay dos problemas. El primero es que las expresiones pueden expandirse dentro de la macro de modo que la precedencia de la evaluación es diferente a lo que cabría esperar. Por ejemplo:

```
    #define FLOOR(x,b) x>=b?0:1
```

Ahora, si usamos expresiones como argumentos:

```
    if (FLOOR(a&0x0f,0x07)) // ...
```

La macro se expandiría a:

```
    if (a&0x0f>=0x07?0:1)
```

La precedencia del & es menor que la del >=, de modo que la evaluación de la macro te sorprenderá. Una vez hayas descubierto el problema, puedes solucionarlo insertando paréntesis a todo lo que hay dentro de la definición de la macro. (Este es un buen método a seguir cuando defina macros de preprocesador), algo como:

```
    #define FLOOR(x,b) ((x)>=(b)?0:1)
```

De cualquier manera, descubrir el problema puede ser difícil, y no dará con él hasta después de haber dado por sentado el comportamiento de la macro en sí misma. En la versión sin paréntesis de la macro anterior, la mayoría de las expresiones van a actuar de manera correcta a causa de la precedencia de >=, que es menor que la mayoría de los operadores como +, /, --, e incluso los operadores de desplazamiento. Por lo que puede pensar que funciona con todas las expresiones, incluyendo aquellas que empleen operadores lógicos a nivel de bit.

El problema anterior puede solucionarse programando cuidadosamente: poner entre paréntesis todo lo que esté definido dentro de una macro. De todos modos el segundo problema es más sutil. Al contrario de una función normal, cada vez que usa argumentos en una macro, dicho argumento es evaluado. Mientras la macro sea llamada solo con variables corrientes, esta evaluación es benigna, pero si la evaluación de un argumento tiene efectos secundarios, entonces los resultados pueden ser inesperados y definitivamente no imitaran el comportamiento de una función.

Por ejemplo, esta macro determina si un argumento entra dentro de cierto rango:

```
      #define BAND(x) (((x)>5 && (x)<10) ? (x) : 0)
```

Mientras use un argumento «ordinario» la macro trabajará de manera bastante similar a una función real. Pero en cuanto se relaje y comience a creer que realmente es una función, comenzarán los problemas. Así:

```
//: C09:MacroSideEffects.cpp
#include "../require.h"
#include <fstream>
using namespace std;

#define BAND(x) (((x)>5 && (x)<10) ? (x) : 0)

int main() {
  ofstream out("macro.out");
  assure(out, "macro.out");
  for(int i = 4; i < 11; i++) {
    int a = i;
    out << "a = " << a << endl << '\t';
    out << "BAND(++a)=" << BAND(++a) << endl;
    out << "\t a = " << a << endl;
  }
} ///:~
```

Observe el uso de caracteres en mayúscula en el nombre de la macro. Este es un buen recurso ya que advierte al lector que esto es una macro y no una función, entonces si hay algún problema, actúa como recordatorio.

A continuación se muestra la salida producida por el programa, que no es para nada lo que se esperaría de una auténtica función:

```
a = 4
  BAND(++a)=0
   a = 5
a = 5
  BAND(++a)=8
   a = 8
a = 6
  BAND(++a)=9
   a = 9
a = 7
  BAND(++a)=10
   a = 10
a = 8
  BAND(++a)=0
   a = 10
a = 9
  BAND(++a)=0
   a = 11
a = 10
  BAND(++a)=0
```

```
a = 12
```

Cuando a es cuatro, sólo ocurre la primera parte de la condición, de modo que la expresión es evaluada sólo una vez, y el efecto resultante de la llamada a la macro es que a será 5, que es lo que se esperaría de una llamada a función normal en la misma situación. De todos modos, cuando el número está dentro del rango, se evalúan ambas condiciones, lo que da como resultado un tercer incremento. Una vez que el número se sale del rango, ambas condiciones siguen siendo evaluadas de manera que se obtienen dos incrementos. Los efectos colaterales son distintos, dependiendo del argumento.

Este no es desde luego el comportamiento que se quiere de una macro que se parece a una llamada a función. En este caso, la solución obviamente es hacer una autentica función, lo que de hecho implica la cabecera extra y puede reducir la eficiencia si se llama demasiado a esa función. Desafortunadamente, el problema no siempre será tan obvio, y sin saberlo. puede estar utilizando una librería que contiene funciones y macros juntas, de modo que un problema como éste puede esconder errores difíciles de encontrar. Por ejemplo, la macro `putc()` de `cstdio` puede llegar a evaluar dos veces su segundo argumento. Esto está especificado en el Estándar C. Además, la implementación descuidada de `toupper()` como una macro puede llegar a evaluar el argumento más de una vez, lo que dará resultados inesperados con `toupper(*p++)`[1].

9.1.1. Macros y acceso

Por supuesto, C requiere codificación cuidadosa y el uso de macros de preprocesador, y se podría hacer lo mismo en C++ si no fuese por un problema: las macros no poseen el concepto de ámbito requerido con los métodos. El preprocesador simplemente hace substitución de texto, de modo que no puede hacer algo como:

```
class X{
  int i;
public:
#define VAL(X::i) // Error
```

ni nada parecido. Además, no habría ninguna indicación del objeto al que se está refiriendo. Simplemente no hay ninguna forma de expresar el ámbito de clase en una macro. No habiendo ninguna alternativa diferente a macros de preprocesador, los programadores se sentirán tentados de crear algunos atributos públicos por el bien de la eficiencia, exponiendo así la implementación subyacente e impidiendo cambios en esa implementación, así como eliminando la protección que proporciona `private`.

9.2. Funciones `inline`

Al resolver el problema que había en C++ con las macros cuando acceden a miembros de clases privada, todos los problemas asociados con las macros de preprocesador fueron eliminados. Esto se ha hecho aplicando el concepto de macros bajo el control del compilador al cual pertenecen. C++ implementa la macro como una función inline, lo que es una función real en todo sentido. Todo comportamiento esperado de

[1] Andrew Koenig entra en más detalles en su libro *C Traps & Pitfalls* (Addison-Wesley, 1989).

una función ordinaria se obtiene con una función inline. La única diferencia es que una función inline se expande en el mismo sitio, como una macro de preprocesador, de modo que la cabecera de una llamada a función es eliminada. Por ello no debería usar macros (casi) nunca, solo funciones inline.

Cualquier función definida en el cuerpo de una clase es automáticamente inline, pero también puede hacer una función inline que no esté dentro del cuerpo de una clase, precediéndola con la palabra clave inline. De todos modos, para que esto tenga algún efecto, debe incluir el cuerpo de la función con la declaración, de otro modo el compilador tratará esa función como una declaración de una función ordinaria. Así:

```
inline int plusOne(int x);
```

no tiene ningún otro efecto que declarar la función (que puede o no obtener una definición inline después). La aproximación correcta proporciona el cuerpo de la función:

```
inline int plusOne(int x) { return ++x; }
```

Observe que el compilador revisará (como siempre lo hace), el uso apropiado de la lista de argumentos de la función y del valor de retorno (haciendo cualquier conversión necesaria), algo que el preprocesador es incapaz de hacer. Además, si intenta escribir lo anterior como una macro de preprocesador, obtendrá un efecto no deseado.

Casi siempre querrá poner las funciones inline en un fichero de cabecera. Cuando el compilador ve una definición como esa pone el tipo de la función (la firma combinada con el valor de retorno) y el cuerpo de la función en su tabla de símbolos. Cuando use la función, el compilador se asegura de que la llamada es correcta y el valor de retorno se está usando correctamente, y entonces sustituye el cuerpo de la función por la llamada a la función, y de ese modo elimina la sobrecarga. El código inline ocupa espacio, pero si la función es pequeña, realmente ocupará menos espacio que el código generado para una llamada a función ordinaria (colocando los argumentos en la pila y ejecutando el CALL).

Una función inline en un fichero de cabecera tiene un estado especial, dado que debe incluir el fichero de cabecera que contiene la función y su definición en cada fichero en donde se use la función, pero eso no provoca un error de definición múltiple (sin embargo, la definición debe ser idéntica en todos los sitios en los que se incluya la función inline).

9.2.1. inline dentro de clases

Para definir una función inline, debe anteponer la palabra clave `inline` al nombre de la función en el momento de definirla. Sin embargo, eso no es necesario cuando se define dentro de una clase. Cualquier función que defina dentro de una clase es inline automáticamente. Por ejemplo:

```
//: C09:Inline.cpp
// Inlines inside classes
#include <iostream>
#include <string>
using namespace std;
```

```
class Point {
  int i, j, k;
public:
  Point(): i(0), j(0), k(0) {}
  Point(int ii, int jj, int kk)
    : i(ii), j(jj), k(kk) {}
  void print(const string& msg = "") const {
    if(msg.size() != 0) cout << msg << endl;
    cout << "i = " << i << ", "
         << "j = " << j << ", "
         << "k = " << k << endl;
  }
};

int main() {
  Point p, q(1,2,3);
  p.print("value of p");
  q.print("value of q");
} ///:~
```

Aquí, los dos constructores y la función `print()` son inline por defecto. Dese cuenta de que usar funciones inline es transparente en `main()`, y así debe ser. El comportamiento lógico de una función debe ser idéntico aunque sea inline (de otro modo su compilador no funcionaría). La única diferencia visible es el rendimiento.

Por supuesto, la tentación es usar declaraciones `inline` en cualquier parte dentro de la case porque ahorran el paso extra de hacer una definición de método externa. Sin embargo, debe tener presente, que la idea de una inline es dar al compilador mejores oportunidades de optimización. Pero, si declara inline una función grande provocará que el código se duplique allí donde se llame, produciendo código [FIXME:bloat] que anulará el beneficio de velocidad obtenido (la única manera de descubrir los efectos del uso de inline en su programa con su compilador es experimentar).

9.2.2. Funciones de acceso

Uno de los usos más importantes de inline dentro de clases son las funciones de acceso. Se trata de pequeñas funciones que le permiten leer o cambiar parte del estado de un objeto, es decir, una o varias variables internas. La razón por la que inline es tan importante para las funciones de acceso se puede ver en el siguiente ejemplo:

```
//: C09:Access.cpp
// Inline access functions

class Access {
  int i;
public:
  int read() const { return i; }
  void set(int ii) { i = ii; }
};

int main() {
  Access A;
  A.set(100);
```

```
  int x = A.read();
} ///:~
```

Aquí, el usuario de la clase nunca tiene contacto directo con las variables de estado internas a la clase, y pueden mantenerse como privadas, bajo el control del diseñador de la clase. Todo el acceso a los atributos se puede controlar a través de los métodos de la interfaz. Además, el acceso es notablemente eficiente. Considere `read()`, por ejemplo. Sin inline, el código generado para la llamada a `read()` podría incluir colocarla en la pila y ejecutar la llamada CALL de ensamblador. En la mayoría de las arquitecturas, el tamaño de ese código sería mayor que el código creado para la variante inline, y el tiempo de ejecución sería mayor con toda certeza.

Sin las funciones inline, un diseñador de clases preocupado por la eficiencia estaría tentado de hacer que `i` fuese un atributo público, eliminado la sobrecarga y permitiendo al usuario acceder directamente a `i`. Desde el punto de vista del diseñador, eso resulta desastroso, `i` sería parte de la interfaz pública, lo cual significa que el diseñador de la clase no podrá cambiarlo en el futuro. Tendrá que cargar con un entero llamado `i`. Esto es un problema porque después puede que considere mejor usar un float en lugar de un int para representar el estado, pero como `i` es parte de la interfaz pública, no podrá cambiarlo. O puede que necesite realizar algún cálculo adicional como parte de la lectura o escritura de `i`, que no podrá hacer si es público. Si, por el contrario, siempre usa métodos para leer y cambiar la información de estado del objeto, podrá modificar la representación subyacente del objeto hasta estar totalmente convencido.

Además, el uso de métodos para controlar atributos le permite añadir código al método para detectar cuando cambia el valor, algo que puede ser muy útil durante la depuración. Si un atributo es público, cualquiera puede cambiarlo en cualquier momento sin que el programador lo sepa.

Accesores y mutadores

Hay gente que divide el concepto de funciones de acceso en dos: accesores (para leer la información de estado de un objeto) y mutadores (para cambiar el estado de un objeto). Además, se puede utilizar la sobrecarga de funciones para tener métodos accesores y mutadores con el mismo nombre; el modo en que se invoque el método determina si se lee o modifica la información de estado. Así,

```
//: C09:Rectangle.cpp
// Accessors & mutators

class Rectangle {
  int wide, high;
public:
  Rectangle(int w = 0, int h = 0)
    : wide(w), high(h) {}
  int width() const { return wide; } // Read
  void width(int w) { wide = w; } // Set
  int height() const { return high; } // Read
  void height(int h) { high = h; } // Set
};

int main() {
  Rectangle r(19, 47);
  // Change width & height:
```

```
  r.height(2 * r.width());
  r.width(2 * r.height());
} ///:~
```

El constructor usa la lista de inicialización (brevemente introducida en el capítulo 8 y ampliamente cubierta en el capitulo 14) para asignar valores a `wide` y `high` (usando el formato de pseudo-constructor para los tipos de datos básicos).

No puede definir métodos que tengan el mismo nombre que los atributos, de modo que puede que se sienta tentado de distinguirlos con un guión bajo al final. Sin embargo, los identificadores con guiones bajos finales están reservados y el programador no debería usarlos.

En su lugar, debería usar «set» y «get» para indicar que los métodos son accesores y mutadores.

```
//: C09:Rectangle2.cpp
// Accessors & mutators with "get" and "set"

class Rectangle {
  int width, height;
public:
  Rectangle(int w = 0, int h = 0)
    : width(w), height(h) {}
  int getWidth() const { return width; }
  void setWidth(int w) { width = w; }
  int getHeight() const { return height; }
  void setHeight(int h) { height = h; }
};

int main() {
  Rectangle r(19, 47);
  // Change width & height:
  r.setHeight(2 * r.getWidth());
  r.setWidth(2 * r.getHeight());
} ///:~
```

Por supuesto, los accesores y mutadores no tienen porqué ser simples tuberías hacia las variables internas. A veces, pueden efectuar cálculos más sofisticados. El siguiente ejemplo usa las funciones de tiempo de la librería C estándar para crear una clase `Time`:

```
//: C09:Cpptime.h
// A simple time class
#ifndef CPPTIME_H
#define CPPTIME_H
#include <ctime>
#include <cstring>

class Time {
  std::time_t t;
  std::tm local;
  char asciiRep[26];
  unsigned char lflag, aflag;
  void updateLocal() {
```

```
    if(!lflag) {
      local = *std::localtime(&t);
      lflag++;
    }
  }
  void updateAscii() {
    if(!aflag) {
      updateLocal();
      std::strcpy(asciiRep,std::asctime(&local));
      aflag++;
    }
  }
public:
  Time() { mark(); }
  void mark() {
    lflag = aflag = 0;
    std::time(&t);
  }
  const char* ascii() {
    updateAscii();
    return asciiRep;
  }
  // Difference in seconds:
  int delta(Time* dt) const {
    return int(std::difftime(t, dt->t));
  }
  int daylightSavings() {
    updateLocal();
    return local.tm_isdst;
  }
  int dayOfYear() { // Since January 1
    updateLocal();
    return local.tm_yday;
  }
  int dayOfWeek() { // Since Sunday
    updateLocal();
    return local.tm_wday;
  }
  int since1900() { // Years since 1900
    updateLocal();
    return local.tm_year;
  }
  int month() { // Since January
    updateLocal();
    return local.tm_mon;
  }
  int dayOfMonth() {
    updateLocal();
    return local.tm_mday;
  }
  int hour() { // Since midnight, 24-hour clock
    updateLocal();
    return local.tm_hour;
  }
  int minute() {
    updateLocal();
    return local.tm_min;
  }
```

```
  int second() {
    updateLocal();
    return local.tm_sec;
  }
};
#endif // CPPTIME_H ///:~
```

Las funciones de la librería C estándar tienen múltiples representaciones para el tiempo, y todas ellas son parte de la clase `Time`. Sin embargo, no es necesario actualizar todos ellos, así que time_t se usa para la representación base, y tm `local` y la representación ASCII `asciiRep` tienen banderas para indicar si han sido actualizadas para el time_t actual. Las dos funciones privadas `updateLocal()` y `updateAscii()` comprueban las banderas y condicionalmente hacen la actualización.

El constructor llama a la función `mark()` (que el usuario puede llamar también para forzar al objeto a representar el tiempo actual), y eso limpia las dos banderas para indicar que el tiempo local y la representación ASCII ya no son válidas. La función `ascii()` llama a `updateAscii()`, que copia el resultado de la función de la librería estándar de C `asctime()` en un buffer local porque `asctime()` usa una área de datos estática que se sobreescribe si la función se llama en otra parte. El valor de retorno de la función `ascii()` es la dirección de ese buffer local.

Todas las funciones que empiezan con `daylightSavings()` usan la función `updateLocal()`, que causa que la composición resultante de inlines sea bastante larga. No parece que valga la pena, especialmente considerando que probablemente no quiera llamar mucho a esas funciones. Sin embargo, eso no significa que todas las funciones deban ser no-inline. Si hace otras funciones no-inline, al menos mantenga `updateLocal()` como `inline` de modo que su código se duplique en las funciones no-inline, eliminando la sobrecarga extra de invocación de funciones.

Este es un pequeño programa de prueba:

```
//: C09:Cpptime.cpp
// Testing a simple time class
#include "Cpptime.h"
#include <iostream>
using namespace std;

int main() {
  Time start;
  for(int i = 1; i < 1000; i++) {
    cout << i << ' ';
    if(i%10 == 0) cout << endl;
  }
  Time end;
  cout << endl;
  cout << "start = " << start.ascii();
  cout << "end = " << end.ascii();
  cout << "delta = " << end.delta(&start);
} ///:~
```

Se crea un objeto `Time`, se hace alguna actividad que consuma tiempo, después se crea un segundo objeto `Time` para marcar el tiempo de finalización. Se usan para mostrar los tiempos de inicio, fin y los intervalos.

9.3. Stash y Stack con inlines

Disponiendo de inlines, podemos modificar las clases `Stash` y `Stack` para hacerlas más eficientes.

```
//: C09:Stash4.h
// Inline functions
#ifndef STASH4_H
#define STASH4_H
#include "../require.h"

class Stash {
  int size;      // Size of each space
  int quantity;  // Number of storage spaces
  int next;      // Next empty space
  // Dynamically allocated array of bytes:
  unsigned char* storage;
  void inflate(int increase);
public:
  Stash(int sz) : size(sz), quantity(0),
    next(0), storage(0) {}
  Stash(int sz, int initQuantity) : size(sz),
    quantity(0), next(0), storage(0) {
    inflate(initQuantity);
  }
  Stash::~Stash() {
    if(storage != 0)
      delete []storage;
  }
  int add(void* element);
  void* fetch(int index) const {
    require(0 <= index, "Stash::fetch (-)index");
    if(index >= next)
      return 0; // To indicate the end
    // Produce pointer to desired element:
    return &(storage[index * size]);
  }
  int count() const { return next; }
};
#endif // STASH4_H ///:~
```

Obviamente las funciones pequeñas funcionan bien como inlines, pero note que las dos funciones más largas siguen siendo no-inline, dado que convertirlas a inline no representaría ninguna mejora de rendimiento.

```
//: C09:Stash4.cpp {O}
#include "Stash4.h"
#include <iostream>
#include <cassert>
using namespace std;
const int increment = 100;

int Stash::add(void* element) {
  if(next >= quantity) // Enough space left?
    inflate(increment);
  // Copy element into storage,
```

```
  // starting at next empty space:
  int startBytes = next * size;
  unsigned char* e = (unsigned char*)element;
  for(int i = 0; i < size; i++)
    storage[startBytes + i] = e[i];
  next++;
  return(next - 1); // Index number
}

void Stash::inflate(int increase) {
  assert(increase >= 0);
  if(increase == 0) return;
  int newQuantity = quantity + increase;
  int newBytes = newQuantity * size;
  int oldBytes = quantity * size;
  unsigned char* b = new unsigned char[newBytes];
  for(int i = 0; i < oldBytes; i++)
    b[i] = storage[i]; // Copy old to new
  delete [](storage); // Release old storage
  storage = b; // Point to new memory
  quantity = newQuantity; // Adjust the size
} ///:~
```

Una vez más, el programa de prueba que verifica que todo funciona correctamente.

```
//: C09:Stash4Test.cpp
//{L} Stash4
#include "Stash4.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
  Stash intStash(sizeof(int));
  for(int i = 0; i < 100; i++)
    intStash.add(&i);
  for(int j = 0; j < intStash.count(); j++)
    cout << "intStash.fetch(" << j << ") = "
         << *(int*)intStash.fetch(j)
         << endl;
  const int bufsize = 80;
  Stash stringStash(sizeof(char) * bufsize, 100);
  ifstream in("Stash4Test.cpp");
  assure(in, "Stash4Test.cpp");
  string line;
  while(getline(in, line))
    stringStash.add((char*)line.c_str());
  int k = 0;
  char* cp;
  while((cp = (char*)stringStash.fetch(k++))!=0)
    cout << "stringStash.fetch(" << k << ") = "
         << cp << endl;
} ///:~
```

Este es el mismo programa de prueba que se usó antes, de modo que la salida debería ser básicamente la misma.

La clase `Stack` incluso hace mejor uso de inline's.

```
//: C09:Stack4.h
// With inlines
#ifndef STACK4_H
#define STACK4_H
#include "../require.h"

class Stack {
  struct Link {
    void* data;
    Link* next;
    Link(void* dat, Link* nxt):
      data(dat), next(nxt) {}
  }* head;
public:
  Stack() : head(0) {}
  ~Stack() {
    require(head == 0, "Stack not empty");
  }
  void push(void* dat) {
    head = new Link(dat, head);
  }
  void* peek() const {
    return head ? head->data : 0;
  }
  void* pop() {
    if(head == 0) return 0;
    void* result = head->data;
    Link* oldHead = head;
    head = head->next;
    delete oldHead;
    return result;
  }
};
#endif // STACK4_H ///:~
```

Note que el destructor `Link`, que se presentó (vacío) en la versión anterior de `Stack`, ha sido eliminado. En `pop()`, la expresión `delete oldHead` simplemente libera la memoria usada por `Link` (no destruye el objeto `data` apuntado por el `Link`).

La mayoría de las funciones inline quedan bastante bien obviamente, en especial para `Link`. Incluso `pop()` parece justificado, aunque siempre que haya sentencias condicionales o variables locales no está claro que las inlines sean beneficiosas. Aquí, la función es lo suficientemente pequeña así que es probable que no haga ningún daño.

Si todas sus funciones son inline, usar la librería se convierte en algo bastante simple porque el enlazado es innecesario, como puede ver en el ejemplo de prueba (fíjese en que no hay `Stack4.cpp`).

```
//: C09:Stack4Test.cpp
//{T} Stack4Test.cpp
#include "Stack4.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // File name is argument
  ifstream in(argv[1]);
  assure(in, argv[1]);
  Stack textlines;
  string line;
  // Read file and store lines in the stack:
  while(getline(in, line))
    textlines.push(new string(line));
  // Pop the lines from the stack and print them:
  string* s;
  while((s = (string*)textlines.pop()) != 0) {
    cout << *s << endl;
    delete s;
  }
} ///:~
```

La gente escribe a veces clases con todas sus funciones inline, así que la clase completa está en el fichero de cabecera (verá en este libro que yo mismo lo hago). Durante el desarrollo de un programa probablemente esto es inofensivo, aunque a veces puede hacer que las compilaciones sean más lentas. Cuando el programa se estabiliza un poco, probablemente querrá volver a hacer las funciones no-inline donde sea conveniente.

9.4. Funciones inline y el compilador

Para comprender cuando es conveniente utilizar inlines, es útil saber lo que hace el compilador cuando encuentra una función inline. Como con cualquier función, el compilador apunta el *tipo* de la función es su tabla de símbolos (es decir, el prototipo de la función incluyendo el nombre y los tipos de los argumentos, en combinación con valor de retorno). Además cuando el compilador ve que la función es inline *y* el cuerpo no contiene errores, el código se coloca también en la tabla de símbolos. El código se almacena en su forma fuente, como instrucciones ensamblador compiladas, o alguna otra representación propia del compilador.

Cuando hace una llamada a una función inline, el compilador se asegura primero de que la llamada se puede hacer correctamente. Es decir, los tipos de todos los argumentos corresponden exactamente con los tipos de la lista de argumentos de la función (o convertible a tipo correcto) y el valor de retorno tiene el tipo correcto (o es convertible al tipo correcto) en la expresión destino. Esto, por supuesto, es exactamente lo mismo que hace el compilador para cualquier función y hay una diferencia considerable respecto de lo que hace el preprocesador, porque el preprocesador no comprueba tipos ni hace conversiones.

Si toda la información del tipo de la función encaja en el contexto de la llamada, entonces la llamada a la función se sustituye directamente por el código inline, eliminando la sobrecarga y permitiendo que el compilador pueda hacer más optimizaciones. Además, si el inline es un método, la dirección del objeto(`this`) se pone en el lugar apropiado, que es, por supuesto, otra acción que el preprocesador es incapaz de hacer.

9.4.1. Limitaciones

Hay dos situaciones en que el compilador no puede efectuar la sustitución de inline. En estos casos, simplemente convierte la función a la forma ordinaria tomando la definición y pidiendo espacio para la función como hace con una función no-inline. Si debe hacerlo en varias unidades de traducción (lo que normalmente causaría un error de definición múltiple), informa al enlazador que ignore esas definiciones múltiples.

En compilador no puede efectuar la sustitución de inline si la función es demasiado complicada. Esto depende de cada compilador particular, pero aunque muchos compiladores lo hagan, no habrá ninguna mejora de eficiencia. En general, se considera que cualquier tipo de bucle es demasiado complicado para expandir como una inline, y si lo piensa, el bucle implica mucho más tiempo que el que conlleva la sobrecarga de la invocación de la función. Si la función es simplemente una colección se sentencias simples, probablemente el compilador no tendrá ningún problema para utilizar inline, pero si hay muchas sentencias, la sobrecarga de llamada será mucho menor que el coste de ejecutar el cuerpo. Y recuerde, cada vez que llame a una función inline grande, el cuerpo completo se inserta en el lugar de la llamada, de modo que el tamaño del código se inflará fácilmente sin que se perciba ninguna mejora de rendimiento. (Note que algunos de los ejemplos de este libro pueden exceder el tamaño razonable para una inline a cambio de mejorar la estética de los listados.

El compilador tampoco efectúa sustituciones inline si la dirección de la función se toma implícita o explícitamente. Si el compilador debe producir una dirección, entonces tendrá que alojar el código de la función y usar la dirección resultante. Sin embargo, cuando no se requiere una dirección, probablemente el compilador hará la sustitución inline.

Es importante comprender que una declaración inline es sólo una sugerencia al compilador; el compilador no está forzado a hacer nada. Un buen compilador hará sustituciones inline para funciones pequeñas y simples mientras que ignorará las que sean demasiado complicadas. Eso le dará lo que espera - la auténtica semántica de una llamada a función con la eficiencia de una macro.

9.4.2. Referencias adelantadas

Si está imaginando que el compilador [FIXME: is doing to implement inlines], puede confundirse pensando que hay más limitaciones que las que existen realmente. En concreto, si una inline hace una referencia adelanta a una función que no ha sido declarada aún en la clase (sea inline o no), puede parecer que el compilador no sabrá tratarla.

```
//: C09:EvaluationOrder.cpp
// Inline evaluation order

class Forward {
```

```
  int i;
public:
  Forward() : i(0) {}
  // Call to undeclared function:
  int f() const { return g() + 1; }
  int g() const { return i; }
};

int main() {
  Forward frwd;
  frwd.f();
} ///:~
```

En `f()`, se realiza una llamada a `g()`, aunque `g()` aún no ha sido declarada. Esto funciona porque la definición del lenguaje dice que las funciones inline en una clase no serán evaluadas hasta la llave de cierre de la declaración de clase.

Por supuesto, si `g()` a su vez llama a `f()`, tendrá un conjunto de llamadas recursivas, que son demasiado complicadas para el compilador pueda hacer inline. (También, tendrá que efectuar alguna comprobación en `f()` o `g()` para forzar en alguna de ellas un caso base, o la recursión será infinita).

9.4.3. Actividades ocultas en contructores y destructores

Constructores y destructores son dos lugares dónde puede engañarse al pensar que una inline es más eficiente de lo que realmente es. Constructores y destructores pueden tener actividades ocultas, porque la clase puede contener subobjetos cuyos constructores y destructores deben invocarse. Estos subobjetos pueden ser objetos miembro (atributos), o pueden existir por herencia (tratado en el Capítulo 14). Como un ejemplo de clase con un objeto miembro:

```
//: C09:Hidden.cpp
// Hidden activities in inlines
#include <iostream>
using namespace std;

class Member {
  int i, j, k;
public:
  Member(int x = 0) : i(x), j(x), k(x) {}
  ~Member() { cout << "~Member" << endl; }
};

class WithMembers {
  Member q, r, s; // Have constructors
  int i;
public:
  WithMembers(int ii) : i(ii) {} // Trivial?
  ~WithMembers() {
    cout << "~WithMembers" << endl;
  }
};

int main() {
  WithMembers wm(1);
```

```
} ///:~
```

El constructor para `Member` es suficientemente simple para ser inline, dado que no hay nada especial en él - ninguna herencia u objeto miembro está provocando actividades ocultas adicionales. Pero en la clase `WithMembers` hay más de lo que se ve a simple vista. Los constructores y destructores para los atributos `q`, `r` y `s` se llaman automáticamente, y esos constructores y destructores también son inline, así que la diferencia es significativa respecto a métodos normales. Esto no significa necesariamente que los constructores y destructores deban ser no-inline; hay casos en que tiene sentido. También, cuando se está haciendo un prototipo inicial de un programa escribiendo código rápidamente, es conveniente a menudo usar inlines. Pero si está preocupado por la eficiencia, es un sitio donde mirar.

9.5. Reducir el desorden

En un libro como éste, la simplicidad y brevedad de poner definiciones inline dentro de las clases es muy útil porque permite meter más en una página o pantalla (en un seminario). Sin embargo, Dan Saks [2] ha apuntado que en un proyecto real esto tiene como consecuencia el desorden de la interfaz de la clase y eso hace que la clase sea más incomoda de usar. Él se refiere a los métodos definidos dentro de las clases usando la expresión *in situ* (en el lugar) e indica que todas las definiciones deberían colocarse fuera de la clase manteniendo la interfaz limpia. La optimización, argumenta él, es una asunto distinto. Si se requiere optimizar, use la palabra reservada `inline`. Siguiente ese enfoque, el ejemplo anterior `Rectangle.cpp` quedaría:

```
//: C09:Noinsitu.cpp
// Removing in situ functions

class Rectangle {
  int width, height;
public:
  Rectangle(int w = 0, int h = 0);
  int getWidth() const;
  void setWidth(int w);
  int getHeight() const;
  void setHeight(int h);
};

inline Rectangle::Rectangle(int w, int h)
  : width(w), height(h) {}

inline int Rectangle::getWidth() const {
  return width;
}

inline void Rectangle::setWidth(int w) {
  width = w;
}

inline int Rectangle::getHeight() const {
  return height;
}
```

[2] Co-autor junto a Tom Plum de C++ *Programming Guidelines*, Plum Hall, 1991.

```
inline void Rectangle::setHeight(int h) {
  height = h;
}

int main() {
  Rectangle r(19, 47);
  // Transpose width & height:
  int iHeight = r.getHeight();
  r.setHeight(r.getWidth());
  r.setWidth(iHeight);
} ///:~
```

Ahora si quiere comparar el efecto de la funciones inline con la versión convencional, simplemente borre la palabra `inline`. (Las funciones inline normalmente deberían aparecen en los ficheros de cabecera, no obstante, las funciones no-inline deberían residir en un propia unidad de traducción). Si quiere poner las funciones en la documentación, es tan simple como un «copiar y pegar». Las funciones *in situ* requieren más trabajo y tienen más posibilidades de provocar errores. Otro argumento para esta propuesta es que siempre puede producir un estilo de formato consistente para las definiciones de función, algo que no siempre ocurre con las funciones *in situ*.

9.6. Más características del preprocesador

Antes, se dijo que *casi* siempre se prefiere usar funciones inline en lugar de macros del preprocesador. Las excepciones aparecen cuando necesita usar tres propiedades especiales del preprocesador de C (que es también el preprocesador de C++): [FIXME(hay más):cadenización?] (*stringizing*), concatenación de cadenas, y encolado de símbolos (*token pasting*). *Stringizing*, ya comentado anteriormente en el libro, se efectúa con la directiva `#` y permite tomar un identificador y convertirlo en una cadena de caracteres. La concatenación de cadenas tiene lugar cuando dos cadenas adyacentes no tienen puntuación, en cuyo caso se combinan. Estas dos propiedades son especialmente útiles cuando se escribe código de depuración. Así,

```
    #define DEBUG(x) cout << #x " = " << x << endl
```

Esto imprime el valor de cualquier variable. Puede conseguir también una traza que imprima las sentencias tal como se ejecutan:

```
    #define TRACE(s) cerr << #s << endl; s
```

El `#s` *cadeniza* la sentencia para la salida, y la segunda `s` hace que la sentencia se ejecute. Por supuesto, este tipo de cosas pueden causar problemas, especialmente bucles `for` de una única línea.

```
    for(int i = 0; i < 100; i++)
     TRACE(f(i));
```

Como realmente hay dos sentencias en la macro `TRACE()`, el bucle `for` de una única línea ejecuta solo la primera. La solución es reemplazar el punto y coma por una coma en la macro.

9.6.1. Encolado de símbolos

El encolado de símbolos, implementado con la directiva `##`, es muy útil cuando se genera código. Permite coger dos identificadores y pegarlos juntos para crear un nuevo identificador automáticamente. Por ejemplo,

```
#define FIELD(a) char* a##_string; int a##_size
class Record {
  FIELD(one);
  FIELD(two);
  FIELD(three);
  // ...
};
```

Cada llamada a la macro `FIELD()` crea un identificador para una cadena de caracteres y otro para la longitud de dicha cadena. No solo es fácil de leer, también puede eliminar errores de codificación y facilitar el mantenimiento.

9.7. Comprobación de errores mejorada

Las funciones de `require.h` se han usado antes de este punto sin haberlas definido (aunque `assert()` se ha usado también para ayudar a detectar errores del programador donde es apropiado). Ahora es el momento de definir este fichero de cabecera. Las funciones inline son convenientes aquí porque permiten colocar todo en el fichero de cabecera, lo que simplifica el proceso para usar el paquete. Simplemente, incluya el fichero de cabecera y se preocupe por enlazar un fichero de implementación.

Debería fijarse que las excepciones (presentadas en detalle en el Volumen 2 de este libro) proporcionan una forma mucho más efectiva de manejar muchos tipos de errores -especialmente aquellos de los que debería recuperarse- en lugar de simplemente abortar el programa. Las condiciones que maneja `require.h`, sin embargo, son algunas que impiden que el programa continúe, como por ejemplo que el usuario no introdujo suficientes argumentos en la línea de comandos o que un fichero no se puede abrir. De modo que es aceptable que usen la función `exit()` de la librería C estándar.

El siguiente fichero de cabecera está en el directorio raíz del libro, así que es fácilmente accesible desde todos los capítulos.

```
//: :require.h
// From Thinking in C++, 2nd Edition
// Available at http://www.BruceEckel.com
// (c) Bruce Eckel 2000
// Copyright notice in Copyright.txt
// Test for error conditions in programs
// Local "using namespace std" for old compilers
#ifndef REQUIRE_H
#define REQUIRE_H
#include <cstdio>
#include <cstdlib>
#include <fstream>
#include <string>

inline void require(bool requirement,
```

```
  const std::string& msg = "Requirement failed"){
  using namespace std;
  if (!requirement) {
    fputs(msg.c_str(), stderr);
    fputs("\n", stderr);
    exit(1);
  }
}

inline void requireArgs(int argc, int args,
  const std::string& msg =
    "Must use %d arguments") {
  using namespace std;
   if (argc != args + 1) {
     fprintf(stderr, msg.c_str(), args);
     fputs("\n", stderr);
     exit(1);
   }
}

inline void requireMinArgs(int argc, int minArgs,
  const std::string& msg =
    "Must use at least %d arguments") {
  using namespace std;
  if(argc < minArgs + 1) {
    fprintf(stderr, msg.c_str(), minArgs);
    fputs("\n", stderr);
    exit(1);
  }
}

inline void assure(std::ifstream& in,
  const std::string& filename = "") {
  using namespace std;
  if(!in) {
    fprintf(stderr, "Could not open file %s\n",
      filename.c_str());
    exit(1);
  }
}

inline void assure(std::ofstream& out,
  const std::string& filename = "") {
  using namespace std;
  if(!out) {
    fprintf(stderr, "Could not open file %s\n",
      filename.c_str());
    exit(1);
  }
}
#endif // REQUIRE_H ///:~
```

Los valores por defecto proporcionan mensajes razonables que se pueden cambiar si es necesario.

Fíjese en que en lugar de usar argumentos char* se utiliza const string&. Esto permite tanto char*, cadenas `string` como argumentos para estas funciones, y así

es más general (quizá quiera utilizar esta forma en su propio código).

En las definiciones para `requireArgs()` y `requireMinArgs()`, se añade uno al número de argumentos que necesita en la línea de comandos porque `argc` siempre incluye el nombre del programa que está ejecutado como argumento cero, y por eso siempre tiene un valor que excede en uno al número real de argumentos de la línea de comandos.

Fíjese en el uso de declaraciones locales `using namespace std` con cada función. Esto es porque algunos compiladores en el momento de escribir este libro incluyen incorrectamente las funciones de la librería C estándar en el espacio de nombres `std`, así que la cualificación explícita podría causar un error en tiempo de compilación. Las declaraciones locales permiten que `require.h` funcione tanto con librerías correctas como con incorrectas sin abrir el espacio de nombres `std` para cualquiera que incluya este fichero de cabecera.

Aquí hay un programa simple para probar `requite.h`:

```
//: C09:ErrTest.cpp
//{T} ErrTest.cpp
// Testing require.h
#include "../require.h"
#include <fstream>
using namespace std;

int main(int argc, char* argv[]) {
  int i = 1;
  require(i, "value must be nonzero");
  requireArgs(argc, 1);
  requireMinArgs(argc, 1);
  ifstream in(argv[1]);
  assure(in, argv[1]); // Use the file name
  ifstream nofile("nofile.xxx");
  // Fails:
//!  assure(nofile); // The default argument
  ofstream out("tmp.txt");
  assure(out);
} ///:~
```

Podría estar tentado a ir un paso más allá para manejar la apertura de ficheros y añadir una macro a `require.h`.

```
    #define IFOPEN(VAR, NAME) \
      ifstream VAR(NAME); \
      assure(VAR, NAME);
```

Que podría usarse entonces así:

```
    IFOPEN(in, argv[1])
```

En principio, esto podría parecer atractivo porque significa que hay que escribir menos. No es terriblemente inseguro, pero es un camino que es mejor evitar. Fíjese que, de nuevo, una macro parece una función pero se comporta diferente; realmente se está creando un objeto `in` cuyo alcance persiste más allá de la macro. Quizá lo entienda, pero para programadores nuevos y mantenedores de código sólo es una

cosa más que ellos deben resolver. C++ es suficientemente complicado sin añadir confusión, así que intente no abusar de las macros del preprocesador siempre que pueda.

9.8. Resumen

Es crítico que sea capaz de ocultar la implementación subyacente de una clase porque puede querer cambiarla después. Hará estos cambios por eficiencia, o porque haya alcanzado una mejor comprensión del problema, o porque hay alguna clase nueva disponible para usar en la implementación. Cualquier cosa que haga peligrar la privacidad de la implementación subyacente reduce la flexibilidad del lenguaje. Por eso, la función inline es muy importante porque prácticamente elimina la necesidad de macros de preprocesador y sus problemas asociados. Con inline, los métodos pueden ser tan eficientes como las macros.

Por supuesto se puede abusan de las funciones inline en las definiciones de clase. El programador está tentado de hacerlo porque es fácil, así que lo hace. Sin embargo, no es un problema grave porque después, cuando se busquen reducciones de tamaño, siempre puede cambiar las inline a funciones convencionales dado que no afecta a su funcionalidad. La pauta debería ser «Primero haz el trabajo, después optimiza».

9.9. Ejercicios

Las soluciones a los ejercicios se pueden encontrar en el documento electrónico titulado «The Thinking in C++ Annotated Solution Guide», disponible por poco dinero en www.BruceEckel.com.

1. Escriba un programa que use la macro `F()` mostrada al principio del capítulo y demuestre que no se expande apropiadamente, tal como describe el texto. Arregle la macro y demuestre que funciona correctamente.
2. Escriba un programa que use la macro `FLOOR()` mostrada al principio del capítulo. Muestre las condiciones en que no funciona apropiadamente.
3. Modifique `MacroSideEffects.cpp` de modo que `BAND()` funcione adecuadamente.
4. Cree dos funciones idénticas, `f1()` y `f2()`. Haga inline a `f1()` y deje `f2()` como no-inline. Use la función `clock()` de la librería C estándar que se encuentra en `<ctime>` para marcar los puntos de comienzo y fin y compare las dos funciones para ver cuál es más rápida. Puede que necesite hacer un bucle de llamadas repetidas para conseguir números representativos.
5. Experimente con el tamaño y complejidad del código de las funciones del ejercicio 4 para ver si puede encontrar el punto donde la función inline y la convencional tardan lo mismo. Si dispone de ellos, inténtelo con compiladores distintos y fíjese en las diferencias.
6. Pruebe que las funciones inline hacen enlazado interno por defecto.
7. Cree una clase que contenga un array de caracteres. Añada un constructor inline que use la función `memset()` de la librería C estándar para inicializar el array al valor dado como argumento del constructor (por defecto será ' '), y un método inline llamado `print()` que imprima todos los caracteres del array.

8. Coja el ejemplo `NestFriend.cpp` del Capítulo 5 y reemplace todos los métodos con inline. No haga métodos inline *in situ*. También cambie las funciones `initialize()` por constructores.

9. Modifique `StringStack.cpp` del Capítulo 8 para usar funciones inline.

10. Cree un enumerado llamado Hue que contenga red, blue y yellow. Ahora cree una clase llamada `Color` que contenga un atributo de tipo Hue y un constructor que dé valor al Hue con su argumento. Añada métodos de acceso al Hue `get()` y `set()`. Haga inline todos los métodos.

11. Modifique el ejercicio 10 para usar el enfoque «accesor» y «mutador».

12. Modifique `Cpptime.cpp` de modo que mida el tiempo desde que comienza el programa hasta que el usuario pulsa la tecla «Intro» o «Retorno».

13. Cree una clase con dos métodos inline, el primero que está definido en la clase llama al segundo, sin necesitar una declaración adelantada. Escriba un `main()` que cree un objeto de esa clase y llame al primer método.

14. Cree una clase `A` con un constructor por defecto inline que se anuncie a sí mismo. Ahora cree una nueva clase `B` y ponga un objeto de `A` como miembro de `B`, y dele a `B` un constructor inline. Cree un array de objetos B y vea qué sucede.

15. Cree una gran cantidad de objetos del ejercicio anterior, y use la clase `Time` para medir las diferencias entre los contructores inline y los no-inline. (Si tiene un perfilador, intente usarlo también).

16. Escriba un programa que tome una cadena por línea de comandos. Escriba un bucle `for` que elimine un carácter de la cadena en cada pasada, y use la macro `DEGUB()` de este capítulo para imprimir la cadena cada vez.

17. Corrija la macro `TRACE()` tal como se explica en el capítulo, y pruebe que funciona correctamente.

18. Modifique la macro `FIELD()` para que también incluya un índice numérico. Cree una clase cuyos miembros están compuestos de llamadas a la macro `FIELD()`. Añada un método que le permita buscar en un campo usando el índice. Escriba un `main()` para probar la clase.

19. Modifique la macro `FIELD()` para que automáticamente genere funciones de acceso para cada campo (`data` debería no obstante ser privado). Cree una clase cuyos miembros estén compuestos de llamadas a la macro `FIELD()`. Escriba un `main()` para probar la clase.

20. Escriba un programa que tome dos argumentos de línea de comandos: el primero es un entero y el segundo es un nombre de fichero. Use `requiere.h` para asegurar que tiene el número correcto de argumentos, que el entero está entre 5 y 10, y que el fichero se puede abrir satisfactoriamente.

21. Escriba un programa que use la macro `IFOPEN()` para abrir un fichero como un flujo de entrada. Fíjese en la creación un objeto `ifstream` y su alcance.

22. (Desafío) Averigüe cómo conseguir que su compilador genere código ensamblador. Cree un fichero que contenga una función muy pequeña y un `main()`. Genere el código ensamblador cuando la función es inline y cuando no lo es, y demuestre que la versión inline no tiene la sobrecarga por la llamada.

10: Control de nombres

La creación de nombres es una actividad fundamental en la programación, y cuando un proyecto empieza a crecer, el número de nombres puede llegar a ser inmanejable con facilidad.

C++ `permite` gran control sobre la creación y visibilidad de nombres, el lugar donde se almacenan y el enlazado de nombres. La palabra clave `static` estaba sobrecargada en C incluso antes de que la mayoría de la gente supiera que significaba el término «sobrecargar». C++ ha añadido además otro significado. El concepto subyacente bajo todos los usos de `static` parece ser «algo que mantiene su posición» (como la electricidad estática), sea manteniendo un ubicación física en la memoria o su visibilidad en un fichero.

En este capítulo aprenderá cómo `static` controla el almacenamiento y la visibilidad, así como una forma mejorada para controlar los nombres mediante el uso de la palabra clave de C++ `namespace`. También descubrirá como utilizar funciones que fueron escritas y compiladas en C.

10.1. Los elementos estáticos de C

Tanto en C como en C++ la palabra clave `static` tiene dos significados básicos que, desafortunadamente, a menudo se confunden:

- Almacenado una sola vez en una dirección de memoria fija. Es decir, el objeto se crea en una área de datos estática especial en lugar de en la pila cada vez que se llama a una función. Éste es el concepto de almacenamiento estático.
- Local a una unidad de traducción particular (y también local para el ámbito de una clase en C++, tal como se verá después). Aquí, `static` controla la visibilidad de un nombre, de forma que dicho nombre no puede ser visto fuera del la unidad de traducción o la clase. Esto también corresponde al concepto de enlazado, que determina qué nombres verá el enlazador.

En esta sección se van a analizar los significados anteriores de `static` tal y como se heredaron de C.

10.1.1. Variables estáticas dentro de funciones

Cuando se crea una variable local dentro de una función, el compilador reserva espacio para esa variable cada vez que se llama a la función moviendo hacia abajo el puntero de pila tanto como sea preciso. Si existe un inicializador para la variable, la inicialización se realiza cada vez que se pasa por ese punto de la secuencia.

No obstante, a veces es deseable retener un valor entre llamadas a función. Esto se puede lograr creando una variable global, pero entonces esta variable no estará únicamente bajo control de la función. C y C++ permiten crear un objeto `static` dentro de una función. El almacenamiento de este objeto no se lleva a cabo en la pila sino en el área de datos estáticos del programa. Dicho objeto sólo se inicializa una vez, la primera vez que se llama a la función, y retiene su valor entre diferentes invocaciones. Por ejemplo, la siguiente función devuelve el siguiente carácter del vector cada vez que se la llama:

```
//: C10:StaticVariablesInfunctions.cpp
#include "../require.h"
#include <iostream>
using namespace std;

char oneChar(const char* charArray = 0) {
  static const char* s;
  if(charArray) {
    s = charArray;
    return *s;
  }
  else
    require(s, "un-initialized s");
  if(*s == '\0')
    return 0;
  return *s++;
}

char* a = "abcdefghijklmnopqrstuvwxyz";

int main() {
  // oneChar(); // require() fails
  oneChar(a); // Initializes s to a
  char c;
  while((c = oneChar()) != 0)
    cout << c << endl;
} ///:~
```

La variable static char* s mantiene su valor entre llamadas a `oneChar()` porque no está almacenada en el segmento de pila de la función, sino que está en el área de almacenamiento estático del programa. Cuando se llama a `oneChar()` con char* como argumento, `s` se asigna a ese argumento de forma que se devuelve el primer carácter del array. Cada llamada posterior a `oneChar()` *sin* argumentos devuelve el valor por defecto cero para `charArray`, que indica a la función que todavía se están extrayendo caracteres del valor previo de `s`. La función continuará devolviendo caracteres hasta que alcance el valor de final del vector, momento en el que para de incrementar el puntero evitando que éste sobrepase la última posición del vector.

Pero ¿qué pasa si se llama a `oneChar()` sin argumentos y sin haber inicializado previamente el valor de `s`? En la definición para `s`, se podía haber utilizado la inicialización,

```
      static char* s = 0;
```

pero si no se incluye un valor inicial para una variable estática de un tipo defini-

do, el compilador garantiza que la variable se inicializará a cero (convertido al tipo adecuado) al comenzar el programa. Así pues, en `oneChar()`, la primera vez que se llama a la función, `s` vale cero. En este caso, se cumplirá la condición `if(!s)`.

La inicialización anterior para `s` es muy simple, pero la inicialización para objetos estáticos (como la de cualquier otro objeto) puede ser una expresión arbitraria, que involucre constantes, variables o funciones previamente declaradas.

Fíjese que la función de arriba es muy vulnerable a problemas de concurrencia. Siempre que diseñe funciones que contengan variables estáticas, deberá tener en mente este tipo de problemas.

Objetos estáticos dentro de funciones

Las reglas son las mismas para objetos estáticos de tipos definidos por el usuario, añadiendo el hecho que el objeto requiere ser inicializado. Sin embargo, la asignación del valor cero sólo tiene sentido para tipos predefinidos. Los tipos definidos por el usuario deben ser inicializados llamando a sus respectivos constructores. Por tanto, si no especifica argumentos en los constructores cuando defina un objeto estático, la clase deberá tener un constructor por defecto. Por ejemplo:

```
//: C10:StaticObjectsInFunctions.cpp
#include <iostream>
using namespace std;

class X {
  int i;
public:
  X(int ii = 0) : i(ii) {} // Default
  ~X() { cout << "X::~X()" << endl; }
};

void f() {
  static X x1(47);
  static X x2; // Default constructor required
}

int main() {
  f();
} ///:~
```

Los objetos estáticos de tipo X dentro de `f()` pueden ser inicializados tanto con la lista de argumentos del constructor como con el constructor por defecto. Esta construcción ocurre únicamente la primera vez que el control llega a la definición.

Destructores de objetos estáticos

Los destructores para objetos estáticos (es decir, cualquier objeto con almacenamiento estático, no sólo objetos estáticos locales como en el ejemplo anterior) son invocados cuando `main()` finaliza o cuando la función de librería estándar de C `exit()` se llama explícitamente. En la mayoría de implementaciones, `main()` simplemente llama a `exit()` cuando termina. Esto significa que puede ser peligroso llamar a `exit()` dentro de un destructor porque podría producirse una invocación recursiva infinita. Los destructores de objetos estáticos no se invocan si se sale del programa utilizando la función de librería estándar de C `abort()`.

Es posible especificar acciones que se lleven a cabo tras finalizar la ejecución de `main()` (o llamando a `exit()`) utilizando la función de librería estándar de C `atexit()`. En este caso, las funciones registradas en `atexit()` serán invocadas antes de los destructores para cualquier objeto construido antes de abandonar `main()` (o de llamar a `exit()`).

Como la destrucción ordinaria, la destrucción de objetos estáticos se lleva a cabo en orden inverso al de la inicialización. Hay que tener en cuenta que sólo los objetos que han sido construidos serán destruidos. Afortunadamente, las herramientas de desarrollo de C++ mantienen un registro del orden de inicialización y de los objetos que han sido construidos. Los objetos globales siempre se construyen antes de entrar en `main()` y se destruyen una vez se sale, pero si existe una función que contiene un objeto local estático a la que nunca se llama, el constructor de dicho objeto nunca fue ejecutado y, por tanto, nunca se invocará su destructor. Por ejemplo:

```
//: C10:StaticDestructors.cpp
// Static object destructors
#include <fstream>
using namespace std;
ofstream out("statdest.out"); // Trace file

class Obj {
  char c; // Identifier
public:
  Obj(char cc) : c(cc) {
    out << "Obj::Obj() for " << c << endl;
  }
  ~Obj() {
    out << "Obj::~Obj() for " << c << endl;
  }
};

Obj a('a'); // Global (static storage)
// Constructor & destructor always called

void f() {
  static Obj b('b');
}

void g() {
  static Obj c('c');
}

int main() {
  out << "inside main()" << endl;
  f(); // Calls static constructor for b
  // g() not called
  out << "leaving main()" << endl;
} ///:~
```

En `Obj`, `char c` actúa como un identificador de forma que el constructor y el destructor pueden imprimir la información acerca del objeto sobre el que actúan. `Obj a` es un objeto global y por tanto su constructor siempre se llama antes de que el control pase a `main()`, pero el constructor para `static Obj b` dentro de `f()`, y el de `static Obj c` dentro de `g()` sólo serán invocados si se llama a esas funciones.

Para mostrar qué constructores y qué destructores serán llamados, sólo se invoca a `f()`. La salida del programa será la siguiente:

```
Obj::Obj() for a
inside main()
Obj::Obj() for b
leaving main()
Obj::~Obj() for b
Obj::~Obj() for a
```

El constructor para `a` se invoca antes de entrar en `main()` y el constructor de `b` se invoca sólo porque existe una llamada a `f()`. Cuando se sale de `main()`, se invoca a los destructores de los objetos que han sido construidos en orden inverso al de su construcción. Esto significa que si llama a `g()`, el orden en el que los destructores para `b` y `c` son invocados depende de si se llamó primero a `f()` o a `g()`.

Nótese que el objeto `out` de tipo `ofstream`, utilizado en la gestión de ficheros, también es un objeto estático (puesto que está definido fuera de cualquier función, reside en el área de almacenamiento estático). Es importante remarcar que su definición (a diferencia de una declaración tipo `extern`) aparece al principio del fichero, antes de cualquier posible uso de `out`. De lo contrario estaríamos utilizando un objeto antes de que estuviese adecuadamente inicializado.

En C++, el constructor de un objeto estático global se invoca antes de entrar en `main()`, de forma que ya dispone de una forma simple y portable de ejecutar código antes de entrar en `main()`, así como ejecutar código después de salir de `main()`. En C, eso siempre implicaba revolver el código ensamblador de arranque del compilador utilizado.

10.1.2. Control del enlazado

Generalmente, cualquier nombre dentro del ámbito del fichero (es decir, no incluido dentro de una clase o de una función) es visible para todas las unidades de traducción del programa. Esto suele llamarse enlazado externo porque durante el enlazado ese nombre es visible desde cualquier sitio, desde el exterior de esa unidad de traducción. Las variables globales y las funciones ordinarias tienen enlazado externo.

Hay veces en las que conviene limitar la visibilidad de un nombre. Puede que desee tener una variable con visibilidad a nivel de fichero de forma que todas las funciones de ese fichero puedan utilizarla, pero quizá no desee que funciones externas a ese fichero tengan acceso a esa variable, o que de forma inadvertida, cause solapes de nombres con identificadores externos a ese fichero.

Un objeto o nombre de función, con visibilidad dentro del fichero en que se encuentra, que es explícitamente declarado como `static` es local a su unidad de traducción (en términos de este libro, el fichero `cpp` donde se lleva a cabo la declaración). Este nombre tiene *enlace interno*. Esto significa que puede usar el mismo nombre en otras unidades de traducción sin confusión entre ellos.

Una ventaja del enlace interno es que el nombre puede situarse en un fichero de cabecera sin tener que preocuparse de si habrá o no un choque de nombres durante el enlazado. Los nombres que aparecen usualmente en los archivos de cabecera, como definiciones `const` y funciones `inline`, tienen por defecto enlazado interno. (De todas formas, `const` tiene por defecto enlazado interno sólo en C++; en C tiene enlazado externo). Nótese que el enlazado se refiere sólo a elementos que tienen direcciones en tiempo de enlazado / carga. Por tanto, las declaraciones de clases y

de variables locales no tienen enlazado.

Confusión

He aquí un ejemplo de cómo los dos significados de `static` pueden confundirse. Todos los objetos globales tienen implícitamente almacenamiento de tipo estático, o sea que si usted dice (en ámbito de fichero)

```
int a = 0;
```

el almacenamiento para `a` se llevará a cabo en el área para datos estáticos del programa y la inicialización para `a` sólo se realizará una vez, antes de entrar en `main()`. Además, la visibilidad de `a` es global para todas las unidades de traducción. En términos de visibilidad, lo opuesto a `static` (visible tan sólo en su :unidad de traducción) es `extern` que establece explícitamente que la visibilidad del nombre se extienda a todas las unidades de traducción. Es decir, la definición de arriba equivale a

```
extern int a = 0;
```

Pero si utilizase

```
static int a = 0;
```

todo lo que habría hecho es cambiar la visibilidad, de forma que `a` tiene enlace interno. El tipo de almacenamiento no se altera, el objeto reside en el área de datos estática aunque en este caso su visibilidad es `static` y en el otro es `extern`.

Cuando pasamos a hablar de variables locales, `static` deja de alterar la visibilidad y pasa a alterar el tipo de almacenamiento.

Si declara lo que parece ser una variable local como `extern`, significa que el almacenamiento existe en alguna otra parte (y por tanto la variable realmente es global a la función). Por ejemplo:

```
//: C10:LocalExtern.cpp
//{L} LocalExtern2
#include <iostream>

int main() {
  extern int i;
  std::cout << i;
} ///:~
```

Para nombres de funciones (sin tener en cuenta las funciones miembro), `static` y `extern` sólo pueden alterar la visibilidad, de forma que si escribe

```
extern void f();
```

es lo mismo que la menos adornada declaración

```
void f();
```

y si utiliza

```
static void f();
```

significa que `f()` es visible sólo para la unidad de traducción, (esto suele llamarse *estático a fichero* (*file static*).

10.1.3. Otros especificadores para almacenamiento de clases

El uso de `static` y `extern` está muy extendido. Existen otros dos especificadores de tipo de almacenamiento bastante menos conocidos. El especificador `auto` no se utiliza prácticamente nunca porque le dice al compilador que esa es una variable local. `auto` es la abreviatura de automático«» y se refiere a la forma en la que el compilador reserva espacio automáticamente para la variable. El compilador siempre puede determinar ese hecho por el contexto en que la variable se define por lo que `auto` es redundante.

El especificador `register` aplicado a una variable indica que es una variable local (`auto`), junto con la pista para el compilador de que esa variable en concreto va a ser ampliamente utilizada por lo que debería ser almacenada en un registro si es posible. Por tanto, es una ayuda para la optimización. Diferentes compiladores responden de diferente manera ante dicha pista; incluso tienen la opción de ignorarla. Si toma la dirección de la variable, el especificador `register` va a ser ignorado casi con total seguridad. Se recomienda evitar el uso de `register` porque, generalmente, el compilador suele realizar las labores de optimización mejor que el usuario.

10.2. Espacios de nombres

Pese a que los nombres pueden estar anidados dentro de clases, los nombres de funciones globales, variables globales y clases se encuentran incluidos dentro de un único espacio de nombres. La palabra reservada `static` le da control sobre éste permitiéndole darle tanto a variables como a funciones enlazado interno (es decir convirtiéndolas en estáticas al fichero). Pero en un proyecto grande, la falta de control sobre el espacio de nombres global puede causar problemas. Con el fin de solventar esos problemas para clases, los programadores suelen crear nombres largos y complicados que tienen baja probabilidad de crear conflictos pero que suponen hartarse a teclear para escribirlos. (Para simplificar este problema se suele utilizar `typedef`). Pese a que el lenguaje la soporta, no es una solución elegante.

En lugar de eso puede subdividir el espacio de nombres global en varias partes más manejables utilizando la característica `namespace` de C++. La palabra reservada `namespace`, de forma similar a `class`, `struct`, `enum` y `union`, sitúa los nombres de sus miembros en un espacio diferente. Mientras que las demás palabras reservadas tienen propósitos adicionales, la única función de `namespace` es la de crear un nuevo espacio de nombres.

10.2.1. Crear un espacio de nombres

La creación de un espacio de nombres es muy similar a la creación de una clase:

```
//: C10:MyLib.cpp
namespace MyLib {
  // Declarations
}
int main() {} ///:~
```

Ese código crea un nuevo espacio de nombres que contiene las declaraciones incluidas entre las llaves. De todas formas, existen diferencias significativas entre `class`, `struct`, `enum` y `union`:

- Una definición con namespace solamente puede aparecer en un rango global de visibilidad o anidado dentro de otro namespace.
- No es necesario un punto y coma tras la llave de cierre para finalizar la definición de namespace.
- Una definición namespace puede ser "continuada" en múltiples archivos de cabecera utilizando una sintaxis que, para una clase, parecería ser la de una redefinición:

```
//: C10:Header1.h
#ifndef HEADER1_H
#define HEADER1_H
namespace MyLib {
  extern int x;
  void f();
  // ...
}

#endif // HEADER1_H ///:~
```

El posible crear alias de un `namespace` de forma que no hace falta que teclee un enrevesado nombre creado por algún frabricante de librerías:

```
//: C10:BobsSuperDuperLibrary.cpp
namespace BobsSuperDuperLibrary {
  class Widget { /* ... */ };
  class Poppit { /* ... */ };
  // ...
}
// Too much to type! I'll alias it:
namespace Bob = BobsSuperDuperLibrary;
int main() {} ///:~
```

No puede crear una instancia de un `namespace` tal y como puede hacer con una clase.

Espacios de nombres sin nombre

Cada unidad de traducción contiene un espacio de nombres sin nombre al que puede referirse escribiendo «`namespace`» sin ningún identificador.

Los nombres en este espacio están disponibles automáticamente en esa unidad de traducción sin cualificación. Se garantiza que un espacio sin nombre es único para cada unidad de traducción. Si usted asigna nombres locales en un espacio de nombres no necesitará darles enlazado interno con `static`.

En C++ es preferible utilizar espacios de nombres sin nombre que *estáticos a fichero*.

Amigas

Es posible añadir una declaración tipo `friend` dentro de un espacio de nombres incluyéndola dentro de una clase:

```
//: C10:FriendInjection.cpp
namespace Me {
  class Us {
    //...
    friend void you();
  };
}
int main() {} ///:~
```

Ahora la función `you()` es un miembro del espacio de nombres `Me`.

Si introduce una declaración tipo `friend` en una clase dentro del espacio de nombres global, dicha declaración se inyecta globalmente.

10.2.2. Cómo usar un espacio de nombres

Puede referirse a un nombre dentro de un espacio de nombres de tres maneras diferentes: especificando el nombre utilizando el operador de resolución de ámbito, con una directiva `using` que introduzca todos los nombres en el espacio de nombres o mediante una declaración `using` para introducir nombres de uno en uno.

Resolución del ámbito

Cualquier nombre en un espacio de nombres puede ser explícitamente especificado utilizando el operador de resolución de ámbito de la misma forma que puede referirse a los nombres dentro de una clase:

```
//: C10:ScopeResolution.cpp
namespace X {
  class Y {
    static int i;
  public:
    void f();
  };
  class Z;
  void func();
}
int X::Y::i = 9;
class X::Z {
  int u, v, w;
public:
  Z(int i);
```

```
  int g();
};
X::Z::Z(int i) { u = v = w = i; }
int X::Z::g() { return u = v = w = 0; }
void X::func() {
  X::Z a(1);
  a.g();
}
int main(){} ///:~
```

Nótese que la definición `X::Y::i` puede referirse también a un atributo de la clase `Y` anidada dentro de la clase `X` en lugar del espacio de nombres `X`.

La directiva `using`

Puesto que teclear toda la especificación para un identificador en un espacio de nombres puede resultar rápidamente tedioso, la palabra clave `using` le permite importar un espacio de nombres entero de una vez. Cuando se utiliza en conjunción con la palabra clave `namespace`, se dice que utilizamos una directiva `using`. Las directivas `using` hacen que los nombres actúen como si perteneciesen al ámbito del espacio de nombres que les incluye más cercano por lo que puede utilizar convenientemente los nombres sin explicitar completamente su especificación. Considere el siguiente espacio de nombres:

```
//: C10:NamespaceInt.h
#ifndef NAMESPACEINT_H
#define NAMESPACEINT_H
namespace Int {
  enum sign { positive, negative };
  class Integer {
    int i;
    sign s;
  public:
    Integer(int ii = 0)
      : i(ii),
        s(i >= 0 ? positive : negative)
    {}
    sign getSign() const { return s; }
    void setSign(sign sgn) { s = sgn; }
    // ...
  };
}
#endif // NAMESPACEINT_H ///:~
```

Un uso de las directivas `using` es incluir todos los nombres en `Int` dentro de otro espacio de nombres, dejando aquellos nombres anidados dentro del espacio de nombres

```
//: C10:NamespaceMath.h
#ifndef NAMESPACEMATH_H
#define NAMESPACEMATH_H
#include "NamespaceInt.h"
namespace Math {
  using namespace Int;
```

```
  Integer a, b;
  Integer divide(Integer, Integer);
  // ...
}
#endif // NAMESPACEMATH_H ///:~
```

También puede declarar todos los nombres en `Int` dentro de la función pero dejando esos nombres anidados dentro de la función:

```
//: C10:Arithmetic.cpp
#include "NamespaceInt.h"
void arithmetic() {
  using namespace Int;
  Integer x;
  x.setSign(positive);
}
int main(){} ///:~
```

Sin la directiva `using`, todos los nombres en el espacio de nombres requerirían estar completamente explicitados.

Hay un aspecto de la directiva `using` que podría parecer poco intuitivo al principio. La visibilidad de los nombres introducidos con una directiva `using` es el rango en el que se crea la directiva. Pero ¡puede hacer caso omiso de los nombres definidos en la directiva `using` como si estos hubiesen sido declarados globalmente para ese ámbito!

```
//: C10:NamespaceOverriding1.cpp
#include "NamespaceMath.h"
int main() {
  using namespace Math;
  Integer a; // Hides Math::a;
  a.setSign(negative);
  // Now scope resolution is necessary
  // to select Math::a :
  Math::a.setSign(positive);
} ///:~
```

Suponga que tiene un segundo espacio de nombres que contiene algunos nombres en `namespace Math`:

```
//: C10:NamespaceOverriding2.h
#ifndef NAMESPACEOVERRIDING2_H
#define NAMESPACEOVERRIDING2_H
#include "NamespaceInt.h"
namespace Calculation {
  using namespace Int;
  Integer divide(Integer, Integer);
  // ...
}
#endif // NAMESPACEOVERRIDING2_H ///:~
```

Dado que este espacio de nombres también se introduce con una directiva `using`, existe la posibilidad de tener una colisión. De todos modos, la ambigüedad aparece en el momento de utilizar el nombre, no en la directiva `using`:

```
//: C10:OverridingAmbiguity.cpp
#include "NamespaceMath.h"
#include "NamespaceOverriding2.h"
void s() {
  using namespace Math;
  using namespace Calculation;
  // Everything's ok until:
  //! divide(1, 2); // Ambiguity
}
int main() {} ///:~
```

Por tanto, es posible escribir directivas `using` para introducir un número de espacios de nombre con nombres conflictivos sin producir ninguna ambigüedad.

La declaración `using`

Puede inyectar nombres de uno en uno en el ámbito actual utilizando una declaración `using`. A diferencia de la directiva `using`, que trata los nombres como si hubiesen sido declarados globalmente para ese ámbito, una declaración `using` es una declaración dentro del ámbito actual. Eso significa que puede sobrescribir nombres de una directiva `using`:

```
//: C10:UsingDeclaration.h
#ifndef USINGDECLARATION_H
#define USINGDECLARATION_H
namespace U {
  inline void f() {}
  inline void g() {}
}
namespace V {
  inline void f() {}
  inline void g() {}
}
#endif // USINGDECLARATION_H ///:~
```

La declaración `using` simplemente da el nombre completamente especificado del identificador pero no da información de tipo. Eso significa que si el espacio de nombres contiene un grupo de funciones sobrecargadas con el mismo nombre, la declaración `using` declara todas las funciones pertenecientes al grupo sobrecargado.

Es posible poner una declaración `using` en cualquier sitio donde podría ponerse una declaración normal. Una declaración `using` funciona de la misma forma que cualquier declaración normal salvo por un aspecto: puesto que no se le da ninguna lista de argumentos, una declaración `using` puede provocar la sobrecarga de una función con los mismos tipos de argumentos (cosa que no está permitida por el procedimiento de sobrecarga normal). De todas formas, esta ambigüedad no se muestra hasta el momento de uso, no apareciendo en el instante de declaración.

Una declaración `using` puede también aparecer dentro de un espacio de nombres y tiene el mismo efecto que en cualquier otro lugar (ese nombre se declara dentro

del espacio):

```
//: C10:UsingDeclaration2.cpp
#include "UsingDeclaration.h"
namespace Q {
  using U::f;
  using V::g;
  // ...
}
void m() {
  using namespace Q;
  f(); // Calls U::f();
  g(); // Calls V::g();
}
int main() {} ///:~
```

Una declaración `using` es un alias. Le permite declarar la misma función en espacios de nombres diferentes. Si acaba redeclarando la misma función importando diferentes espacios de nombres no hay problema, no habrá ambigüedades o duplicados.

10.2.3. El uso de los espacios de nombres

Algunas de las reglas de arriba pueden parecer un poco desalentadoras al principio, especialmente si tiene la impresión que las utilizará constantemente. No obstante, en general es posible salir airoso con el uso de espacios de nombres fácilmente siempre y cuando comprenda como funcionan. La clave a recordar es que cuando introduce una directiva `using` global (vía "using namespace" fuera de cualquier rango) usted ha abierto el espacio de nombres para ese archivo. Esto suele estar bien para un archivo de implementación (un archivo "cpp") porque la directiva `using` sólo afecta hasta el final de la compilación de dicho archivo. Es decir, no afecta a ningún otro archivo, de forma que puede ajustar el control de los espacios de nombres archivo por archivo. Por ejemplo, si usted descubre un cruce de nombres debido a que hay demasiadas directivas `using` en un archivo de implementación particular, es una cuestión simple cambiar dicho archivo para que use calificaciones explícitas o declaraciones `using` para eliminar el cruce sin tener que modificar ningún otro archivo de implementación.

Los ficheros de cabecera ya son otra historia. Prácticamente nunca querrá introducir una directiva `using` global en un fichero de cabecera, puesto que eso significaría que cualquier otro archivo que incluyese la cabecera también tendría el espacio de nombres desplegado (y un fichero de cabecera puede incluir otros ficheros de cabecera).

Por tanto, en los ficheros de cabecera debería utilizar o bien cualificación explícita o bien directivas `using` de ámbito y declaraciones `using`. Éste es el método que encontrará en este libro. Siguiendo esta metodología no «contaminará» el espacio de nombres global, que implicaría volver al mundo pre-espacios de nombres de C++.

10.3. Miembros estáticos en C++

A veces se necesita un único espacio de almacenamiento para utilizado por todos los objetos de una clase. En C, podría usar una variable global pero eso no es muy

seguro. Los datos globales pueden ser modificados por todo el mundo y su nombre puede chocar con otros idénticos si es un proyecto grande. Sería ideal si los datos pudiesen almacenarse como si fuesen globales pero ocultos dentro de una clase y claramente asociados con esa clase.

Esto es posible usando atributos `static`. Existe una única porción de espacio para los atributos `static`, independientemente del número de objetos de dicha clase que se hayan creado. Todos los objetos comparten el mismo espacio de almacenamiento `static` para ese atributo, constituyendo una forma de «comunicarse» entre ellos. Pero los datos `static` pertenecen a la clase; su nombre está restringido al interior de la clase y puede ser `public`, `private` o `protected`.

10.3.1. Definición del almacenamiento para atributos estáticos

Puesto que los datos `static` tienen una única porción de memoria donde almacenarse, independientemente del número de objetos creados, esa porción debe ser definida en un único sitio. El compilador no reservará espacio de almacenamiento por usted. El enlazador reportará un error si un atributo miembro es declarado pero no definido.

La definición debe realizarse fuera de la clase (no se permite el uso de la sentencia `inline`), y sólo está permitida una definición. Es por ello que habitualmente se incluye en el fichero de implementación de la clase. La sintaxis suele traer problemas, pero en realidad es bastante lógica. Por ejemplo, si crea un atributo estático dentro de una clase de la siguiente forma:

```
class A {
  static int i;
public:
  //...
};
```

Deberá definir el almacenamiento para ese atributo estático en el fichero de definición de la siguiente manera:

```
  int A::i = 1;
```

Si quisiera definir una variable global ordinaria, debería utilizar

```
  int i = 1;
```

pero aquí, el operador de resolución de ámbito y el nombre de la clase se utilizan para especificar `A::i`.

Algunas personas tienen problemas con la idea que `A::i` es `private`, y pese a ello parece haber algo que lo está manipulando abiertamente. ¿No rompe esto el mecanismo de protección? Ésta es una práctica completamente segura por dos razones. Primera, el único sitio donde esta inicialización es legal es en la definición. Efectivamente, si el dato `static` fuese un objeto con un constructor, habría llamado al constructor en lugar de utilizar el operador =. Segundo, una vez se ha realizado la definición, el usuario final no puede hacer una segunda definición puesto que el enlazador indicaría un error. Y el creador de la clase está forzado a crear la definición

o el código no enlazaría en las pruebas. Esto asegura que la definición sólo sucede una vez y que es el creador de la clase quien la lleva a cabo.

La expresión completa de inicialización para un atributo estático se realiza en el ámbito de la clase. Por ejemplo,

```
//: C10:Statinit.cpp
// Scope of static initializer
#include <iostream>
using namespace std;

int x = 100;

class WithStatic {
  static int x;
  static int y;
public:
  void print() const {
    cout << "WithStatic::x = " << x << endl;
    cout << "WithStatic::y = " << y << endl;
  }
};

int WithStatic::x = 1;
int WithStatic::y = x + 1;
// WithStatic::x NOT ::x

int main() {
  WithStatic ws;
  ws.print();
} ///:~
```

Aquí el calificador `WithStatic::` extiende el ámbito de `WithStatic` a la definición completa.

Inicialización de vectores estáticos

El capítulo 8 introdujo una variable `static const` que le permite definir un valor constante dentro del cuerpo de una clase. También es posible crear arrays de objetos estáticos, ya sean constantes o no constantes. La sintaxis es razonablemente consistente:

```
//: C10:StaticArray.cpp
// Initializing static arrays in classes
class Values {
  // static consts are initialized in-place:
  static const int scSize = 100;
  static const long scLong = 100;
  // Automatic counting works with static arrays.
  // Arrays, Non-integral and non-const statics
  // must be initialized externally:
  static const int scInts[];
  static const long scLongs[];
  static const float scTable[];
  static const char scLetters[];
  static int size;
```

```
  static const float scFloat;
  static float table[];
  static char letters[];
};

int Values::size = 100;
const float Values::scFloat = 1.1;

const int Values::scInts[] = {
  99, 47, 33, 11, 7
};

const long Values::scLongs[] = {
  99, 47, 33, 11, 7
};

const float Values::scTable[] = {
  1.1, 2.2, 3.3, 4.4
};

const char Values::scLetters[] = {
  'a', 'b', 'c', 'd', 'e',
  'f', 'g', 'h', 'i', 'j'
};

float Values::table[4] = {
  1.1, 2.2, 3.3, 4.4
};

char Values::letters[10] = {
  'a', 'b', 'c', 'd', 'e',
  'f', 'g', 'h', 'i', 'j'
};

int main() { Values v; } ///:~
```

Usando `static const` de tipos enteros puede realizar las definiciones dentro de la clase, pero para cualquier otro tipo (incluyendo listas de enteros, incluso si estos son `static const`) debe realizar una única definición externa para el atributo. Estas definiciones tienen enlazado interno, por lo que pueden incluirse en ficheros de cabecera. La sintaxis para inicializar listas estáticas es la misma que para cualquier agregado, incluyendo el conteo automático`automatic counting`.

También puede crear objetos `static const` de tipos de clase y listas de dichos objetos. De todas formas, no puede inicializarlos utilizando la sintaxis tipo «inline» permitida para `static const` de tipos enteros básicos:

```
//: C10:StaticObjectArrays.cpp
// Static arrays of class objects
class X {
  int i;
public:
  X(int ii) : i(ii) {}
};

class Stat {
```

```
  // This doesn't work, although
  // you might want it to:
//!  static const X x(100);
  // Both const and non-const static class
  // objects must be initialized externally:
  static X x2;
  static X xTable2[];
  static const X x3;
  static const X xTable3[];
};

X Stat::x2(100);

X Stat::xTable2[] = {
  X(1), X(2), X(3), X(4)
};

const X Stat::x3(100);

const X Stat::xTable3[] = {
  X(1), X(2), X(3), X(4)
};

int main() { Stat v; } ///:~
```

La inicialización de listas estáticas de objetos tanto constantes como no constantes debe realizarse de la misma manera, siguiendo la típica sintaxis de definición estática.

10.3.2. Clases anidadas y locales

Puede colocar fácilmente atributos estáticos en clases que están anidadas dentro de otras clases. La definición de tales miembros es intuitiva y obvia (tan sólo utiliza otro nivel de resolución de ámbito). No obstante, no puede tener atributos estáticos dentro de clases locales (una clase local es una clase definida dentro de una función). Por tanto,

```
//: C10:Local.cpp
// Static members & local classes
#include <iostream>
using namespace std;

// Nested class CAN have static data members:
class Outer {
  class Inner {
    static int i; // OK
  };
};

int Outer::Inner::i = 47;

// Local class cannot have static data members:
void f() {
  class Local {
  public:
```

```
//! static int i;  // Error
    // (How would you define i?)
  } x;
}

int main() { Outer x; f(); } ///:~
```

Ya puede ver el problema con atributos estáticos en clases locales: ¿Cómo describirá el dato miembro en el ámbito del fichero para poder definirlo? En la práctica, el uso de clases locales es muy poco común.

10.3.3. Métodos estáticos

También puede crear métodos estáticos que, como los atributos estáticos, trabajan para la clase como un todo en lugar de para un objeto particular de la clase. En lugar de hacer una función global que viva en y «contamine» el espacio de nombres global o local, puede incluir el método dentro de la clase. Cuando crea un método estático, está expresando una asociación con una clase particular.

Puede llamar a un miembro estáticos de la forma habitual, con el punto o la flecha, en asociación con un objeto. De todas formas, es más típico llamar a los métodos estáticos en si mismos, sin especificar ningún objeto, utilizando el operador de resolución de ámbito, como en el siguiente ejemplo:

```
//: C10:SimpleStaticMemberFunction.cpp
class X {
public:
  static void f(){};
};

int main() {
  X::f();
} ///:~
```

Cuando vea métodos estáticos en una clase, recuerde que el diseñador pretendía que esa función estuviese conceptualmente asociada a la clase como un todo.

Un método estático no puede acceder a los atributos ordinarios, sólo a los estáticos. Sólo puede llamar a otros métodos estáticos. Normalmente, la dirección del objeto actual (`this`) se pasa de forma encubierta cuando se llama a cualquier método, pero un miembro `static` no tiene `this`, que es la razón por la cual no puede acceder a los miembros ordinarios. Por tanto, se obtiene el ligero incremento de velocidad proporcionado por una función global debido a que un método estático no implica la carga extra de tener que pasar `this`. Al mismo tiempo, obtiene los beneficios de tener la función dentro de la clase.

Para atributos, `static` indica que sólo existe un espacio de memoria por atributo para todos los objetos de la clase. Esto establece que el uso de `static` para definir objetos dentro de una función significa que sólo se utiliza una copia de una variable local para todas las llamadas a esa función.

Aquí aparece un ejemplo mostrando atributos y métodos estáticos utilizados conjuntamente:

```
//: C10:StaticMemberFunctions.cpp
class X {
  int i;
  static int j;
public:
  X(int ii = 0) : i(ii) {
     // Non-static member function can access
     // static member function or data:
    j = i;
  }
  int val() const { return i; }
  static int incr() {
    //! i++; // Error: static member function
    // cannot access non-static member data
    return ++j;
  }
  static int f() {
    //! val(); // Error: static member function
    // cannot access non-static member function
    return incr(); // OK -- calls static
  }
};

int X::j = 0;

int main() {
  X x;
  X* xp = &x;
  x.f();
  xp->f();
  X::f(); // Only works with static members
} ///:~
```

Puesto que no tienen el puntero `this`, los métodos estáticos no pueden acceder a atributos no estáticos ni llamar a métodos no estáticos.

Note el lector que en `main()` un miembro estático puede seleccionarse utilizando la habitual sintaxis de punto o flecha, asociando la función con el objeto, pero también sin objeto (ya que un miembro estático está asociado con una clase, no con un objeto particular), utilizando el nombre de la clase y el operador de resolución de ámbito.

He aquí una interesante característica: Debido a la forma en la que se inicializan los objetos miembro estáticos, es posible poner un atributos estático de la misma clase dento de dicha clase. He aquí un ejemplo que tan solo permite la existencia de un único objeto de tipo `Egg` definiendo el constructor privado. Puede acceder a este objeto pero no puede crear ningún otro objeto tipo `Egg`:

```
//: C10:Singleton.cpp
// Static member of same type, ensures that
// only one object of this type exists.
// Also referred to as the "singleton" pattern.
#include <iostream>
using namespace std;

class Egg {
```

```
  static Egg e;
  int i;
  Egg(int ii) : i(ii) {}
  Egg(const Egg&); // Prevent copy-construction
public:
  static Egg* instance() { return &e; }
  int val() const { return i; }
};

Egg Egg::e(47);

int main() {
//!  Egg x(1); // Error -- can't create an Egg
  // You can access the single instance:
  cout << Egg::instance()->val() << endl;
} ///:~
```

La inicialización de `e` ocurre una vez se completa la declaración de la clase, por lo que el compilador tiene toda la información que necesita para reservar espacio y llamar al constructor.

Para impedir completamente la creación de cualquier otro objeto, se ha añadido algo más: un segundo constructor privado llamado *constructor de copia*. Llegados a este punto del libro, usted no puede saber porqué esto es necesario puesto que el constructor de copia no se estudiará hasta el siguiente capítulo. De todas formas, como un breve adelanto, si se propusiese retirar el constructor de copia definido en el ejemplo anterior, sería posible crear objetos Egg de la siguiente forma:

```
Egg e = *Egg::instance();
Egg e2(*Egg::instance());
```

Ambos utilizan el constructor de copia, por lo que para evitar esta posibilidad, se declara el constructor de copia como privado (no se requiere definición porque nunca va a ser llamado). Buena parte del siguiente capítulo es una discusión sobre el constructor de copia por lo que esto quedará más claro entonces.

10.4. Dependencia en la inicialización de variables estáticas

Dentro de una unidad de traducción específica, está garantizado que el orden de inicialización de los objetos estáticos será el mismo que el de aparición de sus definiciones en la unidad de traducción.

No obstante, no hay garantías sobre el orden en que se inicializan los objetos estáticos entre diferentes unidades de traducción, y el lenguaje no proporciona ninguna forma de averiguarlo. Esto puede producir problemas significativos. Como ejemplo de desastre posible (que provocará el cuelgue de sistemas operativos primitivos o la necesidad de matar el proceso en otros más sofisticados), si un archivo contiene:

```
//: C10:Out.cpp {O}
// First file
#include <fstream>
std::ofstream out("out.txt"); ///:~
```

y otro archivo utiliza el objeto `out` en uno de sus inicializadores

```
//: C10:Oof.cpp
// Second file
//{L} Out
#include <fstream>
extern std::ofstream out;
class Oof {
public:
  Oof() { out << "ouch"; }
} oof;
int main() {} ///:~
```

el programa puede funcionar, o puede que no. Si el entorno de programación construye el programa de forma que el primer archivo sea inicializado después del segundo, no habrá problemas. Pero si el segundo archivo se inicializa antes que el primero, el constructor para `Oof` se sustenta en la existencia de `out`, que todavía no ha sido construido, lo que causa el caos.

Este problema sólo ocurre con inicializadores de objetos estáticos que dependen el uno del otro. Los estáticos dentro de cada unidad de traducción son inicializados antes de la primera invocación a cualquier función de esa unidad, aunque puede que después de `main()`. No puede estar seguro del orden de inicialización de objetos estáticos si están en archivos diferentes.

Un ejemplo sutil puede encontrarse en ARM.[1] en un archivo que aparece en el rango global:

```
extern int y;
int x = y + 1;
```

y en un segundo archivo también en el ámbitoglobal:

```
extern int x;
int y = x + 1;
```

Para todos los objetos estáticos, el mecanismo de carga-enlazado garantiza una inicialización estática a cero antes de la inicialización dinámica especificada por el programador. En el ejemplo anterior, la inicialización a cero de la zona de memoria ocupada por el objeto `fstream out` no tiene especial relevancia, por lo que realmente no está definido hasta que se llama al constructor. Pese a ello, en el caso de los tipos predefinidos, la inicialización a cero sí tiene importancia, y si los archivos son inicializados en el orden mostrado arriba, y empieza estáticamente inicializada a cero, por lo que `x` se convierte en uno, e `y` es dinámicamente inicializada a dos. Pero si los archivos fuesen inicializados en orden opuesto, `x` sería estáticamente inicializada a cero, `y` dinámicamente inicializada a uno y después, `x` pasaría a valer dos.

Los programadores deben estar al tanto de esto porque puede darse el caso de crear un programa con dependencias de inicialización estáticas que funcionen en una

[1] Bjarne Stroustrup and Margaret Ellis, The Annotated C++ Reference Manual, Addison-Wesley, 1990, pp. 20-21.

plataforma determinada y, de golpe y misteriosamente, compilarlo en otro entorno y que deje de funcionar.

10.4.1. Qué hacer

Existen tres aproximaciones para tratar con este problema:

1. No hacerlo. Evitar las dependencias de inicialización estática es la mejor solución.
2. Si debe hacerlo, coloque las definiciones de objetos estáticos críticos en un único fichero, de forma que pueda controlar, de forma portable, su inicialización colocándolos en el orden correcto.
3. Si está convencido que es inevitable dispersar objetos estáticos entre unidades de traducción diferentes (como en el caso de una librería, donde no puede controlar el programa que la usa), hay dos técnicas de programación para solventar el problema.

Técnica uno

El pionero de esta técnica fue Jerry Schwarz mientras creaba la librería `iostream` (puesto que las definiciones para `cin`, `cout` y `cerr` son `static` y residen en archivos diferentes). Realmente es inferior a la segunda técnica pero ha pululado durante mucho tiempo por lo que puede encontrarse con código que la utilice; así pues, es importante que entienda como funciona.

Esta técnica requiere una clase adicional en su archivo de cabecera. Esta clase es la responsable de la inicialización dinámica de sus objetos estáticos de librería. He aquí un ejemplo simple:

```
//: C10:Initializer.h
// Static initialization technique
#ifndef INITIALIZER_H
#define INITIALIZER_H
#include <iostream>
extern int x; // Declarations, not definitions
extern int y;

class Initializer {
  static int initCount;
public:
  Initializer() {
    std::cout << "Initializer()" << std::endl;
    // Initialize first time only
    if(initCount++ == 0) {
      std::cout << "performing initialization"
                << std::endl;
      x = 100;
      y = 200;
    }
  }
  ~Initializer() {
    std::cout << "~Initializer()" << std::endl;
    // Clean up last time only
    if(--initCount == 0) {
```

```
      std::cout << "performing cleanup"
                << std::endl;
      // Any necessary cleanup here
    }
  }
};

// The following creates one object in each
// file where Initializer.h is included, but that
// object is only visible within that file:
static Initializer init;
#endif // INITIALIZER_H ///:~
```

Las declaraciones para `x` e `y` anuncian tan sólo que esos objetos existen, pero no reservan espacio para los objetos. No obstante, la definición para el `Initializer init` reserva espacio para ese objeto en cada archivo en que se incluya el archivo de cabecera. Pero como el nombre es `static` (en esta ocasión controlando la visibilidad, no la forma en la que se almacena; el almacenamiento se produce a nivel de archivo por defecto), sólo es visible en esa unidad de traducción, por lo que el enlazador no se quejará por múltiples errores de definición.

He aquí el archivo con las definiciones para `x`, `y` e `initCount`:

```
//: C10:InitializerDefs.cpp {O}
// Definitions for Initializer.h
#include "Initializer.h"
// Static initialization will force
// all these values to zero:
int x;
int y;
int Initializer::initCount;
///:~
```

(Por supuesto, una instancia *estática de fichero* de `init` también se incluye en este archivo cuando se incluye el archivo de cabecera. Suponga que otros dos archivos se crean por la librería del usuario:

```
//: C10:Initializer.cpp {O}
// Static initialization
#include "Initializer.h"
///:~
```

y

```
//: C10:Initializer2.cpp
//{L} InitializerDefs Initializer
// Static initialization
#include "Initializer.h"
using namespace std;

int main() {
  cout << "inside main()" << endl;
```

```
  cout << "leaving main()" << endl;
} ///:~
```

Ahora no importa en qué unidad de traducción se inicializa primero. La primera vez que una unidad de traducción que contenga `Initializer.h` se inicialice, `initCount` será cero por lo que la inicialización será llevada a cabo. (Esto depende en gran medida en el hecho que la zona de almacenamiento estático está a cero antes de que cualquier inicialización dinámica se lleve a cabo). Para el resto de unidades de traducción, `initCount` no será cero y se eludirá la inicialización. La limpieza ocurre en el orden inverso, y `~Initializer()` asegura que sólo ocurrirá una vez.

Este ejemplo utiliza tipos del lenguaje como objetos estáticos globales. Esta técnica también funciona con clases, pero esos objetos deben ser inicializados dinámicamente por la clase `Initializer`. Una forma de hacer esto es creando clases sin constructores ni destructores, pero sí con métodos de inicialización y limpieza con nombres diferentes. Una aproximación más común, de todas formas, es tener punteros a objetos y crearlos utilizando `new` dentro de `Initializer()`.

Técnica dos

Bastante después de la aparición de la técnica uno, alguien (no sé quien) llegó con la técnica explicada en esta sección, que es mucho más simple y limpia que la anterior. El hecho de que tardase tanto en descubrirse es un tributo a la complejidad de C++.

Esta técnica se sustenta en el hecho de que los objetos estáticos dentro de funciones (sólo) se inicializan la primera vez que se llama a la función. Tenga presente que el problema que estamos intentando resolver aquí no es cuando se inicializan los objetos estáticos (que se puede controlar separadamente) sino más bien asegurarnos de que la inicialización ocurre en el orden adecuado.

Esta técnica es muy limpia y astuta. Para cualquier dependencia de inicialización, se coloca un objeto estático dentro de una función que devuelve una referencia a ese objeto. De esta forma, la única manera de acceder al objeto estático es llamando a la función, y si ese objeto necesita acceder a otros objetos estáticos de los que depende, debe llamar a sus funciones. Y la primera vez que se llama a una función, se fuerza a llevar a cabo la inicialización. Está garantizado que el orden de la inicialización será correcto debido al diseño del código, no al orden que arbitrariamente decide el enlazador.

Para mostrar un ejemplo, aquí tenemos dos clases que dependen la una de la otra. La primera contiene un bool que sólo se inicializa por el constructor, por lo que se puede decir si se ha llamado el constructor por una instancia estática de la clase (el área de almacenamiento estático se inicializa a cero al inicio del programa, lo que produce un valor `false` para el bool si el constructor no ha sido llamado).

```
//: C10:Dependency1.h
#ifndef DEPENDENCY1_H
#define DEPENDENCY1_H
#include <iostream>

class Dependency1 {
  bool init;
public:
  Dependency1() : init(true) {
    std::cout << "Dependency1 construction"
```

```
              << std::endl;
  }
  void print() const {
    std::cout << "Dependency1 init: "
              << init << std::endl;
  }
};
#endif // DEPENDENCY1_H ///:~
```

El constructor también indica cuando ha sido llamado, y es posible el estado del objeto para averiguar si ha sido inicializado.

La segunda clase es inicializada por un objeto de la primera clase, que es lo que causa la dependencia:

```
//: C10:Dependency2.h
#ifndef DEPENDENCY2_H
#define DEPENDENCY2_H
#include "Dependency1.h"

class Dependency2 {
  Dependency1 d1;
public:
  Dependency2(const Dependency1& dep1): d1(dep1){
    std::cout << "Dependency2 construction ";
    print();
  }
  void print() const { d1.print(); }
};
#endif // DEPENDENCY2_H ///:~
```

El constructor se anuncia a si mismo y imprime el estado del objeto `d1` por lo que puede ver si éste se ha inicializado cuando se llama al constructor.

Para demostrar lo que puede ir mal, el siguiente archivo primero pone las definiciones de los objetos estáticos en el orden incorrecto, tal y como sucedería si el enlazador inicializase el objeto `Dependency2` antes del `Dependency1`. Después se invierte el orden para mostrar que funciona correctamente si el orden resulta ser el correcto. Finalmente, se muestra la técnica dos.

Para proporcionar una salida más legible, se ha creado la función `separator()`. El truco está en que usted no puede llamar a la función globalmente a menos que la función sea utilizada para llevar a cabo la inicialización de la variable, por lo que `separator()` devuelve un valor absurdo que es utilizado para inicializar un par de variables globales.

```
//: C10:Technique2.cpp
#include "Dependency2.h"
using namespace std;

// Returns a value so it can be called as
// a global initializer:
int separator() {
  cout << "----------------------" << endl;
  return 1;
```

```
}

// Simulate the dependency problem:
extern Dependency1 dep1;
Dependency2 dep2(dep1);
Dependency1 dep1;
int x1 = separator();

// But if it happens in this order it works OK:
Dependency1 dep1b;
Dependency2 dep2b(dep1b);
int x2 = separator();

// Wrapping static objects in functions succeeds
Dependency1& d1() {
  static Dependency1 dep1;
  return dep1;
}

Dependency2& d2() {
  static Dependency2 dep2(d1());
  return dep2;
}

int main() {
  Dependency2& dep2 = d2();
} ///:~
```

Las funciones `d1()` y `d2()` contienen instancias estáticas de los objetos `Dependency1` y `Dependency2`. Ahora, la única forma de acceder a los objetos estáticos es llamando a las funciones y eso fuerza la inicialización estática en la primera llamada a la función. Esto significa que se garantiza la inicialización correcta, cosa que verá cuando lance el programa y observe la salida.

He aquí como debe organizar el código para usar esta técnica. Ordinariamente, los objetos estáticos deben ser definidos en archivos diferentes (puesto que se ha visto forzado a ello por alguna razón; recuerde que definir objetos estáticos en archivos diferentes es lo que causa el problema), por lo que definirá las funciones envoltorio `wrapping functions`) en archivos diferentes. Pero éstas necesitan estar declaradas en los archivos de cabecera:

```
//: C10:Dependency1StatFun.h
#ifndef DEPENDENCY1STATFUN_H
#define DEPENDENCY1STATFUN_H
#include "Dependency1.h"
extern Dependency1& d1();
#endif // DEPENDENCY1STATFUN_H ///:~
```

En realidad, el «extern» es redundante para la declaración de la función. Éste es el segundo archivo de cabecera:

```
//: C10:Dependency2StatFun.h
#ifndef DEPENDENCY2STATFUN_H
#define DEPENDENCY2STATFUN_H
```

```
#include "Dependency2.h"
extern Dependency2& d2();
#endif // DEPENDENCY2STATFUN_H ///:~
```

Ahora, en los archivos de implementación donde previamente habría situado las definiciones de los objetos estáticos, situará las definiciones de las funciones envoltorio:

```
//: C10:Dependency1StatFun.cpp {O}
#include "Dependency1StatFun.h"
Dependency1& d1() {
  static Dependency1 dep1;
  return dep1;
} ///:~
```

Presumiblemente, otro código puede también componer esos archivos. He aquí otro archivo:

```
//: C10:Dependency2StatFun.cpp {O}
#include "Dependency1StatFun.h"
#include "Dependency2StatFun.h"
Dependency2& d2() {
  static Dependency2 dep2(d1());
  return dep2;
} ///:~
```

Ahora hay dos archivos que pueden ser enlazados en cualquier orden y si contuviesen objetos estáticos ordinarios podría producirse cualquier orden de inicialización. Pero como contienen funciones envoltorio, no hay posibilidad de inicialización incorrecta:

```
//: C10:Technique2b.cpp
//{L} Dependency1StatFun Dependency2StatFun
#include "Dependency2StatFun.h"
int main() { d2(); } ///:~
```

Cuando ejecute este programa verá que la inicialización del objeto estático `Dependency1` siempre se lleva a cabo antes de la inicialización del objeto estático `Dependency2`. También puede ver que ésta es una solución bastante más simple que la de la uno.

Puede verse tentado a escribir `d1()` y `d2()` como funciones `inline` dentro de sus respectivos archivos de cabecera, pero eso es algo que, definitivamente, no debe hacer. Una función `inline` puede ser duplicada en cada archivo en el que aparezca y esa duplicación incluye la definición de los objetos estáticos. Puesto que las funciones `inline` llevan asociado por defecto enlazado interno, esto provocará la aparición de múltiples objetos estáticos entre las diversas unidades de traducción, lo que ciertamente causará problemas. Es por eso que debe asegurarse que sólo existe una única definición para cada función contenedora, y eso significa no hacerlas `inline`.

10.5. Especificaciones de enlazado alternativo

¿Qué pasa si está escribiendo un programa en C++ y quiere usar una librería de C? Si hace uso de la declaración de funciones de C,

```
float f(int a, char b);
```

el compilador de C++ adornará el nombre como algo tipo `_f_int_char` para permitir la sobrecarga de la función (y el enlazado con verificación de tipos). De todas formas, el compilador de C que compiló su librería C definitivamente no decoró ese nombre, por lo que su nombre interno será `_f`. Así pues, el enlazador no será capaz de resolver sus llamadas tipo C++ a `f()`.

La forma de resolver esto que se propone en C++ es la *especificación de enlazado alternativo*, que se produjo en el lenguaje sobrecargando la palabra clave `extern`. A la palabra clave `extern` le sigue una cadena que especifica el enlazado deseado para la declaración, seguido por la declaración:

```
extern "C" float f(int a, char b);
```

Esto le dice al compilador que `f()` tiene enlazado tipo C, de forma que el compilador no decora el nombre. Las dos únicas especificaciones de enlazado soportadas por el estándar son «C» y «C++», pero algunos vendedores ofrecen compiladores que también soportan otros lenguajes.

Si tiene un grupo de declaraciones con enlazado alternativo, póngalas entre llaves, como a continuación:

```
extern "C" {
    float f(int a, char b);
    double d(int a, char b);
}
```

O, para archivos de cabecera,

```
extern "C" {
    #include "Myheader.h"
}
```

La mayoría de compiladores disponibles de C++ manejan las especificaciones de enlazado alternativo dentro de sus propios archivos de cabecera que trabajan tanto con C como con C++, por lo que no tiene que preocuparse de eso.

10.6. Resumen

La palabra clave `static` puede llevar a confusión porque en algunas situaciones controla la reserva de espacio en memoria, y en otras controla la visibilidad y enlazado del nombre.

Con la introducción de los espacios de nombres de C++, dispone de una alternativa mejorada y más flexible para controlar la proliferación de nombres en proyectos grandes.

El uso de `static` dentro de clases es un método más para controlar los nombres de un programa. Los nombres no colisionan con nombres globales, y la visibilidad y acceso se mantiene dentro del programa, dándole un mayor control para el mantenimiento de su código.

10.7. Ejercicios

Las soluciones a los ejercicios se pueden encontrar en el documento electrónico titulado «The Thinking in C++ Annotated Solution Guide», disponible por poco dinero en www.BruceEckel.com.

1. Cree una función con una variable estática que sea un puntero (con un argumento por defecto igual cero). Cuando la función que realice la llamada proporcione un valor para ese argumento se usará para apuntar al principio de un array de int. Si se llama a la función con el argumento cero (utilizando el argumento por defecto), la función devuelve el siguiente valor del array, hasta que llegue a un valor `-1` en el array (que actuará como señal de final). Experimente con esta función en `main()`.

2. Cree una función que devuelva el siguiente valor de una serie de Fibonacci cada vez que sea llamada. Añada un argumento que de tipo bool con valor por defecto `false` tal que cuando el argumento valga `true` «reinicie» la función al principio de la serie de Fibonacci. Experimente con esta función en `main()`.

3. Cree una clase que contenga un array de int. Especifique la dimensión del array utilizando `static const int` dentro de la clase. Añada una variable `const int` e inicialícela en la lista de inicialización del constructor. Haga al constructor `inline`. Añada un atributo static int e inicialícelo a un valor específico. Añada un método estático que imprima el atributo estático. Añada un miembro `inline` llamado `print()` que imprima todos los valores del array y que llame al método estático. Experimente con esta clase en `main()`.

4. Cree una clase llamada `Monitor` que mantenga el registro del número de veces que ha sido llamado su método `incident()`. Añada un método `print()` que muestre por pantalla el número de incidentes. Ahora cree una función global (no un método) que contenga un objeto estático `Monitor`. Cada vez que llame a la función debe llamar a `incident()`, después al método `print()` para sacar por pantalla el contador de incidentes. Experimente con la función en `main()`.

5. Modifique la clase `Monitor` del Ejercicio 4 de forma que pueda decrementar (`decrement()`) el contador de incidentes. Cree una clase llamada `Monitor2` que tome como argumento del constructor un puntero a `Monitor1`, y que almacene ese puntero y llame a `incident()` y `print()`. En el destructor para `Monitor2`, llame a `decrement()` y `print()`. Cree ahora un objeto estático de `Monitor2` dentro de una función. Dentro de `main()`, experimente llamando y no llamando a la función para ver qué pasa con el destructor de `Monitor2`.

6. Cree un objeto global de clase `Monitor2` y vea qué sucede.

7. Cree una clase con un destructor que imprima un mensaje y después llame a `exit()`. Cree un objeto global de esa clase y vea qué pasa.

8. En `StaticDestructors.cpp`, experimente con el orden de llamada de los constructores y destructores llamando a `f()` y `g()` dentro de `main()` en diferentes órdenes. ¿Su compilador inicializa los objetos de la forma correcta?

9. En `StaticDestructors.cpp`, pruebe el manejo de errores por defecto de su implementación convirtiendo la definición original de `out` dentro de una declaración `extern`, y poniendo la definición real después de la definición de `a` (donde el constructor de `Obj` manda información a `out`). Asegúrese que no hay ningún otro programa importante funcionando en su máquina cuando ejecute el código o que su máquina maneje las faltas robustamente.

10. Pruebe que las variables estáticas de fichero en los archivos de cabecera no chocan entre sí cuando son incluidas en más de un archivo `cpp`.

11. Cree una única clase que contenga un int, un constructor que inicialice el int con su argumento, un método que cambie el valor del int con su argumento y una función `print()` que muestre por pantalla el int. Coloque su clase en un archivo de cabecera e incluya dicho archivo en dos archivos `cpp`. En uno de ellos cree una instancia de la clase y en la otra declare ese identificador como `extern` y pruebe dentro de `main()`. Recuerde, debe enlazar los dos archivos objeto o de lo contrario el enlazador no encontrará el objeto.

12. Cree la instancia del objeto del Ejercicio 11 como `static` y verifique que, debido a eso, el enlazador es incapaz de encontrarla.

13. Declare una función en un archivo de cabecera. Defina la función en un archivo `cpp` y llámela desde `main()` en un segundo archivo `cpp`. Compile y verifique que funciona. Ahora cambie la definición de la función de forma que sea `static` y verifique que el enlazador no puede encontrarla.

14. Modifique `Volatile.cpp` del Capítulo 8 para hacer que `comm::isr()` funcione realmente como una rutina de servicio de interrupción. Pista: una rutina de servicio de interrupción no toma ningún argumento.

15. Escriba y compile un único programa que utilice las palabras clave `auto` y `register`.

16. Cree un archivo de cabecera que contenga un espacio de nombres. Dentro del espacio de nombres cree varias declaraciones de funciones. Cree ahora un segundo archivo de cabecera que incluya el primero y continúe el espacio de nombres, añadiendo varias declaraciones de funciones más. Cree ahora un archivo `cpp` que incluya el segundo archivo de cabecera. Cambie su espacio de nombres a otro nombre (más corto). Dentro de una definición de función, llame a una de sus funciones utilizando la resolución de ámbito. Dentro de una definición de función separada, escriba una directiva `using` para introducir su espacio de nombres en el ámbito de esa función, y demuestre que no necesita utilizar la resolución de ámbito para llamar a las funciones desde su espacio de nombres.

17. Cree un archivo de cabecera con un espacio de nombres sin nombre. Incluya la cabecera en dos archivos `cpp` diferentes y demuestre que un espacio sin nombre es único para cada :unidad de traducción.

18. Utilizando el archivo de cabecera del Ejercicio 17, demuestre que los nombres de un espacio de nombres sin nombre están disponibles automáticamente en una :unidad de traducción sin calificación.

19. Modifique `FriendInjection.cpp` para añadir una definición para la función amiga y para llamar a la función desde `main()`.
20. En `Arithmetic.cpp`, demuestre que la directiva `using` no se extiende fuera de la función en la que fue creada.
21. Repare el problema de `OverridingAmbiguity.cpp`, primero con resolución de ámbito y luego, con una declaración `using` que fuerce al compilador a escojer uno de los nombres de función idénticos.
22. En dos archivos de cabecera, cree dos espacios de nombres, cada uno conteniendo una clase (con todas las definiciones `inline`) con idéntico nombre que el del otro espacio de nombres. Cree un archivo `cpp` que incluya ambos archivos. Cree una función y, dentro de la función, utilice la directiva `using` para introducir ambos espacios de nombres. Pruebe a crear un objeto de la clase y vea que sucede. Haga las directivas `using` globales (fuera de la función) para ver si existe alguna diferencia. Repare el problema usando la resolución de ámbito, y cree objetos de ambas clases.
23. Repare el problema del Ejercicio 22 con una declaración `using` que fuerce al compilador a escojer uno de los nombres de clase idénticos.
24. Extraiga las declaraciones de espacios de nombres de `BobsSuperDuperLibrary.cpp` y `UnnamedNamespaces.cpp` y póngalos en archivos separados, dando un nombre al espacio de nombres sin nombre en el proceso. En un tercer archivo de cabecera, cree un nuevo espacio de nombres que combine los elementos de los otros dos espacios de nombres con declaraciones `using`. En `main()`, introduzca su nuevo espacio de nombres con una directiva `using` y acceda a todos los elementos de su espacio de nombres.
25. Cree un archivo de cabecera que incluya `<string>` y `<iostream>` pero que no use ninguna directiva `using` ni ninguna declaración `using`. Añada guardas de inclusión como ha visto en los archivos de cabecera del libro. Cree una clase con todas las funciones `inline` que muestre por pantalla el string. Cree un archivo `cpp` y ejercite su clase en `main()`.
26. Cree una clase que contenga un static double y long. Escriba un método estático que imprima los valores.
27. Cree una clase que contenga un int, un constructor que inicialice el int con su argumento, y una función `print()` que muestre por pantalla el int. Cree ahora una segunda clase que contenga un objeto estático de la primera. Añada un método estático que llame a la función `print()` del objeto estático. Ejercitu su clase en `main()`.
28. Cree una clase que contenga un array estático de int constante y otro no constante. Escriba métodos estáticos que impriman los arrays. Experimente con su clase en `main()`.
29. Cree una clase que contenga un string, con un constructor que inicialice el string a partir de su argumento, y una función `print()` que imprima el string. Cree otra clase que contenga un array estático, tanto constante como no constante, de objetos de la primera clase, y métodos estáticos para imprimir dichos arrays. Experimente con la segunda clase en `main()`.
30. Cree una `struct` que contenga un int y un constructor por defecto que inicialice el int a cero. Haga ese `struct` local a una función. Dentro de dicha función, cree un array de objetos de su `struct` y demuestre que cada int del array ha sido inicializado a cero automáticamente.

31. Cree una clase que represente una conexión a impresora, y que sólo le permita tener una impresora.

32. En un archivo de cabecera, cree una clase `Mirror` que contiene dos atributos: un puntero a un objeto `Mirror` y un bool. Déle dos constructores: el constructor por defecto inicializa el bool a `true` y el puntero a `Mirror` a cero. El segundo constructor toma como argumento un puntero a un objeto `Mirror`, que asigna al puntero interno del objeto; pone el bool a `false`. Añada un método `test()`: si el puntero del objeto es distinto de cero, devuelve el valor de `test()` llamado a través del puntero. Si el puntero es cero, devuelve el bool. Cree ahora cinco archivos `cpp`, cada uno incluyendo la cabecera `Mirror`. El primer archivo `cpp` define un objeto `Mirror` global utilizando el constructor por defecto. El segundo archivo declara el objeto del primer archivo como `extern`, y define un objeto `Mirror` global utilizando el segundo constructor, con un puntero al primer objeto. Siga haciendo lo mismo hasta que llegue al último archivo, que también contendrá una definición de objeto global. En este archivo, `main()` debe llamar a la función `test()` e informar del resultado. Si el resultado es `true`, encuentre la forma de cambiar el orden de enlazado de su enlazador y cámbielo hasta que el resultado sea `false`.

33. Repare el problema del Ejercicio 32 utilizando la técnica uno mostrada en este libro.

34. Repare el problema del Ejercicio 32 utilizando la técnica dos mostrada en este libro.

35. Sin incluir ningún archivo de cabecera, declare la función `puts()` de la Librería Estándar de C. Llame a esa función desde `main()`.

11: Las referencias y el constructor de copia

Las referencias son como punteros constantes que el compilador de-referencia automáticamente.

Aunque las referencias también existen en Pascal, la versión de C++ se tomó del lenguaje Algol. Las referencias son esenciales en C++ para ayudar en la sintaxis de los operadores sobrecargados (vea el capítulo 12) y, además, son una buena forma para controlar la manera en que los argumentos se pasan a las funciones tanto hacia dentro como hacia fuera.

En este capítulo primero verá la diferencia entre los punteros en C y en C++, y luego se presentarán las referencias. Pero la mayor parte del capítulo ahondará en el asunto un tanto confuso para los programadores de C++ novatos: el constructor de copia, un constructor especial (que usa referencias) y construye un nuevo objeto de otro ya existente del mismo tipo. El compilador utiliza el constructor de copia para pasar y retornar objetos *por valor* a las funciones.

Finalmente, se hablará sobre la característica (un tanto oscura) de los *punteros-a-miembro* de C++.

11.1. Punteros en C++

La diferencia más importante entre los punteros en C y en C++ es que los de C++ están fuertemente tipados. Sobre todo en lo que al tipo void * se refiere. C no permite asignar un puntero de un tipo a otro de forma casual, pero *sí* permite hacerlo mediante un void *. Por ejemplo,

```
Bird* b;
Rock* r;
void* v;
v = r;
b = v;
```

A causa de esta «característica» de C, puede utilizar cualquier tipo como si de otro se tratara sin ningún aviso por parte del compilador. C++ no permite hacer esto; el compilador da un mensaje de error, y si realmente quiere utilizar un tipo como otro diferente, debe hacerlo explícitamente, tanto para el compilador como para el lector, usando molde (*cast* en inglés). (En el capítulo 3 se habló sobre la sintaxis mejorada del molde «explícito»).

11.2. Referencias en C++

Una *referencia* (&) es como un puntero constante que se de-referencia automáticamente. Normalmente se utiliza en la lista de argumentos y en el valor de retorno de de las funciones. Pero también se puede hacer una referencia que apunte a algo que no ha sido asignado. Por ejemplo:

```
//: C11:FreeStandingReferences.cpp
#include <iostream>
using namespace std;

// Ordinary free-standing reference:
int y;
int& r = y;
// When a reference is created, it must
// be initialized to a live object.
// However, you can also say:
const int& q = 12;  // (1)
// References are tied to someone else's storage:
int x = 0;          // (2)
int& a = x;         // (3)
int main() {
  cout << "x = " << x << ", a = " << a << endl;
  a++;
  cout << "x = " << x << ", a = " << a << endl;
} ///:~
```

En la linea (1) el compilador asigna la cantidad necesaria de memoria, la inicializa con el valor 12, y liga la referencia a esa porción de memoria. Lo importante es que una referencia debe estar ligada a la memoria de *alguien*. Cuando se accede a una referencia, se está accediendo a esa memoria. Así pues, si escribe las lineas (2) y (3) incrementará `x` cuando se incremente `a`, tal como se muestra en el `main()`. Lo más fácil es pensar que una referencia es como un puntero de lujo. La ventaja de este «puntero» es que nunca hay que preguntarse si ha sido inicializado (el compilador lo impone) o si hay que destruirlo (el compilador lo hace).

Hay que seguir unas determinadas reglas cuando se utilizan referencias:

1. La referencia de ser inicializada cuando se crea. (Los punteros pueden inicializarse en cualquier momento).

2. Una vez que se inicializa una referencia, ligándola a un objeto, no se puede ligar a otro objeto. (Los punteros pueden apuntar a otro objeto en cualquier momento).

3. No se pueden tener referencias con valor nulo. Siempre ha de suponer que una referencia está conectada a una trozo de memoria ya asignada.

11.2.1. Referencias en las funciones

El lugar más común en el que verá referencias es en los argumentos y valor de retorno de las funciones. Cuando se utiliza una referencia como un argumento de una función, cualquier cambio realizado en la referencia *dentro* de la función se realizará realmente sobre el argumento *fuera* de la función. Por supuesto que podría hacer lo

mismo pasando un puntero como argumento, pero una referencia es sintácticamente más clara. (Si lo desea, puede pensar que una referencia es, nada más y nada menos, más conveniente sintácticamente).

Si una función retorna una referencia, ha de tener el mismo cuidado que si la función retornara un puntero. La referencia que se devuelva debe estar ligada a algo que no sea liberado cuando la función retorne. Si no, la referencia se referirá a un trozo de memoria sobre el que ya no tiene control.

He aquí un ejemplo:

```
//: C11:Reference.cpp
// Simple C++ references

int* f(int* x) {
  (*x)++;
  return x; // Safe, x is outside this scope
}

int& g(int& x) {
  x++; // Same effect as in f()
  return x; // Safe, outside this scope
}

int& h() {
  int q;
//!  return q;  // Error
  static int x;
  return x; // Safe, x lives outside this scope
}

int main() {
  int a = 0;
  f(&a); // Ugly (but explicit)
  g(a);  // Clean (but hidden)
} ///:~
```

La llamada a `f()` no tiene la ventaja ni la claridad que la utilización de referencias, pero está claro que se está pasando una dirección mediante un puntero. En la llamada a `g()`, también se pasa una dirección (mediante una referencia), pero no se ve.

Referencias constantes

El argumento referencia en `Reference.cpp` funciona solamente en caso de que el argumento no sea un objeto constante. Si fuera un objeto constante, la función `g()` no aceptaría el argumento, lo cual es positivo porque la función *modificaría* el argumento que está fuera de la función. Si sabe que la función respetará las constancia un objeto, el hecho de que el argumento sea una referencia constante permitirá que la función se pueda utilizar en cualquier situación. Esto significa que para tipos predefinidos, la función no modificará el argumento, y para tipos definidos por el usuario, la función llamará solamente a métodos constantes, y no modificara ningún atributo público.

La utilización de referencias constantes en argumentos de funciones es especialmente importante porque una función puede recibir un objeto temporal. Éste podría

haber sido creado como valor de retorno de otra función o explícitamente por el usuario de la función. Los objetos temporales son siempre constantes. Por eso, si no utiliza una referencia constante, el compilador se quejará. Como ejemplo muy simple:

```
//: C11:ConstReferenceArguments.cpp
// Passing references as const

void f(int&) {}
void g(const int&) {}

int main() {
//!  f(1); // Error
  g(1);
} ///:~
```

La llamada `f(1)` provoca un error en tiempo de compilación porque el compilador debe crear primero una referencia. Lo hace asignando memoria para un int, iniciánlizándolo a uno y generando la dirección de memoria para ligarla a la referencia. La memoria debe ser constante porque no tendría sentido cambiarlo: no puede cambiarse de nuevo. Puede hacer la misma suposición para todos los objetos temporales: son inaccesibles. Es importante que el compilador le diga cuándo está intentando cambiar algo de este estilo porque podría perder información.

Referencias a puntero

En C, si desea modificar el *contenido* del puntero en sí en vez de modificar a lo que apunta, la declaración de la función sería:

```
void f(int**);
```

y tendría que tomar la dirección del puntero cuando se llamara a la función:

```
int i = 47;
int* ip = &i;
f(&ip);
```

La sintaxis es más clara con las referencias en C++. El argumento de la función pasa a ser de una referencia a un puntero, y así no ha de manejar la dirección del puntero.

```
//: C11:ReferenceToPointer.cpp
#include <iostream>
using namespace std;

void increment(int*& i) { i++; }

int main() {
  int* i = 0;
  cout << "i = " << i << endl;
  increment(i);
  cout << "i = " << i << endl;
} ///:~
```

Al ejecutar este programa se observa que el puntero se incrementa en vez de incrementar a lo que apunta.

11.2.2. Consejos para el paso de argumentos

Cuando se pasa un argumento a un función, lo normal debería ser pasarlo como una referencia constante. Aunque al principio puede parecer que sólo tiene ventajas en términos de eficacia (y normalmente en diseño e implementación inicial no se tiene muy en cuenta la eficacia), además tiene otras: como se podrá ver en el resto del capítulo, se requiere un constructor de copia para pasar un objeto por valor, y esto no siempre es posible.

La eficacia puede mejorar substancialmente por este simple hábito: pasar un argumento por valor necesita una llamada a un constructor y otra a un destructor, pero si no se va a modificar el argumento, el hecho de pasarlo como una referencia constante sólo necesita poner una dirección en la pila.

De hecho, prácticamente la única situación en la que no es preferible pasar la dirección, es cuando provocará tales problemas a un objeto que pasar por valor es la única alternativa segura (en vez de modificar el objeto que está fuera del ámbito de la función, algo que el que llama a la función normalmente no espera). Ese es el tema de la siguiente sección.

11.3. El constructor de copia

Ahora que entiende lo básico de las referencias en C++, está preparado para tratar uno de los conceptos más confusos del lenguaje: el constructor de copia, a menudo denominado `X(X&)` («X de la referencia X»). Este constructor es esencial para controlar el paso y retorno por valor de los tipos definidos por el usuario en las llamadas a funciones. De hecho es tan importante que el compilador crea automáticamente un constructor de copia en caso de que el programador no lo proporcione.

11.3.1. Paso y retorno por valor

Para entender la necesidad del constructor de copia, considere la forma en que C maneja el paso y retorno por valor de variables cuando se llama a una función. Si declara una función y la invoca,

```
int f(int x, char c);
int g = f(a, b);
```

¿cómo sabe el compilador cómo pasar y retornar esas variables? ¡Simplemente lo sabe! El rango de tipos con los que debe tratar es tan pequeño (char, int, float, double, y sus variaciones), que tal información ya está dentro del compilador.

Si averigua cómo hacer que su compilador genere código ensamblador y determina qué instrucciones se usan para la invocación de la función `f()`, obtendrá algo equivalente a:

```
push  b
```

```
push  a
call  f()
add  sp, 4
mov  g, register a
```

Este código se ha simplificado para hacerlo genérico; las expresiones `b` y `a` serán diferentes dependiendo de si las variables son globales (en cuyo caso serían `_b` y `_a`) o locales (el compilador las pondría en la pila). Esto también es cierto para `g`. La sintaxis de la llamada a `f()` dependería de su guía de estilo, y `register a` dependería de cómo su ensamblador llama a los registros de la CPU. A pesar de la simplificación, la lógica del código sería la misma.

Tanto en C como en C++, primero se ponen los argumentos en la pila de derecha a izquierda, y luego se llama a la función. El código de llamada es responsable de recoger los argumentos de la pila (lo cual explica la sentencia `add sp, 4`). Pero tenga en cuenta que cuando se pasan argumentos por valor, el compilador simplemente pone copias en la pila (conoce los tamaños de cada uno, por lo que los puede copiar).

El valor de retorno de `f()` se coloca en un registro. Como el compilador sabe lo que se está retornando, porque la información del tipo ya está en el lenguaje, puede retornarlo colocándolo en un registro. En C, con tipos primitivos, el simple hecho de copiar los bits del valor es equivalente a copiar el objeto.

Paso y retorno de objetos grandes

Considere ahora los tipos definidos por el usuario. Si crea una clase y desea pasar un objeto de esa clase por valor, ¿cómo sabe el compilador lo que tiene que hacer? La información de la clase no está en el compilador, pues lo ha definido el usuario.

Para investigar esto, puede empezar con una estructura simple que, claramente, es demasiado grande para ser devuelta a través de los registros:

```
//: C11:PassingBigStructures.cpp
struct Big {
  char buf[100];
  int i;
  long d;
} B, B2;

Big bigfun(Big b) {
  b.i = 100; // Do something to the argument
  return b;
}

int main() {
  B2 = bigfun(B);
} ///:~
```

La conversión a código ensamblador es un poco más complicada porque la mayoría de los compiladores utilizan funciones «auxiliares» (*helper*) en vez de inline. En la función `main()`, la llamada a `bigfun()` empieza como debe: se coloca el contenido de `B` en la pila. (Aquí podría ocurrir que algunos compiladores carguen registros con la dirección y tamaño de `Big` y luego una función auxiliar se encargue de colocar el `Big` en la pila).

En el fragmento de código fuente anterior, lo único necesario antes de llamar a la

función es colocar los argumentos en la pila. Sin embargo, en el código ensamblador de `PassingBigStructures.cpp` se ve una acción adicional: la dirección de `B2` se coloca en la pila antes de hacer la llamada a la función aunque, obviamente, no sea un argumento. Para entender qué pasa, necesita entender las restricciones del compilador cuando llama a una función.

Marco de pila para llamadas a función

Cuando el compilador genera código para llamar a una función, primero coloca en la pila todos los argumentos y luego hace la llamada. Dentro de la función se genera código para mover el puntero de pila hacia abajo, y así proporciona memoria para las variables locales dentro de la función. («hacia abajo» es relativo, la máquina puede incrementar o decrementar el puntero de pila al colocar un argumento). Pero cuando se hace el `CALL` de ensamblador para llamar a la función, la CPU coloca la dirección desde la que se realiza la llamada, y en el `RETURN` de ensamblador se utiliza esa dirección para volver al punto desde donde se realizó la llamada. Esta dirección es sagrada, porque sin ella el programa se perdería por completo. He aquí es aspecto del marco de pila después de ejecutar `CALL` y poner las variables locales de la función:

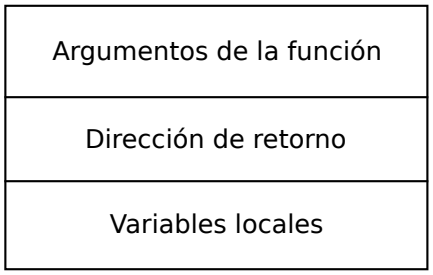

Figura 11.1: Llamada a una función

El código generado por el resto de la función espera que la memoria tenga esta disposición para que pueda utilizar los argumentos y las variables locales sin tocar la dirección de retorno. Llámese a este bloque de memoria, que es todo lo que una función necesita cuando se la llama, *el marco de la función* (*function frame*).

Podría parecer razonable retornar valores mediante la utilización de la pila. El compilador simplemente los colocaría allí y la función devolvería un desplazamiento que indicara dónde empieza el valor de retorno.

Re-entrada

Este problema ocurre porque las funciones en C y C++ pueden sufrir interrupciones; esto es, los lenguajes han de ser (y de hecho son) *re-entrantes*. También permiten llamadas a funciones recursivas. Esto quiere decir que en cualquier punto de ejecución de un programa puede sufrir una interrupción sin que el programa se vea afectado por ello. Obviamente la persona que escribe la rutina de servicio de interrupciones (ISR) es responsable de guardar y restaurar todos los registros que se utilicen en la ISR. Pero si la ISR necesita utilizar la pila, ha de hacerlo con seguridad. (Piense que una ISR es como una función normal sin argumentos y con valor de retorno void que guarda y restaura el estado de la CPU. La ejecución de una ISR suele

producirse por un evento hardware, y no con una invocación dentro del programa de forma explícita).

Ahora imagine que pasaría si una función normal intentara retornar valores mediante la pila. No puede tocar la pila por encima del la dirección de retorno, así que la función tendría que colocar los valores de retorno debajo de la dirección de retorno. Pero cuando se ejecuta el `RETURN`, el puntero de pila debería estar apuntando a la dirección de retorno (o justo debajo, depende de la máquina), así que la función debe subir el puntero de la pila, desechando todas las variables locales. Si intenta retornar valores usando la pila por debajo de la dirección de retorno, en ese momento es vulnerable a una interrupción. La ISR escribiría encima de los valores de retorno para colocar su dirección de retorno y sus variables locales.

Para resolver este problema, el que llama a la función *podría* hacerse responsable de asignar la memoria extra en la pila para los valores de retorno antes de llamar a la función. Sin embargo, C no se diseñó de esta manera y C++ ha de ser compatible. Como verá pronto, el compilador de C++ utiliza un esquema más eficaz.

Otra idea sería retornar el valor utilizando un área de datos global, pero tampoco funcionaría. La re-entrada significa que cualquier función puede ser una rutina de interrupción para otra función, *incluida la función que se está ejecutando*. Por lo tanto, si coloca un valor de retorno en un área global, podría retornar a la misma función, lo cual sobreescribiría el valor de retorno. La misma lógica se aplica a la recursividad.

Los registros son el único lugar seguro para devolver valores, así que se vuelve al problema de qué hacer cuando los registros no son lo suficientemente grandes para contener el valor de retorno. La respuesta es colocar la dirección de la ubicación del valor de retorno en la pila como uno de los argumentos de la función, y dejar que la función copie la información que se devuelve directamente en esa ubicación. Esto no solo soluciona todo los problemas, si no que además es más eficaz. Ésta es la razón por la que el compilador coloca la dirección de `B2` antes de llamar a `bigfun` en la función `main()` de `PassingBigStructures.cpp`. Si viera el código ensamblador de `bigfun()` observaría que la función espera este argumento escondido y lo copia al destino *dentro* de la función.

Copia bit a bit vs. inicialización

Hasta aquí, todo bien. Tenemos un procedimiento para pasar y retornar estructuras simples grandes. Pero note que lo único que tiene es una manera de copiar bits de un lugar a otro, lo que ciertamente funciona bien para la forma (muy primitiva) en que C trata las variables. Sin embargo, en C++ los objetos pueden ser mucho más avanzados que un puñado de bits, pues tienen significado y, por lo tanto, puede que no responda bien al ser copiado.

Considere un ejemplo simple: una clase que conoce cuantos objetos de un tipo existen en cualquier momento. En el Capítulo 10 se vio la manera de hacerlo incluyendo un atributo estático (`static`):

```
//: C11:HowMany.cpp
// A class that counts its objects
#include <fstream>
#include <string>
using namespace std;
ofstream out("HowMany.out");

class HowMany {
  static int objectCount;
```

```
public:
  HowMany() { objectCount++; }
  static void print(const string& msg = "") {
    if(msg.size() != 0) out << msg << ": ";
    out << "objectCount = "
         << objectCount << endl;
  }
  ~HowMany() {
    objectCount--;
    print("~HowMany()");
  }
};

int HowMany::objectCount = 0;

// Pass and return BY VALUE:
HowMany f(HowMany x) {
  x.print("x argument inside f()");
  return x;
}

int main() {
  HowMany h;
  HowMany::print("after construction of h");
  HowMany h2 = f(h);
  HowMany::print("after call to f()");
} ///:~
```

La clase `HowMany` contiene un entero estático llamado `objectCount` y un método estático llamado `print()` para representar el valor de `objectCount`, junto con argumento de mensaje opcional. El constructor incrementa `objectCount` cada vez que se crea un objeto, y el destructor lo disminuye.

Sin embargo la salida no es la que cabría esperar:

```
after construction of h: objectCount = 1
x argument inside f(): objectCount = 1
~HowMany(): objectCount = 0
after call to f(): objectCount = 0
~HowMany(): objectCount = -1
~HowMany(): objectCount = -2
```

Después de crear `h`, el contador es uno, lo cual está bien. Pero después de la llamada a `f()` se esperaría que el contador estuviera a dos, porque `h2` está ahora también dentro de ámbito. Sin embargo, el contador es cero, lo cual indica que algo ha ido muy mal. Esto se confirma por el hecho de que los dos destructores, llamados al final de `main()`, hacen que el contador pase a ser negativo, algo que no debería ocurrir nunca.

Mire lo que ocurre dentro de `f()` después de que el argumento se pase por valor. Esto quiere decir que el objeto original `h` existe fuera del ámbito de la función y, por otro lado, hay un objeto de más *dentro* del ámbito de la función, que es copia del objeto que se pasó por valor. El argumento que se pasó utiliza el primitivo concepto de copia bit a bit de C, pero la clase C++ `HowMany` necesita inicializarse correctamente para mantener su integridad. Por lo tanto, se demuestra que la copia bit a bit no logra el efecto deseado.

Cuando el objeto local sale de ámbito al acabar la función `f()`, se llama a su

destructor, lo cual decrementa `objectCount`, y por lo tanto el `objectCount` se pone a cero. La creación de `h2` se realiza también mediante la copia bit a bit, así que tampoco se llama al constructor, y cuando `h` y `h2` salen de ámbito, sus destructores causan el valor negativo en `objectCount`.

11.3.2. Construcción por copia

El problema se produce debido a que el compilador hace una suposición sobre cómo crear *un nuevo objeto a partir de de otro existente*. Cuando se pasa un objeto por valor, se crea un nuevo objeto, que estará dentro del ámbito de la función, a partir del objeto original existente fuera del ámbito de la función. Esto también se puede aplicar a menudo cuando una función retorna un objeto. En la expresión

```
HowMany h2 = f(h);
```

`h2`, un objeto que no estaba creado anteriormente, se crea a partir del valor que retorna `f();` por tanto también se crea un nuevo objeto a partir de otro existente.

El compilador supone que la creación ha de hacerse con una copia bit a bit, lo que en muchos casos funciona bien, pero en `HowMany` no funciona porque la inicialización va más allá de una simple copia. Otro ejemplo muy común ocurre cuando la clase contiene punteros pues, ¿a qué deben apuntar? ¿debería copiar sólo los punteros o debería asignar memoria nueva y que apuntaran a ella?

Afortunadamente, puede intervenir en este proceso e impedir que el compilador haga una copia bit a bit. Se soluciona definiendo su propia función cuando el compilador necesite crear un nuevo objeto a partir de otro. Lógicamente, está creando un nuevo objeto, por lo que esta función es un constructor, y el único argumento del constructor tiene que ver con el objeto del que se pretende partir para crear el nuevo. Pero no puede pasar ese objeto por valor al constructor porque está intentando *definir* la función que maneja el paso por valor, y, por otro lado, sintácticamente no tiene sentido pasar un puntero porque, después de todo, está creando un objeto a partir de de otro. Aquí es cuando las referencias vienen al rescate, y puede utilizar la referencia del objeto origen. Esta función se llama *constructor de copia*, que también se lo puede encontrar como `X(X&)`, que es el constructor de copia de una clase llamada `X`.

Si crea un constructor de copia, el compilador no realizará una copia bit a bit cuando cree un nuevo objeto a partir de otro. El compilador siempre llamará al constructor de copia. Si no crea el constructor de copia, el compilador intentará hacer algo razonable, pero usted tiene la opción de tener control total del proceso.

Ahora es posible solucionar el problema en `HowMany.cpp`:

```
//: C11:HowMany2.cpp
// The copy-constructor
#include <fstream>
#include <string>
using namespace std;
ofstream out("HowMany2.out");

class HowMany2 {
  string name; // Object identifier
  static int objectCount;
public:
```

```
  HowMany2(const string& id = "") : name(id) {
    ++objectCount;
    print("HowMany2()");
  }
  ~HowMany2() {
    --objectCount;
    print("~HowMany2()");
  }
  // The copy-constructor:
  HowMany2(const HowMany2& h) : name(h.name) {
    name += " copy";
    ++objectCount;
    print("HowMany2(const HowMany2&)");
  }
  void print(const string& msg = "") const {
    if(msg.size() != 0)
      out << msg << endl;
    out << '\t' << name << ": "
        << "objectCount = "
        << objectCount << endl;
  }
};

int HowMany2::objectCount = 0;

// Pass and return BY VALUE:
HowMany2 f(HowMany2 x) {
  x.print("x argument inside f()");
  out << "Returning from f()" << endl;
  return x;
}

int main() {
  HowMany2 h("h");
  out << "Entering f()" << endl;
  HowMany2 h2 = f(h);
  h2.print("h2 after call to f()");
  out << "Call f(), no return value" << endl;
  f(h);
  out << "After call to f()" << endl;
} ///:~
```

Hay unas cuantas cosas nuevas para que pueda hacerse una idea mejor de lo que pasa. Primeramente, el `string name` hace de identificador de objeto cuando se imprima en la salida. Puede poner un identificador (normalmente el nombre del objeto) en el constructor para que se copie en `name` utilizando el constructor con un string como argumento. Por defecto se crea un string vacío. El constructor incrementa `objectCount` y el destructor lo disminuye, igual que en el ejemplo anterior.

Lo siguiente es el constructor de copia, `HowMany2(const HowMany2&)`. El constructor de copia simplemente crea un objeto a partir de otro existente, así que copia en `name` el identificador del objeto origen, seguido de la palabra «copy», y así puede ver de dónde procede. Si mira atentamente, verá que la llamada `name(h.name)` en la lista de inicializadores del constructor está llamando al constructor de copia de la clase string.

Dentro del constructor de copia, se incrementa el contador igual que en el constructor normal. Esto quiere decir que obtendrá un contador de objetos preciso cuando pase y retorne por valor.

La función `print()` se ha modificado para imprimir en la salida un mensaje, el identificador del objeto y el contador de objetos. Como ahora accede al atributo `name` de un objeto concreto, ya no puede ser un método estático.

Dentro de `main()` puede ver que hay una segunda llamada a `f()`. Sin embargo esta llamada utiliza la característica de C para ignorar el valor de retorno. Pero ahora que sabe cómo se retorna el valor (es decir, código *dentro* de la función que maneja el proceso de retorno poniendo el resultado en un lugar cuya dirección se pasa como un argumento escondido), podría preguntarse qué ocurre cuando se ignora el valor de retorno. La salida del programa mostrará alguna luz sobre el asunto.

Pero antes de mostrar la salida, he aquí un pequeño programa que utiliza `iostreams` para añadir números de línea a cualquier archivo:

```
//: C11:Linenum.cpp
//{T} Linenum.cpp
// Add line numbers
#include "../require.h"
#include <vector>
#include <string>
#include <fstream>
#include <iostream>
#include <cmath>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1, "Usage: linenum file\n"
    "Adds line numbers to file");
  ifstream in(argv[1]);
  assure(in, argv[1]);
  string line;
  vector<string> lines;
  while(getline(in, line)) // Read in entire file
    lines.push_back(line);
  if(lines.size() == 0) return 0;
  int num = 0;
  // Number of lines in file determines width:
  const int width =
    int(log10((double)lines.size())) + 1;
  for(int i = 0; i < lines.size(); i++) {
    cout.setf(ios::right, ios::adjustfield);
    cout.width(width);
    cout << ++num << ") " << lines[i] << endl;
  }
} ///:~
```

El archivo se pasa a un vector<string>, utilizando el mismo código fuente que ha visto anteriormente en este libro. Cuando se ponen los números de línea, nos gustaría que todas las líneas estuvieran alineadas, y esto necesita conocer el número de líneas en el archivo para que sea coherente. Se puede conocer el número de líneas con `vector::size()`, pero lo que realmente necesitamos es conocer si hay más líneas de 10, 100, 1000, etc. Si se utiliza el logaritmo en base 10 sobre el número de

líneas en el archivo, se trunca a un entero y se añade uno al valor resultante, eso determinará el ancho máximo en dígitos que un número de línea puede tener.

Nótese que hay un par de llamadas extrañas dentro del bucle `for`: `setf()` y `width()`. Hay llamadas de `ostream` que permiten controlar, en este caso, la justificación y anchura de la salida. Sin embargo se debe llamar cada vez que se imprime línea y por eso están dentro del bucle `for`. El Volumen 2 de este libro tiene un capítulo entero que explica los `iostreams` y que cuenta más sobre estas llamadas así como otras formas de controlar los `iostreams`.

Cuando se aplica `Linenum.cpp` a `HowMany2.out`, resulta:

```
1) HowMany2()
2)   h: objectCount = 1
3) Entering f()
4) HowMany2(const HowMany2&)
5)   h copy: objectCount = 2
6) x argument inside f()
7)   h copy: objectCount = 2
8) Returning from f()
9) HowMany2(const HowMany2&)
10)   h copy copy: objectCount = 3
11) ~HowMany2()
12)   h copy: objectCount = 2
13) h2 after call to f()
14)   h copy copy: objectCount = 2
15) Call f(), no return value
16) HowMany2(const HowMany2&)
17)   h copy: objectCount = 3
18) x argument inside f()
19)   h copy: objectCount = 3
20) Returning from f()
21) HowMany2(const HowMany2&)
22)   h copy copy: objectCount = 4
23) ~HowMany2()
24)   h copy: objectCount = 3
25) ~HowMany2()
26)   h copy copy: objectCount = 2
27) After call to f()
28) ~HowMany2()
29)   h copy copy: objectCount = 1
30) ~HowMany2()
31)   h: objectCount = 0
```

Como se esperaba, la primera cosa que ocurre es que para `h` se llama al constructor normal, el cual incrementa el contador de objetos a uno. Pero entonces, mientras se entra en `f()`, el compilador llama silenciosamente al constructor de copia para hacer el paso por valor. Se crea un nuevo objeto, que es copia de `h` (y por tanto tendrá el identificador «h copy») dentro del ámbito de la función `f()`. Así pues, el contador de objetos se incrementa a dos, por cortesía del constructor de copia.

La línea ocho indica el principio del retorno de `f()`. Pero antes de que se destruya la variable local «h copy» (pues sale de ámbito al final de la función), se debe copiar al valor de retorno, que es `h2`. Por tanto `h2`, que no estaba creado previamente, se crea de un objeto ya existente (la variable local dentro de `f()`) y el constructor de copia vuelve a utilizarse en la línea 9. Ahora el identificador de `h2` es «h copy copy» porque copió el identificador de la variable local de `f()`. Cuando se devuelve el objeto, pero antes de que la función termine, el contador de objetos se incrementa temporalmente a tres, pero la variable local con identificador «h copy» se destruye, disminuyendo a dos. Después de que se complete la llamada a `f()` en la línea 13, sólo hay dos objetos, `h` y `h2`, y puede comprobar, de hecho, que `h2` terminó con el identificador «h copy copy».

Objetos temporales

En la línea 15 se empieza la llamada a `f(h)`, y esta vez ignora el valor de retorno. Puede ver que se invoca el constructor de copia en la línea 16, igual que antes, para pasar el argumento. Y también, igual que antes, en la línea 21 se llama al constructor de copia para el valor de retorno. Pero el constructor de copia necesita una dirección para utilizar como destino (es decir, para trabajar con el puntero `this`). ¿De dónde procede esta dirección?

Esto prueba que el compilador puede crear un objeto temporal cuando lo necesita para evaluar adecuadamente una expresión. En este caso, crea uno que ni siquiera se le ve actuar como destino para el valor ignorado retornado por `f()`. El tiempo de vida de este objeto temporal es tan corto como sea posible para que el programa no se llene de objetos temporales esperando a ser destruidos, lo cual provocaría la utilización ineficaz de recursos valiosos. En algunos casos, el objeto temporal podría pasarse inmediatamente a otra función, pero en este caso no se necesita después de la llamada a la función, así que en cuanto la función termina, llamando al destructor del objeto local (líneas 23 y 24), el objeto temporal también se destruye (líneas 25 y 26).

Finalmente, de la línea 28 a la línea 31, se destruye el objeto `h2`, seguido de `h` y el contador de objetos vuelve a cero.

11.3.3. El constructor de copia por defecto

Como el constructor de copia implementa el paso y retorno por valor, es importante que el compilador cree uno en el caso de estructuras simples (de hecho, es lo mismo que hace C). Sin embargo todo lo que se ha visto es el comportamiento por defecto: una copia bit a bit.

Cuando se utilizan tipos más complejos, el compilador de C++ creará un constructor de copia automáticamente si no se implementa explícitamente. De nuevo, una copia bit a bit no tiene sentido, porque no tiene porqué ser el comportamiento que se necesita.

He aquí un ejemplo para mostrar el comportamiento más inteligente del compilador. Suponga que crea una nueva clase compuesta por objetos de varias clases diferentes. A esto se le denomina *composición*, y es una de las formas en las que se pueden hacer nuevas clases a partir de las ya existentes. Ahora desempeñe el papel de un novato que trata de resolver un problema rápidamente creando una nueva clase de esta manera. No sabe nada sobre los constructores de copia, así que no lo implementa. El ejemplo muestra lo que el compilador hace cuando crea un constructor de copia por defecto para su nueva clase:

```
//: C11:DefaultCopyConstructor.cpp
// Automatic creation of the copy-constructor
#include <iostream>
#include <string>
using namespace std;

class WithCC { // With copy-constructor
public:
  // Explicit default constructor required:
  WithCC() {}
  WithCC(const WithCC&) {
    cout << "WithCC(WithCC&)" << endl;
  }
```

```
};

class WoCC { // Without copy-constructor
  string id;
public:
  WoCC(const string& ident = "") : id(ident) {}
  void print(const string& msg = "") const {
    if(msg.size() != 0) cout << msg << ": ";
    cout << id << endl;
  }
};

class Composite {
  WithCC withcc; // Embedded objects
  WoCC wocc;
public:
  Composite() : wocc("Composite()") {}
  void print(const string& msg = "") const {
    wocc.print(msg);
  }
};

int main() {
  Composite c;
  c.print("Contents of c");
  cout << "Calling Composite copy-constructor"
       << endl;
  Composite c2 = c;  // Calls copy-constructor
  c2.print("Contents of c2");
} ///:~
```

La clase `WithCC` contiene un constructor de copia, que simplemente anuncia que ha sido llamado, y esto demuestra una cuestión interesante: dentro de la clase `Composite` se crea un objeto tipo `WithCC` utilizando el constructor por defecto. Si `WithCC` no tuviera ningún constructor, el compilador crearía uno por defecto automáticamente, el cual, en este caso, no haría nada. No obstante, si añade un constructor por defecto, le está diciendo al compilador que ha de utilizar los constructores disponibles, por lo que él no crea ningún constructor por defecto y se quejará a no ser que explícitamente cree un constructor por defecto, como se hizo en `WithCC`.

La clase `WoCC` no tiene constructor de copia, pero su constructor almacenará un string interno imprimible por la función `print()`. La lista de inicialización del constructor (presentado brevemente en el Capítulo 8 y tratado completamente en el Capítulo 14) de `Composite` llama explícitamente a este constructor. La razón de esto se verá posteriormente.

La clase `Composite` tiene objetos miembro tanto de `WithCC` como de `WoCC` (note que el objeto interno `wocc` se inicializa en la lista de inicializadores del constructor de `Composite`, como debe ser), pero no están inicializados explícitamente en el constructor de copia. Sin embargo un objeto `Composite` se crea en `main()` utilizando el constructor de copia:

```
Composite c2 = c;
```

El compilador ha creado un constructor de copia para `Composite` automática-

mente, y la salida del programa revela la manera en que se crea:

```
Contents of c: Composite()
Calling Composite copy-constructor
WithCC(WithCC&)
Contents of c2: Composite()
```

Para la creación de un constructor de copia para una clase que utiliza composición (y herencia, que se trata en el Capítulo 14), el compilador llama a todos los constructores de copia de todos los miembros objeto y de las clases base de manera recursiva. Es decir, si el miembro también contiene otro objeto, también se llama a su constructor de copia. En el ejemplo, el compilador llama al constructor de copia de `WithCC`. La salida muestra que se llama a este constructor. Como `WoCC` no tiene constructor de copia, el compilador crea uno que realiza simplemente una copia bit a bit para que el constructor de copia de `Composite` lo pueda llamar. La llamada a `Composite::print()` en `main()` muestra que esto ocurre, porque el contenido de `c2.wocc` es idéntico al contenido de `c.wocc`. El proceso que realiza el compilador para crear un constructor de copia se denomina *inicialización de miembros* (*memberwise initialization*).

Se recomienda definir constructor de copia propio en vez de usar el que crea el compilador. Eso garantiza que estará bajo su control.

11.3.4. Alternativas a la construcción por copia

A estas alturas su cabeza debe estar echando humo, y se preguntará cómo es posible que pudiera escribir una clase que funcionase sin saber nada acerca del constructor de copia. No obstante, recuerde que el constructor de copia sólo es necesario cuando la clase se pasa *por valor*. Si esto no va a ocurrir, entonces no lo necesita.

Evitando el paso por valor

«Pero», puede decir, «si no defino el constructor de copia, el compilador lo creará por mí. ¿Cómo sé que un objeto nunca se pasará por valor?»

Existe una técnica simple para evitar el paso por valor: declare un constructor de copia `private`. Ni siquiera necesita definirlo (sólo declararlo), a no ser que un método o una función `friend` necesite realizar un paso por valor. Si el usuario intenta pasar o retornar el objeto por valor, el compilador se quejará con un error porque el constructor de copia es privado. El compilador ya no puede crear un constructor de copia por defecto porque explícitamente ya hay uno creado.

He aquí un ejemplo:

```
//: C11:NoCopyConstruction.cpp
// Preventing copy-construction

class NoCC {
  int i;
  NoCC(const NoCC&); // No definition
public:
  NoCC(int ii = 0) : i(ii) {}
};

void f(NoCC);
```

```
int main() {
  NoCC n;
//! f(n); // Error: copy-constructor called
//! NoCC n2 = n; // Error: c-c called
//! NoCC n3(n); // Error: c-c called
} ///:~
```

Note la utilización de la forma más general

```
NoCC(const NoCC&);
```

utilizando `const`

Funciones que modifican objetos externos

La sintaxis de referencias es más agradable que la de punteros, aunque oculte significado al que lea el código fuente. Por ejemplo, en la librería `iostreams` existe una versión sobrecargada de la función `get()` que tiene como argumento un `char &`, y su cometido es modificar ese argumento y utilizarlo como el valor que retorna `get()`. No obstante, si lee el código fuente de esta función, no es inmediatamente obvio que la variable que se pasa como argumento vaya a ser modificada:

```
char c;
cin.get(c);
```

Parece que a la función se le pasa por valor, lo que sugiere que el argumento que se pasa *no* se modifica.

A causa de esto, es probablemente más seguro, desde el punto de vista de mantenimiento del código fuente, utilizar punteros que pasen la dirección del argumento que se desee modificar. Si *siempre* pasa direcciones como referencias constantes *excepto* cuando intenta modificar el argumento que se pasa a través de la dirección, donde pasaría un puntero no constante, entonces es más fácil para el lector seguir el código fuente.

11.4. Punteros a miembros

Un puntero es una variable que contiene la dirección de alguna ubicación. Se puede cambiar a lo que el puntero apunta en tiempo de ejecución. La ubicación a la que apunta puede ser un dato o función. El *puntero a miembro* de C++ sigue el mismo concepto, excepto que a lo que apunta es una ubicación dentro de una clase. Pero surge el dilema de que un puntero necesita una dirección, pero no hay «dirección» alguna dentro de una clase; La selección de un miembro de una clase se realiza mediante un desplazamiento dentro de la clase. Pero primero hay que conocer la dirección donde comienza un objeto en particular para luego sumarle el desplazamiento y así localizar el miembro de la clase. La sintaxis de los punteros a miembros requiere que usted seleccione un objeto al mismo tiempo que está accediendo al contenido del puntero al miembro.

Para entender esta sintaxis, considere una estructura simple, con un puntero `s-p` y un objeto `so`. Puede seleccionar sus miembros de la misma manera que en el

siguiente ejemplo:

```
//: C11:SimpleStructure.cpp
struct Simple { int a; };
int main() {
  Simple so, *sp = &so;
  sp->a;
  so.a;
} ///:~
```

Ahora suponga que tiene un puntero normal que se llama `ip` y que apunta a un entero. Para acceder a lo que apunta `ip`, ha de estar precedido por un '*':

```
*ip=4;
```

Finalmente, se preguntará qué pasa si tiene un puntero que está apuntando a algo que está dentro de un objeto, incluso si lo que realmente representa es un desplazamiento dentro del objeto. Para acceder a lo que está apuntando, debe preceder el puntero con '*'. Pero como es un desplazamiento dentro de un objeto, también ha de referirse al objeto con el que estamos tratando. Así, el * se combina con el objeto. Por tanto, la nueva sintaxis se escribe ->* para un puntero que apunta a un objeto, y .* para un objeto o referencia, tal como esto:

```
objectPointer->*pointerToMember = 47;
object.*pointerToMember = 47;
```

Pero, ¿cuál es la sintaxis para definir el `pointerToMember`? Pues como cualquier puntero, tiene que decir el tipo al que apuntará, por lo que se utilizaría el '*' en la definición. La única diferencia es que debe decir a qué clase de objetos apuntará ese atributo puntero. Obviamente, esto se consigue con el nombre de la clase y el operador de resolución de ámbito. Así,

```
int ObjectClass::*pointerToMember;
```

define un atributo puntero llamado `pointerToMember` que apunta a cualquier entero dentro de `ObjectClass`. También puede inicializar el puntero cuando lo defina (o en cualquier otro momento):

```
int ObjectClass::*pointerToMember = &ObjectClass::a;
```

Realmente no existe una «dirección» de `ObjectClass::a` porque se está refiriendo a la clase y no a un objeto de esa clase. Así, `&ObjectClass::a` se puede utilizar sólo con la sintaxis de un puntero a miembro.

He aquí un ejemplo que muestra cómo crear y utilizar punteros a atributos:

```
//: C11:PointerToMemberData.cpp
#include <iostream>
using namespace std;

class Data {
```

```
public:
  int a, b, c;
  void print() const {
    cout << "a = " << a << ", b = " << b
         << ", c = " << c << endl;
  }
};

int main() {
  Data d, *dp = &d;
  int Data::*pmInt = &Data::a;
  dp->*pmInt = 47;
  pmInt = &Data::b;
  d.*pmInt = 48;
  pmInt = &Data::c;
  dp->*pmInt = 49;
  dp->print();
} ///:~
```

Obviamente, son muy desagradables de utilizar en cualquier lugar excepto para caso especiales (que es exactamente para lo que se crearon).

Además, los punteros a miembro son bastante limitados: pueden asignarse solamente a una ubicación específica dentro de una clase. No podría, por ejemplo, incrementarlos o compararlos tal como puede hacer con punteros normales.

11.4.1. Funciones

Un ejercicio similar se produce con la sintaxis de puntero a miembro para métodos. Un puntero a una función (presentado al final del Capítulo 3) se define como:

```
int (*fp)(float);
```

Los paréntesis que engloban a `(*fb)` son necesarios para que fuercen la evaluación de la definición apropiadamente. Sin ellos sería una función que devuelve un int*.

Los paréntesis también desempeñan un papel importante cuando se definen y utilizan punteros a métodos. Si tiene una función dentro de una clase, puede definir un puntero a ese método insertando el nombre de la clase y el operador de resolución de ámbito en una definición habitual de puntero a función:

```
//: C11:PmemFunDefinition.cpp
class Simple2 {
public:
  int f(float) const { return 1; }
};
int (Simple2::*fp)(float) const;
int (Simple2::*fp2)(float) const = &Simple2::f;
int main() {
  fp = &Simple2::f;
} ///:~
```

En la definición de `fp2` puede verse que el puntero a un método puede inicializarse cuando se crea, o en cualquier otro momento. A diferencia de las funciones no son miembros, el `&` *no* es opcional para obtener la dirección de un método. Sin embargo, se puede dar el identificador de la función sin la lista de argumentos, porque la sobrecarga se resuelve por el tipo de puntero a miembro.

Un ejemplo

Lo interesante de un puntero es que se puede cambiar el valor del mismo para apuntar a otro lugar en tiempo de ejecución, lo cual proporciona mucha flexibilidad en la programación porque a través de un puntero se puede cambiar el *comportamiento* del programa en tiempo de ejecución. Un puntero a miembro no es distinto; le permite elegir un miembro en tiempo de ejecución. Típicamente, sus clases sólo tendrán métodos visibles públicamente (los atributos normalmente se consideran parte de la implementación que va oculta), de modo que el siguiente ejemplo elige métodos en tiempo de ejecución.

```
//: C11:PointerToMemberFunction.cpp
#include <iostream>
using namespace std;

class Widget {
public:
  void f(int) const { cout << "Widget::f()\n"; }
  void g(int) const { cout << "Widget::g()\n"; }
  void h(int) const { cout << "Widget::h()\n"; }
  void i(int) const { cout << "Widget::i()\n"; }
};

int main() {
  Widget w;
  Widget* wp = &w;
  void (Widget::*pmem)(int) const = &Widget::h;
  (w.*pmem)(1);
  (wp->*pmem)(2);
} ///:~
```

Por supuesto, no es razonable esperar que el usuario casual cree expresiones tan complejas. Si el usuario necesita manipular directamente un puntero a miembro, los `typedef` vienen al rescate. Para dejar aún mejor las cosas, puede utilizar un puntero a función como parte del mecanismo interno de la implementación. He aquí un ejemplo que utiliza un puntero a miembro *dentro* de la clase. Todo lo que el usuario necesita es pasar un número para elegir una función. [1]

```
//: C11:PointerToMemberFunction2.cpp
#include <iostream>
using namespace std;

class Widget {
  void f(int) const { cout << "Widget::f()\n"; }
  void g(int) const { cout << "Widget::g()\n"; }
  void h(int) const { cout << "Widget::h()\n"; }
  void i(int) const { cout << "Widget::i()\n"; }
```

[1] Gracias a Owen Mortensen por este ejemplo

```
  enum { cnt = 4 };
  void (Widget::*fptr[cnt])(int) const;
public:
  Widget() {
    fptr[0] = &Widget::f; // Full spec required
    fptr[1] = &Widget::g;
    fptr[2] = &Widget::h;
    fptr[3] = &Widget::i;
  }
  void select(int i, int j) {
    if(i < 0 || i >= cnt) return;
    (this->*fptr[i])(j);
  }
  int count() { return cnt; }
};

int main() {
  Widget w;
  for(int i = 0; i < w.count(); i++)
    w.select(i, 47);
} ///:~
```

En la interfaz de la clase y en `main()`, puede observar que toda la implementación, funciones incluidas, es privada. El código ha de pedir el `count()` de las funciones. De esta manera, el que implementa la clase puede cambiar la cantidad de funciones en la implementación por debajo sin que afecte al código que utilice la clase.

La inicialización de los punteros a miembro en el constructor puede parecer redundante. ¿No debería ser capaz de poner

```
fptr[1] = &g;
```

porque el nombre `g` es un método, el cual está en el ámbito de la clase? El problema aquí es que no sería conforme a la sintaxis de puntero a miembro. Así todo el mundo, incluido el compilador, puede imaginarse qué está pasando. De igual forma, cuando se accede al contenido del puntero a miembro, parece que

```
(this->*fptr[i])(j);
```

también es redundante; `this` parece redundante. La sintaxis necesita que un puntero a miembro siempre esté ligado a un objeto cuando se accede al contenido al que apunta.

11.5. Resumen

Los punteros en C++ son casi idénticos a los punteros en C, lo cual es bueno. De otra manera, gran cantidad de código C no compilaría en C++. Los únicos errores en tiempo de compilación serán aquéllos que realicen asignaciones peligrosas. Esos errores pueden eliminarse con una simple (pero explícito!) molde.

C++ también añade la *referencia* de Algol y Pascal, que es como un puntero constante que el compilador hace que se acceda directamente al contenido al que apunta.

Una referencia contiene una dirección, pero se trata como un objeto. Las referencias son esenciales para una sintaxis clara con la sobrecarga de operadores (el tema del siguiente capítulo), pero también proporcionan mejoras sintácticas para el paso y retorno de objetos en funciones normales.

El constructor de copia coge una referencia a un objeto existente del mismo tipo que el argumento, y lo utiliza para la creación de un nuevo objeto a partir del que existente. El compilador llama automáticamente al constructor de copia cuando pasa o retorna un objeto por valor. Aunque el compilador crea un constructor de copia automáticamente, si cree que su clase necesita uno, debería definirlo para asegurar un comportamiento apropiado. Si no desea que el objeto se pase o retorne por valor, debería crear un constructor de copia privado.

Los punteros a miembro tienen la misma capacidad que los punteros normales: puede elegir una región de memoria particular (atributo o método) en tiempo de ejecución. Los punteros a miembro funcionan con los miembros de una clase en vez de variables o funciones globales. Ofrecen la suficiente flexibilidad para cambiar el comportamiento en tiempo de ejecución.

11.6. Ejercicios

Las soluciones a los ejercicios se pueden encontrar en el documento electrónico titulado «The Thinking in C++ Annotated Solution Guide», disponible por poco dinero en www.BruceEckel.com.

1. Convierta el fragmento de código «bird & rock» del principio de este capítulo a un programa C (utilizando estructuras para los tipos de datos), y que compile. Ahora intente compilarlo con un compilador de C++ y vea qué ocurre.

2. Coja los fragmentos de código al principio de la sección titulada «Referencias en C++» y póngalos en un `main()`. Añada sentencias para imprimir en la salida para que pueda demostrar usted mismo que las referencias son como punteros que se dereferencian automáticamente.

3. Escriba un programa en el cual intente (1) Crear una referencia que no esté inicializada cuando se crea. (2) Cambiar una referencia para que se refiera a otro objeto después de que se haya inicializado. (3) Crear una referencia nula.

4. Escriba una función que tome un puntero por argumento, modifique el contenido de lo que el apunta puntero, y retorne ese mismo contenido como si de una referencia se tratara.

5. Cree una nueva clase con algunos métodos, y haga que el objeto sea apuntado por el argumento del Ejercicio 4. Haga que el puntero pasado como argumento y algunos métodos sean constantes y pruebe que sólo puede llamar a los métodos constantes dentro de su función. Haga que el argumento de su función sea una referencia en vez de un puntero.

6. Coja los fragmentos de código al principio de la sección «referencias a puntero» y conviértalos en un programa.

7. Cree una función que tome como argumento una referencia a un puntero que apunta a otro puntero y modifique ese argumento. En `main()`, llame a la función.

8. Cree una función que toma un argumento del tipo char& y lo modifica. En el `main()` imprima a la salida una variable char, llame a su función con esa variable e imprima la variable de nuevo para demostrar que ha sido cambiada. ¿Cómo afecta esto a la legibilidad del programa?

9. Escriba una clase que tiene un método constante y otra que no lo tiene. Escriba tres funciones que toman un objeto de esa clase como argumento; la primera lo toma por valor, la segunda lo toma por referencia y la tercera lo toma mediante una referencia constante. Dentro de las funciones, intente llamar a las dos funciones de su clase y explique los resultados.

10. (Algo difícil) Escriba una función simple que toma un entero como argumento, incrementa el valor, y lo retorna. En `main()`, llame a su función. Intente que el compilador genere el código ensamblador e intente entender cómo los argumentos se pasan y se retornan, y cómo las variables locales se colocan en la pila.

11. Escriba una función que tome como argumentos un char, int, float y double. Genere el código ensamblador con su compilador y busque las instrucciones que apilan los argumentos en la pila antes de efectuar la llamada a función.

12. Escriba una función que devuelva un double. Genere el código ensamblador y explique cómo se retorna el valor.

13. Genere el código ensamblador de `PassingBigStructures.cpp`. Recorra y desmitifique la manera en que su compilador genera el código para pasar y devolver estructuras grandes.

14. Escriba una simple función recursiva que disminuya su argumento y retorne cero si el argumento llega a cero, o en otro caso que se llame a sí misma. Genere el código ensamblador para esta función y explique la forma en el compilador implementa la recurrencia.

15. Escriba código para demostrar que el compilador genera un constructor de copia automáticamente en caso de que no lo haga el programador. Demuestre que el constructor de copia generado por el compilador realiza una copia bit a bit de tipos primitivos y llama a los constructores de copia de los tipos definidos por el usuario.

16. Escriba una clase en la que el constructor de copia se anuncia a sí mismo a través de `cout`. Ahora cree una función que pasa un objeto de su nueva clase por valor y otra más que crea un objeto local de su nueva clase y lo devuelve por valor. Llame a estas funciones para demostrar que el constructor de copia es, en efecto, llamado cuando se pasan y retornan objetos por valor.

17. Cree un objeto que contenga un double*. Que el constructor inicialice el double* llamando a `new double` y asignando un valor. Entonces, que el destructor imprima el valor al que apunta, asigne ese valor a -1, llame a `delete` para liberar la memoria y ponga el puntero a cero. Ahora cree una función que tome un objeto de su clase por valor, y llame a esta función desde `main()`. ¿Qué ocurre? Solucione el problema escribiendo un constructor de copia.

18. Cree una clase con un constructor que parezca un constructor de copia, pero que tenga un argumento adicional con un valor por defecto. Muestre que a pesar de eso se utiliza como constructor de copia.

19. Cree una clase con un constructor de copia que se anuncie a sí mismo (es decir que imprima por la salida que ha sido llamado). Haga una segunda clase que contenga un objeto miembro de la primera clase, pero no cree un constructor de copia. Demuestre que el constructor de copia, que genera automáticamente el compilador en la segunda clase, llama al constructor de copia de la primera.

20. Cree una clase muy simple, y una función que devuelva un objeto de esa clase por valor. Cree una segunda función que tome una referencia de un objeto de su clase. Llame a la segunda función pasándole como argumento una llamada a la primera función, y demuestre que la segunda función debe utilizar una referencia constante como argumento.

21. Cree una clase simple sin constructor de copia, y una función simple que tome un objeto de esa clase por valor. Ahora cambie su clase añadiéndole una declaración (sólo declare, no defina) privada de un constructor de copia. Explique lo que ocurre cuando compila la función.

22. Este ejercicio crea una alternativa a la utilización del constructor de copia. Cree una clase `X` y declare (pero no defina) un constructor de copia privado. Haga una función `clone()` pública como un método constante que devuelve una copia del objeto creado utilizando `new`. Ahora escriba una función que tome como argumento un const X& y clone una copia local que puede modificarse. El inconveniente de esto es que es el programador el responsable de destruir explícitamente el objeto clonado (utilizando `delete`) cuando haya terminado con él.

23. Explique qué está mal en `Mem.cpp` y `MemTest.cpp` del Capítulo 7. Solucione el problema.

24. Cree una clase que contenga un double y una función `print()` que imprima el double. Cree punteros a miembro tanto para el atributo como al método de su clase. Cree un objeto de su clase y un puntero a ese objeto, y manipule ambos elementos de la clase a través de los punteros a miembro, utilizando tanto el objeto como el puntero al objeto.

25. Cree una clase que contenga un array de enteros. ¿Puede recorrer el array mediante un puntero a miembro?

26. Modifique `PmemFunDefinition.cpp` añadiendo un método `f()` sobrecargado (puede determinar la lista de argumentos que cause la sobrecarga). Ahora cree un segundo puntero a miembro, asígnelo a la versión sobrecargada de `f()`, y llame al método a través del puntero. ¿Cómo sucede la resolución de la función sobrecargada en este caso?

27. Empiece con la función `FunctionTable.cpp` del Capítulo 3. Cree una clase que contenga un vector de punteros a funciones, con métodos `add()` y `remove()` para añadir y quitar punteros a función. Añada una función denominada `run()` que recorra el vector y llame a todas la funciones.

28. Modifique el Ejercicio 27 para que funcione con punteros a métodos.

12: Sobrecarga de operadores

La sobrecarga de operadores es solamente «azúcar sintáctico», lo que significa que es simplemente otra manera de invocar funciones.

La diferencia es que los argumentos para estas funciones no aparecen entre paréntesis, sino que rodean o siguen a los caracteres que siempre pensó como operadores inalterables.

Hay dos diferencias entre el uso de un operador y el de una llamada a función normal. La sintaxis es diferente: un operador es a menudo «llamado» situándolo entre (o después de) los argumentos. La segunda diferencia es que el compilador determina qué «función» llamar. Por ejemplo, si está usando el operador + con argumentos de punto flotante, el compilador «llama» a la función para realizar una suma de punto flotante (esta «llamada» normalmente consiste en insertar código en linea, o una instrucción de punto flotante del procesador). Si usa el operador + con un número de punto flotante y un entero, el compilador «llama» a una función especial para convertir el int a un float,y entonces «llama» a la función de suma en punto flotante.

Sin embargo, en C++ es posible definir nuevos operadores que trabajen con clases. Esta definición es exactamente como la definición de una función ordinaria, excepto que el nombre de la función consiste en la palabra reservada `operator` seguida del operador. Siendo esta la única diferencia, el operador se convierte en una función como otra cualquiera que el compilador llama cuando ve el prototipo adecuado.

12.1. Precaución y tranquilidad

Es tentador convertirse en un super-entusiasta de la sobrecarga de operadores. Son un juguete divertido, al principio. Pero recuerde que es sólo un endulzamiento sintáctico, otra manera de llamar a una función. Mirándolo desde esa perspectiva, no hay razón para sobrecargar un operador excepto si eso hace al código implicado con la clase más sencillo e intuitivo de escribir y especialmente de leer. (Recuerde, el código se lee mucho más que se escribe). Si éste no es el caso no se moleste.

Otra reacción cmún frente al uso de la sobrecarga de operadores es el pánico: de repente, los operadores de C pierden su significado familiar.«¡Todo ha cambiado y mi código C por completo hará cosas diferentes!». Esto no es verdad. Todos los operadores usados en expresiones que contienen solo tipos de datos incorporados no pueden ser cambiados. Nunca podrá sobrecargar operadores así:

```
1 << 4;
```

para que se comporten de forman diferente, o que

```
  1.414 << 2;
```

tenga significado. Sólo una expresión que contenga tipos de datos definidos por el usuario podrá tener operadores sobrecargados.

12.2. Sintaxis

Definir un operador sobrecargado es como definir una función, pero el nombre de esa función es `operator@` en la que `@` representa el operador que está siendo sobrecargado. El número de argumentos en la lista de argumentos del operador sobrecargado depende de dos factores:

1. Si es un operador unario (un argumento) o un operador binario (dos argumentos)
2. Si el operador está definido como una función global (un argumento para los unarios, dos para los binarios) o un método (cero argumentos para los unarios y uno para los binarios. En este último caso el objeto (`this`) se convierte en el argumento del lado izquierdo al operador).

He aquí una pequeña clase que muestra la sintaxis de la sobrecarga de operadores:

```
//: C12:OperatorOverloadingSyntax.cpp
#include <iostream>
using namespace std;

class Integer {
  int i;
public:
  Integer(int ii) : i(ii) {}
  const Integer
  operator+(const Integer& rv) const {
    cout << "operator+" << endl;
    return Integer(i + rv.i);
  }
  Integer&
  operator+=(const Integer& rv) {
    cout << "operator+=" << endl;
    i += rv.i;
    return *this;
  }
};

int main() {
  cout << "built-in types:" << endl;
  int i = 1, j = 2, k = 3;
  k += i + j;
  cout << "user-defined types:" << endl;
  Integer ii(1), jj(2), kk(3);
  kk += ii + jj;
} ///:~
```

Los dos operadores sobrecargados están definidos como métodos en línea que imprimen un mensaje al ser llamados. El único argumento de estas funciones miembro será el que aparezca del lado derecho del operador binario. Los operadores unarios no tienen argumentos cuando se definen como métodos. El método es invocado por el objeto de la parte izquierda del operador.

Para los operadores incondicionales (los condicionales generalmente devuelven un valor booleano), generalmente se deseará devolver un objeto o una referencia del mismo tipo que está operando, si los dos argumentos son del mismo tipo. (Si no son del mismo tipo, la interpretación de lo que debería pasar es responsabilidad suya). De esta manera, se pueden construir expresiones tan complicadas como la siguiente:

```
    kk += ii + jj;
```

La expresión `operator+` crea un nuevo objeto `Integer` (un temporario) que se usa como argumento `rv` para el operador `operator+=`. Este objeto temporal se destruye tan pronto como deja de necesitarse.

12.3. Operadores sobrecargables

Aunque puede sobrecargar casi todos los operadores disponibles en C, el uso de operadores sobrecargados es bastante restrictivo. En particular, no puede combinar operadores que actualmente no tienen significado en C (como `**` para representar la potencia), no puede cambiar la precedencia de evaluación de operadores, y tampoco el número de argumentos requeridos por un operador. Estas restricciones existen para prevenir que la creación de nuevos operadores ofusquen el significado en lugar de clarificarlo.

Las siguientes dos subsecciones muestran ejemplos de todos los operadores normales, sobrecargados en la forma habitual.

12.3.1. Operadores unarios

El siguiente ejemplo muestra la sintaxis para sobrecargar todos los operadores unarios, en ambas formas: como funciones globales (funciones `friend`, no métodos) y como métodos. Estas expandirán la clase `Integer` vista previamente y añadirá una nueva clase `byte`. El significado de sus operadores particulares dependerá de la forma en que los use, pero considere a los programadores del grupo antes de hacer algo inesperado. He aquí un catálogo de todas las funciones unarias:

```
//: C12:OverloadingUnaryOperators.cpp
#include <iostream>
using namespace std;

// Non-member functions:
class Integer {
  long i;
  Integer* This() { return this; }
public:
  Integer(long ll = 0) : i(ll) {}
  // No side effects takes const& argument:
```

```
  friend const Integer&
    operator+(const Integer& a);
  friend const Integer
    operator-(const Integer& a);
  friend const Integer
    operator~(const Integer& a);
  friend Integer*
    operator&(Integer& a);
  friend int
    operator!(const Integer& a);
  // Side effects have non-const& argument:
  // Prefix:
  friend const Integer&
    operator++(Integer& a);
  // Postfix:
  friend const Integer
    operator++(Integer& a, int);
  // Prefix:
  friend const Integer&
    operator--(Integer& a);
  // Postfix:
  friend const Integer
    operator--(Integer& a, int);
};

// Global operators:
const Integer& operator+(const Integer& a) {
  cout << "+Integer\n";
  return a; // Unary + has no effect
}
const Integer operator-(const Integer& a) {
  cout << "-Integer\n";
  return Integer(-a.i);
}
const Integer operator~(const Integer& a) {
  cout << "~Integer\n";
  return Integer(~a.i);
}
Integer* operator&(Integer& a) {
  cout << "&Integer\n";
  return a.This(); // &a is recursive!
}
int operator!(const Integer& a) {
  cout << "!Integer\n";
  return !a.i;
}
// Prefix; return incremented value
const Integer& operator++(Integer& a) {
  cout << "++Integer\n";
  a.i++;
  return a;
}
// Postfix; return the value before increment:
const Integer operator++(Integer& a, int) {
  cout << "Integer++\n";
  Integer before(a.i);
  a.i++;
  return before;
```

```
}
// Prefix; return decremented value
const Integer& operator--(Integer& a) {
  cout << "--Integer\n";
  a.i--;
  return a;
}
// Postfix; return the value before decrement:
const Integer operator--(Integer& a, int) {
  cout << "Integer--\n";
  Integer before(a.i);
  a.i--;
  return before;
}

// Show that the overloaded operators work:
void f(Integer a) {
  +a;
  -a;
  ~a;
  Integer* ip = &a;
  !a;
  ++a;
  a++;
  --a;
  a--;
}

// Member functions (implicit "this"):
class Byte {
  unsigned char b;
public:
  Byte(unsigned char bb = 0) : b(bb) {}
  // No side effects: const member function:
  const Byte& operator+() const {
    cout << "+Byte\n";
    return *this;
  }
  const Byte operator-() const {
    cout << "-Byte\n";
    return Byte(-b);
  }
  const Byte operator~() const {
    cout << "~Byte\n";
    return Byte(~b);
  }
  Byte operator!() const {
    cout << "!Byte\n";
    return Byte(!b);
  }
  Byte* operator&() {
    cout << "&Byte\n";
    return this;
  }
  // Side effects: non-const member function:
  const Byte& operator++() { // Prefix
    cout << "++Byte\n";
    b++;
```

```
    return *this;
  }
  const Byte operator++(int) { // Postfix
    cout << "Byte++\n";
    Byte before(b);
    b++;
    return before;
  }
  const Byte& operator--() { // Prefix
    cout << "--Byte\n";
    --b;
    return *this;
  }
  const Byte operator--(int) { // Postfix
    cout << "Byte--\n";
    Byte before(b);
    --b;
    return before;
  }
};

void g(Byte b) {
  +b;
  -b;
  ~b;
  Byte* bp = &b;
  !b;
  ++b;
  b++;
  --b;
  b--;
}

int main() {
  Integer a;
  f(a);
  Byte b;
  g(b);
} ///:~
```

Las funciones están agrupadas de acuerdo a la forma en que se pasan los argumentos. Más tarde se darán unas cuantas directrices de cómo pasar y devolver argumentos. Las clases expuestas anteriormente (y las que siguen en la siguiente sección) son las típicas, así que empiece con ellas como un patrón cuando sobrecargue sus propios operadores.

Incremento y decremento

Los operadores de incremento++ y de decremento −− provocan un conflicto porque querrá ser capaz de llamar diferentes funciones dependiendo de si aparecen antes (prefijo) o después (postfijo) del objeto sobre el que actúan. La solución es simple, pero la gente a veces lo encuentra un poco confusa inicialmente. Cuando el compilador ve, por ejemplo, `++a` (un pre-incremento), genera una llamada al `operator++(a)` pero cuando ve `a++`, genera una llamada a `operator++(a, int)`. Así es como el compilador diferencia entre los dos tipos, generando llamadas a funciones sobrecargadas diferentes. En `OverloadingUnaryOperators.cpp` para la versión

de funciones miembro, si el compilador ve `++b`, genera una llamada a `B::operator++()` y si ve `b++`genera una llamada a `B::operator++(int)`.

Todo lo que el usuario ve es que se llama a una función diferente para las versiones postfija y prefija. Internamente, sin embargo, las dos llamadas de funciones tienen diferentes firmas, así que conectan con dos cuerpos diferentes. El compilador pasa un valor constante ficticio para el argumento int (el cual nunca se proporcionada por un identificador porque el valor nunca se usa) para generar las diferentes firmas para la versión postfija.

12.3.2. Operadores binarios

Los siguientes listados repiten el ejemplo de `OverloadingUnaryOperators.cpp` para los operadores binarios presentándole un ejemplo de todos los operadores que pueda querer sobrecargar. De nuevo se muestran ambas versiones, la global y la de método.

```
//: C12:Integer.h
// Non-member overloaded operators
#ifndef INTEGER_H
#define INTEGER_H
#include <iostream>

// Non-member functions:
class Integer {
  long i;
public:
  Integer(long ll = 0) : i(ll) {}
  // Operators that create new, modified value:
  friend const Integer
    operator+(const Integer& left,
              const Integer& right);
  friend const Integer
    operator-(const Integer& left,
              const Integer& right);
  friend const Integer
    operator*(const Integer& left,
              const Integer& right);
  friend const Integer
    operator/(const Integer& left,
              const Integer& right);
  friend const Integer
    operator%(const Integer& left,
              const Integer& right);
  friend const Integer
    operator^(const Integer& left,
              const Integer& right);
  friend const Integer
    operator&(const Integer& left,
              const Integer& right);
  friend const Integer
    operator|(const Integer& left,
              const Integer& right);
  friend const Integer
    operator<<(const Integer& left,
               const Integer& right);
  friend const Integer
```

```
    operator>>(const Integer& left,
              const Integer& right);
  // Assignments modify & return lvalue:
  friend Integer&
    operator+=(Integer& left,
              const Integer& right);
  friend Integer&
    operator-=(Integer& left,
              const Integer& right);
  friend Integer&
    operator*=(Integer& left,
              const Integer& right);
  friend Integer&
    operator/=(Integer& left,
              const Integer& right);
  friend Integer&
    operator%=(Integer& left,
              const Integer& right);
  friend Integer&
    operator^=(Integer& left,
              const Integer& right);
  friend Integer&
    operator&=(Integer& left,
              const Integer& right);
  friend Integer&
    operator|=(Integer& left,
              const Integer& right);
  friend Integer&
    operator>>=(Integer& left,
               const Integer& right);
  friend Integer&
    operator<<=(Integer& left,
               const Integer& right);
  // Conditional operators return true/false:
  friend int
    operator==(const Integer& left,
              const Integer& right);
  friend int
    operator!=(const Integer& left,
              const Integer& right);
  friend int
    operator<(const Integer& left,
             const Integer& right);
  friend int
    operator>(const Integer& left,
             const Integer& right);
  friend int
    operator<=(const Integer& left,
              const Integer& right);
  friend int
    operator>=(const Integer& left,
              const Integer& right);
  friend int
    operator&&(const Integer& left,
              const Integer& right);
  friend int
    operator||(const Integer& left,
              const Integer& right);
```

```
  // Write the contents to an ostream:
  void print(std::ostream& os) const { os << i; }
};
#endif // INTEGER_H ///:~
```

```
//: C12:Integer.cpp {O}
// Implementation of overloaded operators
#include "Integer.h"
#include "../require.h"

const Integer
  operator+(const Integer& left,
            const Integer& right) {
  return Integer(left.i + right.i);
}
const Integer
  operator-(const Integer& left,
            const Integer& right) {
  return Integer(left.i - right.i);
}
const Integer
  operator*(const Integer& left,
            const Integer& right) {
  return Integer(left.i * right.i);
}
const Integer
  operator/(const Integer& left,
            const Integer& right) {
  require(right.i != 0, "divide by zero");
  return Integer(left.i / right.i);
}
const Integer
  operator%(const Integer& left,
            const Integer& right) {
  require(right.i != 0, "modulo by zero");
  return Integer(left.i % right.i);
}
const Integer
  operator^(const Integer& left,
            const Integer& right) {
  return Integer(left.i ^ right.i);
}
const Integer
  operator&(const Integer& left,
            const Integer& right) {
  return Integer(left.i & right.i);
}
const Integer
  operator|(const Integer& left,
            const Integer& right) {
  return Integer(left.i | right.i);
}
const Integer
  operator<<(const Integer& left,
             const Integer& right) {
  return Integer(left.i << right.i);
```

```
}
const Integer
  operator>>(const Integer& left,
             const Integer& right) {
  return Integer(left.i >> right.i);
}
// Assignments modify & return lvalue:
Integer& operator+=(Integer& left,
                    const Integer& right) {
   if(&left == &right) {/* self-assignment */}
   left.i += right.i;
   return left;
}
Integer& operator-=(Integer& left,
                    const Integer& right) {
   if(&left == &right) {/* self-assignment */}
   left.i -= right.i;
   return left;
}
Integer& operator*=(Integer& left,
                    const Integer& right) {
   if(&left == &right) {/* self-assignment */}
   left.i *= right.i;
   return left;
}
Integer& operator/=(Integer& left,
                    const Integer& right) {
   require(right.i != 0, "divide by zero");
   if(&left == &right) {/* self-assignment */}
   left.i /= right.i;
   return left;
}
Integer& operator%=(Integer& left,
                    const Integer& right) {
   require(right.i != 0, "modulo by zero");
   if(&left == &right) {/* self-assignment */}
   left.i %= right.i;
   return left;
}
Integer& operator^=(Integer& left,
                    const Integer& right) {
   if(&left == &right) {/* self-assignment */}
   left.i ^= right.i;
   return left;
}
Integer& operator&=(Integer& left,
                    const Integer& right) {
   if(&left == &right) {/* self-assignment */}
   left.i &= right.i;
   return left;
}
Integer& operator|=(Integer& left,
                    const Integer& right) {
   if(&left == &right) {/* self-assignment */}
   left.i |= right.i;
   return left;
}
Integer& operator>>=(Integer& left,
```

```
                      const Integer& right) {
   if(&left == &right) {/* self-assignment */}
   left.i >>= right.i;
   return left;
}
Integer& operator<<=(Integer& left,
                      const Integer& right) {
   if(&left == &right) {/* self-assignment */}
   left.i <<= right.i;
   return left;
}
// Conditional operators return true/false:
int operator==(const Integer& left,
               const Integer& right) {
    return left.i == right.i;
}
int operator!=(const Integer& left,
               const Integer& right) {
    return left.i != right.i;
}
int operator<(const Integer& left,
              const Integer& right) {
    return left.i < right.i;
}
int operator>(const Integer& left,
              const Integer& right) {
    return left.i > right.i;
}
int operator<=(const Integer& left,
               const Integer& right) {
    return left.i <= right.i;
}
int operator>=(const Integer& left,
               const Integer& right) {
    return left.i >= right.i;
}
int operator&&(const Integer& left,
               const Integer& right) {
    return left.i && right.i;
}
int operator||(const Integer& left,
               const Integer& right) {
    return left.i || right.i;
} ///:~
```

```
//: C12:IntegerTest.cpp
//{L} Integer
#include "Integer.h"
#include <fstream>
using namespace std;
ofstream out("IntegerTest.out");

void h(Integer& c1, Integer& c2) {
  // A complex expression:
  c1 += c1 * c2 + c2 % c1;
  #define TRY(OP) \
```

```
    out << "c1 = "; c1.print(out); \
    out << ", c2 = "; c2.print(out); \
    out << ";  c1 " #OP " c2 produces "; \
    (c1 OP c2).print(out); \
    out << endl;
  TRY(+) TRY(-) TRY(*) TRY(/)
  TRY(%) TRY(^) TRY(&) TRY(|)
  TRY(<<) TRY(>>) TRY(+=) TRY(-=)
  TRY(*=) TRY(/=) TRY(%=) TRY(^=)
  TRY(&=) TRY(|=) TRY(>>=) TRY(<<=)
  // Conditionals:
  #define TRYC(OP) \
    out << "c1 = "; c1.print(out); \
    out << ", c2 = "; c2.print(out); \
    out << ";  c1 " #OP " c2 produces "; \
    out << (c1 OP c2); \
    out << endl;
  TRYC(<) TRYC(>) TRYC(==) TRYC(!=) TRYC(<=)
  TRYC(>=) TRYC(&&) TRYC(||)
}

int main() {
  cout << "friend functions" << endl;
  Integer c1(47), c2(9);
  h(c1, c2);
} ///:~
```

```
//: C12:Byte.h
// Member overloaded operators
#ifndef BYTE_H
#define BYTE_H
#include "../require.h"
#include <iostream>
// Member functions (implicit "this"):
class Byte {
  unsigned char b;
public:
  Byte(unsigned char bb = 0) : b(bb) {}
  // No side effects: const member function:
  const Byte
    operator+(const Byte& right) const {
    return Byte(b + right.b);
  }
  const Byte
    operator-(const Byte& right) const {
    return Byte(b - right.b);
  }
  const Byte
    operator*(const Byte& right) const {
    return Byte(b * right.b);
  }
  const Byte
    operator/(const Byte& right) const {
    require(right.b != 0, "divide by zero");
    return Byte(b / right.b);
  }
```

```
  const Byte
    operator%(const Byte& right) const {
    require(right.b != 0, "modulo by zero");
    return Byte(b % right.b);
  }
  const Byte
    operator^(const Byte& right) const {
    return Byte(b ^ right.b);
  }
  const Byte
    operator&(const Byte& right) const {
    return Byte(b & right.b);
  }
  const Byte
    operator|(const Byte& right) const {
    return Byte(b | right.b);
  }
  const Byte
    operator<<(const Byte& right) const {
    return Byte(b << right.b);
  }
  const Byte
    operator>>(const Byte& right) const {
    return Byte(b >> right.b);
  }
  // Assignments modify & return lvalue.
  // operator= can only be a member function:
  Byte& operator=(const Byte& right) {
    // Handle self-assignment:
    if(this == &right) return *this;
    b = right.b;
    return *this;
  }
  Byte& operator+=(const Byte& right) {
    if(this == &right) {/* self-assignment */}
    b += right.b;
    return *this;
  }
  Byte& operator-=(const Byte& right) {
    if(this == &right) {/* self-assignment */}
    b -= right.b;
    return *this;
  }
  Byte& operator*=(const Byte& right) {
    if(this == &right) {/* self-assignment */}
    b *= right.b;
    return *this;
  }
  Byte& operator/=(const Byte& right) {
    require(right.b != 0, "divide by zero");
    if(this == &right) {/* self-assignment */}
    b /= right.b;
    return *this;
  }
  Byte& operator%=(const Byte& right) {
    require(right.b != 0, "modulo by zero");
    if(this == &right) {/* self-assignment */}
    b %= right.b;
```

```
    return *this;
  }
  Byte& operator^=(const Byte& right) {
    if(this == &right) {/* self-assignment */}
    b ^= right.b;
    return *this;
  }
  Byte& operator&=(const Byte& right) {
    if(this == &right) {/* self-assignment */}
    b &= right.b;
    return *this;
  }
  Byte& operator|=(const Byte& right) {
    if(this == &right) {/* self-assignment */}
    b |= right.b;
    return *this;
  }
  Byte& operator>>=(const Byte& right) {
    if(this == &right) {/* self-assignment */}
    b >>= right.b;
    return *this;
  }
  Byte& operator<<=(const Byte& right) {
    if(this == &right) {/* self-assignment */}
    b <<= right.b;
    return *this;
  }
  // Conditional operators return true/false:
  int operator==(const Byte& right) const {
      return b == right.b;
  }
  int operator!=(const Byte& right) const {
      return b != right.b;
  }
  int operator<(const Byte& right) const {
      return b < right.b;
  }
  int operator>(const Byte& right) const {
      return b > right.b;
  }
  int operator<=(const Byte& right) const {
      return b <= right.b;
  }
  int operator>=(const Byte& right) const {
      return b >= right.b;
  }
  int operator&&(const Byte& right) const {
      return b && right.b;
  }
  int operator||(const Byte& right) const {
      return b || right.b;
  }
  // Write the contents to an ostream:
  void print(std::ostream& os) const {
    os << "0x" << std::hex << int(b) << std::dec;
  }
};
#endif // BYTE_H ///:~
```

```
//: C12:ByteTest.cpp
#include "Byte.h"
#include <fstream>
using namespace std;
ofstream out("ByteTest.out");

void k(Byte& b1, Byte& b2) {
  b1 = b1 * b2 + b2 % b1;

  #define TRY2(OP) \
    out << "b1 = "; b1.print(out); \
    out << ", b2 = "; b2.print(out); \
    out << ";  b1 " #OP " b2 produces "; \
    (b1 OP b2).print(out); \
    out << endl;

  b1 = 9; b2 = 47;
  TRY2(+) TRY2(-) TRY2(*) TRY2(/)
  TRY2(%) TRY2(^) TRY2(&) TRY2(|)
  TRY2(<<) TRY2(>>) TRY2(+=) TRY2(-=)
  TRY2(*=) TRY2(/=) TRY2(%=) TRY2(^=)
  TRY2(&=) TRY2(|=) TRY2(>>=) TRY2(<<=)
  TRY2(=) // Assignment operator

  // Conditionals:
  #define TRYC2(OP) \
    out << "b1 = "; b1.print(out); \
    out << ", b2 = "; b2.print(out); \
    out << ";  b1 " #OP " b2 produces "; \
    out << (b1 OP b2); \
    out << endl;

  b1 = 9; b2 = 47;
  TRYC2(<) TRYC2(>) TRYC2(==) TRYC2(!=) TRYC2(<=)
  TRYC2(>=) TRYC2(&&) TRYC2(||)

  // Chained assignment:
  Byte b3 = 92;
  b1 = b2 = b3;
}

int main() {
  out << "member functions:" << endl;
  Byte b1(47), b2(9);
  k(b1, b2);
} ///:~
```

Puede ver que `operator=` solo puede ser un método. Esto se explica después.

Fíjese que todos los operadores de asignación tienen código para comprobar la auto asignación; ésta es una directiva general. En algunos casos esto no es necesario; por ejemplo, con `operator+=` a menudo querrá decir `A += A` y sumar `A` a sí mismo. El lugar más importante para situar las comprobaciones para la auto asignación es `operator=` porque con objetos complicados pueden ocurrir resultados desastro-

sos. (En algunos casos es correcto, pero siempre debería tenerlo en mente cuando escriba `operator=`).

Todos los operadores mostrados en los dos ejemplos previos son sobrecargados para manejar un tipo simple. También es posible sobrecargar operadores para manejar tipos compuestos, de manera que pueda sumar manzanas a naranjas, por ejemplo. Antes de que empiece una sobrecarga exhaustiva de operadores, no obstante, debería mirar la sección de conversión automática de tipos más adelante en este capitulo. A menudo, una conversión de tipos en el lugar adecuado puede ahorrarle un montón de operadores sobrecargados.

12.3.3. Argumentos y valores de retorno

Puede parecer un poco confuso inicialmente cuando lea los archivos `OverloadingUnaryOperators.cpp`, `Integer.h` y `Byte.h` y vea todas las maneras diferentes en que se pasan y devuelven los argumentos. Aunque usted pueda pasar y devolver argumentos de la forma que prefiera, las decisiones en estos ejemplos no se han realizado al azar. Siguen un patrón lógico, el mismo que querrá usar en la mayoría de sus decisiones.

1. Como con cualquier argumento de función, si sólo necesita leer el argumento y no cambiarlo, lo usual es pasarlo como una referencia `const`. Normalmente operaciones aritméticas (como `+` y `-`, etc.) y booleanas no cambiarán sus argumentos, así que pasarlas como una referencia `const` es lo que veré mayoritariamente. Cuando la función es un método, esto se traduce en una método `const`. Sólo con los operadores de asignación (como `+=`) y `operator=`, que cambian el argumento de la parte derecha, no es el argumento derecho una constante, pero todavía se pasa en dirección porque será cambiado.

2. El tipo de valor de retorno que debe seleccionar depende del significado esperado del operador. (Otra vez, puede hacer cualquier cosa que desee con los argumentos y con los valores de retorno). Si el efecto del operador es producir un nuevo valor, necesitará generar un nuevo objeto como el valor de retorno. Por ejemplo, `Integer::operator+` debe producir un objeto `Integer` que es la suma de los operandos. Este objeto se devuelve por valor como una constante así que el resultado no se puede modificar como un «valor izquierdo».

3. Todas los operadores de asignación modifican el valor izquierdo. Para permitir que el resultado de la asignación pueda ser usado en expresiones encadenadas, como `a = b = c`, se espera que devuelva una referencia al mismo valor izquierdo que acaba de ser modificado. Pero ¿debería ser esta referencia `const` o no `const`?. Aunque lea `a = b = c` de izquierda a derecha, el compilador la analiza de derecha a izquierda, así que no está obligado a devolver una referencia no `const` para soportar asignaciones encadenadas. Sin embargo, la gente a veces espera ser capaz de realizar una operación sobre el elemento de acaba de ser asignado, como `(a = b).func();` para llamar a `func` de `a` después de asignarle `b`. De ese modo, el valor de retorno para todos los operadores de asignación debería ser una referencia no `const` para el valor izquierdo.

4. Para los operadores lógicos, todo el mundo espera obtener en el peor de los casos un tipo int, y en el mejor un tipo bool. (Las librerías desarrolladas antes de que los compiladores de C++ soportaran el tipo incorporado bool usaban un tipo int o un typedef equivalente).

Los operadores de incremento y decremento presentan un dilema a causa de las versiones postfija y prefija. Ambas versiones cambian el objeto y por tanto no pue-

den tratar el objeto como un `const`. La versión prefija devuelve el valor del objeto después de cambiarlo, así que usted espera recuperar el objeto que fue cambiado. De este modo, con la versión prefija puede simplemente revolver `*this` como una referencia. Se supone que la versión postfija devolverá el valor antes de que sea cambiado, luego está forzado a crear un objeto separado para representar el valor y devolverlo. Así que con la versión postfija debe devolverlo por valor si quiere mantener el significado esperado. (Advierta que a veces encontrará operadores de incremento y decremento que devuelven un int o un bool para indicar, por ejemplo, que un objeto preparado para moverse a través de una lista está al final de ella). Ahora la pregunta es: ¿Debería éste devolverse como una referencia `const` o no `const`?. Si permite que el objeto sea modificado y alguien escribe `(a++).func()`, `func` operará en la propia `a`, pero con `(++a).func()`, `func`opera en el objeto temporal devuelto por el operador postfijo `operator++`. Los objetos temporales son automáticamente `const`, así que esto podría ser rechazado por el compilador, pero en favor de la consistencia tendría más sentido hacerlos ambos `const` como hemos hecho aquí. O puede elegir hacer la versión prefija no `const` y la postfija `const`. Debido a la variedad de significados que puede darle a los operadores de incremento y decremento, deben considerarse en términos del caso individual.

Retorno por valor como constante

El retorno por valor como una constante puede parecer un poco sutil al principio, así que es digno de un poco más de explicación. Considere el operador binario `operator+`. Si lo ve en una expresión como `f(a+b)`, el resultado de `a+b` se convierte en un objeto temporal que se usará en la llamada a `f()`. Debido a que es temporal, es automáticamente `const`, así que aunque, de forma explicita, haga el valor de retorno `const` o no, no tendrá efecto.

Sin embargo, también es posible mandar un mensaje al valor de retorno de `a+b`, mejor que simplemente pasarlo a la función. Por ejemplo, puede decir `(a+b).g()` en la que `g()` es algún método de `Integer`, en este caso. Haciendo el valor de retorno `const`, está indicando que sólo un método `const` puede ser llamado sobre ese valor de retorno. Esto es correcto desde el punto de vista del `const`, porque le evita almacenar información potencialmente importante en un objeto que probablemente será destruido.

Optimización del retorno

Advierta la manera que se usa cuando se crean nuevos objetos para ser devueltos por valor. En `operator+`, por ejemplo:

```
    return Integer(left.i + right.i);
```

Esto puede parecer en principio como una «función de llamada a un constructor» pero no lo es. La sintaxis es la de un objeto temporal; la sentencia dice «crea un objeto `Integer` temporal y desvuélvelo». A causa de esto, puede pensar que el resultado es el mismo que crear un objeto local con nombre y devolverlo. Sin embargo, es algo diferente. Si en su lugar escribiera:

```
    Integer tmp(left.i + right.i);
    return tmp;
```

sucederían tres cosas. La primera, se crea el objeto `tmp` incluyendo la llamada a su constructor. La segunda, el constructor de copia duplica `tmp` en la localización del

valor de retorno externo. La tercera, se llama al destructor para `tmp` cuando sale del ámbito.

En contraste, la aproximación de «devolver un objeto temporal» funciona de manera bastante diferente. Cuando el compilador ve eso, sabe que no tiene otra razón para crearlo mas que para devolverlo. El compilador aprovecha la ventaja que ofrece para construir el objeto directamente en la localización del valor de retorno externo a la función. Esto necesita de una sola y ordinaria llamada al constructor (la llamada al constructor de copia no es necesaria) y no hay llamadas al destructor porque nunca se crea un objeto local. De esta manera, no requiere nada más que el conocimiento del programador, y es significativamente mas eficiente. Esto a menudo se llama optimización del valor de retorno.

12.3.4. Operadores poco usuales

Varios operadores adicionales tienen una forma ligeramente diferente de ser sobrecargados.

El subíndice, `operator[]` debe ser un método y precisa de un único argumento. Dado que `operator[]` implica que el objeto que está siendo utilizado como un array, a menudo devolverá una referencia de este operador, así que puede ser convenientemente usado en la parte derecha de un signo de igualdad. Este operador es muy comúnmente sobrecargado; verá ejemplos en el resto del libro.

Los operadores `new` y `delete` controlan la reserva dinámica de almacenamiento y se pueden sobrecargar de muchas maneras diferentes. Este tema se cubre en el capitulo 13.

El operador coma

El operador coma se llama cuando aparece después de un objeto del tipo para el que está definido. Sin embargo, «`operator,`» no se llama para listas de argumentos de funciones, sólo para objetos fuera de ese lugar separados por comas. No parece haber un montón de usos prácticos para este operador, solo es por consistencia del lenguaje. He aquí un ejemplo que muestra como la función coma se puede llamar cuando aparece antes de un objeto, así como después:

```
//: C12:OverloadingOperatorComma.cpp
#include <iostream>
using namespace std;

class After {
public:
  const After& operator,(const After&) const {
    cout << "After::operator,()" << endl;
    return *this;
  }
};

class Before {};

Before& operator,(int, Before& b) {
  cout << "Before::operator,()" << endl;
  return b;
}
```

```
int main() {
  After a, b;
  a, b;  // Operator comma called

  Before c;
  1, c;  // Operator comma called
} ///:~
```

Las funciones globales permiten situar la coma antes del objeto en cuestión. El uso mostrado es bastante oscuro y cuestionable. Probablemente podría una lista separada por comas como parte de una expresión más complicada, es demasiado refinado en la mayoría de las ocasiones.

El operador ->

El operador `->` se usa generalmente cuando quiere hacer que un objeto parezca un puntero. Este tipo de objeto se suele llamar *puntero inteligente* o más a menudo por su equivalente en inglés: *smart pointer*. Resultan especialmente utiles si quiere «envolver» una clase con un puntero para hacer que ese puntero sea seguro, o en la forma común de un `iterador`, que es un objeto que se mueve a través de una `colección` o `contenedor` de otros objetos y los selecciona de uno en uno cada vez, sin proporcionar acceso directo a la implementación del contenedor. (A menudo encontrará iteradores y contenedores en las librerías de clases, como en la Biblioteca Estándar de C++, descrita en el volumen 2 de este libro).

El operador de indirección de punteros (*) debe ser un método. Tiene otras restricciones atípicas: debe devolver un objeto (o una referencia a un objeto) que también tenga un operador de indirección de punteros, o debe devolver un puntero que pueda ser usado para encontrar a lo que apunta la flecha del operador de indireción de punteros. He aquí un ejemplo simple:

```
//: C12:SmartPointer.cpp
#include <iostream>
#include <vector>
#include "../require.h"
using namespace std;

class Obj {
  static int i, j;
public:
  void f() const { cout << i++ << endl; }
  void g() const { cout << j++ << endl; }
};

// Static member definitions:
int Obj::i = 47;
int Obj::j = 11;

// Container:
class ObjContainer {
  vector<Obj*> a;
public:
  void add(Obj* obj) { a.push_back(obj); }
  friend class SmartPointer;
};
```

```
class SmartPointer {
  ObjContainer& oc;
  int index;
public:
  SmartPointer(ObjContainer& objc) : oc(objc) {
    index = 0;
  }
  // Return value indicates end of list:
  bool operator++() { // Prefix
    if(index >= oc.a.size()) return false;
    if(oc.a[++index] == 0) return false;
    return true;
  }
  bool operator++(int) { // Postfix
    return operator++(); // Use prefix version
  }
  Obj* operator->() const {
    require(oc.a[index] != 0, "Zero value "
      "returned by SmartPointer::operator->()");
    return oc.a[index];
  }
};

int main() {
  const int sz = 10;
  Obj o[sz];
  ObjContainer oc;
  for(int i = 0; i < sz; i++)
    oc.add(&o[i]); // Fill it up
  SmartPointer sp(oc); // Create an iterator
  do {
    sp->f(); // Pointer dereference operator call
    sp->g();
  } while(sp++);
} ///:~
```

La clase `Obj` define los objetos que son manipulados en este programa. Las funciones `f()` y `g()` simplemente escriben en pantalla los valores interesantes usando miembros de datos estáticos. Los punteros a estos objetos son almacenados en el interior de los contenedores del tipo `ObjContainer` usando su función `add()`. `ObjContanier` parece un array de punteros, pero advertirá que no hay forma de traer de nuevo los punteros. Sin embargo, `SmartPointer` se declara como una clase `friend`, así que tiene permiso para mirar dentro del contenedor. La clase `SmartPointer` se parece mucho a un puntero inteligente - puede moverlo hacia adelante usando `operator++` (también puede definir un `operator--`, no pasará del final del contenedor al que apunta, y genera (a través del operador de indireccion de punteros) el valor al que apunta. Advierta que `SmartPointer` está hecho a medida sobre el contenedor para el que se crea; a diferencia de un puntero normal, no hay punteros inteligentes de «propósito general». Aprenderá más sobre los punteros inteligentes llamados «iteradores» en el último capitulo de este libro y en el volumen 2 (descargable desde FIXME:url www. BruceEckel. com).

En `main()`, una vez que el contenedor `oc` se rellena con objetos `Obj` se crea un `SmartPointer sp`. La llamada al puntero inteligente sucede en las expresiones:

```
    sp->f();    // Llamada al puntero inteligente
```

```
    sp->g();
```

Aquí, incluso aunque `sp` no tiene métodos `f()` y `g()`, el operador de indirección de punteros automáticamente llama a esas funciones para Obj* que es devuelto por `SmartPointer::operator->`. El compilador realiza todas las comprobaciones pertinentes para asegurar que la llamada a función funciona de forma correcta.

Aunque la mecánica subyacente de los operadores de indirección de punteros es más compleja que la de los otros operadores, el objetivo es exactamente el mismo: proporcionar una sintaxis más conveniente para los usuarios de sus clases.

Un operador anidado

Es más común ver un puntero inteligente o un clase iteradora anidada dentro de la clase a la que sirve. Se puede reescribir el ejemplo anterior para anidar `SmartPointer` dentro de `ObjContainer` así:

```
//: C12:NestedSmartPointer.cpp
#include <iostream>
#include <vector>
#include "../require.h"
using namespace std;

class Obj {
  static int i, j;
public:
  void f() { cout << i++ << endl; }
  void g() { cout << j++ << endl; }
};

// Static member definitions:
int Obj::i = 47;
int Obj::j = 11;

// Container:
class ObjContainer {
  vector<Obj*> a;
public:
  void add(Obj* obj) { a.push_back(obj); }
  class SmartPointer;
  friend class SmartPointer;
  class SmartPointer {
    ObjContainer& oc;
    unsigned int index;
  public:
    SmartPointer(ObjContainer& objc) : oc(objc) {
      index = 0;
    }
    // Return value indicates end of list:
    bool operator++() { // Prefix
      if(index >= oc.a.size()) return false;
      if(oc.a[++index] == 0) return false;
      return true;
    }
    bool operator++(int) { // Postfix
      return operator++(); // Use prefix version
    }
```

```
    Obj* operator->() const {
      require(oc.a[index] != 0, "Zero value "
        "returned by SmartPointer::operator->()");
      return oc.a[index];
    }
  };
  // Function to produce a smart pointer that
  // points to the beginning of the ObjContainer:
  SmartPointer begin() {
    return SmartPointer(*this);
  }
};

int main() {
  const int sz = 10;
  Obj o[sz];
  ObjContainer oc;
  for(int i = 0; i < sz; i++)
    oc.add(&o[i]); // Fill it up
  ObjContainer::SmartPointer sp = oc.begin();
  do {
    sp->f(); // Pointer dereference operator call
    sp->g();
  } while(++sp);
} ///:~
```

Además del anidamiento de la clase, hay solo dos diferencias aquí. La primera es la declaración de la clase para que pueda ser `friend`:

```
      class SmartPointer;
      friend SmartPointer;
```

El compilador debe saber primero que la clase existe, antes de que se le diga que es «amiga».

La segunda diferencia es en `ObjContainer` donde el método `begin()` produce el `SmartPointer` que apunta al principio de la secuencia del `ObjContainer`. Aunque realmente es sólo por conveniencia, es adecuado porque sigue la manera habitual de la librería estándar de C++.

Operador ->*

El operador `->*` es un operador binario que se comporta como todos los otros operadores binarios. Se proporciona para aquellas situaciones en las que quiera imitar el comportamiento producido por la sintaxis incorporada *puntero a miembro*, descrita en el capitulo anterior.

Igual que `operator->`, el operador de indirección de puntero a miembro se usa normalmente con alguna clase de objetos que representan un «puntero inteligente», aunque el ejemplo mostrado aquí será más simple para que sea comprensible. El truco cuando se define `operator->*` es que debe devolver un objeto para el que `operator()` pueda ser llamado con los argumentos para la función miembro que usted llama.

La llamada a función `operator()` debe ser un método, y es único en que permite cualquier número de argumentos. Hace que el objeto parezca realmente una

función. Aunque podría definir varias funciones sobrecargadas `operator()` con diferentes argumentos, a menudo se usa para tipos que solo tienen una operación simple, o al menos una especialmente destacada. En el Volumen2 verá que la Librería Estándar de C++ usa el operador de llamada a función para crear «objetos-función».

Para crear un `operator->*` debe primero crear una clase con un `operator()` que sea el tipo de objeto que `operator->*` devolverá. Esta clase debe, de algún modo, capturar la información necesaria para que cuando `operator()` sea llamada (lo que sucede automáticamente), el puntero a miembro sea indireccionado para el objeto. En el siguiente ejemplo, el constructor de `FunctionObject` captura y almacena el puntero al objeto y el puntero a la función miembro, y entonces `operator()` los usa para hacer la verdadera llamada puntero a miembro:

```
//: C12:PointerToMemberOperator.cpp
#include <iostream>
using namespace std;

class Dog {
public:
  int run(int i) const {
    cout << "run\n";
    return i;
  }
  int eat(int i) const {
     cout << "eat\n";
     return i;
  }
  int sleep(int i) const {
    cout << "ZZZ\n";
    return i;
  }
  typedef int (Dog::*PMF)(int) const;
  // operator->* must return an object
  // that has an operator():
  class FunctionObject {
    Dog* ptr;
    PMF pmem;
  public:
    // Save the object pointer and member pointer
    FunctionObject(Dog* wp, PMF pmf)
      : ptr(wp), pmem(pmf) {
      cout << "FunctionObject constructor\n";
    }
    // Make the call using the object pointer
    // and member pointer
    int operator()(int i) const {
      cout << "FunctionObject::operator()\n";
      return (ptr->*pmem)(i); // Make the call
    }
  };
  FunctionObject operator->*(PMF pmf) {
    cout << "operator->*" << endl;
    return FunctionObject(this, pmf);
  }
};

int main() {
  Dog w;
```

```
  Dog::PMF pmf = &Dog::run;
  cout << (w->*pmf)(1) << endl;
  pmf = &Dog::sleep;
  cout << (w->*pmf)(2) << endl;
  pmf = &Dog::eat;
  cout << (w->*pmf)(3) << endl;
} ///:~
```

`Dog` tiene tres métodos, todos toman un argumento entero y devuelven un entero. `PMC` es un typedef para simplificar la definición de un puntero a miembro para los métodos de `Dog`.

Una `FunctionObject` es creada y devuelta por `operator->*`. Dese cuenta que `operator->*` conoce el objeto para el que puntero a miembro está siendo llamado (`this`) y el puntero a miembro, y los pasa al constructor `FunctionObject` que almacena sus valores. Cuando se llama a `operator->*`, el compilador inmediatamente lo revuelve y llama a `operator()` para el valor de retorno de `operator->*`, pasándole los argumentos que le fueron pasados a `operator->*`. `FunctionObject::operator()` toma los argumentos e desreferencia el puntero a miembro «real» usando los punteros a objeto y a miembro almacenados.

Percátese de que lo que está ocurriendo aquí, justo como con `operator->`, se inserta en la mitad de la llamada a `operator->*`. Esto permite realizar algunas operaciones adicionales si se necesita.

El mecanismo `operator->*` implementado aquí solo trabaja para funciones miembro que toman un argumento entero y devuelven otro entero. Esto es una limitación, pero si intenta crear mecanismos sobrecargados para cada posibilidad diferente, verá que es una tarea prohibitiva. Afortunadamente, el mecanismo de plantillas de C++ (descrito el el ultimo capitulo de este libro, y en el volumen2) está diseñado para manejar semejante problema.

12.3.5. Operadores que no puede sobrecargar

Hay cierta clase de operadores en el conjunto disponible que no pueden ser sobrecargados. La razón general para esta restricción es la seguridad. Si estos operadores fuesen sobrecargables, podría de algún modo arriesgar o romper los mecanismos de seguridad, hacer las cosas mas difíciles o confundir las costumbres existentes.

1. El operador de selección de miembros `operator.`. Actualmente, el punto tiene significado para cualquier miembro de una clase, pero si se pudiera sobrecargar, no se podría acceder a miembros de la forma normal; en lugar de eso debería usar un puntero y la flecha `operator->`.
2. La indirección de punteros a miembros `operator.*` por la misma razón que `operator.`.
3. No hay un operador de potencia. La elección más popular para éste era `operator**` de Fortram, pero provoca casos de análisis gramatical difíciles. C tampoco tiene un operador de potencia, así que C++ no parece tener necesidad de uno porque siempre puede realizar una llamada a una función. Un operador de potencia añadirá una notación adecuada, pero ninguna nueva funcionalidad a cuenta de una mayor complejidad del compilador.
4. No hay operadores definidos por el usuario. Esto es, no puede crear nuevos operadores que no existan ya. Una parte del problema es como determinar la

precedencia, y otra parte es la falta de necesidad a costa del problema inherente.

5. No puede cambiar las reglas de precedencia. Son lo suficientemente difíciles de recordar como son sin dejar a la gente jugar con ellas.

12.4. Operadores no miembros

En algunos de los ejemplos anteriores, los operadores pueden ser miembros o no, y no parece haber mucha diferencia. Esto normalmente provoca la pregunta, «¿Cuál debería elegir?». En general, si no hay ninguna diferencia deberían ser miembros, para enfatizar la asociación entre el operador y su clase. Cuando el operando de la izquierda es siempre un objeto de la clase actual funciona bien.

Sin embargo, a veces querrá que el operando de la izquierda sea un objeto de alguna otra clase. Un caso típico en el que ocurre eso es cuando se sobrecargan los operadores << y >> para los flujos de entrada/salida. Dado que estos flujos son una librería fundamental en C++, probablemente querrá sobrecargar estos operadores para la mayoría de sus clases, por eso el proceso es digno de tratarse:

```
//: C12:IostreamOperatorOverloading.cpp
// Example of non-member overloaded operators
#include "../require.h"
#include <iostream>
#include <sstream> // "String streams"
#include <cstring>
using namespace std;

class IntArray {
  enum { sz = 5 };
  int i[sz];
public:
  IntArray() { memset(i, 0, sz* sizeof(*i)); }
  int& operator[](int x) {
    require(x >= 0 && x < sz,
      "IntArray::operator[] out of range");
    return i[x];
  }
  friend ostream&
    operator<<(ostream& os, const IntArray& ia);
  friend istream&
    operator>>(istream& is, IntArray& ia);
};

ostream&
operator<<(ostream& os, const IntArray& ia) {
  for(int j = 0; j < ia.sz; j++) {
    os << ia.i[j];
    if(j != ia.sz -1)
      os << ", ";
  }
  os << endl;
  return os;
}

istream& operator>>(istream& is, IntArray& ia){
```

```
  for(int j = 0; j < ia.sz; j++)
    is >> ia.i[j];
  return is;
}

int main() {
  stringstream input("47 34 56 92 103");
  IntArray I;
  input >> I;
  I[4] = -1; // Use overloaded operator[]
  cout << I;
} ///:~
```

Esta clase contiene también un operador sobrecargado `operator[]` el cual devuelve una referencia a un valor legítimo en el array. Dado que devuelve una referencia, la expresión:

```
    I[4] = -1;
```

No sólo parece mucho más adecuada que si se usaran punteros, también causa el efecto deseado.

Es importante que los operadores de desplazamiento sobrecargados se pasen y devuelvan *por referencia*, para que los cambios afecten a los objetos externos. En las definiciones de las funciones, expresiones como:

```
    os << ia.i[j];
```

provocan que sean llamadas las funciones de los operadores sobrecargados (esto es, aquellas definidas en `iostream`). En este caso, la función llamada es `ostream& operator<<(ostream&, int)` dado que `ia[i].j` se resuelve a int.

Una vez que las operaciones se han realizado en `istream` o en `ostream` se devuelve para que se pueda usaren expresiones más complicadas.

En `main()` se usa un nuevo tipo de `iostream`: el stringstream (declarado en `<sstream>`). Es una clase que toma una cadena (que se puede crear de un array de char, como se ve aquí) y lo convierte en un `iostream`. En el ejemplo de arriba, esto significa que los operadores de desplazamiento pueden ser comprobados sin abrir un archivo o sin escribir datos en la línea de comandos.

La manera mostrada en este ejemplo para el extractor y el insertador es estándar. Si quiere crear estos operadores para su propia clase, copie el prototipo de la función y los tipos de retorno de arriba y siga el estilo del cuerpo.

12.4.1. Directrices básicas

Murray [1] sugiere estas reglas de estilo para elegir entre miembros y no miembros:

[1] Rob Murray, *C++ Strategies & Tactics* , Addison Wesley, 1993, pagina 47.

Operador	Uso recomendado
Todos los operadores unarios	miembro
= () [] -> ->*	*debe* ser miembro
+= -= /= *= ^= &= |=%= >>= <<=	miembro
El resto de operadores binarios	no miembro

Cuadro 12.1: Directrices para elegir entre miembro y no-miembro

12.5. Sobrecargar la asignación

Una causa común de confusión para los nuevos programadores de C++ es la asignación. De esto no hay duda porque el signo = es una operación fundamental en la programación, directamente hasta copiar un registro en el nivel de máquina. Además, el constructor de copia (descrito en el capítulo 11) [FIXME:referencia] es invocado cuando el signo = se usa así:

```
MyType b;
MyType a = b;
a = b;
```

En la segunda línea, se define el objeto `a`. Se crea un nuevo objeto donde no existía ninguno. Dado que ya sabe hasta que punto es quisquilloso el compilador de C++ respecto a la inicialización de objetos, sabrá que cuando se define un objeto, siempre se invoca un constructor. Pero ¿qué constructor?, `a` se crea desde un objeto existente `MyType` (`b`, en el lado derecho del signo de igualdad), así que solo hay una opción: el constructor de copia. Incluso aunque el signo de igualdad esté involucrado, se llama al constructor de copia.

En la tercera línea, las cosas son diferentes. En la parte izquierda del signo igual, hay un objeto previamente inicializado. Claramente, no se invoca un constructor para un objeto que ya ha sido creado. En este caso `MyType::operator=` se llama para `a`, tomando como argumento lo que sea que aparezca en la parte derecha. (Puede tener varias funciones `operator=` que tomen diferentes argumentos en la parte derecha).

Este comportamiento no está restringido al constructor de copia. Cada vez que inicializa un objeto usando un signo = en lugar de la forma usual de llamada al constructor, el compilador buscará un constructor que acepte lo que sea que haya en la parte derecha:

```
//: C12:CopyingVsInitialization.cpp
class Fi {
public:
  Fi() {}
};

class Fee {
public:
  Fee(int) {}
  Fee(const Fi&) {}
};

int main() {
  Fee fee = 1; // Fee(int)
  Fi fi;
```

```
  Fee fum = fi; // Fee(Fi)
} ///:~
```

Cuando se trata con el signo =, es importante mantener la diferencia en mente: Si el objeto no ha sido creado todavía, se requiere una inicialización; en otro caso se usa el operador de asignación =.

Es incluso mejor evitar escribir código que usa = para la inicialización; en cambio, use siempre la manera del constructor explícito. Las dos construcciones con el signo igual se convierten en:

```
    Fee fee(1);
    Fee fum(fi);
```

De esta manera, evitará confundir a sus lectores.

12.5.1. Comportamiento del `operador =`

En `Integer.h` y en `Byte.h` vimos que el operador = sólo puede ser una función miembro. Está íntimamente ligado al objeto que hay en la parte izquierda del =. Si fuese posible definir `operator=` de forma global, entonces podría intentar redefinir el signo = del lenguaje:

```
    int operator=(int, MyType);   // Global = !No permitido!
```

El compilador evita esta situación obligandole a hacer un método `operator=`.

Cuando cree un `operator=`, debe copiar toda la información necesaria desde el objeto de la parte derecha al objeto actual (es decir, el objeto para el que `operator=` está siendo llamado) para realizar lo que sea que considere «asignación» para su clase. Para objetos simples, esto es trivial:

```
//: C12:SimpleAssignment.cpp
// Simple operator=()
#include <iostream>
using namespace std;

class Value {
  int a, b;
  float c;
public:
  Value(int aa = 0, int bb = 0, float cc = 0.0)
    : a(aa), b(bb), c(cc) {}
  Value& operator=(const Value& rv) {
    a = rv.a;
    b = rv.b;
    c = rv.c;
    return *this;
  }
  friend ostream&
  operator<<(ostream& os, const Value& rv) {
    return os << "a = " << rv.a << ", b = "
      << rv.b << ", c = " << rv.c;
  }
```

```
};

int main() {
  Value a, b(1, 2, 3.3);
  cout << "a: " << a << endl;
  cout << "b: " << b << endl;
  a = b;
  cout << "a after assignment: " << a << endl;
} ///:~
```

Aquí, el objeto de la parte izquierda del igual copia todos los elementos del objeto de la parte derecha, y entonces devuelve una referencia a sí mismo, lo que permite crear expresiones más complejas.

Este ejemplo incluye un error común. Cuando asignane dos objetos del mismo tipo, siempre debería comprobar primero la auto-asignación: ¿Está asignado el objeto a sí mismo?. En algunos casos como éste, es inofensivo si realiza la operación de asignación en todo caso, pero si se realizan cambios a la implementación de la clase, puede ser importante y si no lo toma con una cuestión de costumbre, puede olvidarlo y provocar errores difíciles de encontrar.

Punteros en clases

¿Qué ocurre si el objeto no es tan simple?. Por ejemplo, ¿qué pasa si el objeto contiene punteros a otros objetos?. Sólo copiar el puntero significa que obtendrá dos objetos que apuntan a la misma localización de memoria. En situaciones como ésta, necesita hacer algo de contabilidad.

Hay dos aproximaciones a este problema. La técnica más simple es copiar lo que quiera que apunta el puntero cuando realiza una asignación o una construcción de copia. Esto es sencillo:

```
//: C12:CopyingWithPointers.cpp
// Solving the pointer aliasing problem by
// duplicating what is pointed to during
// assignment and copy-construction.
#include "../require.h"
#include <string>
#include <iostream>
using namespace std;

class Dog {
  string nm;
public:
  Dog(const string& name) : nm(name) {
    cout << "Creating Dog: " << *this << endl;
  }
  // Synthesized copy-constructor & operator=
  // are correct.
  // Create a Dog from a Dog pointer:
  Dog(const Dog* dp, const string& msg)
    : nm(dp->nm + msg) {
    cout << "Copied dog " << *this << " from "
         << *dp << endl;
  }
  ~Dog() {
```

```
    cout << "Deleting Dog: " << *this << endl;
  }
  void rename(const string& newName) {
    nm = newName;
    cout << "Dog renamed to: " << *this << endl;
  }
  friend ostream&
  operator<<(ostream& os, const Dog& d) {
    return os << "[" << d.nm << "]";
  }
};

class DogHouse {
  Dog* p;
  string houseName;
public:
  DogHouse(Dog* dog, const string& house)
   : p(dog), houseName(house) {}
  DogHouse(const DogHouse& dh)
    : p(new Dog(dh.p, " copy-constructed")),
      houseName(dh.houseName
        + " copy-constructed") {}
  DogHouse& operator=(const DogHouse& dh) {
    // Check for self-assignment:
    if(&dh != this) {
      p = new Dog(dh.p, " assigned");
      houseName = dh.houseName + " assigned";
    }
    return *this;
  }
  void renameHouse(const string& newName) {
    houseName = newName;
  }
  Dog* getDog() const { return p; }
  ~DogHouse() { delete p; }
  friend ostream&
  operator<<(ostream& os, const DogHouse& dh) {
    return os << "[" << dh.houseName
      << "] contains " << *dh.p;
  }
};

int main() {
  DogHouse fidos(new Dog("Fido"), "FidoHouse");
  cout << fidos << endl;
  DogHouse fidos2 = fidos; // Copy construction
  cout << fidos2 << endl;
  fidos2.getDog()->rename("Spot");
  fidos2.renameHouse("SpotHouse");
  cout << fidos2 << endl;
  fidos = fidos2; // Assignment
  cout << fidos << endl;
  fidos.getDog()->rename("Max");
  fidos2.renameHouse("MaxHouse");
} ///:~
```

`Dog` es una clase simple que contiene solo una cadena con el nombre del perro.

Sin embargo, generalmente sabrá cuando le sucede algo al perro porque los constructores y destructores imprimen información cuando se invocan. Fíjese que el segundo constructor es un poco como un constructor de copia excepto que toma un puntero a `Dog` en vez de una referencia, y tiene un segundo argumento que es un mensaje a ser concatenado con el nombre del perro. Esto se hace así para ayudar a rastrear el comportamiento del programa.

Puede ver que cuando un método imprime información, no accede a esa información directamente sino que manda `*this` a `cout`. Éste a su vez llama a `ostream operator<<`. Es aconsejable hacer esto así dado que si quiere reformatear la manera en la que información del perro es mostrada (como hice añadiendo el «[» y el «]») solo necesita hacerlo en un lugar.

Una `DogHouse` contiene un `Dog*` y demuestra las cuatro funciones que siempre necesitará definir cuando sus clases contengan punteros: todos los constructores necesarios usuales, el constructor de copia, `operator=` (se define o se deshabilita) y un destructor. `Operator=` comprueba la auto-asignación como una cuestión de estilo, incluso aunque no es estrictamente necesario aquí. Esto virtualmente elimina la posibilidad de que olvide comprobar la auto-asignación si cambia el código.

Contabilidad de referencias

En el ejemplo de arriba, el constructor de copia y el operador = realizan una copia de lo que apunta el puntero, y el destructor lo borra. Sin embargo, si su objeto requiere una gran cantidad de memoria o una gran inicialización fija, a lo mejor puede querer evitar esta copia. Una aproximación común a este problema se llama *conteo de referencias*. Se le da inteligencia al objeto que está siendo apuntado de tal forma que sabe cuántos objetos le están apuntado. Entonces la construcción por copia o la asignación consiste en añadir otro puntero a un objeto existente e incrementar la cuenta de referencias. La destrucción consiste en reducir esta cuenta de referencias y destruir el objeto si la cuenta llega a cero.

¿Pero que pasa si quiere escribir el objeto(`Dog` en el ejemplo anterior)?. Más de un objeto puede estar usando este `Dog` luego podría estar modificando el perro de alguien más a la vez que el suyo, lo cual no parece ser muy amigable. Para resolver este problema de «solapamiento» se usa una técnica adicional llamada *copia-en-escritura*. Antes de escribir un bloque de memoria, debe asegurarse que nadie más lo está usando. Si la cuenta de referencia es superior a uno, debe realizar una copia personal del bloque antes de escribirlo, de tal manera que no moleste el espacio de otro. He aquí un ejemplo simple de conteo de referencias y copia-en-escritura:

```
//: C12:ReferenceCounting.cpp
// Reference count, copy-on-write
#include "../require.h"
#include <string>
#include <iostream>
using namespace std;

class Dog {
  string nm;
  int refcount;
  Dog(const string& name)
    : nm(name), refcount(1) {
    cout << "Creating Dog: " << *this << endl;
  }
  // Prevent assignment:
  Dog& operator=(const Dog& rv);
```

```
public:
  // Dogs can only be created on the heap:
  static Dog* make(const string& name) {
    return new Dog(name);
  }
  Dog(const Dog& d)
    : nm(d.nm + " copy"), refcount(1) {
    cout << "Dog copy-constructor: "
         << *this << endl;
  }
  ~Dog() {
    cout << "Deleting Dog: " << *this << endl;
  }
  void attach() {
    ++refcount;
    cout << "Attached Dog: " << *this << endl;
  }
  void detach() {
    require(refcount != 0);
    cout << "Detaching Dog: " << *this << endl;
    // Destroy object if no one is using it:
    if(--refcount == 0) delete this;
  }
  // Conditionally copy this Dog.
  // Call before modifying the Dog, assign
  // resulting pointer to your Dog*.
  Dog* unalias() {
    cout << "Unaliasing Dog: " << *this << endl;
    // Don't duplicate if not aliased:
    if(refcount == 1) return this;
    --refcount;
    // Use copy-constructor to duplicate:
    return new Dog(*this);
  }
  void rename(const string& newName) {
    nm = newName;
    cout << "Dog renamed to: " << *this << endl;
  }
  friend ostream&
  operator<<(ostream& os, const Dog& d) {
    return os << "[" << d.nm << "], rc = "
      << d.refcount;
  }
};

class DogHouse {
  Dog* p;
  string houseName;
public:
  DogHouse(Dog* dog, const string& house)
   : p(dog), houseName(house) {
    cout << "Created DogHouse: "<< *this << endl;
  }
  DogHouse(const DogHouse& dh)
    : p(dh.p),
      houseName("copy-constructed " +
        dh.houseName) {
    p->attach();
```

```
    cout << "DogHouse copy-constructor: "
         << *this << endl;
  }
  DogHouse& operator=(const DogHouse& dh) {
    // Check for self-assignment:
    if(&dh != this) {
      houseName = dh.houseName + " assigned";
      // Clean up what you're using first:
      p->detach();
      p = dh.p; // Like copy-constructor
      p->attach();
    }
    cout << "DogHouse operator= : "
         << *this << endl;
    return *this;
  }
  // Decrement refcount, conditionally destroy
  ~DogHouse() {
    cout << "DogHouse destructor: "
         << *this << endl;
    p->detach();
  }
  void renameHouse(const string& newName) {
    houseName = newName;
  }
  void unalias() { p = p->unalias(); }
  // Copy-on-write. Anytime you modify the
  // contents of the pointer you must
  // first unalias it:
  void renameDog(const string& newName) {
    unalias();
    p->rename(newName);
  }
  // ... or when you allow someone else access:
  Dog* getDog() {
    unalias();
    return p;
  }
  friend ostream&
  operator<<(ostream& os, const DogHouse& dh) {
    return os << "[" << dh.houseName
      << "] contains " << *dh.p;
  }
};

int main() {
  DogHouse
    fidos(Dog::make("Fido"), "FidoHouse"),
    spots(Dog::make("Spot"), "SpotHouse");
  cout << "Entering copy-construction" << endl;
  DogHouse bobs(fidos);
  cout << "After copy-constructing bobs" << endl;
  cout << "fidos:" << fidos << endl;
  cout << "spots:" << spots << endl;
  cout << "bobs:" << bobs << endl;
  cout << "Entering spots = fidos" << endl;
  spots = fidos;
  cout << "After spots = fidos" << endl;
```

```
  cout << "spots:" << spots << endl;
  cout << "Entering self-assignment" << endl;
  bobs = bobs;
  cout << "After self-assignment" << endl;
  cout << "bobs:" << bobs << endl;
  // Comment out the following lines:
  cout << "Entering rename(\"Bob\")" << endl;
  bobs.getDog()->rename("Bob");
  cout << "After rename(\"Bob\")" << endl;
} ///:~
```

La clase `Dog` es el objeto apuntado por `DogHouse`. Contiene una cuenta de referencias y métodos para controlar y leer la cuenta de referencias. Hay un constructor de copia de modo que puede crear un nuevo `Dog` a partir de uno existente.

La función `attach()` incrementa la cuenta de referencias de un `Dog` para indicar que hay otro objeto usándolo. La función `detach()` decrementa la cuenta de referencias. Si llega a cero, entonces nadie mas lo está usando, así que el método destruye su propio objeto llamando a `delete this`.

Antes de que haga cualquier modificación (como renombrar un `Dog`), debería asegurarse de que no está cambiando un `Dog` que algún otro objeto está usando. Hágalo llamando a `DogHouse::unalias()` , que llama a `Dog::unalias()`. Esta última función devolverá el puntero a `Dog` existente si la cuenta de referencias es uno (lo que significa que nadie mas está usando ese `Dog`), pero duplicará `Dog` si esa cuenta es mayor que uno.

El constructor de copia, además de pedir su propia memoria, asigna `Dog` al `Dog` del objeto fuente. Entonces, dado que ahora hay un objeto más usando ese bloque de memoria, incrementa la cuenta de referencias llamando a `Dog::attach()`.

El operador = trata con un objeto que ha sido creado en la parte izquierda del =, así que primero debe limpiarlo llamando a `detach()` para ese `Dog`, lo que destruirá el `Dog` viejo si nadie más lo está usando. Entonces `operator=` repite el comportamiento del constructor de copia. Advierta que primero realiza comprobaciones para detectar cuando está asignando el objeto a sí mismo.

El destructor llama a `detach()` para destruir condicionalmente el `Dog`.

Para implementar la copia-en-escritura, debe controlar todas las operaciones que escriben en su bloque de memoria. Por ejemplo, el método `renameDog()` le permite cambiar valores en el bloque de memoria. Pero primero, llama a `unalias()` para evitar la modificación de un `Dog` solapado (un `Dog` con más de un objeto `DogHouse` apuntándole). Y si necesita crear un puntero a un `Dog` desde un `DogHouse` debe llamar primero a `unalias()` para ese puntero.

La función `main()` comprueba varias funciones que deben funcionar correctamente para implementar la cuenta de referencias: el constructor, el constructor de copia, `operator=` y el destructor. También comprueba la copia-en-escritura llamando a `renameDog()`.

He aquí la salida (después de un poco de reformateo):

```
Creando Dog: [Fido],  rc = 1
CreadoDogHouse: [FidoHouse]
contiene [Fido],  rc = 1
Creando Dog: [Spot],  rc = 1
CreadoDogHouse: [SpotHouse]
```

```
    contiene [Spot],  rc = 1
    Entrando en el constructor de copia
    Dog ñaadido:[Fido],  rc = 2
    DogHouse constructor de copia
    [construido por copia FidoHouse]
    contiene [Fido],  rc = 2
    Despues de la óconstruccin por copia de Bobs
    fidos:[FidoHouse] contiene [Fido],  rc = 2
    spots:[SpotHouse] contiene [Spot],  rc = 1
    bobs:[construido por copia FidoHouse]
    contiene[Fido],  rc = 2
    Entrando spots = fidos
    Eliminando perro: [Spot],  rc = 1
    Borrando Perro: [Spot],  rc = 0ñ
    Aadido Dog: [Fido],  rc = 3
    DogHouse operador= : [FidoHouse asignado]
    contiene[Fido],  rc = 3
    Despues de  spots = fidos
    spots:[FidoHouse asignado] contiene [Fido], rc = 3
    Entrando en la auto óasignacin
    DogHouse operador= : [construido por copia FidoHouse]
    contiene [Fido],  rc = 3
    Despues de la auto óasignacin
    bobs:[construido por copia FidoHouse]
    contiene [Fido],  rc = 3
    Entando rename("Bob")
    Despues de rename("Bob")
    DogHouse destructor: [construido por copia FidoHouse]
    contiene [Fido],  rc = 3
    Eliminando perro: [Fido],  rc = 3
    DogHouse destructor: [FidoHouse asignado]
    contiene [Fido],  rc = 2
    Eliminando perro: [Fido],  rc = 2
    DogHouse destructor: [FidoHouse]
    contiene [Fido],  rc = 1
    Eliminando perro: [Fido],  rc = 1
    Borrando perro: [Fido],  rc = 0
```

Estudiando la salida, rastreando el código fuente y experimentando con el programa, podrá ahondar en la comprensión de estas técnicas.

Creación automática del operador =

Dado que asignar un objeto a otro *del mismo tipo* es una operación que la mayoría de la gente espera que sea posible, el compilador automáticamente creará un `type::operator=(type)` si usted el programador no proporciona uno. El comportamiento de este operador imita el del constructor de copia creado automáticamente; si la clase contiene objetos (o se deriva de otra clase), se llama recursivamente a `operator=` para esos objetos. A esto se le llama *asignación miembro a miembro*. Por ejemplo:

```
//: C12:AutomaticOperatorEquals.cpp
#include <iostream>
using namespace std;

class Cargo {
public:
```

```
  Cargo& operator=(const Cargo&) {
    cout << "inside Cargo::operator=()" << endl;
    return *this;
  }
};

class Truck {
  Cargo b;
};

int main() {
  Truck a, b;
  a = b; // Prints: "inside Cargo::operator=()"
} ///:~
```

El `operador=` generado automáticamente para `Truck` llama a `Cargo::operator=`.

En general, no querrá que el compilador haga esto por usted. Con clases de cualquier sofisticación (¡Especialmente si contienen punteros!) querrá crear de forma explicita un `operator=`. Si realmente no quiere que la gente realice asignaciones, declare `operator=` como una método privado. (No necesita definirla a menos que la esté usando dentro de la clase).

12.6. Conversión automática de tipos

En C y C++, si el compilador encuentra una expresión o una llamada a función que usa un tipo que no es el que se requiere, a menudo podrá realizar una conversión automática de tipos desde el tipo que tiene al tipo que necesita. En C++, puede conseguir este mismo efecto para los tipos definidos por el usuario creando funciones de conversión automática de tipos. Estas funciones se pueden ver en dos versiones: un tipo particular de constructores y un operador sobrecargado.

12.6.1. Conversión por constructor

Si define un constructor que toma como su único argumento un objeto (o referencia) de otro tipo, ese constructor permite al compilador realizar una conversión automática de tipos. Por ejemplo:

```
//: C12:AutomaticTypeConversion.cpp
// Type conversion constructor
class One {
public:
  One() {}
};

class Two {
public:
  Two(const One&) {}
};

void f(Two) {}
```

```
int main() {
  One one;
  f(one); // Wants a Two, has a One
} ///:~
```

Cuando el compilador ve que `f()` es invocada pasando un objeto `One`, mira en la declaración de `f()` y ve que requiere un `Two`. Entonces busca si hay alguna manera de conseguir un `Two` a partir de un `One`, encuentra el constructor `Two::Two(One)` y lo llama. Pasa el objeto `Two` resultante a `f()`.

En este caso, la conversión automática de tipos le ha salvado del problema de definir dos versiones sobrecargadas de `f()`. Sin embargo el coste es la llamada oculta al constructor de `Two`, que puede ser importante si está preocupado por la eficiencia de las llamadas a `f()`,

Evitar la conversión por constructor

Hay veces en que la conversión automática de tipos vía constructor puede ocasionar problemas. Para desactivarlo, modifique el constructor anteponiéndole la palabra reservada `explicit` (que sólo funciona con constructores). Así se ha hecho para modificar el constructor de la clase `Two` en el ejemplo anterior:

```
//: C12:ExplicitKeyword.cpp
// Using the "explicit" keyword
class One {
public:
  One() {}
};

class Two {
public:
  explicit Two(const One&) {}
};

void f(Two) {}

int main() {
  One one;
//!  f(one); // No auto conversion allowed
  f(Two(one)); // OK -- user performs conversion
} ///:~
```

Haciendo el constructor de `Two` explícito, se le dice al compilador que no realice ninguna conversión automática de tipos usando ese constructor en particular (sí se podrían usar otros constructores no explícitos de esa clase para realizar conversiones automáticas). Si el usuario quiere que ocurra esa conversión, debe escribir el codigo necesario. En el código de arriba, `f(Two(one))` crea un objeto temporal de tipo `Two` a partir de `one`, justo como el compilador hizo automáticamente en la versión anterior.

12.6.2. Conversión por operador

La segunda forma de producir conversiones automáticas de tipo es a través de la sobrecarga de operadores. Puede crear un método que tome el tipo actual y lo convierta al tipo deseado usando la palabra reservada `operator` seguida del tipo al que quiere convertir. Esta forma de sobrecarga de operadores es única porque parece que no se especifica un tipo de retorno -- el tipo de retorno es el nombre del operador que está sobrecargando. He aquí un ejemplo:

```
//: C12:OperatorOverloadingConversion.cpp
class Three {
  int i;
public:
  Three(int ii = 0, int = 0) : i(ii) {}
};

class Four {
  int x;
public:
  Four(int xx) : x(xx) {}
  operator Three() const { return Three(x); }
};

void g(Three) {}

int main() {
  Four four(1);
  g(four);
  g(1);  // Calls Three(1,0)
} ///:~
```

Con la técnica del constructor, la clase destino realiza la conversión, pero con los operadores, la realiza la clase origen. El valor de la técnica del constructor es que puede añadir una nueva ruta de conversión a un sistema existente al crear una nueva clase. Sin embargo, creando un constructor con un único argumento siempre define una conversión automática de tipos (incluso si requiere más de un argumento si el resto de los argumentos tiene un valor por defecto), que puede no ser lo que desea (en cuyo caso puede desactivarlo usando `explicit`). Además, no hay ninguna forma de usar una conversión por constructor desde un tipo definido por el usuario a un tipo incorporado; eso sólo es posible con la sobrecarga de operadores.

Reflexividad

Una de las razones más convenientes para usar operadores sobrecargados globales en lugar de operadores miembros es que en la versión global, la conversión automática de tipos puede aplicarse a cualquiera de los operandos, mientras que con objetos miembro, el operando de la parte izquierda debe ser del tipo apropiado. Si quiere que ambos operandos sean convertidos, la versión global puede ahorrar un montón de código. He aquí un pequeño ejemplo:

```
//: C12:ReflexivityInOverloading.cpp
class Number {
  int i;
public:
```

```
  Number(int ii = 0) : i(ii) {}
  const Number
  operator+(const Number& n) const {
    return Number(i + n.i);
  }
  friend const Number
    operator-(const Number&, const Number&);
};

const Number
  operator-(const Number& n1,
            const Number& n2) {
    return Number(n1.i - n2.i);
}

int main() {
  Number a(47), b(11);
  a + b; // OK
  a + 1; // 2nd arg converted to Number
//! 1 + a; // Wrong! 1st arg not of type Number
  a - b; // OK
  a - 1; // 2nd arg converted to Number
  1 - a; // 1st arg converted to Number
} ///:~
```

La clase `Number` tiene tanto un miembro `operator+` como un `friend operator-`. Dado que hay un constructor que acepta un argumento int simple, se puede convertir un int automáticamente a `Number`, pero sólo bajo las condiciones adecuadas. En `main()`, puede ver que sumar un `Number` a otro `Number` funciona bien dado que tiene una correspondencia exacta con el operador sobrecargado. Además, cuando el compilador ve un `Number` seguido de un `+` y de un int, puede hacer la correspondencia al método `Number::operator+` y convertir el argumento int an `Number` usando el constructor. Pero cuando ve un int, un `+` y un `Number`, no sabe qué hacer porque todo lo que tiene es `Number::operator+` que requiere que el operando de la izquierda sea ya un objeto `Number`. Así que, el compilador genera un error.

Con `friend operator-` las cosas son diferentes. El compilador necesita rellenar ambos argumentos como quiera; no está restringido a tener un `Number` como argumento de la parte izquierda. así que si ve:

```
    1 - a
```

puede convertir el primer argumento a `Number` usando el constructor.

A veces querrá ser capaz de restringir el uso de sus operadores haciéndolos métodos. Por ejemplo, cuando multiplique una matriz por un vector, el vector debe ir a la derecha. Pero si quiere que sus operadores sean capaces de convertir cualquier argumento, haga el operador una función `friend`.

Afortunadamente, el compilador cogerá la expresión `1-1` y convertirá ambos argumentos a objetos `Number` y después llamará a `operator-`. Eso significaría que el código C existente podría empezar a funcionar de forma diferente. El compilador intenta primero la correspondencia «más simple», es decir, en este caso el operador incorporado para la expresión `1-1`.

12.6.3. Ejemplo de conversión de tipos

Un ejemplo en el que la conversión automática de tipos es extremadamente útil es con cualquier clase que encapsule una cadena de caracteres (en este caso, simplemente implementaremos la clase usando la clase estándar de C++ string dado que es simple). Sin la conversión automática de tipos, si quiere usar todas las funciones existentes de string de la librería estándar de C, tiene que crear un método para cada una, así:

```
//: C12:Strings1.cpp
// No auto type conversion
#include "../require.h"
#include <cstring>
#include <cstdlib>
#include <string>
using namespace std;

class Stringc {
  string s;
public:
  Stringc(const string& str = "") : s(str) {}
  int strcmp(const Stringc& S) const {
    return ::strcmp(s.c_str(), S.s.c_str());
  }
  // ... etc., for every function in string.h
};

int main() {
  Stringc s1("hello"), s2("there");
  s1.strcmp(s2);
} ///:~
```

Aquí, sólo se crea la función `strcmp()`, pero tendría que crear las funciones correspondientes para cada una de `<cstring>` que necesite. Afortunadamente, puede proporcionar una conversión automática de tipos permitiendo el acceso a todas las funciones de `<cstring>`.

```
//: C12:Strings2.cpp
// With auto type conversion
#include "../require.h"
#include <cstring>
#include <cstdlib>
#include <string>
using namespace std;

class Stringc {
  string s;
public:
  Stringc(const string& str = "") : s(str) {}
  operator const char*() const {
    return s.c_str();
  }
};

int main() {
```

```
  Stringc s1("hello"), s2("there");
  strcmp(s1, s2); // Standard C function
  strspn(s1, s2); // Any string function!
} ///:~
```

Ahora cualquier función que acepte un argumento char* puede aceptar también un argumento `Stringc` porque el compilador sabe cómo crear un char* a partir de `Stringc`.

12.6.4. Las trampas de la conversión automática de tipos

Dado que el compilador debe decidir cómo realizar una conversión de tipos, puede meterse en problemas si el programador no diseña las conversiones correctamente. Una situación obvia y simple sucede cuando una clase `X` que puede convertirse a sí misma en una clase `Y` con un `operator Y()`. Si la clase `Y` tiene un constructor que toma un argumento simple de tipo `X`, esto representa la conversión de tipos idéntica. El compilador ahora tiene dos formas de ir de `X` a `Y`, así que se generará una error de ambigüedad:

```
//: C12:TypeConversionAmbiguity.cpp
class Orange; // Class declaration

class Apple {
public:
  operator Orange() const; // Convert Apple to Orange
};

class Orange {
public:
  Orange(Apple); // Convert Apple to Orange
};

void f(Orange) {}

int main() {
  Apple a;
//! f(a); // Error: ambiguous conversion
} ///:~
```

La solución obvia a este problema es no hacerla. Simplemente proporcione una ruta única para la conversión automática de un tipo a otro.

Un problema más difícil de eliminar sucede cuando proporciona conversiones automáticas a más de un tipo. Esto se llama a veces *acomodamiento* (FIXME):

```
//: C12:TypeConversionFanout.cpp
class Orange {};
class Pear {};

class Apple {
public:
  operator Orange() const;
  operator Pear() const;
```

```
};

// Overloaded eat():
void eat(Orange);
void eat(Pear);

int main() {
  Apple c;
//! eat(c);
  // Error: Apple -> Orange or Apple -> Pear ???
} ///:~
```

La clase `Apple` tiene conversiones automáticas a `Orange` y a `Pear`. El elemento capcioso aquí es que no hay problema hasta que alguien inocentemente crea dos versiones sobrecargadas de `eat()`. (Con una única versión el codigo en `main()` funciona correctamente).

De nuevo la solución -- y el lema general de la conversión automática de tipos -- es proporcionar solo una conversión automática de un tipo a otro. Puede tener conversiones a otros tipos, sólo que no deberían ser *automáticas*. Puede crear llamadas a funciones explícitas con nombres como `makeA()` y `makeB()`.

Actividades ocultas

La conversión automática de tipos puede producir mas actividad subyacente de la que se podría esperar. Mire esta modificación de `CopyingVsInitialization.cpp` como un pequeño rompecabezas:

```
//: C12:CopyingVsInitialization2.cpp
class Fi {};

class Fee {
public:
  Fee(int) {}
  Fee(const Fi&) {}
};

class Fo {
  int i;
public:
  Fo(int x = 0) : i(x) {}
  operator Fee() const { return Fee(i); }
};

int main() {
  Fo fo;
  Fee fee = fo;
} ///:~
```

No hay un constructor para crear `Fee fee` de un objeto `Fo`. Sin embargo, `Fo` tiene una conversión automática de tipos a `Fee`. No hay un constructor de copia para crear un `Fee` a partir de un `Fee`, pero ésa es una de las funciones especiales que el compilador puede crear. (El constructor por defecto, el constructor de copia y `operator=`) y el destructor pueden sintetizarse automáticamente por el compilador.

Así que, para la relativamente inocua expresión:

```
Fee fee = fo;
```

se invoca el operador de conversión automático de tipo, y se crea un constructor de copia.

Use la conversión automática de tipos con precaución. Como con toda la sobrecarga de operadores, es excelente cuando reduce la tarea de codificación significativamente, pero no vale la pena usarla de forma gratuita.

12.7. Resumen

El motivo de la existencia de la sobrecarga de operadores es para aquellas situaciones en la que la vida. No hay nada particularmente mágico en ello; los operadores sobrecargados son solo funciones con nombres divertidos, y el compilador realiza las invocaciones a esas funciones cuando aparece el patrón adecuado. Pero si la sobrecarga de operadores no proporciona un beneficio significativo el creador de la clase o para el usuario de la clase, no complique el asunto añadiéndolos.

12.8. Ejercicios

Las soluciones a los ejercicios se pueden encontrar en el documento electrónico titulado «The Thinking in C++ Annotated Solution Guide», disponible por poco dinero en www.BruceEckel.com.

1. Cree una clase sencilla con un operador sobrecargado ++. Intente llamar a este operador en la forma prefija y postfija y vea qué clase de advertencia obtiene del compilador.
2. Cree una clase sencilla que contenga un int y sobrecargue el operador + como un método. Cree también un método `print()` que tome un `ostream&` como un argumento y lo imprima a un `ostream&`. Pruebe su clase para comprobar que funciona correctamente.
3. Añada un operador binario − al ejercicio 2 como un método. Demuestre que puede usar sus objetos en expresiones complejas como `a + b -c`.
4. Añada un operador ++ y otro −− al ejercicio 2, ambos con las versiones prefijas y postfijas, tales que devuelvan el objeto incrementado o decrementado. Asegúrese de que la versión postfija devuelve el valor correcto.
5. Modifique los operadores de incremento y decremento del ejercicio 4 para que la versión prefija devuelva una referencia no `const` y la postfija devuelva un objeto `const`. Muestre que funcionan correctamente y explique porqué esto se puede hacer en la práctica.
6. Cambie la función `print()` del ejercicio2 para que use el operador sobrecargado << como en `IostreamOperatorOverloading.cpp`.
7. Modifique el ejercicio 3 para que los operadores + y − no sean métodos. Demuestre que todavía funcionan correctamente.
8. Añada el operador unario − al ejercicio 2 y demuestre que funciona correctamente.

9. Cree una clase que contenga un único private char. Sobrecargue los operadores de flujos de entrada/salida << y >> (como en `IostreamOperatorOverloading.cpp`) y pruébelos. Puede probarlos con `fstreams`, `stringstreams` y `cin` y `cout`.

10. Determine el valor constante ficticio que su compilador pasa a los operadores postfijos ++ y --.

11. Escriba una clase `Number` que contenga un double y añada operadores sobrecargados para `+`, `-`, `*`, `/` y la asignación. Elija los valores de retorno para estas funciones para que las expresiones se puedan encadenar y que sea eficiente. Escriba una conversión automática de tipos operator int().

12. Modifique el ejercicio 11 para que use la *optimización del valor de retorno*, si todavía no lo ha hecho.

13. Cree una clase que contenga un puntero, y demuestre que si permite al compilador sintetizar el operador = el resultado de usar ese operador serán punteros que estarán solapados en la misma ubicación de memoria. Ahora arregle el problema definiendo su propio operador = y demuestre que corrige el solapamiento. Asegúrese que comprueba la auto-asignación y que maneja el caso apropiadamente.

14. Escriba una clase llamada `Bird` que contenga un miembro string y un static int. En el constructor por defecto, use el int para generar automáticamente un identificador que usted construya en el string junto con el nombre de la clase(`Bird #1`, `Bird #2`, etc). Añada un operador <; < para flujos de salida para imprimir los objetos `Bird`-Escriba un operador de asignación = y un constructor de copia. En `main()` verifique que todo funciona correctamente.

15. Escriba una clase llamada `BirdHouse` que contenga un objeto, un puntero y una referencia para la clase `Bird` del ejercicio 14. El constructor debería tomar 3 `Birds` como argumentos. Añada un operador << de flujo de salida para `BirdHouse`. Deshabilite el operador de asignación = y el constructor de copia. En `main()` verifique que todo funciona correctamente.

16. Añada un miembro de datos int a `Bird` y a `BirdHouse` en el ejercicio 15. Añada operadores miembros `+`, `-`, `*` y `/` que usen el miembro int para realizar las operaciones en los respectivos miembros. Verifique que funcionan.

17. Repita el ejercicio 16 usando operadores no miembro.

18. Añada un operador `-` a `SmartPointer.cpp` y a `NestedSmartPointer.cpp`.

19. Modifique `CopyingVsInitialization.cpp` para que todos los constructores impriman un mensaje que explique qué está pasando. Ahora verifique que las dos maneras de llamar al constructor de copia (la de asignación y la de paréntesis) son equivalentes.

20. Intente crear un operador no miembro = para una clase y vea qué clase de mensaje del compilador recibe.

21. Cree una clase con un operador de asignación que tenga un segundo argumento, un `string` que tenga un valor por defecto que diga `op = call`. Cree una función que asigne un objeto de su clase a otro y muestre que su operador de asignación es llamado correctamente.

22. En `CopyingWithPointers.cpp` elimine el operador = en `DogHouse` y muestre que el operador = sintetizado por el compilador copia correctamente `string` pero es simplemente un alias del puntero `Dog`.

23. En `ReferenceCounting.cpp` añada un static int y un int ordinario como atributos a `Dog` y a `DogHouse`. En todos los constructores para ambas clases, incremente el static int y asigne el resultado al int ordinario para mantener un seguimiento del número de objetos que están siendo creados. Haga las modificaciones necesarias para que todas las sentencias de impresión muestren los identificadores int de los objetos involucrados.

24. Cree una clase que contenga un `string` como atributo. Inicialice el `string` en el constructor, pero no cree un constructor de copia o un operador =. Haga una segunda clase que tenga un atributo de su primera clase; no cree un constructor de copia o un operador = para esta clase tampoco. Demuestre que el constructor de copia y el operador = son sintetizados correctamente por el compilador.

25. Combine las clases en `OverloadingUnaryOperators.cpp` y en `Integer.cpp`.

26. Modifique `PointerToMemmberOperator.cpp` añadiendo dos nuevas funciones miembro a `Dog` que no tomen argumentos y devuelvan void. Cree y compruebe un operador sobrecargado `->*` que funcione con sus dos nuevas funciones.

27. Añada un operador `->*` a `NestedSmartPointer.cpp`.

28. Cree dos clases, `Apple` y `Orange`. En `Apple`, cree un constructor que tome una `Orange` como argumento. Cree una función que tome un `Apple` y llame a esa función con una una `Orange` para demostrar que funciona. Ahora haga explícito el constructor de `Apple` para demostrar que así se evita la conversión automática de tipos. Modifique la llamada a su función para que la la conversión se haga explícitamente y de ese modo, funcione.

29. Añada un operador global `*` a `ReflexivityInOverloading.cpp` y demuestre que es reflexivo.

30. Cree dos clases y un operador + y las funciones de conversión de tal manera que la adicción sea reflexiva para las dos clases.

31. Arregle `TypeConversionFanout.cpp` creando una función explícita para realizar la conversión de tipo, en lugar de uno de los operadores de conversión automáticos.

32. Escriba un código simple que use los operadores `+`, `-`, `*`, `/` para double. Imagine cómo el compilador genera el codigo ensamblador y mire el ensamblador que se genera en realidad para descubrir y explicar qué está ocurriendo «bajo el capó».

13: Creación dinámica de objetos

A veces se conoce la cantidad exacta exacta, el tipo y duración de la vida de los objetos en un programa, pero no siempre es así.

¿Cuántos aviones tendrá que supervisar un sistema de control de tráfico aéreo? ¿Cuántas formas o figuras se usarán en un sistema CAD? ¿Cuántos nodos habrá en una red?

Para resolver un problema general de programación, es esencial poder crear y destruir objetos en tiempo de ejecución. Por supuesto, C proporciona las funciones de asignación dinámica de memoria `malloc()` y sus variantes, y `free()`, que permiten obtener y liberar bloques en el espacio de memoria del *montículo* (también llamado *espacio libre*[1] mientras se ejecuta el programa.

Este método sin embargo, no funcionará en C++. El constructor no le permite manipular la dirección de memoria a inicializar, y con motivo. De permitirse, sería posible:

1. Olvidar la llamada al constructor. Con lo cual no sería posible garantizar la inicialización de los objetos en C++.
2. Usar accidentalmente un objeto que aún no ha sido inicializado, esperando que todo vaya bien.
3. Manipular un objeto de tamaño incorrecto.

Y por supuesto, incluso si se hizo todo correctamente, cualquiera que modifique el programa estaría expuesto a cometer esos mismos errores. Una gran parte de los problemas de programación tienen su origen en la inicialización incorrecta de objetos, lo que hace especialmente importante garantizar la llamada a los constructores para los objetos que han de ser creados en el montículo.

¿Cómo se garantiza en C++ la correcta inicialización y limpieza, permitiendo la creación dinámica de objetos?

La respuesta está en integrar en el lenguaje mismo la creación dinámica de objetos. `malloc()` y `free()` son funciones de biblioteca y por tanto, están fuera del control del compilador. Si se dispone de un *operador* que lleve a cabo el acto combinado de la asignación dinámica de memoria y la inicialización, y de otro operador que realice el acto combinado de la limpieza y de liberación de memoria, el compilador podrá garantizar la llamada a los constructores y destructores de los objetos.

En este capítulo verá cómo se resuelve de modo elegante este problema con los operadores `new` y `delete` de C++.

[1] N.T. espacio de almacenamiento libre (*free store*)

13.1. Creación de objetos

La creación de un objeto en C++ tiene lugar en dos pasos:

1. Asignación de memoria para el objeto.
2. Llamada al constructor.

Aceptemos por ahora que este segundo paso ocurre *siempre*. C++ lo fuerza, debido a que el uso de objetos no inicializados es una de las causas más frecuentes de errores de programación. Siempre se invoca al constructor, sin importar cómo ni dónde se crea el objeto.

El primero de estos pasos puede ocurrir de varios modos y en diferente momento:

1. Asignación de memoria en la zona de almacenamiento estático, que tiene lugar durante la carga del programa. El espacio de memoria asignado al objeto existe hasta que el programa termina.
2. Asignación de memoria en la pila, cuando se alcanza algún punto determinado durante la ejecución del programa (la llave de apertura de un bloque). La memoria asignada se vuelve a liberar de forma automática en cuanto se alcanza el punto de ejecución complementario (la llave de cierre de un bloque). Las operaciones de manipulación de la pila forman parte del conjunto de instrucciones del procesador y son muy eficientes. Por otra parte, es necesario saber cuantas variables se necesitan mientras se escribe el programa de modo que el copilador pueda generar el código correspondiente.
3. Asignación dinámica, en una zona de memoria libre llamada montículo (*heap* o *free store*). Se reserva espacio para un objeto en esta zona mediante la llamada a una función durante la ejecución del programa; esto significa que se puede decidir en cualquier momento que se necesita cierta cantidad de memoria. Esto conlleva la responsabilidad de determinar el momento en que ha de liberarse la memoria, lo que implica determinar el tiempo de vida de la misma que, por tanto, ya no está bajo control de las reglas de ámbito.

A menudo, las tres regiones de memoria referidas se disponen en una zona contigua de la memoria física: área estática, la pila, y el montículo, en un orden determinado por el escritor del compilador. No hay reglas fijas. La pila puede estar en una zona especial, y puede que las asignaciones en el montículo se obtengan mediante petición de bloques de la memoria del sistema operativo. Estos detalles quedan normalmente ocultos al programador puesto que todo lo que se necesita conocer al respecto es que esa memoria estará disponible cuando se necesite.

13.1.1. Asignación dinámica en C

C proporciona las funciones de su biblioteca estándar `malloc()` y sus variantes `calloc()` y `realloc()` para asignar, y `free()` para liberar bloques de memoria dinámicamente en tiempo de ejecución. Estas funciones son pragmáticas pero rudimentarias por lo que requieren comprensión y un cuidadoso manejo por parte del programador. El listado que sigue es un ejemplo que ilustra el modo de crear una instancia de una clase con estas funciones de C:

```
//: C13:MallocClass.cpp
```

```
// Malloc with class objects
// What you'd have to do if not for "new"
#include "../require.h"
#include <cstdlib> // malloc() & free()
#include <cstring> // memset()
#include <iostream>
using namespace std;

class Obj {
  int i, j, k;
  enum { sz = 100 };
  char buf[sz];
public:
  void initialize() { // Can't use constructor
    cout << "initializing Obj" << endl;
    i = j = k = 0;
    memset(buf, 0, sz);
  }
  void destroy() const { // Can't use destructor
    cout << "destroying Obj" << endl;
  }
};

int main() {
  Obj* obj = (Obj*)malloc(sizeof(Obj));
  require(obj != 0);
  obj->initialize();
  // ... sometime later:
  obj->destroy();
  free(obj);
} ///:~
```

Observe el uso de `malloc()` para la obtención de espacio para el objeto:

```
  Obj* obj = (Obj*)malloc(sizeof(Obj));
```

Se debe pasar como parámetro a `malloc()` el tamaño del objeto. El tipo de retorno de `malloc()` es void*, pues es sólo un puntero a un bloque de memoria, no un objeto. En C++ no se permite la asignación directa de un void* a ningún otro tipo de puntero, de ahí la necesidad de la conversión explícita de tipo (molde).

Puede ocurrir que `malloc()` no encuentre un bloque adecuado, en cuyo caso devolverá un puntero nulo, de ahí la necesidad de comprobar la validez del puntero devuelto.

El principal escollo está en la línea:

```
  obj->initialize();
```

El usuario deberá asegurarse de inicializar el objeto antes de su uso. Obsérvese que no se ha usado el constructor debido a que éste no puede ser llamado de modo explícito [2]; es llamado por el compilador cuando se crea un objeto. El problema es

[2] Existe una sintaxis especial llamada *placement-new* que permite llamar al constructor para un bloque de memoria preasignando. Se verá más adelante, en este mismo capítulo.

que el usuario puede olvidar inicializar el objeto antes de usarlo, introduciendo así una importante fuente de problemas.

Como consecuencia, muchos programadores encuentran muy confusas y complicadas las funciones de asignación dinámica de la memoria en C. No es muy difícil encontrar programadores que, usando máquinas con memoria virtual, usan vectores enormes en el área de almacenamiento estático para evitar tener que tratar con la asignación dinámica. Dado que C++ intenta facilitar el uso de la biblioteca a los programadores ocasionales, no es aceptable la forma de abordar la asignación dinámica en C.

13.1.2. El operador `new`

La solución que ofrece C++ consiste en combinar la serie de acciones necesarias para la creación de un objeto en un único operador llamado `>new`. Cuando se crea un objeto mediante el operador `>new`, éste se encarga de obtener el espacio necesario para el objeto y de llamar a su constructor. Cuando se ejecuta el código:

```
MyType *fp = new MyType(1,2);
```

se asigna espacio mediante alguna llamada equivalente a `>malloc(sizeof(MyType))` --con frecuencia es así, literalmente--, y usando la dirección obtenida como puntero `>this`, y `(1, 2)` como argumentos, se llama al constructor de la clase `MyType`. Para cuando está disponible, el valor de retorno de `new` es ya un puntero válido a un objeto inicializado. Además es del tipo correcto, lo que hace innecesaria la conversión.

El operador `new` por defecto, comprueba el éxito o fracaso de la asignación de memoria como paso previo a la llamada al constructor, haciendo innecesaria y redundante la posterior comprobación. Más adelante en este capítulo se verá qué sucede si se produce este fallo.

En las expresiones con `new` se puede usar cualquiera de los constructores disponibles para una clase. Si éste no tiene argumentos, se escribe la expresión sin lista de argumentos

```
MyType *fp = new MyType;
```

Es notable la simpleza alcanzada en la creación dinámica de objetos: una única expresión realiza todo el trabajo de cálculo de tamaño, asignación, comprobaciones de seguridad y conversión de tipo. Esto hace que la creación dinámica de objetos sea tan sencilla como la creación en la pila.

13.1.3. El operador `delete`

El complemento a la expresión `new` es la expresión `delete`, que primero llama al destructor y después libera la memoria (a menudo mediante una llamada a `free(-)`). El argumento para una expresión con `delete` debe ser una dirección: un puntero a objeto creado mediante `new`.

```
delete fp;
```

Esta expresión destruye el objeto y después libera el espacio dinámicamente asignado al objeto MyType

El uso del operador `delete` debe limitarse a los objetos que hayan sido creados mediante `new`. Las consecuencias de aplicar el operador `delete` a los objetos creados con `malloc()`, `calloc()` o `realloc()` no están definidas. Dado que la mayoría de las implementaciones por defecto de `new` y `delete` usan `malloc()` y `free()`, el resultado será probablemente la liberación de la memoria sin la llamada al destructor.

No ocurre nada si el puntero que se le pasa a `delete` es nulo. Por esa razón, a menudo se recomienda asignar cero al puntero inmediatamente después de usar `delete`; se evita así que pueda ser usado de nuevo como argumento para `delete`. Tratar de destruir un objeto más de una vez es un error de consecuencias imprevisibles.

13.1.4. Un ejemplo sencillo

El siguiente ejemplo demuestra que la inicialización tiene lugar:

```
//: C13:Tree.h
#ifndef TREE_H
#define TREE_H
#include <iostream>

class Tree {
  int height;
public:
  Tree(int treeHeight) : height(treeHeight) {}
  ~Tree() { std::cout << "*"; }
  friend std::ostream&
  operator<<(std::ostream& os, const Tree* t) {
    return os << "Tree height is: "
              << t->height << std::endl;
  }
};
#endif // TREE_H ///:~
```

Se puede probar que el constructor es invocado imprimiendo el valor de `Tree`. Aquí se hace sobrecargando el `operator <<` para usarlo con un `ostream` y un Tree*. Note, sin embargo, que aunque la función está declarada como `friend`, está definida como una `inline`!. Esto es así por conveniencia --definir una función amiga como `inline` a una clase no cambia su condición de amiga o el hecho de que es una función global y no un método. También resaltar que el valor de retorno es el resultado de una expresión completa (el ostream&), y así debe ser, para satisfacer el tipo del valor de retorno de la función.

13.1.5. Trabajo extra para el gestor de memoria

Cuando se crean objetos automáticos en la pila, el tamaño de los objetos y su tiempo de vida queda fijado en el código generado, porque el compilador conoce su tipo, cantidad y alcance. Crear objetos en el montículo implica una sobrecarga adicional, tanto en tiempo como en espacio. Veamos el escenario típico (Puede reemplazar

`malloc()` con `calloc()` o `realloc()`).

Se invoca `malloc()`, que pide un bloque de memoria. (Este código realmente puede ser parte de `malloc()`).

Ahora tiene lugar la búsqueda de un bloque de tamaño adecuado de entre los bloques libres. Esto requiere la comprobación de un mapa o directorio de algún tipo que lleve el registro de los bloques disponibles y de los que están en uso. Es un proceso rápido, pero puede que necesite varias pruebas, es pues un proceso no determinista. Dicho de otro modo, no se puede contar con que `malloc()` tarde siempre exactamente el mismo tiempo en cada búsqueda.

Antes de entregar el puntero del bloque obtenido, hay que registrar en alguna parte su tamaño y localización para que `malloc()` no lo vuelva a usar y para que cuando se produzca la llamada a `free()`, el sistema sepa cuánto espacio ha de liberar.

El modo en que se implementan todas estas operaciones puede variar mucho. No hay nada que impida que puedan implementarse las primitivas de asignación de memoria en el conjunto de instrucciones del procesador. Si es suficientemente curioso, pueden escribir programas que permitan averiguar cómo está implementada `malloc()`. Si dispone de él, puede leer el código fuente de la biblioteca de funciones de C, si no, siempre está disponible el de GNU C.

13.2. Rediseño de los ejemplos anteriores

Puede reescribirse el ejemplo `Stash` que vimos anteriormente en el libro, haciendo uso de los operadores `new` y `delete`, con las características que se han visto desde entonces. A la vista del nuevo código se pueden repasar estas cuestiones.

Hasta este punto del libro, ninguna de las clases `Stash` ni `Stack` poseerán los objetos a los que apuntan; es decir, cuando el objeto `Stash` o `Stack` sale de ámbito, no se invoca `delete` para cada uno de los objetos a los que apunta. La razón por la que eso no es posible es porque, en un intento de conseguir más generalidad, utilizan punteros void. Usar `delete` con punteros `void` libera el bloque de memoria pero, al no existir información de tipo, el compilador no sabe qué destructor debe invocar.

13.2.1. `delete` void* probablemente es un error

Es necesario puntualizar que, llamar a `delete` con un argumento void* es casi con seguridad un error en el programa, a no ser que el puntero apunte a un objeto muy simple; en particular, que no tenga un destructor. He aquí un ejemplo ilustrativo:

```
//: C13:BadVoidPointerDeletion.cpp
// Deleting void pointers can cause memory leaks
#include <iostream>
using namespace std;

class Object {
  void* data; // Some storage
  const int size;
  const char id;
public:
  Object(int sz, char c) : size(sz), id(c) {
```

```
    data = new char[size];
    cout << "Constructing object " << id
         << ", size = " << size << endl;
  }
  ~Object() {
    cout << "Destructing object " << id << endl;
    delete []data; // OK, just releases storage,
    // no destructor calls are necessary
  }
};

int main() {
  Object* a = new Object(40, 'a');
  delete a;
  void* b = new Object(40, 'b');
  delete b;
} ///:~
```

La clase `Object` contiene la variable `data` de tipo void* que es inicializada para apuntar a un objeto simple que no tiene destructor. En el destructor de `Object` se llama a `delete` con este puntero, sin que tenga consecuencias negativas puesto que lo único que se necesita aquí es liberar la memoria.

Ahora bien, se puede ver en `main()` la necesidad de que `delete` conozca el tipo del objeto al que apunta su argumento. Esta es la salida del programa:

```
Construyendo objeto a, ñtamao = 40
Destruyendo objeto a
Construyendo objeto b, ñtamao = 40
```

Como `delete` sabe que `a` es un puntero a `Object`, se lleva a cabo la llamada al destructor de `Object`, con lo que se libera el espacio asignado a `data`. En cambio, cuando se manipula un objeto usando un void*, como es el caso en `delete b`, se libera el bloque de `Object`, pero no se efectúa la llamada a su destructor, con lo que tampoco se liberará el espacio asignado a `data`, miembro de `Object`. Probablemente no se mostrará ningún mensaje de advertencia al compilar el programa; no hay ningún error sintáctico. Como resultado obtenemos un programa con una silenciosa fuga de memoria.

Cuando se tiene una fuga de memoria, se debe buscar entre todas las llamadas a `delete` para comprobar el tipo de puntero que se le pasa. Si es un void*, puede estar ante una de las posibles causas (Sin embargo, C++ proporciona otras muchas oportunidades para la fuga de memoria).

13.2.2. Responsabilidad de la limpieza cuando se usan punteros

Para hacer que los contenedores `Stack` y `Stash` sean flexibles, capaces de recibir cualquier tipo de objeto, se usan punteros de tipo void*. Esto hace necesario convertir al tipo adecuado los punteros devueltos por las clases `Stash` y `Stack`, antes de que sean usados. Hemos visto en la sección anterior, que los punteros deben ser convertidos al tipo correcto incluso antes de ser entregados a `delete`, para evitar posibles fugas de memoria.

Hay otro problema, derivado de la necesidad de llamar a `delete` para cada pun-

tero a objeto almacenado en el contenedor. El contenedor no puede realizar la limpieza para los punteros que almacena puesto que son punteros void*. Esto puede derivar en un serio problema si a un contenedor se le pasan punteros a objetos automáticos junto con punteros a objetos dinámicos; el resultado de usar `delete` sobre un puntero que no haya sido obtenido del montículo es imprevisible. Más aún, al obtener del contenedor un puntero cualquiera, existirán dudas sobre el origen, automático, dinámico o estático, del objeto al que apunta. Esto implica que hay que asegurarse del origen dinámico de los punteros que se almacenen en la siguiente versión de `Stash` y `Stack`, bien sea mediante una programación cuidadosa, o bien por la creación de clases que sólo puedan ser construidas en el montículo.

Es muy importante asegurarse también de que el programador cliente se responsabilice de la limpieza de los punteros del contenedor. Se ha visto en ejemplos anteriores que la clase `Stack` comprobaba en su destructor que todos los objetos `Link` habían sido desapilados. Un objeto `Stash` para punteros requiere un modo diferente de abordar el problema.

13.2.3. Stash para punteros

Esta nueva versión de la clase `Stash`, que llamamos `PStash`, almacena punteros a objetos existentes en el montículo, a diferencia de la vieja versión, que guardaba una copia por valor de los objetos. Usando `new` y `delete`, es fácil y seguro almacenar punteros a objetos creados en el montículo.

He aquí el archivo de cabecera para «`Stash` para punteros»:

```
//: C13:PStash.h
// Holds pointers instead of objects
#ifndef PSTASH_H
#define PSTASH_H

class PStash {
  int quantity; // Number of storage spaces
  int next; // Next empty space
   // Pointer storage:
  void** storage;
  void inflate(int increase);
public:
  PStash() : quantity(0), storage(0), next(0) {}
  ~PStash();
  int add(void* element);
  void* operator[](int index) const; // Fetch
  // Remove the reference from this PStash:
  void* remove(int index);
  // Number of elements in Stash:
  int count() const { return next; }
};
#endif // PSTASH_H ///:~
```

Los elementos de datos subyacentes no han cambiado mucho, pero ahora el almacenamiento se hace sobre un vector de punteros void, que se obtiene mediante `new` en lugar de `malloc()`. En la expresión

```
  void** st = new void*[ quantity + increase ];
```

se asigna espacio para un vector de punteros a void.

El destructor de la clase libera el espacio en el que se almacenan los punteros sin tratar de borrar los objetos a los que hacen referencia, ya que esto, insistimos, liberaría el espacio asignado a los objetos, pero no se produciría la necesaria llamada a sus destructores por la falta de información de tipo.

El otro cambio realizado es el reemplazo de la función `fetch()` por `operator[]`, más significativo sintácticamente. Su tipo de retorno es nuevamente void*, por lo que el usuario deberá recordar el tipo de los objetos a que se refieren y efectuar la adecuada conversión al extraerlos del contenedor. Resolveremos este problema en capítulos posteriores.

Sigue la definición de los métodos de `PStash`:

```
//: C13:PStash.cpp {O}
// Pointer Stash definitions
#include "PStash.h"
#include "../require.h"
#include <iostream>
#include <cstring> // 'mem' functions
using namespace std;

int PStash::add(void* element) {
  const int inflateSize = 10;
  if(next >= quantity)
    inflate(inflateSize);
  storage[next++] = element;
  return(next - 1); // Index number
}

// No ownership:
PStash::~PStash() {
  for(int i = 0; i < next; i++)
    require(storage[i] == 0,
      "PStash not cleaned up");
  delete []storage;
}

// Operator overloading replacement for fetch
void* PStash::operator[](int index) const {
  require(index >= 0,
    "PStash::operator[] index negative");
  if(index >= next)
    return 0; // To indicate the end
  // Produce pointer to desired element:
  return storage[index];
}

void* PStash::remove(int index) {
  void* v = operator[](index);
  // "Remove" the pointer:
  if(v != 0) storage[index] = 0;
  return v;
}

void PStash::inflate(int increase) {
  const int psz = sizeof(void*);
  void** st = new void*[quantity + increase];
```

```
  memset(st, 0, (quantity + increase) * psz);
  memcpy(st, storage, quantity * psz);
  quantity += increase;
  delete []storage; // Old storage
  storage = st; // Point to new memory
} ///:~
```

La función `add()` es, en efecto, la misma que antes si exceptuamos el hecho de que lo que se almacena ahora es un puntero a un objeto en lugar de una copia del objeto.

El código de `inflate()` ha sido modificado para gestionar la asignación de memoria para un vector de void*, a diferencia del diseño previo, que sólo trataba con bytes. Aquí, en lugar de usar el método de copia por el índice del vector, se pone primero a cero el vector usando la función `memset()` de la biblioteca estándar de C, aunque esto no sea estrictamente necesario ya que, presumiblemente, `PStash` manipulará la memoria de forma adecuada, pero a veces no es muy costoso añadir un poco más de seguridad. A continuación, se copian al nuevo vector usando `memcpy()` los datos existentes en el antiguo. Con frecuencia verá que las funciones `memcpy()` y `memset()` han sido optimizadas en cuanto al tiempo de proceso, de modo que pueden ser más rápidas que los bucles anteriormente vistos. No obstante, una función como `inflate()` no es probable que sea llamada con la frecuencia necesaria para que la diferencia sea palpable. En cualquier caso, el hecho de que las llamadas a función sean más concisas que los bucles, puede ayudar a prevenir errores de programación.

Para dejar definitivamente la responsabilidad de la limpieza de los objetos sobre los hombros del programador cliente, se proporcionan dos formas de acceder a los punteros en `PStash`: el operador `[]`, que devuelve el puntero sin eliminarlo del contenedor, y un segundo método `remove()` que además de devolver el puntero lo elimina del contenedor, poniendo a cero la posición que ocupaba. Cuando se produce la llamada al destructor de `PStash`, se prueba si han sido previamente retirados todos los punteros, si no es así, se notifica, de modo que es posible prevenir la fuga de memoria. Se verán otras soluciones mas elegantes en capítulos posteriores.

Una prueba

Aquí aparece el programa de prueba de `Stash`, reescrito para `PStash`:

```
//: C13:PStashTest.cpp
//{L} PStash
// Test of pointer Stash
#include "PStash.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  PStash intStash;
  // 'new' works with built-in types, too. Note
  // the "pseudo-constructor" syntax:
  for(int i = 0; i < 25; i++)
    intStash.add(new int(i));
```

```
  for(int j = 0; j < intStash.count(); j++)
    cout << "intStash[" << j << "] = "
         << *(int*)intStash[j] << endl;
  // Clean up:
  for(int k = 0; k < intStash.count(); k++)
    delete intStash.remove(k);
  ifstream in ("PStashTest.cpp");
  assure(in, "PStashTest.cpp");
  PStash stringStash;
  string line;
  while(getline(in, line))
    stringStash.add(new string(line));
  // Print out the strings:
  for(int u = 0; stringStash[u]; u++)
    cout << "stringStash[" << u << "] = "
         << *(string*)stringStash[u] << endl;
  // Clean up:
  for(int v = 0; v < stringStash.count(); v++)
    delete (string*)stringStash.remove(v);
} ///:~
```

Igual que antes, se crean y rellenan varias `Stash`, pero esta vez con los punteros obtenidos con `new`. En el primer caso, véase la línea:

```
      intStash.add(new int(i));
```

Se ha usado una forma de pseudo constructor en la expresión `new int(i)`, con lo que además de crear un objeto int en el área de memoria dinámica, le asigna el valor inicial `i`.

Para imprimir, es necesario convertir al tipo adecuado el puntero obtenido de `PStash::operator[]`; lo mismo se repite con el resto de los objetos de `PStatsh` del programa. Es la consecuencia indeseable del uso de punteros void como representación subyacente, que se corregirá en capítulos posteriores.

En la segunda prueba, se lee línea a línea el propio archivo fuente. Mediante `getline()` se lee cada línea de texto en una variable de cadena, de la que se crea una copia independiente. Si le hubiéramos pasado cada vez la dirección de `line`, tendríamos un montón de copias del mismo puntero, referidas a la última línea leída.

En, en la recuperación de los punteros, verá la expresión:

```
      *(string*)stringStash[v];
```

El puntero obtenido por medio de `operator[]` debe ser convertido a string* para tener el tipo adecuado. Después el string* es de-referenciado y es visto por el compilador como un objeto string que se envía a `cout`.

Antes de destruir los objetos, se han de eliminar las referencias correspondientes mediante el uso de `remove()`. De no hacerse así, `PStash` notificará que no se ha efectuado la limpieza correctamente. Véase que en el caso de los punteros a int, no es necesaria la conversión de tipo al carecer de destructor, y lo único que se necesita es liberar la memoria:

```
        delete intStash.remove(k);
```

En cambio, para los punteros a string, hace falta la conversión de tipo, so pena de crear otra (silenciosa) fuga de memoria, de modo que el molde es esencial:

```
        delete (string*) stringStash.remove(k);
```

Algunas de estas dificultades pueden resolverse mediante el uso de plantillas, que veremos en el capítulo 16. FIXME:ref

13.3. `new` y `delete` para vectores

En C++ es igual de fácil crear vectores de objetos en la pila o en el montículo, con la certeza de que se producirá la llamada al constructor para cada uno de los objetos del vector. Hay una restricción: debe existir un constructor por defecto, o sea, sin argumentos, que será invocado para cada objeto.

Cuando se crean vectores de objetos dinámicamente, usando `new`, hay otras cosas que hay que tener en cuenta. Como ejemplo de este tipo de vectores véase

```
        MyType* fp = new MyType[100];
```

Esta sentencia asigna espacio suficiente en el montículo para 100 objetos `MyType` y llama al constructor para cada uno de ellos. Lo que se ha obtenido es simplemente un MyType*, exactamente lo mismo que hubiera obtenido de esta otra forma, que crea un único objeto:

```
        MyType* fp2 = new MyType;
```

El escritor del programa sabe que `fp` es la dirección del primer elemento de un vector, por lo que tiene sentido seleccionar elementos del mismo mediante una expresión como `fp[3]`, pero ¿qué pasa cuando destruimos el vector?. Las sentencias

```
  delete fp2;  // Correcta
  delete fp;   // Ésta no átendr el efecto deseado
```

parecen iguales, y sus efectos serán los mismos. Se llamará al destructor del objeto `MyType` al que apunta el puntero dado y después se liberará el bloque asignado. Esto es correcto para `fp2`, pero no lo es para `fp`, significa que los destructores de los 99 elementos restantes del vector no se invocarán. Sin embargo, sí se liberará toda la memoria asignada al vector, ya que fue obtenida como un único gran bloque cuyo tamaño quedó anotado en alguna parte por las rutinas de asignación.

Esto se soluciona indicando al compilador que el puntero que pasamos es la dirección de inicio de un vector, usando la siguiente sintaxis:

```
        delete [] fp;
```

Los corchetes indican al compilador la necesidad de generar el código para obtener el número de objetos en el vector, que fue guardado en alguna parte cuando se

creó, y llamar al destructor para cada uno de dichos elementos. Esta es una mejora sobre la sintaxis primitiva, que puede verse ocasionalmente en el código de viejos programas:

```
delete [100] fp;
```

que forzaba al programador a incluir el número de objetos contenidos en el vector, introduciendo con ello una posible fuente de errores. El esfuerzo adicional que supone para el compilador tener en esto en cuenta es pequeño, y por eso se consideró preferible especificar el número de objetos en un lugar y no en dos.

13.3.1. Cómo hacer que un puntero sea más parecido a un vector

Como defecto colateral, existe la posibilidad de modificar el puntero `fp` anteriormente definido, para que apunte a cualquier otra cosa, lo que no es consistente con el hecho de ser la dirección de inicio de un vector. Tiene más sentido definirlo como una constante, de modo que cualquier intento de modificación sea señalado como un error. Para conseguir este efecto se podría probar con:

```
int const* q = new int[10];
```

o bien:

```
const int* q = new int[10];
```

pero en ambos casos el especificador `const` quedaría asociado al int, es decir, al valor al que apunta, en lugar de al puntero en sí. Si se quiere conseguir el efecto deseado, en lugar de las anteriores, se debe poner:

```
int* const q = new int[10];
```

Ahora es posible modificar el valor de los elementos del vector, siendo ilegal cualquier intento posterior de modificar `q`, como `q++` por ejemplo, al igual que ocurre con el identificador de un vector ordinario.

13.3.2. Cuando se supera el espacio de almacenamiento

¿Qué ocurre cuando `new()` no puede encontrar un bloque contiguo suficientemente grande para alojar el objeto? En este caso se produce la llamada a una función especial: el manejador de errores de `new` o *new-handler*. Para ello comprueba si un determinado puntero a función es nulo, si no lo es, se efectúa la llamada a la función a la que apunta.

El comportamiento por defecto del manejador de errores de `new` es disparar una excepción, asunto del que se tratará en el Volumen 2. Si se piensa usar la asignación dinámica, conviene al menos reemplazar el manejador de errores de `new` por una función que advierta de la falta de memoria y fuerce la terminación del programa. De este modo, durante la depuración del programa, se podrá seguir la pista de lo sucedido. Para la versión final del programa, será mejor implementar una recuperación de errores más elaborada.

La forma de reemplazar el manejador de *new-handler* por defecto consiste en incluir el archivo `new.h` y hacer una llamada a la función `set_new_handler()` con la dirección de la función que se desea instalar:

```
//: C13:NewHandler.cpp
// Changing the new-handler
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;

int count = 0;

void out_of_memory() {
  cerr << "memory exhausted after " << count
    << " allocations!" << endl;
  exit(1);
}

int main() {
  set_new_handler(out_of_memory);
  while(1) {
    count++;
    new int[1000]; // Exhausts memory
  }
} ///:~
```

La función a instalar debe retornar void y no tomar argumentos. El bucle `while` seguirá pidiendo bloques de int hasta consumir la memoria libre disponible, sin hacer nada con ellos. Justo a la siguiente llamada a `new`, no habrá espacio para asignar y se producirá la llamada al *manejador de `new`*.

Este comportamiento del *new-handler* está asociado al `operator new()`, de modo que si se sobrecarga `operator new()` (asunto que se trata en la siguiente sección), no se producirá la llamada al *manejador de `new`*. Si se desea que se produzca dicha llamada será necesario que lo haga en el `operator new()` que substituya al original.

Por supuesto, es posible escribir manejadores `new` más sofisticados, incluso alguno que intente reclamar los bloques asignados que no se usan (conocidos habitualmente como *recolectores de basura*). Pero este no es un trabajo adecuado para programadores noveles.

13.3.3. Sobrecarga de los operadores `new` y `delete`

Cuando se ejecuta una *expresión con `new`*, ocurren dos cosas. Primero se asigna la memoria al ejecutar el código del `operator new()` y después se realiza la llamada al constructor. En el caso de una *expresión con `delete`*, se llama primero al destructor y después se libera la memoria con el operador `operator delete()`. Las llamadas al constructor y destructor no están bajo el control del programador, pero se *pueden* cambiar las funciones `opertator new()` y `operatator delete()`.

El sistema de asignación de memoria usado por `new` y `delete` es un sistema de propósito general. En situaciones especiales, puede que no funcione como se requie-

re. Frecuentemente la razón para cambiar el asignador es la eficiencia; puede que se necesite crear y destruir tantos objetos de la misma clase que lo haga ineficaz en términos de velocidad: un cuello de botella. En C++ es posible sobrecargar `new` y `delete` para implementar un esquema particular más adecuado que permita manejar situaciones como ésta.

Otra cuestión es la fragmentación del montículo. Cuando los objetos tienen tamaños diferentes es posible llegar a dividir de tal modo el área de memoria libre que se vuelva inútil. Es decir, el espacio puede estar disponible, pero debido al nivel de fragmentación alcanzado, no exista ningún bloque del tamaño requerido. Es posible asegurarse de que esto no llegue a ocurrir mediante la creación de un asignador para una clase específica.

En los sistemas de tiempo real y en los sistemas integrados, suele ser necesario que los programas funcionen por largo tiempo con recursos muy limitados. Tales sistemas pueden incluso requerir que cada asignación tome siempre la misma cantidad de tiempo, y que no esté permitida la fragmentación ni el agotamiento en el área dinámica. La solución a este problema consiste en utilizar un asignador «personalizado»; de otro modo, los programadores evitarían usar `new` y `delete` es estos casos y desperdiciarían un recurso muy valioso de C++.

A la hora de sobrecargar `operator new()` y `operator delete()` es importante tener en cuenta que lo único que se está cambiando es la forma en que se realiza la asignación del espacio. El compilador llamará a la nueva versión de `new` en lugar de al original, para asignar espacio, llamando después al constructor que actuará sobre él. Así que, aunque el compilador convierte una expresión `new` en código para asignar el espacio y para llamar al constructor, todo lo que se puede cambiar al sobrecargar `new` es la parte correspondiente a la asignación. `delete` tiene una limitación similar.

Cuando se sobrecarga `operator new()`, se está reemplazando también el modo de tratar los posibles fallos en la asignación de la memoria. Se debe decidir qué acciones va a realizar en tal caso: devolver cero, un bucle de reintento con llamada al *new-handler*, o lo que es más frecuente, disparar una excepción *bad_alloc* (tema que se trata en el Volumen 2).

La sobrecarga de `new` y `delete` es como la de cualquier otro operador. Existe la posibilidad de elegir entre sobrecarga global y sobrecarga para una clase determinada.

Sobrecarga global de `new` y `delete`

Este es el modo más drástico de abordar el asunto, resulta útil cuando el comportamiento de `new` y `delete` no es satisfactorio para la mayor parte del sistema. Al sobrecargar la versión global, quedan inaccesibles las originales, y ya no es posible llamarlas desde dentro de las funciones sobrecargadas.

El `new` sobrecargado debe tomar un argumento del tipo size_t (el estándar de C) para tamaños. Este argumento es generado y pasado por el compilador, y se refiere al tamaño del objeto para el que ahora tenemos la responsabilidad de la asignación de memoria. Debe devolver un puntero a un bloque de ese tamaño, (o mayor, si hubiera motivos para hacerlo así), o cero en el caso de no se encontrara un bloque adecuado. Si eso sucede, no se producirá la llamada al constructor. Por supuesto, hay que hacer algo más informativo que sólo devolver cero, por ejemplo llamar al «new-handler» o disparar una excepción, para indicar que hubo un problema.

El valor de retorno de `operator new()` es void*, no un puntero a un tipo particular. Lo que hace es obtener un bloque de memoria, no un objeto definido, no hasta

que que sea llamado el constructor, un acto que el compilador garantiza y que está fuera del control de este operador.

El operador `operator delete()` toma como argumento un puntero void* a un bloque obtenido con el `operator new()`. Es un void* ya que el `delete` obtiene el puntero sólo *después* de que haya sido llamado el destructor, lo que efectivamente elimina su caracter de objeto convirtiéndolo en un simple bloque de memoria. El tipo de retorno para `delete` es void.

A continuación se expone un ejemplo del modo de sobrecargar globalmente `new` y `delete`:

```
//: C13:GlobalOperatorNew.cpp
// Overload global new/delete
#include <cstdio>
#include <cstdlib>
using namespace std;

void* operator new(size_t sz) {
  printf("operator new: %d Bytes\n", sz);
  void* m = malloc(sz);
  if(!m) puts("out of memory");
  return m;
}

void operator delete(void* m) {
  puts("operator delete");
  free(m);
}

class S {
  int i[100];
public:
  S() { puts("S::S()"); }
  ~S() { puts("S::~S()"); }
};

int main() {
  puts("creating & destroying an int");
  int* p = new int(47);
  delete p;
  puts("creating & destroying an s");
  S* s = new S;
  delete s;
  puts("creating & destroying S[3]");
  S* sa = new S[3];
  delete []sa;
} ///:~
```

Aquí puede verse la forma general de sobrecarga de operadores `new` y `delete`. Estos operadores sustitutivos usan las funciones `malloc()` y `free()` de la biblioteca estándar de C, que es probablemente lo que ocurre en los operadores originales. Imprimen también mensajes sobre lo que están haciendo. Nótese que no se han usado `iostreams` sino `printf()` y `puts()`. Esto se hace debido a que los objetos iostream como los globales `cin`, `cout` y `cerr` llaman a `new` para obtener memoria

[3]. Usar printf() evita el fatal bloqueo, ya que no hace llamadas a new.

En main(), se crean algunos objetos de tipos básicos para demostrar que también en estos casos se llama a los operadores new y delete sobrecargados. Posteriormente, se crean un objeto simple y un vector, ambos de tipo S. En el caso del vector se puede ver, por el número de bytes pedidos, que se solicita algo de memoria extra para incluir información sobre el número de objetos que tendrá. En todos los casos se efectúa la llamada a las versiones globales sobrecargadas de new y delete.

Sobrecarga de new y delete específica para una clase

Aunque no es necesario poner el modificador static, cuando se sobrecarga new y delete para una clase se están creando métodos estáticos (métodos de clase). La sintaxis es la misma que para cualquier otro operador. Cuando el compilador encuentra una expresión new para crear un objeto de una clase, elige, si existe, un método de la clase llamado operator new() en lugar del new global. Para el resto de tipos o clases se usan los operadores globales (a menos que tengan definidos los suyos propios).

En el siguiente ejemplo se usa un primitivo sistema de asignación de almacenamiento para la clase Framis. Se reserva un bloque de memoria en el área de datos estática FIXME , y se usa esa memoria para asignar alojamiento para los objetos de tipo Framis. Para determinar qué bloques se han asignado, se usa un sencillo vector de bytes, un byte por bloque.

```
//: C13:Framis.cpp
// Local overloaded new & delete
#include <cstddef> // Size_t
#include <fstream>
#include <iostream>
#include <new>
using namespace std;
ofstream out("Framis.out");

class Framis {
  enum { sz = 10 };
  char c[sz]; // To take up space, not used
  static unsigned char pool[];
  static bool alloc_map[];
public:
  enum { psize = 100 };  // frami allowed
  Framis() { out << "Framis()\n"; }
  ~Framis() { out << "~Framis() ... "; }
  void* operator new(size_t) throw(bad_alloc);
  void operator delete(void*);
};
unsigned char Framis::pool[psize * sizeof(Framis)];
bool Framis::alloc_map[psize] = {false};

// Size is ignored -- assume a Framis object
void*
Framis::operator new(size_t) throw(bad_alloc) {
  for(int i = 0; i < psize; i++)
    if(!alloc_map[i]) {
      out << "using block " << i << " ... ";
      alloc_map[i] = true; // Mark it used
```

[3] Provocaría una serie continua de llamadas a new hasta agotar la pila y abortaría el programa.

```
      return pool + (i * sizeof(Framis));
    }
  out << "out of memory" << endl;
  throw bad_alloc();
}

void Framis::operator delete(void* m) {
  if(!m) return; // Check for null pointer
  // Assume it was created in the pool
  // Calculate which block number it is:
  unsigned long block = (unsigned long)m
    - (unsigned long)pool;
  block /= sizeof(Framis);
  out << "freeing block " << block << endl;
  // Mark it free:
  alloc_map[block] = false;
}

int main() {
  Framis* f[Framis::psize];
  try {
    for(int i = 0; i < Framis::psize; i++)
      f[i] = new Framis;
    new Framis; // Out of memory
  } catch(bad_alloc) {
    cerr << "Out of memory!" << endl;
  }
  delete f[10];
  f[10] = 0;
  // Use released memory:
  Framis* x = new Framis;
  delete x;
  for(int j = 0; j < Framis::psize; j++)
    delete f[j]; // Delete f[10] OK
} ///:~
```

El espacio de almacenamiento para el montículo `Framis` se crea sobre el bloque obtenido al declarar un vector de tamaño suficiente para contener `psize` objetos de clase `Framis`. Se ha declarado también una variable lógica para cada uno de los `psize` bloques en el vector. Todas estas variables lógicas son inicializadas a `false` usando el truco consistente en inicializar el primer elemento para que el compilador lo haga automáticamente con los restantes iniciándolos a su valor por defecto, `false`, en el caso de variables lógicas.

El `operador new()` local usa la misma sintaxis que el global. Lo único que hace es buscar una posición libre, es decir, un valor `false` en el mapa de localización `alloc_map`. Si la encuentra, cambia su valor a `true` para marcarla como ocupada, y devuelve la dirección del bloque correspondiente. En caso de no encontrar ningún bloque libre, envía un mensaje al fichero de trazas y dispara una excepción de tipo bad_alloc.

Este es el primer ejemplo con excepción que aparece en este libro. En el Volumen 2 se verá una discusión detallada del tratamiento de excepciones, por lo que en este ejemplo se hace un uso muy simple del mismo. En el `operador new` hay dos expresiones relacionadas con el tratamiento de excepciones. Primero, a la lista de argumentos de función le sigue la expresión `throw(bad_alloc)`, esto informa

al compilador que la función puede disparar una excepción del tipo indicado. En segundo lugar, si efectivamente se agota la memoria, la función alcanzará la sentencia `throw bad_alloc()` lanzando la excepción. En el caso de que esto ocurra, la función deja de ejecutarse y se cede el control del programa a la rutina de tratamiento de excepción que se ha definido en una cláusula `catch(bad_alloc)`.

En `main()` se puede ver la cláusula *try-catch* que es la otra parte del mecanismo. El código que puede lanzar la excepción queda dentro del bloque `try`; en este caso, llamadas a `new` para objetos `Framis`. Justo a continuación de dicho bloque sigue una o varias cláusulas `catch`, especificando en cada una la excepción a la que se destina. En este caso, `catch(bad_alloc)` indica que en ese bloque se tratarán las excepciones de tipo bad_alloc. El código de este bloque sólo se ejecutará si se dispara la excepción, continuando la ejecución del programa justo después de la última del grupo de cláusulas `catch` que existan. Aquí sólo hay una, pero podría haber más.

En este ejemplo, el uso de `iostream` es correcto ya que el `operator new()` global no ha sido modificado.

El `operator delete()` asume que la dirección de `Framis` ha sido obtenida de nuestro almacén particular. Una asunción justa, ya que cada vez que se crea un objeto `Framis` simple se llama al `operator new()` local; pero cuando se crea un vector de tales objetos se llama al `new` global. Esto causaría problemas si el usuario llamara accidentalmente al operador `delete` sin usar la sintaxis para destrucción de vectores. Podría ser que incluso estuviera tratando de borrar un puntero a un objeto de la pila. Si cree que estas cosas puedan suceder, conviene pensar en añadir una línea que asegurare que la dirección está en el intervalo correcto (aquí se demuestra el potencial que tiene la sobrecarga de los operadores `new` y `delete` para la localización de fugas de memoria).

`operador delete()` calcula el bloque al que el puntero representa y después pone a `false` la bandera correspondiente en el mapa de localización, para indicar que dicho bloque está libre.

En la función `main()`, se crean dinámicamente suficientes objetos `Framis` para agotar la memoria. Con esto se prueba el comportamiento del programa en este caso. A continuación, se libera uno de los objetos y se crea otro para mostrar la reutilización del bloque recién liberado.

Este esquema específico de asignación de memoria es probablemente mucho más rápido que el esquema de propósito general que usan los operadores `new` y `delete` originales. Se debe advertir, no obstante, que este enfoque no es automáticamente utilizable cuando se usa herencia, un tema que verá en el Capítulo 14 (FIXME).

Sobrecarga de `new` y `delete` para vectores

Si se sobrecargan los operadores `new` y `delete` para una clase, esos operadores se llaman cada vez que se crea un objeto simple de esa clase. Sin embargo, al crear un vector de tales objetos se llama al `operator new()` global para obtener el espacio necesario para el vector, y al `operator delete()` global para liberarlo. Es posible controlar también la asignación de memoria para vectores sobrecargando los métodos `operator new[]` y `operator delete[]`; se trata de versiones especiales para vectores. A continuación se expone un ejemplo que muestra el uso de ambas versiones.

```
//: C13:ArrayOperatorNew.cpp
// Operator new for arrays
#include <new> // Size_t definition
```

```
#include <fstream>
using namespace std;
ofstream trace("ArrayOperatorNew.out");

class Widget {
  enum { sz = 10 };
  int i[sz];
public:
  Widget() { trace << "*"; }
  ~Widget() { trace << "~"; }
  void* operator new(size_t sz) {
    trace << "Widget::new: "
          << sz << " bytes" << endl;
    return ::new char[sz];
  }
  void operator delete(void* p) {
    trace << "Widget::delete" << endl;
    ::delete []p;
  }
  void* operator new[](size_t sz) {
    trace << "Widget::new[]: "
          << sz << " bytes" << endl;
    return ::new char[sz];
  }
  void operator delete[](void* p) {
    trace << "Widget::delete[]" << endl;
    ::delete []p;
  }
};

int main() {
  trace << "new Widget" << endl;
  Widget* w = new Widget;
  trace << "\ndelete Widget" << endl;
  delete w;
  trace << "\nnew Widget[25]" << endl;
  Widget* wa = new Widget[25];
  trace << "\ndelete []Widget" << endl;
  delete []wa;
} ///:~
```

Si exceptuamos la información de rastreo que se añade aquí, las llamadas a las versiones globales de `new` y `delete` causan el mismo efecto que si estos operadores no se hubieran sobrecargado. Como se ha visto anteriormente, es posible usar cualquier esquema conveniente de asignación de memoria en estos operadores modificados.

Se puede observar que la sintaxis de `new` y `delete` para vectores es la misma que la usada para objetos simples añadiéndoles el operador subíndice `[]`. En ambos casos se le pasa a `new` como argumento el tamaño del bloque de memoria solicitado. A la versión para vectores se le pasa el tamaño necesario para albergar todos sus componentes. Conviene tener en cuenta que lo único que se requiere del `operator new()` es que devuelva un puntero a un bloque de memoria suficientemente grande. Aunque es posible inicializar el bloque referido, eso es trabajo del constructor, que se llamará automáticamente por el compilador.

El constructor y el destructor simplemente imprimen mensajes para que pueda verse que han sido llamados. A continuación se muestran dichos mensajes:

```
new Widget
Widget::new: 40 bytes
*
delete Widget
~Widget::delete
new Widget[25]
Widget::new: 1004 bytes
************************
delete []Widget
~~~~~~~~~~~~~~~~~~~~~~~~Widget::delete[]
```

La creación de un único objeto `Widget` requiere 40 bytes, tal y como se podría esperar para una máquina que usa 32 bits para un int. Se invoca al `operator new()` y luego al constructor, que se indica con la impresión del carácter «*». De forma complementaria, la llamada a `delete` provoca primero la invocación del destructor y sólo después, la de `operator delete()`.

Cuando lo que se crea es un vector de objetos `Widget`, se observa el uso de la versión de `operator new()` para vectores, de acuerdo con lo dicho anteriormente. Se observa que el tamaño del bloque solicitado en este caso es cuatro bytes mayor que el esperado. Es en estos cuatro bytes extra donde el compilador guarda la información sobre el tamaño del vector. De ese modo, la expresión

```
delete []Widget;
```

informa al compilador que se trata de un vector, con lo cual, generará el código para extraer la información que indica el número de objetos y para llamar otras tantas veces al destructor. Obsérvese que aunque se llame solo una vez a `operator new()` y `operator delete()` para el vector, se llama al constructor y al destructor una vez para cada uno de los objetos del vector.

Llamadas al constructor

Considerando que

```
MyType* f = new MyType;
```

llama a `new` para obtener un bloque del tamaño de `MyType` invocando después a su constructor, ¿qué pasaría si la asignación de memoria falla en `new`?. En tal caso, no habrá llamada al constructor al que se le tendría que pasar un puntero `this` nulo, para un objeto que no se ha creado . He aquí un ejemplo que lo demuestra:

```
//: C13:NoMemory.cpp
// Constructor isn't called if new fails
#include <iostream>
#include <new> // bad_alloc definition
using namespace std;

class NoMemory {
public:
  NoMemory() {
    cout << "NoMemory::NoMemory()" << endl;
  }
  void* operator new(size_t sz) throw(bad_alloc){
```

```
    cout << "NoMemory::operator new" << endl;
    throw bad_alloc(); // "Out of memory"
  }
};

int main() {
  NoMemory* nm = 0;
  try {
    nm = new NoMemory;
  } catch(bad_alloc) {
    cerr << "Out of memory exception" << endl;
  }
  cout << "nm = " << nm << endl;
} ///:~
```

Cuando se ejecuta, el programa imprime los mensajes del `operator new()` y del manejador de excepción, pero no el del constructor. Como `new` nunca retorna, no se llama al constructor y por tanto no se imprime su mensaje.

Para asegurar que no se usa indebidamente, Es importante inicializar `nm` a cero, debido a que `new` no se completa. El código de manejo de excepciones debe hacer algo más que imprimir un mensaje y continuar como si el objeto hubiera sido creado con éxito. Idealmente, debería hacer algo que permitiera al programa recuperarse del fallo, o al menos, provocar la salida después de registrar un error.

En las primeras versiones de C++, el comportamiento estándar consistía en hacer que `new` retornara un puntero nulo si la asignación de memoria fallaba. Esto podía impedir que se llamara al constructor. Si se intenta hacer esto con un compilador que sea conforme al estándar actual, le informará de que en lugar de devolver un valor nulo, debe disparar una excepción de tipo bad_alloc.

Operadores `new` y `delete` de [FIXME emplazamiento (situación)]

He aquí otros dos usos, menos comunes, para la sobrecarga de `operador new-()`:

1. Puede ocurrir que necesite emplazar un objeto en un lugar específico de la memoria. Esto puede ser importante en programas en los que algunos de los objetos se refieren o son sinónimos de componentes hardware mapeados sobre una zona de la memoria.

2. Si se quiere permitir la elección entre varios asignadores de memoria (allocators) en la llamada a `new`.

Ambas situaciones se resuelven mediante el mismo mecanismo: la función `operator new()` puede tomar más de un argumento. Como se ha visto, el primer argumento de `new` es siempre el tamaño del objeto, calculado en secreto y pasado por el compilador. El resto de argumentos puede ser de cualquier otro tipo que se necesite: la dirección en la que queremos emplazar el objeto, una referencia a una función de asignación de memoria, o cualquiera otra cosa que se considere conveniente.

Al principio puede parecer curioso el modo en que se pasan los argumentos extra al `operator new()`. Después de la palabra clave `new` y antes del nombre de clase del objeto que se pretende crear, se pone la lista de argumentos, sin contar con el

correspondiente al size_t del objeto, que le pasa el compilador. Por ejemplo, la expresión:

```
    X* xp = new(a) X;
```

pasará a como segundo argumento al operador operator new(). Por supuesto, sólo funcionará si ha sido declarado el operator new() adecuado.

He aquí un ejemplo demostrativo de cómo se usa esto para colocar un objeto en una posición particular:

```
//: C13:PlacementOperatorNew.cpp
// Placement with operator new()
#include <cstddef> // Size_t
#include <iostream>
using namespace std;

class X {
  int i;
public:
  X(int ii = 0) : i(ii) {
    cout << "this = " << this << endl;
  }
  ~X() {
    cout << "X::~X(): " << this << endl;
  }
  void* operator new(size_t, void* loc) {
    return loc;
  }
};

int main() {
  int l[10];
  cout << "l = " << l << endl;
  X* xp = new(l) X(47); // X at location l
  xp->X::~X(); // Explicit destructor call
  // ONLY use with placement!
} ///:~
```

Observe que lo único que hace el operador new es retornar el puntero que se pasa. Por tanto, es posible especificar la dirección en la que se quiere construir el objeto.

Aunque este ejemplo muestra sólo un argumento adicional, nada impide añadir otros, si se considera conveniente para sus propósitos.

Al tratar de destruir estos objetos surge un problema. Sólo hay una versión del operador delete, de modo que no hay forma de decir: "Usa mi función de liberación de memoria para este objeto". Se requiere llamar al destructor, pero sin utilizar el mecanismo de memoria dinámica, ya que el objeto no está alojado en el montículo.

La solución tiene una sintaxis muy especial. Se debe llamar explícitamente al destructor, tal como se muestra:

```
   xp->X::~X();  //Llamada íexplcita al destructor
```

Hay que hacer una llamada de atención al respecto. Algunas personas ven esto como un modo de destruir objetos en algún momento anterior al determinado por las reglas de ámbito, en lugar de ajustar el ámbito, o más correctamente, en lugar de usar asignación dinámica como medio de determinar la duración del objeto en tiempo de ejecución. Esto es un error, que puede provocar problemas si se trata de destruir de esta manera un objeto ordinario creado en la pila, ya que el destructor será llamado de nuevo cuando se produzca la salida del ámbito correspondiente. Si se llama de esta forma directa al destructor de un objeto creado dinámicamente, se llevará a cabo la destrucción, pero no la liberación del bloque de memoria, lo que probablemente no es lo que se desea. La única razón para este tipo de llamada explícita al destructor es permitir este uso especial del operador `new`, para emplazamiento en memoria.

Existe también una forma de operador `delete` de emplazamiento que sólo es llamada en caso de que el constructor dispare una excepción, con lo que la memoria se libera automáticamente durante la excepción. El operador `delete` de emplazamiento usa una lista de argumentos que se corresponde con la del operador `new` de emplazamiento que fue llamado previamente a que el constructor lanzase la excepción. Este asunto se tratará en el Volumen 2, en un capítulo dedicado al tratamiento de excepciones.

13.4. Resumen

La creación de objetos en la pila es eficaz y conveniente, pero para resolver el problema general de programación es necesario poder crear y destruir objetos en cualquier momento en tiempo de ejecución, en particular, para que pueda responder a la información externa al programa. Aunque C ofrece funciones de asignación dinámica, éstas no proporcionan la facilidad de uso ni la construcción garantizada de objetos que se necesita en C++. Al llevar al núcleo mismo del lenguaje gracias al uso de los operadores `new` y `delete`, la creación dinámica de objetos se hace tan fácil como la creación de objetos en la pila, añadiendo además una gran flexibilidad. Se puede modificar el comportamiento de `new` y `delete` si no se ajusta a los requerimientos, particularmente para mejorar la eficiencia, y también es posible definir su comportamiento en caso de agotarse la memoria libre.

13.5. Ejercicios

Las soluciones a los ejercicios se pueden encontrar en el documento electrónico titulado «The Thinking in C++ Annotated Solution Guide», disponible por poco dinero en www.BruceEckel.com.

1. Crear una clase `Counted` que contenga un `int id` y un `static int count`. El constructor por defecto debe empezar con `Counted():id(count++){`. También deberá mostrar mensajes con su `id`, además de alguno que muestre que se está creando. El destructor debe mostrar que está siendo destruido y su `id`. Probar su funcionamiento.
2. Compruebe que `new` y `delete` llaman siempre a constructores y destructores, creando mediante el uso de `new` un objeto de la clase `Counted` del ejercicio 1, y destruyéndolo después con `delete`. Cree y destruya un vector de `Counted` en el montículo.
3. Cree un objeto de la clase `PStash`, y llénelo de los objetos del ejercicio 1. Observe lo que sucede cuando el objeto `PStash` sale de su ámbito y es llamado

su destructor.

4. Cree un vector de `Counted*` y cárguelo con punteros a objetos `Counted`. Recorra el vector llamando imprimiendo cada objeto, repita este paso y elimínelos uno a uno.
5. Repita el ejercicio 4 añadiendo una función miembro `f()` de `Counted` que muestre un mensaje. Recorra el vector llamando a `f()` para cada objeto del vector.
6. Repita el ejercicio 5 usando un objeto `PStash`.
7. Repita el ejercicio 5 usando `Stack4.h` del capítulo 9.
8. Cree mediante asignación dinámica un vector de objetos de clase `Counted`. Llame a `delete` con el puntero resultante como argumento, sin usar el operador subíndice []. Explique el resultado.
9. Cree un objeto de clase `Counted` mediante `new`, convierta el puntero resultante a `void*` y luego bórrelo. Explique el resultado.
10. Compile y ejecute el programa `NewHandler.cpp` en su ordenador. A partir del número resultante, calcule la cantidad de memoria libre disponible para su programa.
11. Cree una clase y defina en ella operadores de sobrecarga para `new` y `delete`, para objetos simples y para vectores de objetos. Demuestre que ambas versiones funcionan.
12. Diseñe un test que le permita evaluar de forma aproximada la mejora en velocidad obtenida en `Framis.cpp` con el uso de las versiones adaptadas de `new` y `delete`, respecto de la obtenida con las globales .
13. Modifique `NoMemory.cpp` para que contenga un vector de enteros y realmente obtenga memoria en lugar de disparar `bad_alloc`. Establezca un bucle `while` en el cuerpo de `main()` similar al que existe en `NewHandler.cpp` para agotar la memoria. Observe lo que sucede en el caso de que su operador `new` no compruebe el éxito de la asignación de memoria. Añada después esa comprobación a su operador `new` y la llamada a `throw bad_alloc`.
14. Cree una clase y defina un operador `new` de emplazamiento, con un string como segundo argumento. Defina un vector de string, en el que se almacenará este segundo argumento a cada llamada a `new`. El operador `new` de emplazamiento asignará bloques de manera normal. En `main()`, haga llamadas a este operador `new` pasándole como argumentos cadenas de caracteres que describan las llamadas. Para ello, puede hacer uso de las macros `__FILE__` y `__LINE__` del preprocesador.
15. Modifique `ArrayOperatorNew.cpp` definiendo un vector estático de Widget* que añada la dirección de cada uno de los objetos `Widget` asignados con `new`, y la retire cuando sea liberada mediante `delete`. Puede que necesite buscar información sebre vectores en la documentación de la biblioteca estándar de C++, o en el segundo volumen de este libro que está disponible en la web del autor. Cree una segunda clase a la que llamará `MemoryChecker`, que contenga un destructor que muestre el número de punteros a `Widget` en su vector. Diseñe un programa con una única instancia global de `MemoryChecker`, y en `main()`, cree y destruya dinámicamente varios objetos y vectores de objetos `Widget`. Observe que `MemoryCheck` revela fugas de memoria.

14: Herencia y Composición

Una de las características más importantes acerca de C++ es la reutilización de código. Pero para ser revolucionario, necesita ser capaz de hacer algo más que copiar código y modificarlo.

Este es un enfoque de C y no fue demasiado bien. Como en la mayoría de los casos en C++, la solución gira alrededor de la clase. Se reutiliza código creando nuevas clases, pero en vez de crearlas desde la nada, utilizará clases existentes que alguien ha realizado y comprobado que funcionan correctamente.

La clave consiste en utilizar estas clases sin modificarlas. En este capítulo, aprenderá los dos modos de hacerlo. El primero es bastante directo: simplemente cree objetos de la clase existente dentro de la nueva clase. A esto se le llama composición porqué la nueva clase esta compuesta por objetos de clases ya existentes.

La segunda forma es mucho más sutil. Crear la nueva clase como un tipo de una clase existente. Literalmente se toma la forma de la clase existente y se añade código, pero sin modificar la clase ya existente. A este hecho mágico se le llama herencia, y la mayoría del trabajo es realizado por el compilador. La herencia es uno de los pilares de la programación orientada a objetos y tiene extensiones adicionales que serán exploradas en el capítulo 15.

Esto es, resulta que gran parte de la sintaxis y el comportamiento son similares tanto en la composición como en la herencia (lo cual tiene sentido; ambas son dos formas de crear nuevos tipos utilizando tipos ya existentes). En este capítulo, aprenderá acerca de los mecanismos para la reutilización de código.

14.1. Sintaxis de la composición

Realmente, ha utilizado la composición a lo largo de la creación de una clase. Ha estado construyendo clases principalmente con tipos predefinidos (y en ocasiones cadenas). Por esto, resulta fácil usar la composición con tipos definidos por el usuario.

Considere la siguiente clase:

```
//: C14:Useful.h
// A class to reuse
#ifndef USEFUL_H
#define USEFUL_H

class X {
  int i;
public:
```

```
  X() { i = 0; }
  void set(int ii) { i = ii; }
  int read() const { return i; }
  int permute() { return i = i * 47; }
};
#endif // USEFUL_H ///:~
```

En esta clase los miembros son privados, y entonces, es completamente seguro declarar un objeto del tipo X público en la nueva clase, y por ello, permitir una interfaz directa:

```
//: C14:Composition.cpp
// Reuse code with composition
#include "Useful.h"

class Y {
  int i;
public:
  X x; // Embedded object
  Y() { i = 0; }
  void f(int ii) { i = ii; }
  int g() const { return i; }
};

int main() {
  Y y;
  y.f(47);
  y.x.set(37); // Access the embedded object
} ///:~
```

Para acceder a las funciones miembro alojadas en el objeto (referido como subobjeto) simplemente requiere otra selección del miembro.

Es habitual hacer privado el objeto alojado, y por ello, formar parte de la capa de implementación (lo que significa que es posible cambiar la implementación si se desea). La interfaz de funciones de la nueva clase implica el uso del objeto alojado, pero no necesariamente imita a la interfaz del objeto.

```
//: C14:Composition2.cpp
// Private embedded objects
#include "Useful.h"

class Y {
  int i;
  X x; // Embedded object
public:
  Y() { i = 0; }
  void f(int ii) { i = ii; x.set(ii); }
  int g() const { return i * x.read(); }
  void permute() { x.permute(); }
};

int main() {
  Y y;
  y.f(47);
```

```
  y.permute();
} ///:~
```

Aquí, la función permute() se ha añadido a la interfaz de la clase, pero el resto funciones de X son utilizadas dentro de los miembros de Y.

14.2. Sintaxis de la herencia

La sintaxis en la composición es bastante obvia, en cambio en la herencia, la sintaxis es nueva y diferente.

Cuando hereda, realmente se expresa "Esta nueva clase es como esta otra vieja clase". Se comienza el código proporcionando el nombre de la clase, como se realiza normalmente, pero antes de abrir la llave del cuerpo de la clase, se colocan dos puntos y el nombre de la clase base (o de las clases bases, separadas por comas, para herencia múltiple). Una vez realizado, automáticamente se consiguen todos los miembros y las funciones de la clase base. Ejemplo:

```
//: C14:Inheritance.cpp
// Simple inheritance
#include "Useful.h"
#include <iostream>
using namespace std;

class Y : public X {
  int i; // Different from X's i
public:
  Y() { i = 0; }
  int change() {
    i = permute(); // Different name call
    return i;
  }
  void set(int ii) {
    i = ii;
    X::set(ii); // Same-name function call
  }
};

int main() {
  cout << "sizeof(X) = " << sizeof(X) << endl;
  cout << "sizeof(Y) = "
       << sizeof(Y) << endl;
  Y D;
  D.change();
  // X function interface comes through:
  D.read();
  D.permute();
  // Redefined functions hide base versions:
  D.set(12);
} ///:~
```

Como se puede observar, Y hereda de X, que significa que Y contendrá todos los miembros de X y todas las funciones de X. De hecho, Y contiene un subobjeto X como

si se hubiese creado un objeto X dentro de la clase Y en vez de heredar de X. Tanto los miembros objetos y la clase base son conocidos como subobjetos.

Todos los elementos privados de X continúan siendo privados en Y; esto es, aunque Y hereda de X no significa que Y pueda romper el mecanismo de protección. Los elementos privados de X continúan existiendo, ocupando su espacio - sólo que no se puede acceder a ellos directamente.

En main() observamos que los datos de Y están combinados con los datos de X porque sizeof(Y) es el doble de grande que el sizeof(X).

Observará que la clase base es precedida por public. Durante la herencia, por defecto, todo es privado. Si la clase base no estuviese precedida por public, significaría que todos los miembros públicos de la clase base serían privados en la clase derivada. Esto, en la mayoría de ocasiones no es lo deseado [51]; el resultado que se desea es mantener todos los miembros públicos de la clase base en la clase derivada. Para hacer esto, se usa la palabra clave public durante la herencia.

En change(), se utiliza a la función de la clase base permute(). La clase derivada tiene acceso directo a todas las funciones públicas de la clase base.

La función set() en la clase derivada redefine la función set() de la clase base. Esto es, si llama a las funciones read() y permute() de un objeto Y, conseguirá las versiones de la clase base (esto es lo que esta ocurriendo dentro de main()). Pero si llamamos a set() en un objeto Y, conseguiremos la versión redefinida. Esto significa que si no deseamos un comportamiento de una función durante la herencia, se puede cambiar. (También se pueden añadir funciones completamente nuevas como change().)

Sin embargo, cuando redefinimos una función, puede ser que desee llamar a la versión de la clase base. Si, dentro de set(), simplemente llama a set(), conseguiremos una versión local de la función - una función recursiva. Para llamar a la versión de la clase base, se debe explícitamente utilizar el nombre de la clase base y el operador de resolución de alcance.

14.3. Lista de inicializadores de un constructor

Hemos visto lo importante que es en C++ garantizar una correcta inicialización, y esto no va a cambiar en la composición ni en la herencia. Cuando se crea un objeto, el compilador garantiza la ejecución todos los constructores para cada uno de los subobjetos. Hasta ahora, en los ejemplos, todos los subobjetos tienen un constructor por defecto, que es ejecutado por el compilador automáticamente. Pero que ocurre si uno de nuestros subobjetos no tiene constructores por defecto, o si queremos cambiar los parámetros por defecto de un constructor. Esto supone un problema, porque el constructor de la nueva clase no tiene permiso para acceder a los miembros privados del subobjeto y por ello, no puede inicializarlos directamente.

La solución es simple: ejecutar el constructor del subobjeto. C++ proporciona una sintaxis especial para ello, la lista de inicializadores de un constructor. La forma de la lista de inicializadores de un constructor demuestra como actúa como la herencia. Con la herencia, las clases bases son colocadas después de dos puntos y justo después, puede abrir la llave para empezar con el cuerpo de la clase. En la lista de inicializadores de un constructor, se coloca las llamadas a los constructores de los subobjetos, después los argumentos del constructor y los dos puntos, pero todo esto, antes de abrir el brazo del cuerpo de la función. En una clase MyType, que hereda de Bar, sería de la siguiente manera:

```
MyType::MyType(int i) : Bar(i) { // ...
```

si el constructor de Bar tuviera un solo parámetro del tipo int.

14.3.1. Inicialización de objetos miembros

La inicialización de objetos miembros de una clase utiliza la misma sintaxis cuando se usa la composición. Para la composición, se proporcionan los nombres de los objetos en lugar de los nombres de las clases. Si se tiene más de una llamada al constructor en la lista de inicializadores, las llamadas se separan con comas:

```
MyType2::MyType2(int i) : Bar(i), m(i+1) { // ...
```

Esta sería la forma de un constructor de la clase MyType2, la cual hereda de Bar y contiene un miembro objeto llamado m. Fíjese que mientras podemos ver el tipo de la clase base en la lista de inicializadores del constructor, sólo podemos ver el miembro identificador objeto.

14.3.2. Tipos predefinidos en la lista de inicializadores

La lista de inicializadores del constructor permite invocar explícitamente a los constructores de los objetos miembros. De hecho, no existe otra forma de llamar a esos constructores. La idea es que los constructores son llamados antes de la ejecución del cuerpo del constructor de la nueva clase. De esta forma, cualquier llamada que hagamos a las funciones miembros de los subobjetos siempre serán objetos inicializados. No existe otra manera de acceder al cuerpo del constructor sin que ningún constructor llame a todos los miembros objetos y los objetos de la clase base, es más, el compilador crea un constructor oculto por defecto. Esto es otra característica de C++, que garantiza que ningún objeto (o parte de un objeto) puedan estar desde un principio sin que su constructor sea llamado.

La idea de que todos los objetos miembros estén inicializados al inicio del constructor es una buena ayuda para programar. Una vez en el inicio del constructor, puede asumir que todos los subobjetos están correctamente inicializados y centrarse en las tareas que se desean realizar en el constructor. Sin embargo, existe un contratiempo: ¿Qué ocurre con los objetos predefinidos, aquellos que no tienen constructor?

Para hacer una sintaxis sólida, piense en los tipos predefinidos como si tuviesen un solo constructor, con un solo parámetro: una variable del mismo tipo como el que esta inicializando. Esto es

```
//: C14:PseudoConstructor.cpp
class X {
  int i;
  float f;
  char c;
  char* s;
public:
  X() : i(7), f(1.4), c('x'), s("howdy") {}
};

int main() {
```

```
  X x;
  int i(100);  // Applied to ordinary definition
  int* ip = new int(47);
} ///:~
```

El propósito de esta "pseudo-llamadas a los constructores" es una simple asignación. Es una técnica recomendada y un buen estilo de programación, que usted verá usar a menudo.

ncluso es posible utilizar esta sintaxis cuando se crean variables de tipos predefinidos fuera de la clase:

```
int i(100);
int* ip = new int(47);
```

De esta forma, los tipos predefinidos actúan, más o menos, como los objetos. Sin embargo, recuerde que no son constructores reales. En particular, si usted no realiza una llamada explícita al constructor, no se ejecutará ninguna inicialización.

14.3.3. Combinación de composición y herencia

Por supuesto, usted puede usar la composición y la herencia a la vez. El siguiente ejemplo muestra la creación de una clase más compleja utilizando composición y herencia.

```
//: C14:Combined.cpp
// Inheritance & composition

class A {
  int i;
public:
  A(int ii) : i(ii) {}
  ~A() {}
  void f() const {}
};

class B {
  int i;
public:
  B(int ii) : i(ii) {}
  ~B() {}
  void f() const {}
};

class C : public B {
  A a;
public:
  C(int ii) : B(ii), a(ii) {}
  ~C() {} // Calls ~A() and ~B()
  void f() const {  // Redefinition
    a.f();
    B::f();
  }
};
```

```
int main() {
  C c(47);
} ///:~
```

C hereda de B y tiene un objeto miembro ("esta compuesto de") del tipo A. Puede comparar que la lista de inicializadores contiene las llamadas al constructor de la clase base y las constructores de los objetos miembros.

La función C::f() redefine B::f(), que era heredada, y también llama a la versión de la clase base. Además, se llama a a.f(). Fíjese que durante todo este tiempo estamos hablando acerca de la redefinición de funciones durante la herencia; con un objeto miembro que sólo se puede manipular su interfaz pública, no redefinirla. Además, al llamar a f() en un objeto de la clase C no podrá llamar a a.f() si C::f() no ha sido definido, mientras que sería posible llamar a B::f().

Llamadas automáticas al destructor

Aunque muy a menudo sea necesario realizar llamadas explicitas a los constructores en la inicialización, nunca será necesario realizar una llamada explicita a los destructores porque sólo existe un destructor para cada clase y éste no tiene parámetros. Sin embargo, el compilador asegura que todos los destructores son llamados, esto significa que todos los destructores de la jerarquía, desde el destructor de la clase derivada y retrocediendo hasta la raíz, serán ejecutados.

Es necesario destacar que los constructores y destructores son un poco inusuales en el modo en que llaman a su jerarquía, en una función miembro normal sólo la función en si es llamada, ninguna versión de la clase base. Si usted desea llamar a la versión de la clase base de una función miembro normal, deberá sobrecargarla y deberá llamarla explícitamente.

14.3.4. Orden de llamada de constructores y destructores

Es importante conocer el orden de las llamadas de los constructores y destructores de un objeto con varios subobjetos. El siguiente ejemplo muestra cómo funciona:

```
//: C14:Order.cpp
// Constructor/destructor order
#include <fstream>
using namespace std;
ofstream out("order.out");

#define CLASS(ID) class ID { \
public: \
  ID(int) { out << #ID " constructor\n"; } \
  ~ID() { out << #ID " destructor\n"; } \
};

CLASS(Base1);
CLASS(Member1);
CLASS(Member2);
CLASS(Member3);
CLASS(Member4);

class Derived1 : public Base1 {
```

```
  Member1 m1;
  Member2 m2;
public:
  Derived1(int) : m2(1), m1(2), Base1(3) {
    out << "Derived1 constructor\n";
  }
  ~Derived1() {
    out << "Derived1 destructor\n";
  }
};

class Derived2 : public Derived1 {
  Member3 m3;
  Member4 m4;
public:
  Derived2() : m3(1), Derived1(2), m4(3) {
    out << "Derived2 constructor\n";
  }
  ~Derived2() {
    out << "Derived2 destructor\n";
  }
};

int main() {
  Derived2 d2;
} ///:~
```

Primero, se crea un objeto ofstream para enviar la salida a un archivo. Entonces, para no teclear tanto y demostrar la técnica de las macros que será sustituida por otra mucho más mejorada en el capítulo 16, se crea una para construir varias clases que utilizan herencia y composición. Cada constructor y destructor escribe información en el archivo de salida. Fíjense que los constructores no son constructores por defecto; cada uno tiene un parámetro del tipo int. Y además, el argumento no tiene nombre; la única razón de su existencia es forzar la llamada al constructor en la lista de inicializadores del constructor. (Eliminando el identificador evita que el compilador informe con mensajes de advertencia)

La salida de este programa es

```
Base1 constructor
Member1 constructor
Member2 constructor
Derived1 constructor
Member3 constructor
Member4 constructor
Derived2 constructor
Derived2 destructor
Member4 destructor
Member3 destructor
Derived1 destructor
Member2 destructor
Member1 destructor
Base1 destructor
```

omo puede observar, la construcción empieza desde la raíz de la jerarquía de clases y en cada nivel, el constructor de la clase base se ejecuta primero, seguido por los constructores de los objetos miembro. Los destructores son llamados exactamente en el orden inverso que los constructores -esto es importante debido a los problemas de dependencias (en el constructor de la clase derivada o en el destructor, se debe

asumir que el subobjeto de la clase base esta todavía disponible para su uso, y ha sido construido - o no se ha destruido todavía).

También es interesante que el orden de las llamadas al constructor para los objetos miembro no afecten para nada el orden de las llamadas en la lista de inicializadores de un constructor. El orden es determinado por el orden en que los objetos miembros son declarados en la clase. Si usted pudiera cambiar el orden del constructor en la lista de inicializadores de un constructor, usted podría tener dos secuencias diferentes de llamada en dos constructores diferentes, pero el destructor no sabría como invertir el orden para llamarse correctamente y nos encontraríamos con problemas de dependencias.

14.4. Ocultación de nombres

Si se ha heredado de una clase y se proporciona una nueva definición para alguna de sus funciones miembros, existen dos posibilidades. La primera es proporcionar los mismos argumentos y el mismo tipo de retorno en la definición de la clase derivada como la clase base. Esto es conocido como redefinición para funciones miembro ordinarias y sobrecarga, cuando la función miembro de la clase es una función virtual (las funciones virtuales son un caso normal y serán tratadas en detalle en el capítulo 15). Pero ¿qué ocurre cuando se modifican los argumentos de la función miembro o el tipo de retorno en una clase derivada? Aquí esta un ejemplo:

```
//: C14:NameHiding.cpp
// Hiding overloaded names during inheritance
#include <iostream>
#include <string>
using namespace std;

class Base {
public:
  int f() const {
    cout << "Base::f()\n";
    return 1;
  }
  int f(string) const { return 1; }
  void g() {}
};

class Derived1 : public Base {
public:
  void g() const {}
};

class Derived2 : public Base {
public:
  // Redefinition:
  int f() const {
    cout << "Derived2::f()\n";
    return 2;
  }
};

class Derived3 : public Base {
public:
```

```
  // Change return type:
  void f() const { cout << "Derived3::f()\n"; }
};

class Derived4 : public Base {
public:
  // Change argument list:
  int f(int) const {
    cout << "Derived4::f()\n";
    return 4;
  }
};

int main() {
  string s("hello");
  Derived1 d1;
  int x = d1.f();
  d1.f(s);
  Derived2 d2;
  x = d2.f();
//!  d2.f(s); // string version hidden
  Derived3 d3;
//!  x = d3.f(); // return int version hidden
  Derived4 d4;
//!  x = d4.f(); // f() version hidden
  x = d4.f(1);
} ///:~
```

En Base se observa una función sobrecargada f(), en Derived1 no se realiza ningún cambio a f() pero se redefine la función g(). En main(), se observa que ambas funciones f() están disponibles en Derived1. Sin embargo, Derived2 redefine una versión sobrecargada de f() pero no la otra, y el resultado es que la segunda forma de sobrecarga no esta disponible. En Derived3, se ha cambiado el tipo de retorno y esconde ambas versiones de la clase base, y Derived4 muestra que al cambiar la lista de argumentos también se esconde las versiones de la clase base. En general, usted puede expresar cada vez que redefine una función sobrecargada de la clase base, que todas las otras versiones son automáticamente escondidas en la nueva clase. En el capítulo 15, verá como añadir la palabra reservada virtual que afecta un significativamente a la sobrecarga de una función.

Si cambia la interfaz de la clase base modificando la signatura y/o el tipo de retorno de una función miembro desde la clase base, entonces esta utilizando la clase de una forma diferente en que la herencia esta pensado para realizar normalmente. Esto no quiere decir que lo que este haciendo sea incorrecto, esto es que el principal objetivo de la herencia es soportar el polimorfismo, y si usted cambia la signatura de la función o el tipo de retorno entonces realmente esta cambiando la interfaz de la clase base. Si esto es lo que esta intentando hacer entonces esta utilizando la herencia principalmente para la reutilización de código, no para mantener interfaces comunes en la clase base (que es un aspecto esencial del poliformismo). En general, cuando usa la herencia de esta forma significa que esta en una clase de propósito general y la especialización para una necesidad particular - que generalmente, pero no siempre, se considera parte de la composición.

Por ejemplo, considere la clase Stack del capítulo 9. Uno de los problemas con esta clase es que se tenía que realizar que convertir cada vez que se conseguía un puntero desde el contenedor. Esto no es solo tedioso, también inseguro - se puede

convertir a cualquier cosa que desee.

Un procedimiento que a primera vista parece mejor es especializar la clase general Stack utilizando la herencia. Aquí esta un ejemplo que utiliza la clase del capítulo 9.

```
//: C14:InheritStack.cpp
// Specializing the Stack class
#include "../C09/Stack4.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class StringStack : public Stack {
public:
  void push(string* str) {
    Stack::push(str);
  }
  string* peek() const {
    return (string*)Stack::peek();
  }
  string* pop() {
    return (string*)Stack::pop();
  }
  ~StringStack() {
    string* top = pop();
    while(top) {
      delete top;
      top = pop();
    }
  }
};

int main() {
  ifstream in("InheritStack.cpp");
  assure(in, "InheritStack.cpp");
  string line;
  StringStack textlines;
  while(getline(in, line))
    textlines.push(new string(line));
  string* s;
  while((s = textlines.pop()) != 0) { // No cast!
    cout << *s << endl;
    delete s;
  }
} ///:~
```

Como todas las funciones miembros en Stack4.h son inline, no es necesario ser enlazadas.

StringStack especializa Stack para que push() acepte solo punteros a String. Antes, Stack acepta punteros a void, y así el usuario no tenía que realizar una comprobación de tipos para asegurarse que el punteros fuesen insertados. Además, peek() and pop() ahora retornan punteros a String en vez de punteros a void, entonces no es necesario realizar la conversión para utilizar el puntero.

orprendido ¡este mecanismo de comprobación de tipos seguro es gratuito, en push(), peek() y pop! Al compilador se le proporciona información extra acerca de los tipos y éste lo utiliza en tiempo de compilación, pero las funciones son inline y no es necesario ningún código extra.

La ocultación de nombres entra en acción en la función push() que tiene la signatura diferente: la lista de argumentos. Si se tuviesen dos versiones de push() en la misma clase, tendrían que ser sobrecargadas, pero en este caso la sobrecarga no es lo que deseamos porque todavía permitiría pasar cualquier tipo de puntero a push como void *. Afortunadamente, C++ esconde la versión push (void *) en la clase base en favor de la nueva versión que es definida en la clase derivada, de este modo, solo se permite utilizar la función push() con punteros a String en StringStack.

Ahora podemos asegurar que se conoce exactamente el tipo de objeto que esta en el contenedor, el destructor funcionará correctamente y problema esta resuelto - o al menos, un parte del procedimiento. Si utiliza push( ) con un puntero a String en StringStack, entonces (según el significado de StringStack) también se esta pasando el puntero a StringStack. Si utiliza pop(), no solo consigue puntero, sino que a la vez el propietario. Cualquier puntero que se haya dejado en StringStack será borrado cuando el destructor sea invocado. Puesto que siempre son punteros a Strings y la declaración delete esta funcionando con punteros a String en vez de punteros a void, la destrucción se realiza de forma adecuada y todo funciona correctamente.

Esto es una desventaja: esta clase solo funciona con punteros de cadenas. Si se desea un Stack que funcione con cualquier variedad de objetos, se debe escribir una nueva versión de la clase que funcione con ese nuevo tipo de objeto. Esto puede convertirse rápidamente en una tarea tediosa, pero finalmente es resulta utilizando plantillas como se vera en el capítulo 16.

Existen consideraciones adicionales sobre este ejemplo: el cambio de la interfaz en Stack en el proceso de herencia. Si la interfaz es diferente, entonces StringStack no es realmente un Stack, y nunca será posible usar de forma correcta un StringStack como Stack. Esto hace que el uso de la herencia sea cuestionable en este punto; si no se esta creando un StringStack del tipo Stack, entonces, porque hereda de él. Más adelante, sen este mismo capítulo se mostrará una versión más adecuada.

14.5. Funciones que no heredan automáticamente

No todas las funciones son heredadas automáticamente desde la clase base a la clase derivada. Los constructores y destructores manejan la creación y la destrucción de un objeto y sólo ellos saben que hacer con los aspectos de un objeto en sus clases particulares y por ello los constructores y destructores inferiores de la jerarquía deben llamarlos. Así, los constructores y destructores no se heredan y deben ser creados específicamente en cada clase derivada.

Además, operator= tampoco se hereda porque realiza una acción parecida al constructor. Esto es, sólo porque conoce como asignar todos los miembros de un objeto, la parte izquierda del = a la parte derecha del otro objeto, no significa que la asignación tendrá el mismo significado después de la herencia.

En la herencia, estas funciones son creadas por el compilador si no son creadas por usted. (Con constructores, no se pueden crear constructores para que el compilador cree el constructor por defecto y el constructor copia.) Esto fue brevemente descrito en el capítulo 6. Los constructores creados se usan en inicialización de sus miembros y la creación del operator= usa la asignación de los miembros. A continuación, un ejemplo de las funciones que son creadas por el compilador.

```
//: C14:SynthesizedFunctions.cpp
// Functions that are synthesized by the compiler
#include <iostream>
using namespace std;

class GameBoard {
public:
  GameBoard() { cout << "GameBoard()\n"; }
  GameBoard(const GameBoard&) {
    cout << "GameBoard(const GameBoard&)\n";
  }
  GameBoard& operator=(const GameBoard&) {
    cout << "GameBoard::operator=()\n";
    return *this;
  }
  ~GameBoard() { cout << "~GameBoard()\n"; }
};

class Game {
  GameBoard gb; // Composition
public:
  // Default GameBoard constructor called:
  Game() { cout << "Game()\n"; }
  // You must explicitly call the GameBoard
  // copy-constructor or the default constructor
  // is automatically called instead:
  Game(const Game& g) : gb(g.gb) {
    cout << "Game(const Game&)\n";
  }
  Game(int) { cout << "Game(int)\n"; }
  Game& operator=(const Game& g) {
    // You must explicitly call the GameBoard
    // assignment operator or no assignment at
    // all happens for gb!
    gb = g.gb;
    cout << "Game::operator=()\n";
    return *this;
  }
  class Other {}; // Nested class
  // Automatic type conversion:
  operator Other() const {
    cout << "Game::operator Other()\n";
    return Other();
  }
  ~Game() { cout << "~Game()\n"; }
};

class Chess : public Game {};

void f(Game::Other) {}

class Checkers : public Game {
public:
  // Default base-class constructor called:
  Checkers() { cout << "Checkers()\n"; }
  // You must explicitly call the base-class
  // copy constructor or the default constructor
```

```
  // will be automatically called instead:
  Checkers(const Checkers& c) : Game(c) {
    cout << "Checkers(const Checkers& c)\n";
  }
  Checkers& operator=(const Checkers& c) {
    // You must explicitly call the base-class
    // version of operator=() or no base-class
    // assignment will happen:
    Game::operator=(c);
    cout << "Checkers::operator=()\n";
    return *this;
  }
};

int main() {
  Chess d1;  // Default constructor
  Chess d2(d1); // Copy-constructor
//! Chess d3(1); // Error: no int constructor
  d1 = d2; // Operator= synthesized
  f(d1); // Type-conversion IS inherited
  Game::Other go;
//!  d1 = go; // Operator= not synthesized
           // for differing types
  Checkers c1, c2(c1);
  c1 = c2;
} ///:~
```

Los constructores y el operator= de GameBoard y Game se describen por si solos y por ello distinguirá cuando son utilizados por el compilador. Además, el operador Other() ejecuta una conversión automática de tipo desde un objeto Game a un objeto de la clase anidada Other. La clase Chess simplemente hereda de Game y no crea ninguna función (sólo para ver como responde el compilador) La función f() coge un objeto Other para comprobar la conversión automática del tipo.

En main(), el constructor creado por defecto y el constructor copia de la clase derivada Chess son ejecutados. Las versiones de Game de estos constructores son llamados como parte de la jerarquía de llamadas a los constructores. Aun cuando esto es parecido a la herencia, los nuevos constructores son realmente creados por el compilador. Como es de esperar, ningún constructor con argumentos es ejecutado automáticamente porque es demasiado trabajo para el compilador y no es capaz de intuirlo.

El operator= es también es creado como una nueva función en Chess usando la asignación (así, la versión de la clase base es ejecutada) porque esta función no fue explícitamente escrita en la nueva clase. Y, por supuesto el destructor es creado automáticamente por el compilador.

El porqué de todas estas reglas acerca de la reescritura de funciones en relación a la creación de un objeto pueden parecer un poco extrañas en una primera impresión y como se heredan las conversiones automáticas de tipo. Pero todo esto tiene sentido - si existen suficiente piezas en Game para realizar un objeto Other, aquellas piezas están todavía en cualquier objeto derivado de Game y el tipo de conversión es válido (aun cuando puede, si lo desea, redefinirlo).

El operator= es creado automáticamente sólo para asignar objeto del mismo tipo. Si desea asignar otro tipo, deberá escribir el operator= usted mismo.

Si mira con detenimiento Game, observará que el constructor copia y la asignación tienen llamadas explicitas a constructor copia del objeto miembro y al operador de asignación. En la mayoría de ocasiones, deberá hacer esto porque sino, en vez del constructor copia, será llamado el constructor por defecto del objeto miembro, y en el caso del operador de asignación, ¡ninguna asignación se hará en los objetos miembros!

Por último, fíjese en Checkers, dónde explícitamente se escribe un constructor por defecto, constructor copia y los operadores de asignación. En el caso del constructor por defecto, el constructor por defecto de la clase base se llama automáticamente, y esto es lo normalmente que se desea hacer. Pero, aquí existe un punto importante, tan pronto que se decide escribir nuestro propio constructor copia y operador de asignación, el compilador asume que usted sabe lo que esta haciendo y no ejecutará automáticamente las versiones de la clase base así como las funciones creadas automáticamente. Si se quiere ejecutar las versiones de la clase base, debe llamarlas explícitamente. En el constructor copia de Checkers, esta llamada aparece en la lista de inicialización del constructor:

```
Checkers(const Checkers& c) : Game(c) {
```

En el operador de asignación de Checkers, la clase base se llama en la primera línea del cuerpo de la función:

```
Game::operator=(c);
```

Estas llamadas deben seguirse de forma canónica cuando usa cualquier clase derivada.

14.5.1. Herencia y métodos estáticos

Las funciones miembro estáticas funcionan igual que las funciones miembros no estáticas:

1. Son heredadas en la clase derivada.
2. Si redefine un miembro estático, el resto de funciones sobrecargadas en la clase base son ocultas.
3. Si cambia la signatura de una función en la clase base, todas las versiones con ese nombre de función en la clase base son ocultadas (esto es realmente una variación del punto anterior).

Sin embargo, las funciones miembro estáticas no pueden ser virtuales (este tema se cubrirá detenidamente en el capítulo 15).

14.5.2. Composición vs. herencia

La composición y la herencia colocan subobjetos dentro de la clase. Ambos usan la lista de inicialización del constructor para construir esos subobjetos. Pero se preguntará cuál es la diferencia entre los dos, y cuando escoger una y no la otra.

La composición generalmente se usa cuando se quieren las características de una clase existente dentro se su clase, pero no en su interfaz. Esto es, aloja un objeto para

implementar características en su clase, pero el usuario de su clase ve el interfaz que se ha definido, en vez del interfaz de la clase original. Para hacer esto, se sigue el típico patrón de alojar objetos privados de clases existentes en su nueva clase.

En ocasiones, sin embargo, tiene sentido permitir que el usuario de la clase acceda a la composición de su clase, esto es, hacer públicos los miembros objeto. Los miembros objeto usan su control de accesos, entonces es seguro y cuando el usuario conoce que esta formando un conjunto de piezas, hace que la interfaz sea más fácil de entender. Un buen ejemplo es la clase Car:

```
//: C14:Car.cpp
// Public composition

class Engine {
public:
  void start() const {}
  void rev() const {}
  void stop() const {}
};

class Wheel {
public:
  void inflate(int psi) const {}
};

class Window {
public:
  void rollup() const {}
  void rolldown() const {}
};

class Door {
public:
  Window window;
  void open() const {}
  void close() const {}
};

class Car {
public:
  Engine engine;
  Wheel wheel[4];
  Door left, right; // 2-door
};

int main() {
  Car car;
  car.left.window.rollup();
  car.wheel[0].inflate(72);
} ///:~
```

Como la composición de Car es parte del análisis del problema (y no una simple capa del diseño), hace públicos los miembros y ayudan al programador a entender como se utiliza la clase y requiere menos complejidad de código para el creador de la clase.

Si piensa un poco, observará que no tiene sentido componer un Car usando un

objeto "vehículo" - un coche no contiene un vehículo, es un vehículo. La relación "es-un" es expresado con la herencia y la relación "tiene un" es expresado con la composición.

Subtipado

Ahora suponga que desea crear un objeto del tipo ifstream que no solo abre un fichero sino que también guarde el nombre del fichero. Puede usar la composición e alojar un objeto ifstream y un string en la nueva clase:

```
//: C14:FName1.cpp
// An fstream with a file name
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class FName1 {
  ifstream file;
  string fileName;
  bool named;
public:
  FName1() : named(false) {}
  FName1(const string& fname)
    : fileName(fname), file(fname.c_str()) {
    assure(file, fileName);
    named = true;
  }
  string name() const { return fileName; }
  void name(const string& newName) {
    if(named) return; // Don't overwrite
    fileName = newName;
    named = true;
  }
  operator ifstream&() { return file; }
};

int main() {
  FName1 file("FName1.cpp");
  cout << file.name() << endl;
  // Error: close() not a member:
//!  file.close();
} ///:~
```

Sin embargo, existe un problema. Se intenta permitir el uso de un objeto FName1 en cualquier lugar dónde se utilice un objeto ifstream, incluyendo una conversión automática del tipo desde FName1 a ifstream&. Pero en main, la línea

```
file.close();
```

no compilará porque la conversión automática de tipo sólo ocurre cuando se llama a la función, no durante la selección del miembro. Por ello, esta manera no funcionará.

Una segunda manera es añadir la definición Close() a FName1:

```
void close() { file.close(); }
```

Esto funcionará si sólo existen unas cuantas funciones a las que se desea hacer funcionar como una clase ifstream. En este caso, solo una parte de la clase y la composición apropiada.

Pero ¿qué ocurre si se quiere que todo funcione cómo la clase deseada? A eso se le llama subtipos porque esta creando un nuevo tipo desde uno ya existente y lo que se quiere es que el nuevo tipo tenga la misma interfaz que el tipo existente (además de otras funciones que se deseen añadir) para que se pueda utilizar en cualquier lugar donde se utilizaba el tipo existente. Aquí es dónde la herencia es esencial. Puede ver que el subtipo resuelve perfectamente el problema anterior:

```
//: C14:FName2.cpp
// Subtyping solves the problem
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class FName2 : public ifstream {
  string fileName;
  bool named;
public:
  FName2() : named(false) {}
  FName2(const string& fname)
    : ifstream(fname.c_str()), fileName(fname) {
    assure(*this, fileName);
    named = true;
  }
  string name() const { return fileName; }
  void name(const string& newName) {
    if(named) return; // Don't overwrite
    fileName = newName;
    named = true;
  }
};

int main() {
  FName2 file("FName2.cpp");
  assure(file, "FName2.cpp");
  cout << "name: " << file.name() << endl;
  string s;
  getline(file, s); // These work too!
  file.seekg(-200, ios::end);
  file.close();
} ///:~
```

Ahora cualquier función que este disponible para el objeto sfstream también esta disponible para el objeto FName2. Asimismo se observan funciones no miembro como getline() que esperan un objeto ifstream y que pueden funcionar con un objeto FName2. Esto es porque FName2 es un tipo de ifstream; esto no significa simple-

mente que lo contiene. Esto es un tema muy importante que será explorado al final de este capitulo y el siguiente.

Herencia privada

Puede heredar utilizando una clase base de forma privada borrando public en la lista de la clase base o explícitamente utilizando private (definitivamente la mejor política a tomar pues indica al usuario lo que desea hacer). Cuando se hereda de forma privada, esta "implementado en términos de", esto es, se esta creando una nueva clase que tiene todos los datos y funcionalidad de la clase base, pero la funcionalidad esta oculta, solo una parte de capa de implementación. La clase derivada no tiene acceso a la capa de funcionalidad y un objeto no puede ser creado como instancia de la clase base (como ocurrió en FName2.cpp).

Se sorprenderá del propósito de la herencia privada, porque la alternativa, usar la composición para crear un objeto privado en la nueva clase parece más apropiada. La herencia privada esta incluida para completar el lenguaje pero para reducir confusión, normalmente se usará la composición en vez de la herencia privada. Sin embargo, existen ocasiones donde se desea el mismo interfaz como la clase base y anular tratamiento del objeto. La herencia privada proporciona esta habilidad.

14.5.2.2.1. Publicar los miembros heredados de forma privada Cuando se hereda de forma privada, todos los miembros públicos de la clase base llegan como privados. Si desea que cualquiera de ellos sea visible, solo use sus nombres (sin argumentos o valores de retorno) junto con la palabra clave using en una sección pública de la clase derivada:

```
//: C14:PrivateInheritance.cpp
class Pet {
public:
  char eat() const { return 'a'; }
  int speak() const { return 2; }
  float sleep() const { return 3.0; }
  float sleep(int) const { return 4.0; }
};

class Goldfish : Pet { // Private inheritance
public:
  using Pet::eat; // Name publicizes member
  using Pet::sleep; // Both members exposed
};

int main() {
  Goldfish bob;
  bob.eat();
  bob.sleep();
  bob.sleep(1);
//! bob.speak();// Error: private member function
} ///:~
```

Así, la herencia privada es útil si desea esconder parte de la funcionalidad de la clase base.

Fíjese que si expone el nombre de una función sobrecargada, expone todas las versiones sobrecargadas de la función en la clase base.

Debe pensar detenidamente antes de utilizar la herencia privada en vez de la composición; la herencia privada tiene complicaciones particulares cuando son combinadas con la identificación de tipos en tiempo de ejecución (es un tema de un capítulo en el volumen 2, disponible en www.BruceEckel.com)

14.6. Protected

Ahora que ya sabe que es la herencia, la palabra reservada protected ya tiene significado. En un mundo ideal, los miembros privados siempre serian fijos-y-rápidos, pero en los proyectos reales hay ocasiones cuando se desea ocultar algo a todo el mundo y todavía permitir accesos a los miembros de la clase derivada. La palabra clave protected es un movimiento al pragmatismo: este dice "Esto es privado como la clase usuario en cuestión, pero disponible para cualquiera que hereda de esta clase.

La mejor manera es dejar los miembros de datos privados - siempre debe preservar su derecho de cambiar la capa de implementación. Entonces puede permitir acceso controlado a los herederos de su clase a través de funciones miembro protegidas:

```
//: C14:Protected.cpp
// The protected keyword
#include <fstream>
using namespace std;

class Base {
  int i;
protected:
  int read() const { return i; }
  void set(int ii) { i = ii; }
public:
  Base(int ii = 0) : i(ii) {}
  int value(int m) const { return m*i; }
};

class Derived : public Base {
  int j;
public:
  Derived(int jj = 0) : j(jj) {}
  void change(int x) { set(x); }
};

int main() {
  Derived d;
  d.change(10);
} ///:~
```

Encontrará ejemplos de la necesidad de uso de protected más adelante y en el volumen 2.

14.6.1. Herencia protegida

Cuando se hereda, por defecto la clase base es privada, que significa que todos las funciones miembro publicas son privadas para el usuario en la nueva clase. Nor-

malmente, heredará públicamente para que el interfaz de la clase base sea también el interfaz de la clase derivada. Sin embargo, puede usar la palabra clave protected durante la herencia.

Derivar de forma protegida significa "implementado en términos de" para otras clases pero "es-una" para las clases derivadas y amigas. Es algo que no utilizará muy a menudo, pero esta en el lenguaje para completarlo.

14.7. Herencia y sobrecarga de operadores

Excepto el operador de asignación, el resto de operadores son heredados automáticamente en la clase derivada. Esto se puede demostrar heredando de C12:Byte.h:

```
//: C14:OperatorInheritance.cpp
// Inheriting overloaded operators
#include "../C12/Byte.h"
#include <fstream>
using namespace std;
ofstream out("ByteTest.out");

class Byte2 : public Byte {
public:
  // Constructors don't inherit:
  Byte2(unsigned char bb = 0) : Byte(bb) {}
  // operator= does not inherit, but
  // is synthesized for memberwise assignment.
  // However, only the SameType = SameType
  // operator= is synthesized, so you have to
  // make the others explicitly:
  Byte2& operator=(const Byte& right) {
    Byte::operator=(right);
    return *this;
  }
  Byte2& operator=(int i) {
    Byte::operator=(i);
    return *this;
  }
};

// Similar test function as in C12:ByteTest.cpp:
void k(Byte2& b1, Byte2& b2) {
  b1 = b1 * b2 + b2 % b1;

  #define TRY2(OP) \
    out << "b1 = "; b1.print(out); \
    out << ", b2 = "; b2.print(out); \
    out << ";  b1 " #OP " b2 produces "; \
    (b1 OP b2).print(out); \
    out << endl;

  b1 = 9; b2 = 47;
  TRY2(+) TRY2(-) TRY2(*) TRY2(/)
  TRY2(%) TRY2(^) TRY2(&) TRY2(|)
  TRY2(<<) TRY2(>>) TRY2(+=) TRY2(-=)
  TRY2(*=) TRY2(/=) TRY2(%=) TRY2(^=)
  TRY2(&=) TRY2(|=) TRY2(>>=) TRY2(<<=)
```

```
  TRY2(=) // Assignment operator

  // Conditionals:
  #define TRYC2(OP) \
    out << "b1 = "; b1.print(out); \
    out << ", b2 = "; b2.print(out); \
    out << ";  b1 " #OP " b2 produces "; \
    out << (b1 OP b2); \
    out << endl;

  b1 = 9; b2 = 47;
  TRYC2(<) TRYC2(>) TRYC2(==) TRYC2(!=) TRYC2(<=)
  TRYC2(>=) TRYC2(&&) TRYC2(||)

  // Chained assignment:
  Byte2 b3 = 92;
  b1 = b2 = b3;
}

int main() {
  out << "member functions:" << endl;
  Byte2 b1(47), b2(9);
  k(b1, b2);
} ///:~
```

El código de prueba anterior es idéntico a C12:ByteTest.cpp excepto que Byte2 se usa en vez de Byte. De esta forma todos los operadores son verificados para trabajar con Byte2 a través de la herencia.

Cuando examina la clase Byte2, verá que se ha definido explícitamente el constructor y que solo se ha credo el operator= que asigna un Byte2 a Byte2; cualquier otro operador de asignación tendrá que se realizado por usted.

14.8. Herencia múltiple

Si puede heredar de una clase, tendría sentido heredar de más de una clase a la vez. De hecho, puede hacerlo, pero si tiene sentido como parte del diseño es un tema que todavía se esta debatiendo. Una cosa en que generalmente se esta de acuerdo: debe evitar intentarlo hasta que haya programado bastante y comprenda el lenguaje en profundidad. Por ahora, probablemente no le importa cuando debe absolutamente utilizar la herencia múltiple y siempre puede utilizar la herencia simple

Inicialmente, la herencia múltiple parece bastante simple: se añade las clases en la lista de clases base durante la herencia separadas por comas. Sin embargo, la herencia múltiple introduce un número mayor de ambigüedades, y por eso, un capítulo del Volumen 2 hablará sobre el tema.

14.9. Desarrollo incremental

Una de las ventajas de la herencia y la composición es el soporte al desarrollo incremental permitiendo introducir nuevo código sin causar fallos en el ya existente. Si aparecen fallos, éstos son rectificados con nuevo código. Heredando de (o componiendo con) clases y funciones existentes y añadiendo miembros de datos y fun-

ciones miembros (y redefiniendo las funciones existentes durante la herencia) puede dejar el código existente - por otro que todavía se esta usando - que alguien todavía lo este utilizando. Si ocurre algún error, ahora sabe donde esta el nuevo código, y entonces podrá leerlo mas rápido y fácilmente que si lo hubiera modificado en el cuerpo del código existente.

Es sorprendente como las clases son limpiamente separadas. Incluso no es necesario añadir el código fuente con funciones miembros para reutilizar el código, solamente el fichero de cabecera describiendo la clase y el fichero objeto o el fichero de librería con las funciones miembros compiladas. (Esto es válido tanto para la herencia como para la composición.)

Esto es importante para hacer que el desarrollo sea un proceso incremental, como el aprendizaje de una persona. Puede hacer tantos análisis como desee pero todavía no sabrá todas las respuestas cuando configure un proyecto. Tendrá más éxito y un progresión inmediata - si su proyecto empieza a crecer como una criatura orgánica, evolutiva, parecerá mas bien que esa construyendo algo como un rascacielos de cristal [52]

Sin embargo la herencia es una técnica útil para la experimentación, en algún punto donde las cosas están estabilizadas, necesita echar un nuevo vistazo a la jerarquía de clases para colapsarla dentro de una estructura sensible [53]. Recuerde que, por encima de todo, la herencia significa expresar una relación que dice "Esta nueva clase es un tipo de esta vieja". Su programa no debe preocuparse de cómo mueve pedazos de bits por alrededor, en vez debe crear y manipular objetos de varios tipos para expresar un modelo en los términos dados para su problema.

14.10. Upcasting

Anteriormente en este capitulo, observo como un objeto de una clase que derivaba de ifstream tenia todas las características y conductas de un objeto ifstream. En FName2.cpp, cualquier función miembro de ifstream podría ser llamada por cualquier objeto FName2.

El aspecto más importante de la herencia no es proporcionar nuevas funciones miembro a la nueva clase. Es la relación expresada entre la nueva clase y la clase base. Esta relación puede ser resumida diciendo "La nueva clase es de un tipo de una clase existente".

Esta no es una descripción fantasiosa de explicar la herencia - esta directamente soportada por el compilador. Un ejemplo, considere una clase base llamada Instrument que representa instrumentos musicales y una clase derivada llamada Wind. Dado que la herencia significa que todas las funciones en la clase base están también disponibles en la clase derivada, cualquier mensaje que envié a la clase base puede ser también enviado desde la derivada. Entonces si la clase Instrument tiene una función miembro play(), también existirá en los instrumentos de Wind. Esto significa precisamente que un objeto Wind es un tipo de Instrument. El siguiente ejemplo muestra como el compilador soporta esta idea:

```
//: C14:Instrument.cpp
// Inheritance & upcasting
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  void play(note) const {}
```

```
};

// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {};

void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Upcasting
} ///:~
```

Lo interesante en este ejemplo es la función tune(), que acepta una referencia Instrument. Sin embargo, en main() la función tune() se llama utilizando una referencia a un objeto Wind. Dado que C++ es un muy peculiar sobre la comprobación de tipos, parece extraño que una función que acepta solo un tipo pueda aceptar otro tipo, al menos que sepa que un objeto Instrument es tambien un objeto Instrument.

14.10.1. ¿Por qué «upcasting»?

La razón de este término es histórica y esta basada en la manera en que se dibuja la herencia: con la raíz en la parte superior de la pagina y hacia abajo (por supuesto que puede pintar su diagrama de cualquier modo que le sea útil). El diagrama para `Instrument.cpp` es:

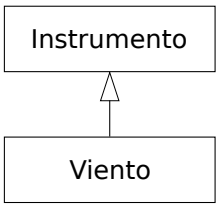

Figura 14.1: Upcasting

El hecho de pasar de la clase derivada a la clase base, esto es, desplazarse hacia arriba en el diagrama de la herencia, es normalmente conocido como upcasting. Upcasting es siempre seguro porque se esta desplazando de un tipo desde un tipo mas especifico a otro tipo mas general. - únicamente puede ocurrir es que la interfaz de la clase pierda algunas funciones miembro, pero no ganarlas. Esto es porque el compilador permite el upcasting sin ninguna conversión explicita o notación especial.

14.10.2. FIXME Upcasting y el constructor de copia

Si permite que el compilador cree un constructor copia de una clase derivada, éste llamara automáticamente al constructor copia de la clase base, y entones ,a los

constructores copia para todos los miembros objeto (o realizará una copia de bits en los tipos predefinidos) entonces conseguirá la conducta correcta:

```
//: C14:CopyConstructor.cpp
// Correctly creating the copy-constructor
#include <iostream>
using namespace std;

class Parent {
  int i;
public:
  Parent(int ii) : i(ii) {
    cout << "Parent(int ii)\n";
  }
  Parent(const Parent& b) : i(b.i) {
    cout << "Parent(const Parent&)\n";
  }
  Parent() : i(0) { cout << "Parent()\n"; }
  friend ostream&
    operator<<(ostream& os, const Parent& b) {
    return os << "Parent: " << b.i << endl;
  }
};

class Member {
  int i;
public:
  Member(int ii) : i(ii) {
    cout << "Member(int ii)\n";
  }
  Member(const Member& m) : i(m.i) {
    cout << "Member(const Member&)\n";
  }
  friend ostream&
    operator<<(ostream& os, const Member& m) {
    return os << "Member: " << m.i << endl;
  }
};

class Child : public Parent {
  int i;
  Member m;
public:
  Child(int ii) : Parent(ii), i(ii), m(ii) {
    cout << "Child(int ii)\n";
  }
  friend ostream&
    operator<<(ostream& os, const Child& c){
    return os << (Parent&)c << c.m
              << "Child: " << c.i << endl;
  }
};

int main() {
  Child c(2);
  cout << "calling copy-constructor: " << endl;
  Child c2 = c; // Calls copy-constructor
  cout << "values in c2:\n" << c2;
```

```
} ///:~
```

El operador<< de Child es interesante por la forma en que llama al operador<< del padre dentro de éste: convirtiendo el objeto Child a Parent& (si lo convierte a un objeto de la clase base en vez de una referencia, probablemente obtendrá resultados no deseados)

```
return os << (Parent&)c << c.m
```

Dado que el compilador lo ve como Parent, éste llama al operador<< Parent.

Puede observar que Child no tiene explícitamente definido un constructor copia. El compilador crea el constructor copia (es una de las cuatro funciones que sintetiza, junto con el constructor del defecto - si no creas a ninguna constructores - el operator= y el destructor) llamando el constructor copia de Parent y el constructor copia de Member. Esto muestra la siguiente salida

```
Parent(int ii)
Member(int ii)
Child(int ii)
calling copy-constructor:
Parent(const Parent&)
Member(const Member&)
values in c2:
Parent: 2
Member: 2
Child: 2
```

Sin embargo, si escribe su propio constructor copia para Child puede tener un error inocente y funcionar incorrectamente:

```
Child(const Child& c) : i(c.i), m(c.m) {}
```

entonces el constructor por defecto será llamado automáticamente por la clase base por parte de Child, aquí es dónde el compilador muestra un error cuando no tienen otra (recuerde que siempre algun constructor se ejecuta para cada objeto, sin importar si es un subobjeto de otra clase). La salida será entonces:

```
Parent(int ii)
Member(int ii)
Child(int ii)
calling copy-constructor:
Parent()
Member(const Member&)
values in c2:
Parent: 0
Member: 2
Child: 2
```

Esto probablemente no es lo que espera, generalmente deseará que la parte de la clase base sea copiada del objeto existente al nuevo objeto como parte del constructor copia.

Para arreglar el problema debe recordar como funciona la llamada al constructor copia de la clase base (como el compilador lo hace) para que escriba su propio constructor copia. Este puede parecer un poco extraño a primera vista pero es otro ejemplo de upcasting.

```
Child(const Child& c)
   : Parent(c), i(c.i), m(c.m) {
   cout << "Child(Child&)\n";
}
```

La parte extraña es cómo el constructor copia es ejecutado: Parent(c). ¿Qué significa pasar un objeto Child al constructor padre? Child hereda de Parent, entonces una referencia de Child es una referencia Parent. El constructor copia de la clase base convierte a una referencia de Child a una referencia de Parent y la utiliza en el construcción de la copia. Cuando escribe su propio constructor copia la mayoría de ocasiones deseará lo mismo.

14.10.3. Composición vs. herencia FIXME (revisited)

Una de las maneras más claras de determinar cuando debe utilizar la composición o la herencia es preguntando cuando será necesaria la conversión desde su nueva clase. Anteriormente, en esta capitulo, la clase Stack fue especializada utilizando la herencia. Sin embargo, los cambios en StringStack serán utilizados son como contenedores de string y nunca serán convertidos, pero ello, es mucho mas apropiado utilizas la alternativa de la composición.

```
//: C14:InheritStack2.cpp
// Composition vs. inheritance
#include "../C09/Stack4.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class StringStack {
  Stack stack; // Embed instead of inherit
public:
  void push(string* str) {
    stack.push(str);
  }
  string* peek() const {
    return (string*)stack.peek();
  }
  string* pop() {
    return (string*)stack.pop();
  }
};

int main() {
  ifstream in("InheritStack2.cpp");
  assure(in, "InheritStack2.cpp");
  string line;
  StringStack textlines;
  while(getline(in, line))
    textlines.push(new string(line));
  string* s;
  while((s = textlines.pop()) != 0) // No cast!
    cout << *s << endl;
} ///:~
```

```
```

El fichero es idéntico a InheritStack.cpp, excepto que un objeto Stack es alojado en StringStack y se utilizan las funciones miembros para llamarlo. No se consume tiempo o espacio porque el subobjeto tiene el mismo tamaño y todas las comprobaciones de tipos han ocurrido en tiempo de compilación.

Sin embargo, esto tiende a confusión, podría también utilizar la herencia privada para expresar "implementado en términos de". Esto también resolvería el problema de forma adecuada. Un punto importante es cuando la herencia múltiple puede ser garantizada. En este caso, si observa un diseño en que la composición pueda utilizarse en vez de la herencia, debe eliminar la necesidad de utilizar herencia múltiple.

14.10.4. FIXME Upcasting de punteros y referencias

En Instrument.cpp, la conversión ocurre durante la llamada a la función - un objeto Wind fuera de la función se toma como referencia y se convierte en una referencia Instrument dentro de la función. La conversión puede también ocurrir durante una asignación a un puntero o una referencia.

```
Wind w;
Instrument* ip = &w; // Upcast
Instrument& ir = w; // Upcast
```

Como en la llamada a la función, ninguno de estos casos requiere una conversión explicita.

14.10.5. Una crisis

Por supuesto, cualquier conversión pierde la información del tipo sobre el objeto. Si dice

```
Wind w;
Instrument* ip = &w;
```

el compilador puede utilizar ip solo como un puntero a Instrumento y nada mas. Esto es, éste no puede conocer que ip realmente esta apuntando a un objeto Wind. Entonces cuando llame a la función miembro play() diciendo

```
ip->play(middleC);
```

el compilador solo puede conocer que la llamada a play() es de un puntero a Instrument y llamara a la versión de la clase base Instrument::play() en vez de lo que debería hacer, que es llamar a Wind::play(). Así, no conseguirá una conducta adecuada.

esto es un problema importante; es resulto en el Capitulo 15, introducción al tercer pilar de la programación orientada a objetos: poliformismo (implementado en C++ con funciones virtuales).

14.11. Resumen

Herencia y composición le permiten crear nuevos tipos desde tipos existentes y ambos incluyen subobjetos de tipos existentes dentro del nuevo tipo. Sin embargo, normalmente, utilizara la composición para reutilizar tipos existentes como parte de la capa de implementación de un nuevo tipo y la herencia cuando desea forzar al nuevo tipo a ser del mismo tipo que la clase base (la equivalencia de tipos garantiza la equivalencia de la interfaz). Como las clases derivadas tienen el interfaz de la clase base, esta puede ser convertidas a la base, lo cual es crítico para el poliformismo como verá el Capítulo 15.

Aunque la reutilización de código a través de la composición y la herencia es muy útil para el desarrollo rápido de proyectos, generalmente deseara rediseñar su jerarquía de clases antes de permitir a otros programadores dependan de ella. Su objetivo es crear una jerarquía en que cada clase tenga un uso especifico y sin ser demasiado grande (esforzándose más en la funcionalidad que en la dificultad de la reutilización...), ni pequeña, (no se podrá usar por si mismo o sin añadir funcionalidad).

14.12. Ejercicios

Las soluciones a los ejercicios se pueden encontrar en el documento electrónico titulado «The Thinking in C++ Annotated Solution Guide», disponible por poco dinero en www.BruceEckel.com.

1. Modificar Car.cpp para que herede desde una clase llamada Vehicle, colocando correctamente las funciones miembro en Vehicle (esto es, añadir algunas funciones miembro). Añada un constructor (no el de por defecto) a Vehicle, que debe ser llamado desde dentro del constructor Car

2. Crear dos clases, A y B, con constructor por defectos notificándose ellos mismos. Una nueva clase llamada C que hereda de A, y cree un objeto miembro B dentro de C, pero no cree un constructor para C. Cree un objeto de la clase C y observe los resultados.

3. Crear una jerarquía de clases de tres niveles con constructores por defecto y con destructores, ambos notificándose utilizando cout. Verificar que el objeto más alto de la jerarquía, los tres constructores y destructores son ejecutados automáticamente. Explicar el orden en que han sido realizados.

4. Modificar Combined.cpp para añadir otro nivel de herencia y un nuevo objeto miembro. Añadir el código para mostrar cuando los constructores y destructores son ejecutados.

5. En Combined.cpp, crear una clase D que herede de B y que tenga un objeto miembro de la clase C. Añadir el código para mostrar cuando los constructores y los destructores son ejecutados.

6. Modificar Order.cpp para añadir otro nivel de herencia Derived3 con objetos miembro de la clase Member4 y Member5. Compruebe la salida del programa.

7. En NameHidding.cpp, verificar que Derived2, Derived3 y Derived4, ninguna versión de la clase base de f() esta disponible.

8. Modificar NameHiding.cpp añadiendo tres funciones sobrecargadas llamadas h() en Base y mostrar como redefiniendo una de ellas en una clase derivada oculta las otras.

9. Crear una clase StringVector que herede de vector<void*> y redefinir push_back y el operador [] para aceptar y producir string*. ¿Qué ocurre si intenta utilizar push_back() un void*?

10. Escribir una clase que contenga muchos tipos y utilice una llamada a una función pseudo-constructor que utiliza la misma sintaxis de un constructor.Utilizarla en el constructor para inicializar los tipos.

11. Crear una clase llamada Asteroid. Utilizar la herencia para especializar la clase PStash del capítulo 13 (PStash.h y PStash.cpp) para que la acepte y retorne punteros a Asteroid. También modifique PStashTest.cpp para comprobar sus clases. Cambiar la clase para que PStash sea un objeto miembro.

12. Repita el ejercicio 11 con un vector en vez de la clase PStash.

13. En SynthesizedFunctions.cpp, modifique Chess para proporcionarle un constructor por defecto, un constructor copia y un operador de asignación. Demostrar que han sido escritos correctamente.

14. Crear dos clases llamadas Traveler y Pager sin constructores por defecto, pero con constructores que toman un argumento del tipo string, el cual simplemente lo copia a una variable interna del tipo string. Para cada clase, escribir correctamente un constructor copia y el operador de asignación. Entonces cree la clase BusinessTraveler que hereda de Traveler y crear ub objeto miembro Pager dentro ella. Escribir correctamente el constructor por defecto, un constructor que tome una cadena como argumento, un constructor copia y un operador de asignación.

15. Crear una clase con dos funciones miembro estáticas. Herede de estas clases y redefina una de las funciones miembro. Mostrar que la otra función se oculta en la clase derivada.

16. Mejorar las funciones miembro de ifstream. En FName2.cpp, intentar suprimirlas del objeto file.

17. Utilice la herencia privada y protegida para crear dos nuevas clases desde la clase base. Intente convertir los objetos de las clases derivadas en la clase base. Explicar lo que ocurre.

18. En Protected.cpp, añadir una función miembro en Derived que llame al miembro protegido de Base read().

19. Cambiar Protected.cpp para que Derived utilice herencia protegida. Compruebe si puede llamar a value() desde un objeto Derived.

20. Crear una clase llamada SpaceShip con un metodo fly(). Crear Shuttle que hereda de SpaceShip y añadir el metodo land(). Creat un nuevo Shuttle, convertirlo por puntero o referenciaa SpaceShip e intente llamar al metodo land(). Explicar los resultados.

21. Modificar Instrument.cpp para añadir un método prepare() a Instrument. Llamar a prepare () dentro de tune().

22. Modificar Instrument.cpp para que play() muestre un mensaje con cout y que Wind redefina play() para que muestra un mensaje diferente con cout. Ejecute el programa y explique porque probamenteble no deseara esta conducta. Ahora ponga la palabra reservada virtual (la cual aprenderá en el capitulo 15) delante de de la declaración de play en Instrument y observe el cambio en el comportamiento.

23. En CopyConstructor.cpp, herede una nueva clase de Child y proporcionarle un miembro m. Escribir un constructor correcto, un constructor copia, operator= y operator<< de ostreams y comprobar la clase en main().

24. Tomar como ejemplo CopyConstructor.cpp y modifíquelo añadiendo su propio constructor copia a Child sin llamar el constructor copia de clase base y comprobar que ocurre. Arregle el problema añadiendo una llamada explicita al constructor copia de la clase base en la lista de inicialización del constructor del constructor copia de Child.

25. Modificar InheritStack2.cpp para utilizar un vector<string> en vez de Stack.

26. Crear una clase Rock con un constructor por defecto, un constructor copia y un operador de asignación y un destructor, todos ellos mostrándose para saber que han sido ejecutados. En main(), crear un vector<Rock> (esto es, tener objetos Rock por valor) y añadir varios Rocks. Ejecutar el programa y explicar los resultados obtenidos. Fijarse cuando los destructores son llamados desde los objetos Rock en el vector. Ahora repita el ejercicio con un vector<Rock*>. ¿Es posible crear un vector<Rock&>?

27. En este ejercicio cree un patrón de diseño llamado proxy. Comience con la clase base Subject y proporciónele tres funciones: f(), g() y h(). Ahora herede una clase Proxy y dos clases Implementation1 e Implementacion2 de Subject. Proxy tendría que contener un puntero a un Suboject y todos los miembros de Proxy (usualmente el constructor). En main(), crear dos objetos Proxy diferentes que usen las dos implementaciones diferentes. Modificar Proxy para que dinámicamente cambie las implementaciones.

28. Modificar ArrayOperatorNew del Capítulo 13 para mostrar que si deriva de Widget, la reserva de memoria todavía funciona correctamente. Explicar porque la herencia en Framis.cpp no funcionaria correctamente.

29. Modificar Framis.cpp del Capítulo 13 para que herede de Framis y crear nuevas versiones de new y delete para su clase derivada. Demostrar como todo ello funciona correctamente.

15: Polimorfismo y Funciones virtuales

El Polimorfismo (implementado en C++ con funciones `virtuales`) es la tercera característica esencial de un lenguaje orientado a objetos, después de la abstracción de datos y la herencia.

De hecho, nos provee de otra dimensión para la separación entre interfaz y la implementación, desacoplando el *qué* del *cómo*. El Polimorfismo permite mejorar la organización del código y su legibilidad así como la creación de programas *extensibles* que pueden "crecer" no sólo durante el desarrollo del proyecto, si no también cuando se deseen nuevas características.

La encapsulación crea nuevos tipos de datos combinando características y comportamientos. El control de acceso separa la interfaz de la implementación haciendo privados (`private`) los detalles. Estos tipos de organización son fácilmente entendibles por cualquiera que venga de la programación procedimental. Pero las funciones virtuales tratan de desunir en términos de *tipos*. En el Capítulo 14, usted vió como la herencia permitía tratar a un objeto como su propio tipo *o* como a su tipo base. Esta habilidad es básica debido a que permite a diferentes tipos (derivados del mismo tipo base) ser tratados como si fueran un único tipo, y un único trozo de código es capaz de trabajar indistintamente con todos. Las funciones virtuales permiten a un tipo expresar sus diferencias con respecto a otro similar si ambos han sido derivados del mismo tipo base. Esta distinción se consigue modificando las conductas de las funciones a las que se puede llamar a través de la clase base.

En este capítulo aprenderá sobre las funciones virtuales, empezando con ejemplos simples que le mostrará lo "desvirtual" del programa.

15.1. Evolución de los programadores de C++

Los programadores de C parecen conseguir pasarse a C++ en tres pasos. Al principio, como un "C mejorado", debido a que C++ le fuerza a declarar todas las funciones antes de usarlas y a que es mucho más sensible a la forma de usar las variables. A menudo se pueden encontrar errores en un programa C simplemente recompilándolo con un compilador de C++.

El segundo paso es la "programación basada en objetos", que significa que se pueden ver fácilmente los beneficios de la organización del código al agrupar estructuras de datos junto con las funciones que las manejan, la potencia de los constructores y los destructores, y quizás algo de herencia simple. La mayoría de los programadores que han trabajado durante un tiempo con C ven la utilidad de esto porque es lo que intentan hacer cuando crean una librería. Con C++ usted recibe la ayuda del

compilador.

Usted se puede encontrar atascado en el nivel de "programación basada en objetos" debido a que es de fácil acceso y no requiere mucho esfuerzo mental. Es también sencillo sentir cómo está creando tipos de datos - usted hace clases y objetos, envía mensajes a esos objetos, y todo es bonito y pulcro.

Pero no sea tonto. Si se para aquí, se está perdiendo una de las más importantes partes del lenguaje, que significa el salto a la verdadera programación orientada a objetos. Y esto se consigue únicamente con las funciones virtuales.

Las funciones virtuales realzan el concepto de tipo en lugar de simplemente encapsular código dentro de estructuras y dejarlo detrás de un muro, por lo que son, sin lugar a dudas, el concepto más difícil a desentrañar por los nuevos programadores en C++. Sin embargo, son también el punto decisivo para comprender la programación orientada a objetos. Si no usa funciones virtuales, todavía no entiende la POO.

Debido a que las funciones virtuales están intimamente unidas al concepto de tipo, y los tipos son el núcleo de la programación orientada a objetos, no existe analogía a las funciones virtuales dentro de los lenguajes procedurales. Como programador procedural, usted no tiene referente con el que comparar las funciones virtuales, al contrario de las otras características del lenguaje. Las características de un lenguaje procedural pueden ser entendidas en un nivel algorítmico, pero las funciones virtuales deben ser entendidas desde el punto de vista del diseño.

15.2. Upcasting

En el Capítulo 14 se vió como un objeto puede ser usado como un objeto de su propio tipo o como un objeto de su tipo base. Además el objeto puede ser manejado a través de su tipo base. Tomar la dirección de un objeto (o un puntero o una referencia) y tratarlo como la dirección de su tipo base se conoce como *upcasting* [1] debido al camino que se genera en los árboles de herencia que se suelen pintar con la clase base en la cima.

También se vió surgir un problema el cuál está encarnado en el siguiente código:

```
//: C15:Instrument2.cpp
// Inheritance & upcasting
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat }; // Etc.

class Instrument {
public:
  void play(note) const {
    cout << "Instrument::play" << endl;
  }
};

// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {
```

[1] N del T: Por desgracia `upcasting` es otro de los términos a los que no he encontrado una traducción convincente (¿¿amoldar hacia arriba??) y tiene el agravante que deriva de una expresión ampliamente usada por los programadores de C (¿Quién no ha hecho nunca un `cast a void*` ;-) ?. Se aceptan sugerencias.

```
public:
  // Redefine interface function:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
};

void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Upcasting
} ///:~
```

La función `tune()` acepta (por referencia) un `Instrument`, pero también acepta cualquier cosa que derive de `Instrument`. En el `main()`, se puede ver este comportamiento cuando se pasa un objeto `Wind` a `afinar()` sin que se necesite ningún molde. La interfaz de `Instrument` tiene que existir en `Wind`, porque `Wind` hereda sus propiedades de `Instrument`. Moldear en sentido ascendente (*Upcasting*) de `Wind` a `Instrument` puede "reducir" la interfaz, pero nunca puede ser menor que la interfaz de `Instrument`.

Los mismos argumentos son ciertos cuando trabajamos con punteros; la única diferencia es que el usuario debe indicar la dirección de los objtos de forma explícita cuando se pasen a una función.

15.3. El problema

El problema con `Instrument2.cpp` puede verse al ejecutar el programa. La salida es `Instrument::play`. Claramente, esta no es la salida deseada, porque el objeto es actualmente un `Wind` y no solo un `Instrument`. La llamada debería producir un `Wind::play`. Por este motivo, cualquier objeto de una clase que derive de la clase `Instrument` debería usar su propia versión de `play()`, de acuerdo a la situación.

El comportamiento de `Instrument2.cpp` no es sorprendente dada la aproximación de C a las funciones. Para entender el resultado es necesario comprender el concepto de *binding* (ligadura).

15.3.1. Ligadura de las llamadas a funciones

Conectar una llamada a una función al cuerpo de la función se conoce como *binding* (vincular). Cuando la vinculación se realiza antes de ejecutar el programa (por el compilador y el linker), se la conoce como *early binding* (ligadura temprana). Puede no haber escuchado anteriormente este término debido a que no es posible con los lenguajes procedurales: los compiladores de C sólo admiten un tipo de vinculación que es la vinculación anticipada.

El problema en el programa anterior es causado por la vinculación anticipada porque el compilador no conoce la correcta función a la que debería llamar cuando sólo es una dirección de `Instrument`.

La solución se conoce como ligadura tardía (*late binding*), que significa que la ligadura se produce en tiempo de ejecución basándose en el tipo de objeto. También es conocida como *ligadura dinámica* o *ligadura en tiempo de ejecución*. Cuando un lenguaje implementa la ligadura dinámica debe existir algún tipo de mecanismo para determinar el tipo del objeto en tiempo de ejecución y llamar a la función miembro apropiada. En el caso de un lenguaje compilado, el compilador todavía no conoce el tipo actual del objeto, pero inserta código que lo averigua y llama al cuerpo correcto de la función. El mecanismo de la ligadura dinámica varía de un lenguaje a otro, pero se puede imaginar que algún tipo de información debe ser introducida en los objetos. Se verá como trabaja posteriormente.

15.4. Funciones virtuales

Para que la ligadura dinámica tenga efecto en una función particular, C++ necesita que se use la palabra reservada `virtual` cuando se declara la función en la clase base. La ligadura en tiempo de ejecución funciona unícamente con las funciones `virtual` es, y sólo cuando se está usando una dirección de la clase base donde exista la función `virtual`, aunque puede ser definida también en una clase base anterior.

Para crear una función miembro como `virtual`, simplemente hay que preceder a la declaración de la función con la palabra reservada `virtual`. Sólo la declaración necesita la palabra reservada `virtual`, y no la definición. Si una función es declarada como `virtual`, en la clase base, será entonces `virtual` en todas las clases derivadas. La redefinición de una función `virtual` en una clase derivada se conoce como *overriding*.

Hay que hacer notar que sólo es necesario declarar la función como `virtual` en la clase base. Todas las funciones de las clases derivadas que encajen con la declaración que esté en la clase base serán llamadas usando el mecanismo virtual. Se *puede* usar la palabra reservada `virtual` en las declaraciones de las clases derivadas (no hace ningún mal), pero es redundante y puede causar confusión.

Para conseguir el comportamiento deseado de `Instrument2.cpp`, simplemente hay que añadir la palabra reservada `virtual` en la clase base antes de `play()`.

```
//: C15:Instrument3.cpp
// Late binding with the virtual keyword
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  virtual void play(note) const {
    cout << "Instrument::play" << endl;
  }
};

// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {
public:
  // Override interface function:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
```

```
};

void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Upcasting
} ///:~
```

Este archivo es idéntico a `Instrument2.cpp` excepto por la adición de la palabra reservada `virtual` y, sin embargo, el comportamiento es significativamente diferente: Ahora la salida es `Wind::play`.

15.4.1. Extensibilidad

Con `play()` definido como `virtual` en la clase base, se pueden añadir tantos nuevos tipos como se quiera sin cambiar la función `play()`. En un programa orientado a objetos bien diseñado, la mayoría de las funciones seguirán el modelo de `play()` y se comunicarán únicamente a través de la interfaz de la clase base. Las funciones que usen la interfaz de la clase base no necesitarán ser cambiadas para soportar a las nuevas clases.

Aquí está el ejemplo de los instrumentos con más funciones virtuales y un mayor número de nuevas clases, las cuales trabajan de manera correcta con la antigua (sin modificaciones) función `play()`:

```
//: C15:Instrument4.cpp
// Extensibility in OOP
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  virtual void play(note) const {
    cout << "Instrument::play" << endl;
  }
  virtual char* what() const {
    return "Instrument";
  }
  // Assume this will modify the object:
  virtual void adjust(int) {}
};

class Wind : public Instrument {
public:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
  char* what() const { return "Wind"; }
  void adjust(int) {}
};
```

```
class Percussion : public Instrument {
public:
  void play(note) const {
    cout << "Percussion::play" << endl;
  }
  char* what() const { return "Percussion"; }
  void adjust(int) {}
};

class Stringed : public Instrument {
public:
  void play(note) const {
    cout << "Stringed::play" << endl;
  }
  char* what() const { return "Stringed"; }
  void adjust(int) {}
};

class Brass : public Wind {
public:
  void play(note) const {
    cout << "Brass::play" << endl;
  }
  char* what() const { return "Brass"; }
};

class Woodwind : public Wind {
public:
  void play(note) const {
    cout << "Woodwind::play" << endl;
  }
  char* what() const { return "Woodwind"; }
};

// Identical function from before:
void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

// New function:
void f(Instrument& i) { i.adjust(1); }

// Upcasting during array initialization:
Instrument* A[] = {
  new Wind,
  new Percussion,
  new Stringed,
  new Brass,
};

int main() {
  Wind flute;
  Percussion drum;
  Stringed violin;
  Brass flugelhorn;
  Woodwind recorder;
```

```
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  tune(recorder);
  f(flugelhorn);
} ///:~
```

Se puede ver que se ha añadido otro nivel de herencia debajo de `Wind`, pero el mecanismo `virtual` funciona correctamente sin importar cuantos niveles haya. La función `adjust()` *no* está redefinida (*override*) por `Brass` y `Woodwind`. Cuando esto ocurre, se usa la definición más "cercana" en la jerarquía de herencia - el compilador garantiza que exista *alguna* definición para una función virtual, por lo que nunca acabará en una llamada que no esté enlazada con el cuerpo de una función (lo cual sería desatroso).

El array `A[]` contiene punteros a la clase base `Instrument`, lo que implica que durante el proceso de inicialización del array habrá *upcasting*. Este array y la función `f()` serán usados en posteriores discusiones.

En la llamada a `tune()`, el *upcasting* se realiza en cada tipo de objeto, haciendo que se obtenga siempre el comportamiento deseado. Se puede describir como "enviar un mensaje a un objeto y dejar al objeto que se preocupe sobre qué hacer con él". La función `virtual` es la lente a usar cuando se está analizando un proyecto: ¿Dónde deben estar las clases base y cómo se desea *extender* el programa? Sin embargo, incluso si no se descubre la interfaz apropiada para la clase base y las funciones virtuales durante la creación del programa, a menudo se descubrirán más tarde, incluso mucho más tarde cuando se desee ampliar o se vaya a hacer funciones de mantenimiento en el programa. Esto no implica un error de análisis o de diseño; simplemente significa que no se conocía o no se podía conocer toda la información al principio. Debido a la fuerte modularización de C++, no es mucho problema que esto suceda porque los cambios que se hagan en una parte del sistema no tienden a propagarse a otras partes como sucede en C.

15.5. Cómo implementa C++ la ligadura dinámica

¿Cómo funciona la ligadura dinámica? Todo el trabajo se realiza detrás del telón gracias al compilador, que instala los mecanismos necesarios de la ligadura dinámica cuando se crean funciones virtuales. Debido a que los programadores se suelen beneficiar de la comprensión del mecanismo de las funciones virtuales en C++, esta sección mostrará la forma en que el compilador implementa este mecanismo.

La palabra reservada `virtual` le dice al compilador que no debe realizar ligadura estática. Al contrario, debe instalar automáticamente todos los mecanismos necesarios para realizar la ligadura dinámica. Esto significa que si se llama a `play()` para un objeto `Brass` a *través una dirección a la clase base* `Instrument`, se usará la función apropiada.

Para que funcione, el compilador típico [2] crea una única tabla (llamada VTABLE) por cada clase que contenga funciones `virtuales`. El compilador coloca las direcciones de las funciones virtuales de esa clase en concreto en la VTABLE. En cada

[2] Los compiladores pueden implementar el comportamiento virtual como quieran, pero el modo aquí descrito es una aproximación casi universal.

clase con funciones virtuales el compilador coloca de forma secreta un puntero llamado `vpointer` (de forma abreviada VPTR), que apunta a la VTABLE de ese objeto. Cuando se hace una llamada a una función virtual a través de un puntero a la clase base (es decir, cuando se hace una llamada usando el polimorfismo), el compilador silenciosamente añade código para buscar el VPTR y así conseguir la dirección de la función en la VTABLE, con lo que se llama a la función correcta y tiene lugar la ligadura dinámica.

Todo esto - establecer la VTABLE para cada clase, inicializar el VPTR, insertar código para la llamada a la función virtual - sucede automáticamente sin que haya que preocuparse por ello. Con las funciones virtuales, se llama a la función apropiada de un objeto, incluso aunque el compilador no sepa el tipo exacto del objeto.

15.5.1. Almacenando información de tipo

Se puede ver que no hay almacenada información de tipo de forma explícita en ninguna de las clases. Pero los ejemplos anteriores, y la simple lógica, dicen que debe existir algún tipo de información almacenada en los objetos; de otra forma el tipo no podría ser establecido en tiempo de ejecución. Es verdad, pero la información de tipo está oculta. Para verlo, aquí está un ejemplo que muestra el tamaño de las clases que usan funciones virtuales comparadas con aquellas que no las usan:

```
//: C15:Sizes.cpp
// Object sizes with/without virtual functions
#include <iostream>
using namespace std;

class NoVirtual {
  int a;
public:
  void x() const {}
  int i() const { return 1; }
};

class OneVirtual {
  int a;
public:
  virtual void x() const {}
  int i() const { return 1; }
};

class TwoVirtuals {
  int a;
public:
  virtual void x() const {}
  virtual int i() const { return 1; }
};

int main() {
  cout << "int: " << sizeof(int) << endl;
  cout << "NoVirtual: "
       << sizeof(NoVirtual) << endl;
  cout << "void* : " << sizeof(void*) << endl;
  cout << "OneVirtual: "
       << sizeof(OneVirtual) << endl;
  cout << "TwoVirtuals: "
```

```
        << sizeof(TwoVirtuals) << endl;
} ///:~
```

Sin funciones virtuales el tamaño del objeto es exactamente el que se espera: el tamaño de un único [3] int. Con una única función virtual en `OneVirtual`, el tamaño del objeto es el tamaño de `NoVirtual` más el tamaño de un puntero a void. Lo que implica que el compilador añade un único puntero (el VPTR) en la estructura si se tienen una *o más* funciones virtuales. No hay diferencia de tamaño entre `OneVirtual` y `TwoVirtuals`. Esto es porque el VPTR apunta a una tabla con direcciones de funciones. Se necesita sólo una tabla porque todas las direcciones de las funciones virtuales están contenidas en esta tabla.

Este ejemplo requiere como mínimo un miembro de datos. Si no hubiera miembros de datos, el compilador de C++ hubiera forzado a los objetos a ser de tamaño no nulo porque cada objeto debe tener direcciones distintas (¿se imagina cómo indexar un array de objetos de tamaño nulo?). Por esto se inserta en el objeto un miembro "falso" ya que de otra forma tendríá un tamaño nulo. Cuando se inserta la información de tipo gracias a la palabra reservada `virtual`, ésta ocupa el lugar del miembro "falso". Intente comentar el `int a` en todas las clases del ejemplo anterior para comprobarlo.

15.5.2. Pintar funciones virtuales

Para entender exactamente qué está pasando cuando se usan funciones virtuales, es útil ver la actividad que hay detrás del telón. Aquí se muestra el array de punteros `A[]` in `Instrument4.cpp`:

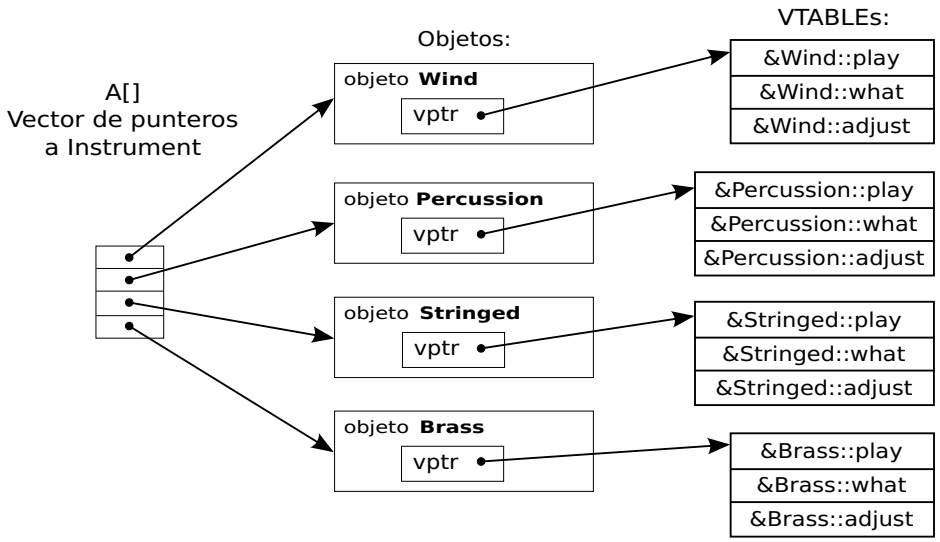

Figura 15.1: Funciones virtuales

El array de punteros a `Instrument`s no tiene información específica de tipo; cada uno de ellos apunta a un objeto de tipo `Instrument`. `Wind`, `Percussion`, `S-`

[3] Algunos compiladores pueden aumentar el tamaño pero sería raro.

`tringed`, y `Brass` encajan en esta categoría porque derivan de `Instrument` (esto hace que tengan la misma interfaz de `Instrument`, y puedan responder a los mismos mensajes), lo que implica que sus direcciones pueden ser metidas en el array. Sin embargo, el compilador no sabe que sean otra cosa que objetos de tipo `Instrument`, por lo que normalmente llamará a las versiones de las funciones que estén en la clase base. Pero en este caso, todas las funciones han sido declaradas con la palabra reservada `virtual`, por lo que ocurre algo diferente. Cada vez que se crea una clase que contiene funciones virtuales, o se deriva de una clase que contiene funciones virtuales, el compilador crea para cada clase una única VTABLE, que se puede ver a la derecha en el diagrama. En ésta tabla se colocan las direcciones de todas las funciones que son declaradas virtuales en la clase o en la clase base. Si no se sobreescribe una función que ha sido declarada como virtual, el compilador usa la dirección de la versión que se encuentra en la clase base (esto se puede ver en la entrada `adjusta` de la VTABLE de `Brass`). Además, se coloca el VPTR (descubierto en `Sizes.cpp`) en la clase. Hay un único VPTR por cada objeto cuando se usa herencia simple como es el caso. El VPTR debe estar inicializado para que apunte a la dirección inicial de la VTABLE apropiada (esto sucede en el constructor que se verá más tarde con mayor detalle).

Una vez que el VPTR ha sido inicializado a la VTABLE apropiada, el objeto "sabe" de que tipo es. Pero este autoconocimiento no tiene valor a menos que sea usado en el momento en que se llama a la función virtual.

Cuando se llama a una función virtual a través de la clase base (la situación que se da cuando el compilador no tiene toda la información necesaria para realizar la ligadura estática), ocurre algo especial. En vez de realizarse la típica llamada a función, que en lenguaje ensamblador es simplemente un `CALL` a una dirección en concreto, el compilador genera código diferente para ejecutar la llamada a la función. Aquí se muestra a lo que se parece una llamada a `adjust()` para un objeto `Brass`, si se hace a través de un puntero a `Instrument` (una referencia a `Instrument` produce el mismo efecto):

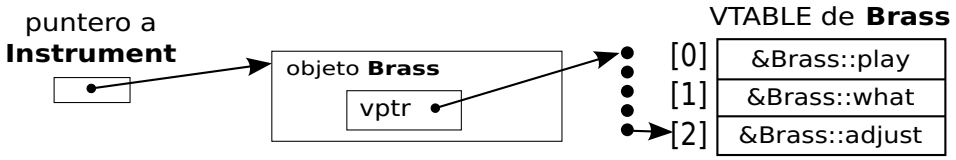

Figura 15.2: Tabla de punteros virtuales

El compilador empieza con el puntero a `Instrument`, que apunta a la dirección inicial del objeto. Todos los objetos `Instrument` o los objetos derivados de `Instrument` tienen su VPTR en el mismo lugar (a menudo al principio del objeto), de tal forma que el compilador puede conseguir el VPTR del objeto. El VPTR apunta a la la dirección inicial de VTABLE. Todas las direcciones de funciones de las VTABLE están dispuestas en el mismo orden, a pesar del tipo específico del objeto. `play()` es el primero, `what()` es el segundo y `adjust()` es el tercero. El compilador sabe que a pesar del tipo específico del objeto, la función `adjust()` se encuentra localizada en VPTR+2. Debido a esto, en vez de decir, "Llama a la función en la dirección absoluta `Instrument::adjust()` (ligadura estática y acción equivocada), se genera código que dice "Llama a la función que se encuentre en VPTR+2". Como la búsqueda del VPTR y la determinación de la dirección de la función actual ocurre en tiempo de ejecución, se consigue la deseada ligadura dinámica. Se envía un mensaje al objeto,

y el objeto se figura que debe hacer con él.

15.5.3. Detrás del telón

Puede ser útil ver el código ensamblador que se genera con la llamada a una función virtual, para poder ver como funciona la ligadura dinámica. Aquí está la salida de un compilador a la llamada

```
i.adjust(1);
```

dentro de la función `f(Instrument& i)`:

```
push 1
push si
mov  bx, word ptr [si]
call word ptr [bx+4]
add  sp, 4
```

Los argumentos de una llamada a una función C++, como los de a una función C, son colocados en la pila de derecha a izquierda (este orden es necesario para poder soportar las listas de argumentos variables de C), por lo que el argumento `1` se pone al principio en la pila. En este punto en la función, el registro `si` (que es parte de la arquitectura del procesador Intel™ X86) contiene la dirección de `i`. También se introduce en la pila porque es la dirección de comienzo del objeto de interés. Hay que recordar que la dirección del comienzo del objeto corresponde al valor de `this`, y `this` es introducido en la pila de manera oculta antes de cualquier llamada a función, por lo que la función miembro sabe sobre qué objeto en concreto está trabajando. Debido a esto se verá siempre uno más que el número de argumentos introducidos en la pila antes de una llamada a una función miembro (excepto para las funciones miembro `static`, que no tienen `this`).

Ahora se puede ejecutar la llamada a la función virtual. Primero hay que producir el VPTR para poder encontrar la VTABLE. Para el compilador el VPTR se inserta al principio del objeto, por lo que el contenido de `this` corresponde al VPTR. La línea

```
mov bx, word ptr [si]
```

busca la dirección *(word)* a la que apunta `si`, que es el VPTR y la coloca dentro del registro `bx`.

El VPTR contenido en `bx` apunta a la dirección inicial de la VTABLE, pero el puntero de la función a llamar no se encuentra en la posición cero de la VTABLE, si no en la segunda posición (debido a que es la tercera función en la lista). Debido al modelo de memoria cada puntero a función ocupa dos bytes, por lo que el compilador suma cuatro al VPTR para calcular donde está la dirección de la función apropiada. Hay que hacer notar que este es un valor constante establecido en tiempo de compilación, por lo que lo único que ocurre es que el puntero a función que está en la posición dos apunta a `adjust()`. Afortunadamente, el compilador se encarga de todo y se asegura de que todos los punteros a funciones en todas las VTABLEs de una jerarquía particular se creen con el mismo orden, a pesar del orden en que se hayan sobreescrito las funciones en las clases derivadas.

Una vez se ha calculado en la VTABLE la dirección del puntero apropiado, se llama a la función a la que apunta el puntero. Esto es, se busca la dirección y se llama

de una sola vez con la sentencia:

```
call word ptr [bx+4]
```

Finalmente, se retrocede el puntero de la pila para limpiar los argumentos que se pusieron antes de la llamada. En código ensamblador de C y de C++ se ve a menudo la instrucción para limpiar la lista de argumentos pero puede variar dependiendo del procesador o de la implementación del compilador.

15.5.4. Instalar el vpointer

Debido a que el VPTR determina el comportamiento virtual de las funciones en un objeto, es crítico que el VPTR siempre esté apuntando a la VTABLE apropiada. No tendría sentido hacer una llamada a una función virtual antes de que esté inicializado apropiadamente a su correspondiente VTABLE. Por supuesto, el lugar donde se puede garantizar esta inicialización es en el constructor, pero ninguno de los ejemplos `Instrument` tiene constructor.

Aquí es donde la creación del constructor por defecto es esencial. En los ejemplos `Instrument`, el compilador crea un constructor por defecto que no hace nada más que inicializar el VPTR. Este constructor es, obviamente, llamado autormáticamente por todos los objetos `Instrument` antes de que se pueda hacer nada con ellos, lo que asegura el buen comportamiento con las llamadas a funciones virtuales.

Las implicaciones de la inicialización automática del VPTR dentro de un constructor se discute en un sección posterior.

15.5.5. Los objetos son diferentes

Es importante darse cuenta de que el *upcasting* sólo maneja direcciones. Si el compilador tiene un objeto, sabe su tipo concreto y además (en C++) no se usará la ligadura dinámica para ninguna de las llamadas a funciones - o como mínimo el compilador no *necesitará* usar la ligadura dinámica. Por cuestiones de eficiencia, la mayoría de los compiladores usarán la ligadura estática cuando esten haciendo una llamada a una función virtual de un objeto porque saben su tipo concreto. Aquí hay un ejemplo:

```
//: C15:Early.cpp
// Early binding & virtual functions
#include <iostream>
#include <string>
using namespace std;

class Pet {
public:
  virtual string speak() const { return ""; }
};

class Dog : public Pet {
public:
  string speak() const { return "Bark!"; }
};

int main() {
```

```
  Dog ralph;
  Pet* p1 = &ralph;
  Pet& p2 = ralph;
  Pet p3;
  // Late binding for both:
  cout << "p1->speak() = " << p1->speak() <<endl;
  cout << "p2.speak() = " << p2.speak() << endl;
  // Early binding (probably):
  cout << "p3.speak() = " << p3.speak() << endl;
} ///:~
```

En `p1->speak()` y en `p2.speak()`, se usan direcciones, lo que significa que la información es incompleta: `p1` y `p2` pueden representar la dirección de una `Pet` o algo que derivee de una `Pet`, por lo que el debe ser usado el mecanismo virtual. Cuando se llama a `p3.speak` no existe ambigüedad. El compilador conoce el tipo exacto del objeto, lo que hace imposible que sea un objeto derivado de `Pet` - es *exactamente* una `Pet`. Por esto, probablemente se use la ligadura estática. Sin embargo, si el compilador no quiere trabajar mucho, puede usar la ligadura dinámica y el comportamiento será el mismo.

15.6. ¿Por qué funciones virtuales?

A estas alturas usted se puede hacer una pregunta: "Si esta técnica es tan importante, y si se ejecuta la función *correcta* todo el tiempo, ¿por qué es una opción? ¿por qué es necesario conocerla?"

Es una buena pregunta, y la respuesta se debe a la filosofía fudamental de C++: "Debido a que no es tan eficiente". Se puede ver en el código en lenguaje ensamblador que se generan, en vez de un simple `CALL` a una dirección absoluta, dos instrucciones ensamblador necesarias para preparar la llamada a función. Esto requiere más código y tiempo de ejecución.

Algunos lenguajes orientado a objetos han decidido que la aproximación a la ligadura dinámica es intrínseca a la programación orientada a objetos, que siempre debe tener lugar, que no puede ser opcional, y que el usuario no tiene por qué conocerlo. Esta es una decisión de diseño cuando se crea un lenguaje, y éste camino particular es adecuado para varios lenguajes[4]. Sin embargo, C++ tiene una tara por venir de C, donde la eficiencia es crítica. Después de todo, C fué creado para sustituir al lenguaje ensamblador para la implementación de un sistema operativo (haciendo a este sistema operativo - Unix - mucho más portable que sus antecesores). Y una de las principales razones para la invención de C++ fue hacer más eficientes a los programadores de C[5]. Y la primera pregunta cuando un programador de C se pasa a C++ es, "¿Cómo me afectará el cambio en velocidad y tamaño? Si la respuesta fuera, "Todo es magnífico excepto en las llamadas a funciones donde siempre tendrá un pequeña sobrecarga extra", mucha gente se hubiera aguantado con C antes que hacer el cambio a C++. Además las funciones inline no serían posibles, porque las funciones virtuales deben tener una dirección para meter en la VTABLE. Por lo tanto las funciones virtuales son opcionales, *y* por defecto el lenguaje no es `virtual`, porque es la configuración más eficiente. Stroustrup expuso que su línea de trabajo

[4] Smalltalk, Java y Python, por ejemplo, usan esta aproximación con gran éxito.

[5] En los laboratorios Bell, donde se inventó C, hay *un montón* de programadores de C. Hacerlos más eficientes, aunque sea sólo un poco, ahorra a la compañía muchos millones.

era, "Si no lo usa, no lo pague".

Además la palabra reservada `virtual` permite afinar el rendimiento. Cuando se diseñan las clases, sin embargo, no hay que preocuparse por afinarlas. Si va a usar el polimorfismo, úselo en todos los sitios. Sólo es necesario buscar funciones que se puedan hacer no virtuales cuando se esté buscando modos de acelerar el código (y normalmente hay mucho más que ganar en otras áreas - una buena idea es intentar adivinar dónde se encuentran los cuellos de botella).

Como anécdota la evidencia sugiere que el tamaño y la velocidad de C++ sufren un impacto del 10 por ciento con respecto a C, y a menudo están mucho más cerca de ser parejos. Además otra razón es que se puede diseñar un programa en C++ más rápido y más pequeño que como sería en C.

15.7. Clases base abstractas y funciones virtuales puras

A menudo en el diseño, se quiere la clase base para presentar *sólo* una interfaz para sus clases derivadas. Esto es, se puede querer que nadie pueda crear un objeto de la clase base y que ésta sirva únicamente para hacer un *upcast* hacia ella, y poder tener una interfaz. Se consigue haciendo a la clase *abstract* (abstracta), poniendo como mínimo una *función virtual pura*. Se puede reconocer a una función virtual pura porque usa la palabra reservada `virtual` y es seguida por `=0`. Si alguien intenta hacer un objeto de una clase abstracta, el compilador lo impide. Esta es una utilidad que fuerza a un diseño en concreto.

Cuando se hereda una clase abstracta, hay que implementar todas las funciones virtuales, o la clase que hereda se convierte en una nueva clase abstracta. Crear una función virtual pura permite poner una fución miembro en una interfaz sin forzar a proveer un cuerpo con código sin significado para esa función miembro. Al mismo tiempo, una función virtual fuerza a las clases que la hereden a que implemente una definición para ellas.

En todos los ejemplos de los intrumentos, las funciones en la clase base `Instrument` eran siempre funciones «tontas». Si esas funciones hubieran sido llamadas algo iba mal. Esto es porque la intención de la clase `Instrument` es crear una interfaz común para todas las clases que deriven de ella.

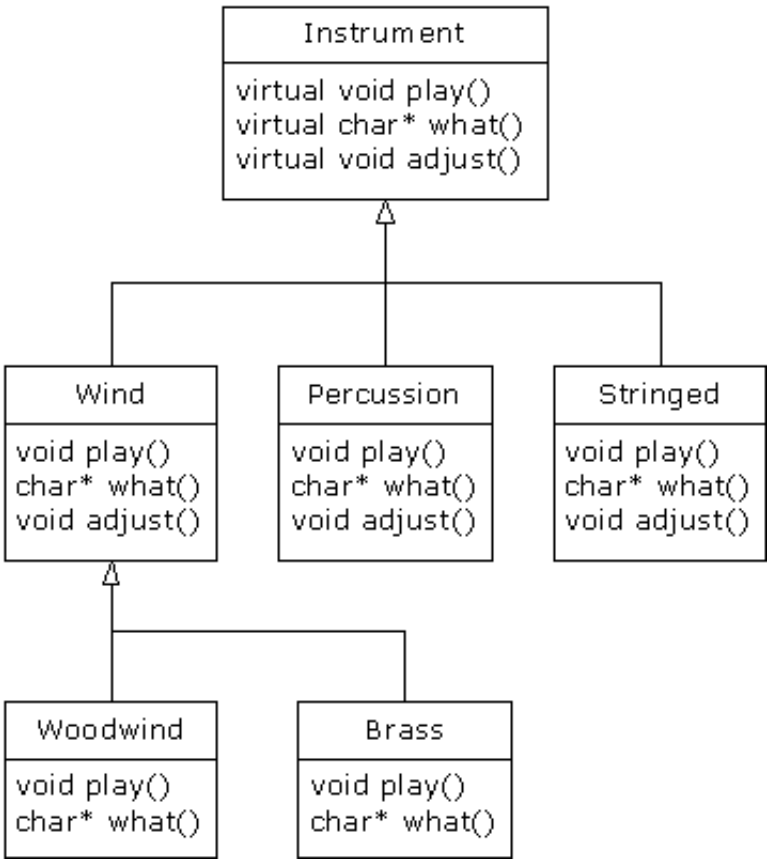

Figura 15.3: Clase abstracta

La única razón para establecer una interfaz común es que después se pueda expresar de forma diferente en cada subtipo. Se crea una forma básica que tiene lo que está en común con todas las clases derivadas y nada más. Por esto, `Instrument` es un candidato perfecto para ser una clase abstracta. Se crea una clase abstracta sólo cuando se quiere manipular un conjunto de clases a través de una interfaz común, pero la interfaz común no necesita tener una implementación (o como mucho, no necesita una implementación completa).

Si se tiene un concepto como `Instrument` que funciona como clase abstracta, los objetos de esa clase casi nunca tendrán sentido. Es decir, `Instrument` sirve solamente para expresar la interfaz, y no una implementación particular, por lo que crear un objeto que sea únicamente un `Instrument` no tendrá sentido, y probablemente se quiera prevenir al usuario de hacerlo. Se puede solucionar haciendo que todas las funciones virtuales en `Instrument` muestren mensajes de error, pero retrasa la aparición de los errores al tiempo de ejecución lo que obligará a un testeo exhaustivo por parte del usuario. Es mucho más productivo cazar el problema en tiempo de compilación.

Aquí está la sintaxis usada para una función virtual pura:

```
virtual void f() = 0;
```

Haciendo esto, se indica al compilador que reserve un hueco para una función en la VTABLE, pero que no ponga una dirección en ese hueco. Incluso aunque sólo una

función en una clase sea declarada como virtual pura, la VTABLE estará incompleta.

Si la VTABLE de una clase está incompleta, ¿qué se supone que debe hacer el compilador cuando alguien intente crear un objeto de esa clase? No sería seguro crear un objeto de esa clase abstracta, por lo que se obtendría un error de parte del compilador. Dicho de otra forma, el compilador garantiza la pureza de una clase abstracta. Hacer clases abstractas asegura que el programador cliente no puede hacer mal uso de ellas.

Aquí tenemos `Instrument4.cpp` modificado para usar funciones virtuales puras. Debido a que la clase no tiene otra cosa que no sea funciones virtuales, se la llama *clase abstracta pura*:

```
//: C15:Instrument5.cpp
// Pure abstract base classes
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  // Pure virtual functions:
  virtual void play(note) const = 0;
  virtual char* what() const = 0;
  // Assume this will modify the object:
  virtual void adjust(int) = 0;
};
// Rest of the file is the same ...

class Wind : public Instrument {
public:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
  char* what() const { return "Wind"; }
  void adjust(int) {}
};

class Percussion : public Instrument {
public:
  void play(note) const {
    cout << "Percussion::play" << endl;
  }
  char* what() const { return "Percussion"; }
  void adjust(int) {}
};

class Stringed : public Instrument {
public:
  void play(note) const {
    cout << "Stringed::play" << endl;
  }
  char* what() const { return "Stringed"; }
  void adjust(int) {}
};

class Brass : public Wind {
public:
```

```
  void play(note) const {
    cout << "Brass::play" << endl;
  }
  char* what() const { return "Brass"; }
};

class Woodwind : public Wind {
public:
  void play(note) const {
    cout << "Woodwind::play" << endl;
  }
  char* what() const { return "Woodwind"; }
};

// Identical function from before:
void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

// New function:
void f(Instrument& i) { i.adjust(1); }

int main() {
  Wind flute;
  Percussion drum;
  Stringed violin;
  Brass flugelhorn;
  Woodwind recorder;
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  tune(recorder);
  f(flugelhorn);
} ///:~
```

Las funciones virtuales puras son útiles porque hacen explícita la abstracción de una clase e indican al usuario y al compilador cómo deben ser usadas.

Hay que hacer notar que las funciones virtuales puras previenen a una clase abstracta de ser pasadas a una función *por valor*, lo que es una manera de prevenir el *object slicing* (que será descrito de forma reducida). Convertir una clase en abstracta también permite garantizar que se use siempre un puntero o una referencia cuando se haga *upcasting* a esa clase.

Sólo porque una función virtual pura impida a la VTABLE estar completa no implica que no se quiera crear cuerpos de función para alguna de las otras funciones. A menudo se querrá llamar a la versión de la función que esté en la clase base, incluso aunque ésta sea virtual. Es una buena idea poner siempre el código común tan cerca como sea posible de la raiz de la jerarquía. No sólo ahorra código, si no que permite fácilmente la propagación de cambios.

15.7.1. Definiciones virtuales puras

Es posible proveer una definición para una función virtual pura en la clase base. Todavía implica decirle al compilador que no permita crear objetos de esa clase base abstracta, y que las funciones virtuales puras deben ser definidas en las clases derivadas para poder crear objetos. Sin embargo, puede haber un trozo de código en común que se quiera llamar desde todas, o algunas de las clases derivadas en vez de estar duplicando código en todas las funciones.

Este es un ejemplo de definición de funciones virtuales.

```
//: C15:PureVirtualDefinitions.cpp
// Pure virtual base definitions
#include <iostream>
using namespace std;

class Pet {
public:
  virtual void speak() const = 0;
  virtual void eat() const = 0;
  // Inline pure virtual definitions illegal:
  //!  virtual void sleep() const = 0 {}
};

// OK, not defined inline
void Pet::eat() const {
  cout << "Pet::eat()" << endl;
}

void Pet::speak() const {
  cout << "Pet::speak()" << endl;
}

class Dog : public Pet {
public:
  // Use the common Pet code:
  void speak() const { Pet::speak(); }
  void eat() const { Pet::eat(); }
};

int main() {
  Dog simba;  // Richard's dog
  simba.speak();
  simba.eat();
} ///:~
```

El hueco en la VTABLE de `Pet` todavía está vacío, pero tiene funciones a las que se puede llamar desde la clase derivada.

Otra ventaja de esta característica es que perimite cambiar de una función virtual corriente a una virtual pura sin destrozar el código existente (es una forma para localizar clases que no sobreescriban a esa función virtual).

15.8. Herencia y la VTABLE

Es fácil imaginar lo que sucede cuando hay herencia y se sobreescriben algunas de las funciones virtuales. El compilador crea una nueva VTABLE para la nueva clase, e inserta las nuevas direcciones de las funciones usando además las direcciones de las funciones de la clase base para aquellas funciones virtuales que no se hayan sobreescrito. De un modo u otro, para todos los objetos que se puedan crear (es decir, aquellos que no tengan funciones virtuales puras) existe un conjunto completo de direcciones de funciones en la VTABLE, por lo que será imposible hacer llamadas a una dirección que no esté en la VTABLE (lo cual sería desastroso).

Pero ¿qué ocurre cuando se hereda y añade una nueva función virtual en la clase derivada? Aquí hay un sencillo ejemplo:

```
//: C15:AddingVirtuals.cpp
// Adding virtuals in derivation
#include <iostream>
#include <string>
using namespace std;

class Pet {
  string pname;
public:
  Pet(const string& petName) : pname(petName) {}
  virtual string name() const { return pname; }
  virtual string speak() const { return ""; }
};

class Dog : public Pet {
  string name;
public:
  Dog(const string& petName) : Pet(petName) {}
  // New virtual function in the Dog class:
  virtual string sit() const {
    return Pet::name() + " sits";
  }
  string speak() const { // Override
    return Pet::name() + " says 'Bark!'";
  }
};

int main() {
  Pet* p[] = {new Pet("generic"),new Dog("bob")};
  cout << "p[0]->speak() = "
       << p[0]->speak() << endl;
  cout << "p[1]->speak() = "
       << p[1]->speak() << endl;
//! cout << "p[1]->sit() = "
//!      << p[1]->sit() << endl; // Illegal
} ///:~
```

La clase `Pet` tiene dos funciones virtuales: `speak()` y `name()`. `Dog` añade una tercera función virtual llamada `sit()`, y sobreescribe el significado de `speak()`. Un diagrama ayuda a visualizar qué está ocurriendo. Se muestran las VTABLEs creadas por el compilador para `Pet` y `Dog`:

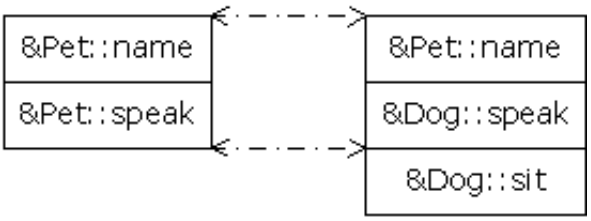

Figura 15.4: Una nueva función virtual

Hay que hacer notar, que el compilador mapea la dirección de `speak()` en exactamente el mismo lugar tanto en la VTABLE de `Dog` como en la de `Pet`. De igual forma, si una clase `Pug` heredara de `Dog`, su versión de `sit()` ocuparía su lugar en la VTABLE en la misma posición que en `Dog`. Esto es debido a que el compilador genera un código que usa un simple desplazamiento numérico en la VTABLE para seleccionar una función virtual, como se vio con el ejemplo en lenguaje ensamblador. Sin importar el subtipo en concreto del objeto, su VTABLE está colocada de la misma forma por lo que llamar a una función virtual se hará siempre del mismo modo.

En este caso, sin embargo, el compilador está trabajando sólo con un puntero a un objeto de la clase base. La clase base tiene únicamente las funciones `speak()` y `name()`, por lo que son a las únicas funciones a las que el compilador permitirá acceder. ¿Cómo es posible saber que se está trabajando con un objeto `Dog` si sólo hay un puntero a un objeto de la clase base? El puntero podría apuntar a algún otro tipo, que no tenga una función `sit()`. En este punto, puede o no tener otra dirección a función en la VTABLE, pero en cualquiera de los casos, hacer una llamada a una función virtual de esa VTABLE no es lo que se desea hacer. De modo que el compilador hace su trabajo impidiendo hacer llamadas virtuales a funciones que sólo existen en las clases derivadas.

Hay algunos poco comunes casos en los cuales se sabe que el puntero actualmente apunta al objeto de una subclase específica. Si se quiere hacer una llamada a una función que sólo exista en esa subclase, entonces hay que hacer un molde (*cast*) del puntero. Se puede quitar el mensaje de error producido por el anterior programa con:

```
((Dog *) p[1])->sit()
```

Aquí, parece saberse que `p[1]` apunta a un objeto `Dog`, pero en general no se sabe. Si el problema consiste en averiguar el tipo exacto de todos los objetos, hay que volver a pensar porque posiblemente no se estén usando las funciones virtuales de forma apropiada. Sin embargo, hay algunas situaciones en las cuales el diseño funciona mejor (o no hay otra elección) si se conoce el tipo exacto de todos los objetos, por ejemplo aquellos incluidos en un contenedor genérico. Este es el problema de la *run time type identification* o RTTI (identificación de tipos en tiempo de ejecución).

RTTI sirve para moldear un puntero de una clase base y "bajarlo" a un puntero de una clase derivada ("arriba" y "abajo", en inglés "up" y "down" respectivamente, se refieren al típico diagrama de clases, con la clase base arriba). Hacer el molde hacia arriba (*upcast*) funciona de forma automática, sin coacciones, debido a que es completamente seguro. Hacer el molde en sentido descendente (*downcast*) es inseguro porque no hay información en tiempo de compilación sobre los tipos actuales, por lo que hay que saber exactamente el tipo al que pertenece. Si se hace un molde al tipo

equivocado habrá problemas.

RTTI se describe posteriormente en este capítulo, y el Volumen 2 de este libro tiene un capítulo dedicado al tema.

15.8.1. FIXME: Object slicing

Existe una gran diferencia entre pasar una dirección de un objeto a pasar el objeto por valor cuando se usa el polimorfismo. Todos los ejemplos que se han visto, y prácticamente todos los ejemplos que se verán, se pasan por referencia y no por valor. Esto se debe a que todas las direcciones tienen el mismo tamaño[6], por lo que pasar la dirección de un tipo derivado (que normalmente será un objeto más grande) es lo mismo que pasar la dirección de un objeto del tipo base (que es normalmente más pequeño). Como se explicó anteriormente, éste es el objetivo cuando se usa el polimorfismo - el código que maneja un tipo base puede, también manejar objetos derivados de forma transparente

Si se hace un *upcast* de un objeto en vez de usar un puntero o una referencia, pasará algo que puede resultar sorprendente: el objeto es "truncado", recortado, hasta que lo que quede sea un subobjeto que corresponda al tipo destino del molde. En el siguiente ejemplo se puede ver que ocurre cuando un objeto es truncado (*object slicing*):

```
//: C15:ObjectSlicing.cpp
#include <iostream>
#include <string>
using namespace std;

class Pet {
  string pname;
public:
  Pet(const string& name) : pname(name) {}
  virtual string name() const { return pname; }
  virtual string description() const {
    return "This is " + pname;
  }
};

class Dog : public Pet {
  string favoriteActivity;
public:
  Dog(const string& name, const string& activity)
    : Pet(name), favoriteActivity(activity) {}
  string description() const {
    return Pet::name() + " likes to " +
      favoriteActivity;
  }
};

void describe(Pet p) { // Slices the object
  cout << p.description() << endl;
}
```

[6] Actualmente, no todos los punteros tienen el mismo tamaño en todos las máquinas. Sin embargo, en el contexto de esta discusión se pueden considerar iguales.

```
int main() {
  Pet p("Alfred");
  Dog d("Fluffy", "sleep");
  describe(p);
  describe(d);
} ///:~
```

La función `describe()` recibe un objeto de tipo `Pet` *por valor*. Después llama a la función virtual `description()` del objeto `Pet`. En el `main()`, se puede esperar que la primera llamada produzca "This is Alfred", y que la segunda produzca "Fluffy likes to sleep". De hecho, ambas usan la versión `description()` de la clase base.

En este programa están sucediendo dos cosas. Primero, debido a que `describe()` acepta un objeto `Pet` (en vez de un puntero o una referencia), cualquier llamada a `describe()` creará un objeto del tamaño de `Pet` que será puesto en la pila y posteriormente limpiado cuando acabe la llamada. Esto significa que si se pasa a `describe()` un objeto de una clase heredada de `Pet`, el compilador lo acepta, pero copia únicamente el fragmento del objeto que corresponda a una `Pet`. Se deshecha el fragmento derivado del objeto:

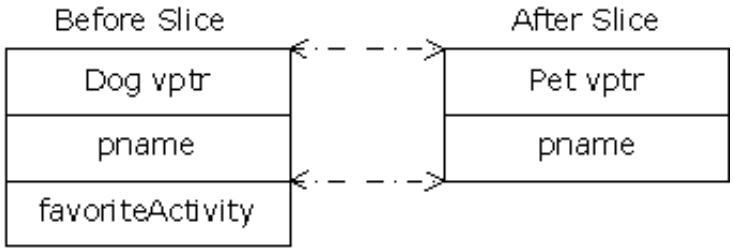

Figura 15.5: Object slicing

Ahora queda la cuestión de la llamada a la función virtual. `Dog::description()` hace uso de trozos de `Pet` (que todavía existe) y de `Dog`, ¡el cual no existe porque fue truncado!. Entonces, ¿Qué ocurre cuando se llama a la función virtual?

El desastre es evitado porque el objeto es pasado por valor. Debido a esto, el compilador conoce el tipo exacto del objeto porque el objeto derivado ha sido forzado a transformarse en un objeto de la clase base. Cuando se pasa por valor, se usa el constructor de copia del objeto `Pet`, que se encarga de inicializar el VPTR a la VTABLE de `Pet` y copia sólo las partes del objeto que correspondan a `Pet`. En el ejemplo no hay un constructor de copia explícito por lo que el compilador genera uno. Quitando interpretaciones, el objeto se convierte realmente en una `Pet` durante el truncado.

El *Object Slicing* quita parte del objeto existente y se copia en un nuevo objeto, en vez de simplemente cambiar el significado de una dirección cuando se usa un puntero o una referencia. Debido a esto, el *upcasting* a un objeto no se usa a menudo; de hecho, normalmente, es algo a controlar y prevenir. Hay que resaltar que en este ejemplo, si `description()` fuera una función virtual pura en la clase base (lo cual es bastante razonable debido a que realmente no hace nada en la clase base), entonces el compilador impedirá el *object slicing* debido a que no se puede "crear" un objeto de la clase base (que al fin y al cabo es lo que sucede cuando se hace un upcast por valor). ésto podría ser el valor más importante de las funciones virtuales puras: prevenir el *object slicing* generando un error en tiempo de compilación si alguien lo

intenta hacer.

15.9. Sobrecargar y redefinir

En el capítulo 14 se vio que redefinir una función sobrecargada en la función base oculta todas las otras versiones de esa función. Cuando se involucra a las funciones virtuales el comportamiento es un poco diferente. Consideremos una versión modificada del ejemplo `NameHiding.cpp` del capítulo 14:

```
//: C15:NameHiding2.cpp
// Virtual functions restrict overloading
#include <iostream>
#include <string>
using namespace std;

class Base {
public:
  virtual int f() const {
    cout << "Base::f()\n";
    return 1;
  }
  virtual void f(string) const {}
  virtual void g() const {}
};

class Derived1 : public Base {
public:
  void g() const {}
};

class Derived2 : public Base {
public:
  // Overriding a virtual function:
  int f() const {
    cout << "Derived2::f()\n";
    return 2;
  }
};

class Derived3 : public Base {
public:
  // Cannot change return type:
  //! void f() const{ cout << "Derived3::f()\n";}
};

class Derived4 : public Base {
public:
  // Change argument list:
  int f(int) const {
    cout << "Derived4::f()\n";
    return 4;
  }
};

int main() {
  string s("hello");
```

```
  Derived1 d1;
  int x = d1.f();
  d1.f(s);
  Derived2 d2;
  x = d2.f();
//!  d2.f(s); // string version hidden
  Derived4 d4;
  x = d4.f(1);
//!  x = d4.f(); // f() version hidden
//!  d4.f(s); // string version hidden
  Base& br = d4; // Upcast
//!  br.f(1); // Derived version unavailable
  br.f(); // Base version available
  br.f(s); // Base version abailable
} ///:~
```

La primera cosa a resaltar es que en `Derived3`, el compilador no permitirá cambiar el tipo de retorno de una función sobreescrita (lo permitiría si `f()` no fuera virtual). ésta es una restricción importante porque el compilador debe garantizar que se pueda llamar de forma "polimórfica" a la función a través de la clase base, y si la clase base está esperando que `f()` devuelva un int, entonces la versión de `f()` de la clase derivada debe mantener ese compromiso o si no algo fallará.

La regla que se enseño en el capítulo 14 todavía funciona: si se sobreescribe una de las funciones miembro sobrecargadas de la clase base, las otras versiones sobrecargadas estarán ocultas en la clase derivada. En el `main()` el código de `Derived4` muestra lo que ocurre incluso si la nueva versión de `f()` no está actualmente sobreescribiendo una función virtual existente de la interfaz - ambas versiones de `f()` en la clase base estan ocultas por `f(int)`. Sin embargo, si se hace un upcast de `d4` a `Base`, entonces únicamente las versiones de la clase base estarán disponibles (porque es el compromiso de la clase base) y la versión de la clase derivada no está disponible (debido a que no está especificada en la clase base).

15.9.1. Tipo de retorno variante

La clase `Derived3` de arriba viene a sugerir que no se puede modificar el tipo de retorno de una función virtual cuando es sobreescrita. En general es verdad, pero hay un caso especial en el que se puede modificar ligeramente el tipo de retorno. Si se está devolviendo un puntero o una referencia a una clase base, entonces la versión sobreescrita de la función puede devolver un puntero o una referencia a una clase derivada. Por ejemplo:

```
//: C15:VariantReturn.cpp
// Returning a pointer or reference to a derived
// type during ovverriding
#include <iostream>
#include <string>
using namespace std;

class PetFood {
public:
  virtual string foodType() const = 0;
};
```

```
class Pet {
public:
  virtual string type() const = 0;
  virtual PetFood* eats() = 0;
};

class Bird : public Pet {
public:
  string type() const { return "Bird"; }
  class BirdFood : public PetFood {
  public:
    string foodType() const {
      return "Bird food";
    }
  };
  // Upcast to base type:
  PetFood* eats() { return &bf; }
private:
  BirdFood bf;
};

class Cat : public Pet {
public:
  string type() const { return "Cat"; }
  class CatFood : public PetFood {
  public:
    string foodType() const { return "Birds"; }
  };
  // Return exact type instead:
  CatFood* eats() { return &cf; }
private:
  CatFood cf;
};

int main() {
  Bird b;
  Cat c;
  Pet* p[] = { &b, &c, };
  for(int i = 0; i < sizeof p / sizeof *p; i++)
    cout << p[i]->type() << " eats "
         << p[i]->eats()->foodType() << endl;
  // Can return the exact type:
  Cat::CatFood* cf = c.eats();
  Bird::BirdFood* bf;
  // Cannot return the exact type:
//!  bf = b.eats();
  // Must downcast:
  bf = dynamic_cast<Bird::BirdFood*>(b.eats());
} ///:~
```

La función miembro `Pet::eats()` devuelve un puntero a `PetFood`. En `Bird`, ésta función miembro es sobreescrita exactamente como en la clase base, incluyendo el tipo de retorno. Esto es, `Bird::eats()` hace un >*upcast* de `BirdFood` a `PetFood` en el retorno de la función.

Pero en `Cat`, el tipo devuelto por `eats()` es un puntero a `CatFood`, que es un tipo derivado de `PetFood`. El hecho de que el tipo de retorno esté heredado del tipo

de retorno la función de la clase base es la única razón que hace que esto compile. De esta forma el acuerdo se cumple totalmente: `eats()` siempre devuelve un puntero a `PetFood`.

Si se piensa de forma polimórfica lo anterior no parece necesario. ¿Por qué no simplemente se hacen upcast de todos los tipos retornados a PetFood* como lo hace `Bird::eats()`? Normalmente esa es una buena solución, pero al final del `main()` se puede ver la diferencia: `Cat::eats()` puede devolver el tipo exacto de `PetFood`, mientras que al valor retornado por `Bird::eats()` hay que hacerle un *downcast* al tipo exacto.

Devolver el tipo exacto es un poco más general y además no pierde la información específica de tipo debida al upcast automático. Sin embargo, devolver un tipo de la clase base generalmente resuelve el problema por lo que esto es una característica bastante específica.

15.10. funciones virtuales y constructores

Cuando se crea un objeto que contiene funciones virtuales, su VPTR debe ser inicializado para apuntar a la correcta VTABLE. Esto debe ser hecho antes de que exista la oportunidad de llamar a una función virtual. Como se puede adivinar, debido a que el constructor tiene el trabajo de traer a la existencia al objeto, también será trabajo del constructor inicializar el VPTR. El compilador de forma secreta añade código al principio del constructor para inicializar el VPTR. Y como se describe en el capítulo 14, si no se crea un constructor de una clase de forma explícita, el compilador genera uno de forma automática. Si la clase tiene funciones virtuales, el constructor incluirá el código apropidado para la inicialización del VPTR. Esto tiene varias consecuencias.

La primera concierne a la eficiencia. La razón de que existan funciones `inline` es reducir la sobrecarga que produce llamar a funciones pequeñas. Si C++ no proporciona funciones `inline`, el preprocesador debe ser usado para crear estas "macros". Sin embargo, el preprocesador no tiene los conceptos de accesos o clases, y además no puede ser usado para crear macros con funciones miembro. Además, con los constructores que deben tener código oculto insertado por el compilador, una macro del preprocesador no funcionaría del todo.

Hay que estar precavidos cuando se estén buscando agujeros de eficiencia porque el compilador está insertando código oculto en los constructores. No sólo hay que inicializar el VPTR, también hay que comprobar el valor de `this` (en caso de que el operador `new` devuelva cero), y llamar al constructor de la clase base. Todo junto, éste código puede tener cierto impacto cuando se pensaba que era una simple función `inline`. En particular, el tamaño del constructor puede aplastar al ahorro que se consigue al reducir la sobrecarga en las llamadas. Si se hacen un monton de llamadas a constructores inline, el tamaño del código puede crecer sin ningún beneficio en la velocidad.

Cuando esté afinando el código recuerde considerar el quitar los constructores en línea.

15.10.1. Orden de las llamadas a los constructores

La segunda faceta interesante de los constructores y las funciones virtuales tiene que ver con el orden de las llamadas a los constructores y el modo en que las llamadas virtuales se hacen dentro de los constructores.

Todos los constructores de la clase base son siempre llamados en el constructor de una clase heredada. Esto tiene sentido porque el constructor tiene un trabajo especial: ver que el objeto está construido de forma apropiada. Una clase derivada sólo tiene acceso a sus propios miembros, y no a los de la clase base. únicamente el constructor de la clase base puede inicializar de forma adecuada a sus propios elementos. Por lo tanto es esencial que se llame a todos los constructores; de otra forma el objeto no estará construido de forma adecuada. Esto es por lo que el compilador obliga a hacer una llamada por cada trozo en una clase derivada. Se llamará al constructor por defecto si no se hace una llamada explícita a un constructor de la clase base. Si no existe constructor por defecto, el compilador lo creará.

El orden de las llamadas al constructor es importante. Cuando se hereda, se sabe todo sobre la clase base y se puede acceder a todos los miembros públicos y protegidos (`public` y `protected`) de la clase base. ésto significa que se puede asumir que todos los miembros de la clase base son válidos cuando se está en la clase derivada. En una función miembro normal, la construcción ya ha ocurrido, por lo que todos los miembros de todas las partes del objeto ya han sido construidos. Dentro del constructor, sin embargo, hay que asumir que todos los miembros que se usen han sido construidos. La única manera de garantizarlo es llamando primero al constructor de la clase base. Entonces cuando se esté en el constructor de la clase derivada, todos los miembros a los que se pueda acceder en la clase base han sido inicializados. "Saber que todos los miembros son válidos" dentro del constructor es también la razón por la que, dentro de lo posible, se debe inicializar todos los objetos miembros (es decir, los objetos puestos en la clase mediante composición). Si se sigue ésta práctica, se puede asumir que todos los miembros de la clase base y los miembros objetos del objeto actual han sido inicializados.

15.10.2. Comportamiento de las funciones virtuales dentro de los constructores

La jerarquía de las llamadas a los constructores plantea un interesante dilema. ¿Qué ocurre si se está dentro de un constructor y se llama a una función virtual? Dentro de una función miembro ordinaria se puede imaginar que ocurrirá - la llamada virtual es resuelta en tiempo de ejecución porque el objeto no puede conocer si la función miembro es de la clase en la que está o es de una clase derivada. Por consistencia, se podría pensar que también es lo que debería ocurrir dentro de los constructores.

No es el caso. Si se llama a una función virtual dentro de un constructor, sólo se usa la versión local de la función. Es decir, el mecanismo virtual no funciona dentro del constructor.

éste comportamiento tiene sentido por dos motivos. Conceptualmente, el trabajo del constructor es dar al objeto una *existencia*. Dentro de cualquier constructor, el objeto puede ser formado sólo parcialmente - se puede saber sólo que los objetos de la clase base han sido inicializados, pero no se puede saber que clases heredan de ésta. Una función virtual, sin embargo, se mueve "arriba" y "abajo" dentro de la jerarquía de herencia. Llama a una función de una clase derivada. Si se pudiera hacer esto dentro de un constructor, se estaría llamando a una función que debe manejar miembros que todavía no han sido inicializados, una receta segura para el desastre.

El segundo motivo es mecánico. Cuando se llama a un constructor, una de las primeras cosas que hace es inicializar su VPTR. Sin embargo, sólo puede saber que es del tipo "actual" - el tipo para el que se ha escrito el constructor. El código del constructor ignora completamente si el objeto está en la base de otra clase. Cuando

el compilador genera código para ese constructor, se genera código para un constructor de esa clase, no para la clase base, ni para una clase derivada (debido a que una clase no puede saber quién la hereda). Por eso, el VPTR que use debe apuntar a la VTABLE de *esa* clase. El VPTR permanece inicializado a la VTABLE para el resto de vida del objeto a menos que no sea la última llamada al constructor. Si posteriormente se llama a un constructor de una clase derivada, éste constructor pone el VPTR a *su* VTABLE, y así hasta que el último constructor termine. El estado del VPTR es determinado por el constructor que sea llamado en último lugar. Otra razón por la que los constructores son llamados en orden desde la base al más derivado.

Pero mientras que toda esta serie de llamadas al constructor tiene lugar, cada constructor ha puesto el VPTR a su propia VTABLE. Si se usa el mecanismo virtual para llamar a funciones, producirá sólo una llamada a través de su propia VTABLE, y no de la VTABLE del más derivado (como debería suceder después de que todos los constructores hayan sido llamados). Además, muchos compiladores reconocen cuando se hace una llamada a una función virtual dentro de un constructor, y realizan una ligadura estática porque saben que la ligadura dinámica producirá una llamada a una función local. En todo caso, no se conseguirán los resultados que se podían esperar inicialmente de la llamada a una función virtual dentro de un constructor.

15.10.3. Destructores y destructores virtuales

No se puede usar la palabra reservada `virtual` con los constructores, pero los destructores pueden, y a menudo deben, ser virtuales.

El constructor tiene el trabajo especial de iniciar un objeto poco a poco, primero llamando al constructor base y después a los constructores derivados en el orden de la herencia. De manera similar, el destructor tiene otro trabajo especial: desmontar un objeto, el cual puede pertenecer a una jerarquía de clases. Para hacerlo, el compilador genera código que llama a todos los destructores, pero en el orden *inverso* al que son llamados en los constructores. Es decir, el constructor empieza en la clase más derivada y termina en la clase base. ésta es la opción deseable y segura debido a que el destructor siempre sabe que los miembros de la clase base están vivos y activos. Si se necesita llamar a una función miembro de la clase base dentro del destructor, será seguro hacerlo. De esta forma, el destructor puede realizar su propio limpiado, y entonces llamar al siguiente destructor, el cual hará *su* propio limpiado, etc. Cada destructor sabe de que clase deriva, pero no cuales derivan de él.

Hay que tener en cuenta que los constructores y los destructores son los únicos lugares donde tiene que funcionar ésta jerarquía de llamadas (que es automáticamente generada por el compilador). En el resto de las funciones, sólo esa función, sea o no virtual, será llamada (y no las versiones de la clase base). La única forma para acceder a las versiones de la clase base de una función consiste en llamar de forma *explicita* a esa funciones.

Normalmente, la acción del destructor es adecuada. Pero ¿qué ocurre si se quiere manipular un objeto a través de un puntero a su clase base (es decir, manipular al objeto a través de su interfaz genérica)? Este tipo de actividades es uno de los objetivos de la programación orientada a objetos. El problema viene cuando se quiere hacer un `delete` (eliminar) de un puntero a un objeto que ha sido creado en el *montón* (>*heap*) con `new`. Si el puntero apunta a la clase base, el compilador sólo puede conocer la versión del destructor que se encuentre en la clase base durante el `delete`. ¿Suena familiar? Al fin y al cabo, es el mismo problema por las que fueron creadas las funciones virtuales en el caso general. Afortunadamente, las funciones virtuales

funcionan con los destructores como lo hacen para las otras funciones excepto los constructores.

```
//: C15:VirtualDestructors.cpp
// Behavior of virtual vs. non-virtual destructor
#include <iostream>
using namespace std;

class Base1 {
public:
  ~Base1() { cout << "~Base1()\n"; }
};

class Derived1 : public Base1 {
public:
  ~Derived1() { cout << "~Derived1()\n"; }
};

class Base2 {
public:
  virtual ~Base2() { cout << "~Base2()\n"; }
};

class Derived2 : public Base2 {
public:
  ~Derived2() { cout << "~Derived2()\n"; }
};

int main() {
  Base1* bp = new Derived1; // Upcast
  delete bp;
  Base2* b2p = new Derived2; // Upcast
  delete b2p;
} ///:~
```

Cuando se ejecuta el programa, se ve que `delete bp` sólo llama al destructor de la clase base, mientras que `delete b2p` llama al destructor de la clase derivada seguido por el destructor de la clase base, que es el comportamiento que deseamos. Olvidar hacer `virtual` a un destructor es un error peligroso porque a menudo no afecta directamente al comportamiento del programa, pero puede introducir de forma oculta agujeros de memoria. Además, el hecho de que *alguna* destrucción está teniendo lugar puede enmascarar el problema.

Es posible que el destructor sea virtual porque el objeto sabe de que tipo es (lo que no ocurre durante la construcción del objeto). Una vez que el objeto ha sido construido, su VPTR es inicializado y se pueden usar las funciones virtuales.

15.10.4. Destructores virtuales puros

Mientras que los destructores virtuales puros son legales en el Standard C++, hay una restricción añadida cuando se usan: hay que proveer de un cuerpo de función a los destructores virtuales puros. Esto parece antinatural; ¿Cómo puede una función virtual ser "pura" si necesita el cuerpo de una función? Pero si se tiene en cuenta que los constructores y los destructores son operaciones especiales tiene más sentido, especialmente si se recuerda que todos los destructores en una jerarquía de clases

son llamados siempre. Si se quita la definición de un destructor virtual puro, ¿a qué cuerpo de función se llamará durante la destrucción? Por esto, es absolutamente necesario que el compilador y el enlazador requieran la existencia del cuerpo de una función para un destructor virtual puro.

Si es puro, pero la función tiene cuerpo ¿cuál es su valor? La única diferencia que se verá entre el destructor virtual puro y el no-puro es que el destructor virtual puro convierte a la clase base en abstracta, por lo que no se puede crear un objeto de la clase base (aunque esto también sería verdad si cualquier otra función miembro de esa clase base fuera virtual pura).

Sin embargo, las cosas son un poco confusas cuando se hereda una clase de otra que contenga un destructor puro virtual. Al contrario que en el resto de las funciones virtuales puras, *no* es necesario dar una definición de un destructor virtual puro en la clase derivada. El hecho de que el siguiente código compile es la prueba:

```
//: C15:UnAbstract.cpp
// Pure virtual destructors
// seem to behave strangely

class AbstractBase {
public:
  virtual ~AbstractBase() = 0;
};

AbstractBase::~AbstractBase() {}

class Derived : public AbstractBase {};
// No overriding of destructor necessary?

int main() { Derived d; } ///:~
```

Normalmente, una función virtual pura en una clase base causará que la clase derivada sea abstracta a menos que esa (y todas las demás funciones virtuales puras) tengan una definición. Pero aquí, no parece ser el caso. Sin embargo, hay que recordar que el compilador crea *automáticamente* una definición del destructor en todas las clases si no se crea una de forma explícita. Esto es lo que sucede aquí - el destructor de la clase base es sobreescrito de forma oculta, y una definición es puesta por el compilador por lo que `Derived` no es abstracta.

Esto nos brinda una cuestión interesante: ¿Cuál es el sentido de un destructor virtual puro? Al contrario que con las funciones virtuales puras ordinarias en las que hay que *dar* el cuerpo de una función, en una clase derivada de otra con un destructor virtual puro, no se está obligado a implementar el cuerpo de la función porque el compilador genera automáticamente el destructor. Entonces ¿Cuál es la diferencia entre un destructor virtual normal y un destructor virtual puro?

La única diferencia ocurre cuando se tiene una clase que sólo tiene una función virtual pura: el destructor. En este caso, el único efecto de la *pureza* del destructor es prevenir la instanciación de la clase base, pero si no existen otros destructores en las clase heredadas, el destructor virtual se ejecutará. Por esto, mientras que el añadir un destructor virtual es esencial, el hecho de que sea puro o no lo sea no es tan importante.

Cuando se ejecuta el siguiente ejemplo, se puede ver que se llama al cuerpo de la función virtual pura después de la versión que está en la clase derivada, igual que

con cualquier otro destructor.

```
//: C15:PureVirtualDestructors.cpp
// Pure virtual destructors
// require a function body
#include <iostream>
using namespace std;

class Pet {
public:
  virtual ~Pet() = 0;
};

Pet::~Pet() {
  cout << "~Pet()" << endl;
}

class Dog : public Pet {
public:
  ~Dog() {
    cout << "~Dog()" << endl;
  }
};

int main() {
  Pet* p = new Dog; // Upcast
  delete p; // Virtual destructor call
} ///:~
```

Como guía, cada vez que se tenga una función virtual en una clase, se debería añadir inmediatamente un destructor virtual (aunque no haga nada). De esta forma se evitan posteriores sorpresas.

15.10.5. Mecanismo virtual en los destructores

Hay algo que sucede durante la destrucción que no se espera de manera intuitiva. Si se está dentro de una función miembro y se llama a una función virtual, esa función es ejecutada usando el mecanismo de la ligadura dinámica. Esto no es verdad con los destructores, virtuales o no. Dentro de un destructor, sólo se llama a la función miembro "local"; el mecanismo virtual es ignorado.

```
//: C15:VirtualsInDestructors.cpp
// Virtual calls inside destructors
#include <iostream>
using namespace std;

class Base {
public:
  virtual ~Base() {
    cout << "Base1()\n";
    f();
  }
  virtual void f() { cout << "Base::f()\n"; }
};
```

```
class Derived : public Base {
public:
  ~Derived() { cout << "~Derived()\n"; }
  void f() { cout << "Derived::f()\n"; }
};

int main() {
  Base* bp = new Derived; // Upcast
  delete bp;
} ///:~
```

Durante la llamada al destructor virtual, no se llama a `Derived::f()`, incluso aunque `f()` es virtual.

¿A qué es debido ésto? Supongamos que fuera usado el mecanismo virtual dentro del destructor. Entonces sería posible para la llamada virtual resolver una función que está "lejana" (más derivada) en la jerarquía de herencia que el destructor actual. Pero los destructores son llamados de "afuera a dentro" (desde el destructor más derivado hacia el destructor de la clase base), por lo que la llamada actual a la función puede intentar acceder a fragmentos de un objeto que *!ya ha sido destruido!* En vez de eso, el compilador resuelve la llamada en tiempo de compilación y llama sólo a la versión local de la función. Hay que resaltar que lo mismo es también verdad para el constructor (como se explicó anteriormente), pero en el caso del constructor el tipo de información no estaba disponible, mientras que en el destructor la información está ahí (es decir, el VPTR) pero no es accesible.

15.10.6. Creación una jerarquía basada en objetos

Un asunto que ha aparecido de forma recurrente a lo largo de todo el libro cuando se usaban las clases `Stack` y `Stash` es el "problema de la propiedad". El "propietario" se refiere a quien o al que sea responsable de llamar al `delete` de aquellos objetos que hayan sido creados dinámicamente (usando `new`). El problema cuando se usan contenedores es que es necesario ser lo suficientemente flexible para manejar distintos tipos de objetos. Para conseguirlo, los contenedores manejan punteros a void por lo que no pueden saber el tipo del objeto que están manejando. Borrar un puntero a void no llama al destructor, por lo que el contenedor no puede ser responsable de borrar sus objetos.

Una solución fue presentada en el ejemplo `C14:InheritStack.cpp`, en el que `Stack` era heredado en una nueva clase que aceptaba y producía únicamente objetos string, por lo que se les podía borrar de manera adecuada. Era una buena solución pero requería heredar una nueva clase contenedera por cada tipo que se quisiera manejar en el contenedor. (Aunque suene un poco tedioso funciona bastante bien como se verá en el capítulo 16 cuando las plantillas o templates sean introducidos).

El problema es que se quiere que el contenedor maneje más de un tipo, pero sólo se quieren usar punteros a void. Otra solución es usar polimorfismo forzando a todos los objetos incluidos en el contenedor a ser heredados de la misma clase base. Es decir, el contenedor maneja los objetos de la clase base, y sólo hay que usar funciones virtuales - en particular, se pueden llamar a destructores virtuales para solucionar el problema de pertenencia.

Esta solución usa lo que se conoce como "jerarquía de raiz única" (*singly-rooted hierarchy*) o "jerarquía basada en objetos" (*object-based hierarchy*), siendo el último nom-

bre debido a que la clase raiz de la jerarquía suele ser llamada "Objeto". Además, el usar jerarquía de raiz única, tiene como resultado otros beneficios: de hecho, cualquier otro lenguaje orientado a objetos que no sea el C++ obliga a usar una jerarquía - cuando se crea una clase se hereda automáticamente de forma directa o indirecta de una clase base común, una clase base que fue establecida por los creadores del lenguaje. En C++, se penso que forzar a tener una base clase común crearía demasiada sobrecarga, por lo que se desestimó. Sin embargo, se puede elegir usar en nuestros proyectos una clase base común, y esta materia será tratada en el segundo volumen de este libro.

Para solucionar el problema de pertenencia, se puede crear una clase base `Object` extremadamente simple, que sólo tiene un destructor virtual. De esta forma `Stack` puede manejar objetos que hereden de `Object`:

```
//: C15:OStack.h
// Using a singly-rooted hierarchy
#ifndef OSTACK_H
#define OSTACK_H

class Object {
public:
  virtual ~Object() = 0;
};

// Required definition:
inline Object::~Object() {}

class Stack {
  struct Link {
    Object* data;
    Link* next;
    Link(Object* dat, Link* nxt) :
      data(dat), next(nxt) {}
  }* head;
public:
  Stack() : head(0) {}
  ~Stack(){
    while(head)
      delete pop();
  }
  void push(Object* dat) {
    head = new Link(dat, head);
  }
  Object* peek() const {
    return head ? head->data : 0;
  }
  Object* pop() {
    if(head == 0) return 0;
    Object* result = head->data;
    Link* oldHead = head;
    head = head->next;
    delete oldHead;
    return result;
  }
};
#endif // OSTACK_H ///:~
```

Para simplificar las cosas se crea todo en el fichero cabecera, la definición (requerida) del destructor virtual puro es introducida en línea el el fichero cabecera, y `pop()` también está en línea aunque podría ser considearado como demasiado largo para ser incluido así.

Los objetos `Link` (lista) ahora manejan punteros a `Object` en vez de punteros a void, y la `Stack` (pila) sólo aceptará y devolverá punteros a `Object`. Ahora `Stack` es mucho más flexible, ya que puede manejar un montón de tipos diferentes pero además es capaz de destruirá cualquier objeto dejado en la pila. La nueva limitación (que será finalmente eliminada cuando las plantillas se apliquen al problema en el capítulo 16) es que todo lo que se ponga en la pila debe ser heredado de `Object`. Esto está bien si se crea una clase desde la nada, pero ¿qué pasa si se tiene una clase como `string` y se quiere ser capaz de meterla en la pila? En este caso, la nueva clase debe ser al mismo tiempo un `string` y un `Object`, lo que significa que debe heredar de ambas clases. Esto se conoce como *herencia múltiple* y es materia para un capítulo entero en el Volumen 2 de este libro (se puede bajar de www.BruceEckel.com). cuando se lea este capítulo, se verá que la herencia múltiple genera un montón de complejidad, y que es una característica que hay que usar con cuentagotas. Sin embargo, ésta situación es lo suficiéntemente simple como para no tener problemas al usar herencia múltiple:

```
//: C15:OStackTest.cpp
//{T} OStackTest.cpp
#include "OStack.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

// Use multiple inheritance. We want
// both a string and an Object:
class MyString: public string, public Object {
public:
  ~MyString() {
    cout << "deleting string: " << *this << endl;
  }
  MyString(string s) : string(s) {}
};

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // File name is argument
  ifstream in(argv[1]);
  assure(in, argv[1]);
  Stack textlines;
  string line;
  // Read file and store lines in the stack:
  while(getline(in, line))
    textlines.push(new MyString(line));
  // Pop some lines from the stack:
  MyString* s;
  for(int i = 0; i < 10; i++) {
    if((s=(MyString*)textlines.pop())==0) break;
    cout << *s << endl;
    delete s;
  }
  cout << "Letting the destructor do the rest:"
```

```
    << endl;
} ///:~
```

Aunque es similar a la versión anterior del programa de pruebas de `Stack`, se puede ver que sólo se han sacado 10 elementos de la pila, lo que implica que probablemente quede algún elemento. Como la pila ahora maneja `Object`s, el destructor puede limpiarlos de forma adecuada, como se puede ver en la salida del programa gracias a que los objetos `MyString` muestran un mensaje cuando son destruidos.

Crear contenedores que manejen `Object`s es una aproximación razonable - si se tiene una jerarquía de raiz única (debido al lenguaje o por algún requerimiento que obligue a que todas las clases hereden de `Object`). En este caso, está garantizado que todo es un `Object` y no es muy complicado usar contenedores. Sin embargo, en C++ no se puede esperar este comportamiento de todas las clases, por lo que se está abocado a usar herencia múltiple si se quiere usar esta aproximación. Se verá en el capítulo 16 que las plantillas solucionan este problema de una forma más simple y elegante.

15.11. Sobrecarga de operadores

Se pueden crear operadores `virtual`es de forma análoga a otras funciones miembro. Sin embargo implementar operadores `virtual`es se vuelve a menudo confuso porque se está operando sobre dos objetos, ambos sin tipos conocidos. Esto suele ser el caso de los componentes matemáticos (para los cuales se suele usar la sobrecarga de operadores). Por ejemplo, considere un sistema que usa matrices, vectores y valores escalares, todos ellos heredados de la clase `Math`:

```
//: C15:OperatorPolymorphism.cpp
// Polymorphism with overloaded operators
#include <iostream>
using namespace std;

class Matrix;
class Scalar;
class Vector;

class Math {
public:
  virtual Math& operator*(Math& rv) = 0;
  virtual Math& multiply(Matrix*) = 0;
  virtual Math& multiply(Scalar*) = 0;
  virtual Math& multiply(Vector*) = 0;
  virtual ~Math() {}
};

class Matrix : public Math {
public:
  Math& operator*(Math& rv) {
    return rv.multiply(this); // 2nd dispatch
  }
  Math& multiply(Matrix*) {
    cout << "Matrix * Matrix" << endl;
    return *this;
  }
```

```
  Math& multiply(Scalar*) {
    cout << "Scalar * Matrix" << endl;
    return *this;
  }
  Math& multiply(Vector*) {
    cout << "Vector * Matrix" << endl;
    return *this;
  }
};

class Scalar : public Math  {
public:
  Math& operator*(Math& rv) {
    return rv.multiply(this); // 2nd dispatch
  }
  Math& multiply(Matrix*) {
    cout << "Matrix * Scalar" << endl;
    return *this;
  }
  Math& multiply(Scalar*) {
    cout << "Scalar * Scalar" << endl;
    return *this;
  }
  Math& multiply(Vector*) {
    cout << "Vector * Scalar" << endl;
    return *this;
  }
};

class Vector : public Math  {
public:
  Math& operator*(Math& rv) {
    return rv.multiply(this); // 2nd dispatch
  }
  Math& multiply(Matrix*) {
    cout << "Matrix * Vector" << endl;
    return *this;
  }
  Math& multiply(Scalar*) {
    cout << "Scalar * Vector" << endl;
    return *this;
  }
  Math& multiply(Vector*) {
    cout << "Vector * Vector" << endl;
    return *this;
  }
};

int main() {
  Matrix m; Vector v; Scalar s;
  Math* math[] = { &m, &v, &s };
  for(int i = 0; i < 3; i++)
    for(int j = 0; j < 3; j++) {
      Math& m1 = *math[i];
      Math& m2 = *math[j];
      m1 * m2;
    }
} ///:~
```

Para simplificar sólo se ha sobrecargado el operator*. El objetivo es ser capaz de multiplicar dos objetos `Math` cualquiera y producir el resultado deseado - hay que darse cuenta que multiplicar una matriz por un vector es una operación totalmente distinta a la de multiplicar un vector por una matriz.

El problema es que, en el `main()`, la expresión `m1 * m2` contiene dos referencias `Math`, y son dos objetos de tipo desconocido. Una función virtual es sólo capaz de hacer una única llamada - es decir, determinar el tipo de un único objeto. Para determinar ambos tipos en este ejemplo se usa una técnica conocida como despachado múltiple (*multiple dispatching*), donde lo que parece ser una única llamada a una función virtual se convierte en una segunda llamada a una función virtual. Cuando la segunda llamada se ha ejecutado, ya se han determinado ambos tipos de objetos y se puede ejecutar la actividad de forma correcta. En un principio no es transparante, pero después de un rato mirando el código empieza a cobrar sentido. Esta materia es tratada con más profundidad en el capítulo de los patrones de diseño en el Volumen 2 que se puede bajar de >www.BruceEckel.com.

15.12. Downcasting

Como se puede adivinar, desde el momento que existe algo conocido como upcasting - mover en sentido ascendente por una jerarquía de herencia - debe existir el *downcasting* para mover en sentido descendente en una jerarquía. Pero el upcasting es sencillo porque al movernos en sentido ascendente en la jerarquía de clases siempre convergemos en clases más generales. Es decir, cuando se hace un upcast siempre se está en una clase claramente derivada de un ascendente (normalmente solo uno, excepto en el caso de herencia múltiple) pero cuando se hace downcast hay normalmente varias posibilidades a las que amoldarse. Mas concretamente, un `Circulo` es un tipo de `Figura` (que sería su *upcast*), pero si se intenta hacer un *downcast* de una `Figura` podría ser un `Circulo`, un `Cuadrado`, un `Triángulo`, etc. El problema es encontrar un modo seguro de hacer *downcast* (aunque es incluso más importante preguntarse por qué se está usando *downcasting* en vez de usar el polimorfismo para que adivine automáticamente el tipo correcto. En el Volumen 2 de este libro se trata como evitar el *downcasting*.

C++ proporciona un moldeado explícito especial (introducido en el capítulo 3) llamado "moldeado dinámico" (*dynamic_cast*) que es una operación segura. Cuando se usa *moldeado dinámico* para intentar hacer un molde a un tipo en concreto, el valor de retorno será un puntero al tipo deseado sólo si el molde es adecuado y tiene éxito, de otra forma devuelve cero para indicar que no es del tipo correcto. Aquí tenemos un ejemplo mínimo:

```
//: C15:DynamicCast.cpp
#include <iostream>
using namespace std;

class Pet { public: virtual ~Pet(){}};
class Dog : public Pet {};
class Cat : public Pet {};

int main() {
  Pet* b = new Cat; // Upcast
  // Try to cast it to Dog*:
```

```
  Dog* d1 = dynamic_cast<Dog*>(b);
  // Try to cast it to Cat*:
  Cat* d2 = dynamic_cast<Cat*>(b);
  cout << "d1 = " << (long)d1 << endl;
  cout << "d2 = " << (long)d2 << endl;
} ///:~
```

Cuando se use *moldeado dinámico*, hay que trabajar con una jerarquía polimórfica real - con funciones virtuales - debido a que el *modeado dinámico* usa información almacenada en la VTABLE para determinar el tipo actual. Aquí, la clase base contiene un destructor virtual y esto es suficiente. En el `main()`, un puntero a `Cat` es elevado a `Pet`, y después se hace un downcast tanto a puntero `Dog` como a puntero a `Cat`. Ambos punteros son imprimidos, y se puede observar que cuando se ejecuta el programa el *downcast* incorrecto produce el valor cero. Por supuesto somos los responsables de comprobar que el resultado del cast no es cero cada vez que se haga un downcast. Además no hay que asumir que el puntero será exactamente el mismo, porque a veces se realizan ajustes de punteros durante el *upcasting* y el *downcasting* (en particular, con la herencia múltiple).

Un *moldeado dinámico* requiere un poco de sobrecarga extra en ejecución; no mucha, pero si se está haciendo mucho *moldeado dinámico* (en cuyo caso debería ser cuestionado seriamente el diseño del programa) se convierte en un lastre en el rendimiento. En algunos casos se puede tener alguna información especial durante el downcasting que permita conocer el tipo que se está manejando, con lo que la sobrecarga extra del *modeado dinámico* se vuelve innecesario, y se puede usar de manera alternativa un *moldeado estático*. Aquí se muestra como funciona:

```
//: C15:StaticHierarchyNavigation.cpp
// Navigating class hierarchies with static_cast
#include <iostream>
#include <typeinfo>
using namespace std;

class Shape { public: virtual ~Shape() {}; };
class Circle : public Shape {};
class Square : public Shape {};
class Other {};

int main() {
  Circle c;
  Shape* s = &c; // Upcast: normal and OK
  // More explicit but unnecessary:
  s = static_cast<Shape*>(&c);
  // (Since upcasting is such a safe and common
  // operation, the cast becomes cluttering)
  Circle* cp = 0;
  Square* sp = 0;
  // Static Navigation of class hierarchies
  // requires extra type information:
  if(typeid(s) == typeid(cp)) // C++ RTTI
    cp = static_cast<Circle*>(s);
  if(typeid(s) == typeid(sp))
    sp = static_cast<Square*>(s);
  if(cp != 0)
    cout << "It's a circle!" << endl;
```

```
  if(sp != 0)
    cout << "It's a square!" << endl;
  // Static navigation is ONLY an efficiency hack;
  // dynamic_cast is always safer. However:
  // Other* op = static_cast<Other*>(s);
  // Conveniently gives an error message, while
  Other* op2 = (Other*)s;
  // does not
} ///:~
```

En este programa, se usa una nueva característica que no será completamente descrita hasta el Volumen 2 de este libro, donde hay un capítulo que cubre este tema: *Información de tipo en tiempo de ejecución en C++* o mecanismo RTTI (*run time type information*). RTTI permite descubrir información de tipo que ha sido perdida en el upcasting. El *moldeado dinámico* es actualmente una forma de RTTI. Aquí se usa la palabra reservada `typeid` (declarada en el fichero cabecera `typeinfo`) para detectar el tipo de los punteros. Se puede ver que el tipo del puntero a `Figura` es comparado de forma sucesiva con un puntero a `Circulo` y con un `Cuadrado` para ver si existe alguna coincidencia. Hay más RTTI que el `typeid`, y se puede imaginar que es fácilmente implementable un sistema de información de tipos usando una función virtual.

Se crea un objeto `Circulo` y la dirección es elevada a un puntero a `Figura`; la segunda versión de la expresión muestra como se puede usar *modeado estático* para ser más explícito con el upcast. Sin embargo, desde el momento que un upcast siempre es seguro y es una cosa que se hace comunmente, considero que un cast explícito para hacer upcast ensucia el código y es innecesario.

Para determinar el tipo se usa RTTI, y se usa *modelado estático* para realizar el downcast. Pero hay que resaltar que, efectivamente, en este diseño el proceso es el mismo que usar el *moldeado dinámico*, y el programador cliente debe hacer algún test para descubrir si el cast tuvo éxito. Normalmente se prefiere una situación más determinista que la del ejemplo anterior para usar el *modeado estático* antes que el *moldeado dinámico* (y hay que examinar detenidamente el diseño antes de usar *moldeado dinámico*).

Si una jerarquía de clases no tiene funciones `virtual`es (que es un diseño cuestionable) o si hay otra información que permite hacer un downcast seguro, es un poco más rápido hacer el downcast de forma estática que con el *moldeado dinámico*. Además, *modeado estático* no permitirá realizar un cast fuera de la jerarquía, como un cast tradicional permitiría, por lo que es más seguro. Sin enbargo, navegar de forma estática por la jerarquía de clases es siempre arriesgado por lo que hay que usar *moldeado dinámico* a menos que sea una situación especial.

15.13. Resumen

Polimorfismo - implementado en C++ con las funciones virtuales - significa "formas diferentes". En la programación orientada a objetos, se tiene la misma vista (la interfaz común en la clase base) y diferentes formas de usarla: las diferentes versiones de las funciones virtuales.

Se ha visto en este capítulo que es imposible entender, ni siquiera crear, un ejemplo de polimorfismo sin usar la abstracción de datos y la herencia. El polimorfismo es una característica que no puede ser vista de forma aislada (como por ejemplo las

sentencias `const` y `switch`), pero sin embargo funciona únicamente de forma conjunta, como una parte de un "gran cuadro" de relaciones entre clases. La gente se vuelve a menudo confusa con otras características no orientadas a objetos de C++ como es la sobrecarga y los argumentos por defecto, los cuales son presentados a veces como orientado a objetos. No nos liemos; si no hay ligadura dinámica, no hay polimorfismo.

Para usar el polimorfismo - y por lo tanto, técnicas orientadas a objetos - en los programas hay que ampliar la visión de la programación para incluir no solo miembros y mensajes entre clases individuales, si no también sus puntos en común y las relaciones entre ellas. Aunque requiere un esfuerzo significativo, es recompensado gracias a que se consigue mayor velocidad en el desarrollo, mejor organización de código, programas extensibles, y mayor mantenibilidad.

El polimorfismo completa las características de orientación a objetos del lenguaje, pero hay dos características fundamentales más en C++: plantillas (introducidas en el capítulo 16 y cubiertas en mayor detalle en el segundo volumen de este libro), y manejo de excepciones (cubierto en el Volumen 2). Estas características nos proporcionan un incremento de poder de cada una de las características de la orientación a objetos: tipado abstracto de datos, herencia, y polimorfismo.

15.14. Ejercicios

Las soluciones a los ejercicios se pueden encontrar en el documento electrónico titulado «The Thinking in C++ Annotated Solution Guide», disponible por poco dinero en www.BruceEckel.com.

1. Cree una jerarquía simple "figura": una clase base llamada `Figura` y una clases derivadas llamadas `Circulo`, `Cuadrado`, y `Triangulo`. En la clase base, hay que hacer una función virtual llamada `dibujar()`, y sobreescribirla en las clases derivadas. Hacer un array de punteros a objetos `Figura` que se creen en el montón (heap) y que obligue a realizar upcasting de los punteros, y llamar a `dibujar()` a través de la clase base para verificar el comportamiento de las funciones virtuales. Si el depurador lo soporta, intente ver el programa paso a paso.

2. Modifique el Ejercicio 1 de tal forma que `dibujar()` sea una función virtual pura. Intente crear un objeto de tipo `Figura`. Intente llamar a la función virtual pura dentro del constructor y mire lo que ocurre. Dejándolo como una función virtual pura cree una definición para `dibujar()`.

3. Aumentando el Ejercicio 2, cree una función que use un objeto `Figura` *por valor* e intente hacer un upcast de un objeto derivado como argumento. Vea lo que ocurre. Arregle la función usando una referencia a un objeto `Figura`.

4. Modifique `C14:Combined.cpp` para que `f()` sea `virtual` en la clase base. Cambie el `main()` para que se haga un *upcast* y una llamada virtual.

5. Modifique `Instrument3.cpp` añadiendo una función virtual `preparar()`. Llame a `preparar()` dentro de `tune()`.

6. Cree una jerarquía de herencia de `Roedores`: `Raton`, `Gerbo`, `Hamster`, etc. En la clase base, proporcione los métodos que son comunes a todos los roedores, y redefina aquellos en las clases derivadas para que tengan diferentes comportamientos dependiendo del tipo específico de roedor. Cree un array de punteros

a `Roedor`, rellenelo con distintos tipos de roedores y llame a los métodos de la clase base para ver lo que ocurre.

7. Modifique el Ejercicio 6 para que use un `vector<Roedor*>` en vez de un array de punteros. Asegurese que se hace un limpiado correcto de la memoria.
8. Empezando con la jerarquía anterior de `Roedor`, herede un `HamsterAzul` de `Hamster` (si, existe algo así, tuve uno cuando era niño), sobreescriba los métodos de la clase base y muestre que el código que llama a los métodos de clase base no necesitan cambiar para adecuarse el nuevo tipo.
9. A partir de la jerarquía `Roedor` anterior, añadaun destructor no virtual, cree un objeto de la `Hamster` usando `new`, haga un upcast del puntero a Roedor*, y borre el puntero con `delete` para ver si no se llama a los destructores en la jerarquía. Cambie el destructor a `virtual` y demuestre que el comportamiento es ahora correcto.
10. Modifique `Roedor` para convertirlo en una clase base pura abstracta.
11. Cree un sistema de control aéreo con la clase base `Avion` y varios tipos derivados. Cree una clase `Torre` con un `vector<Avion*>` que envie los mensajes adecuados a los distintos aviones que están bajo su control.
12. Cree un modelo de invernadero heredando varios tipos de `Plantas` y construyendo mecanismos en el invernadero que se ocupen de las plantas.
13. En `Early.cpp`, haga a `Pet` una clase base abstracta pura.
14. En `AddingVirtuals.cpp`, haga a todas las funciones miembro de `Pet` virtuales puras, pero proporcione una definición para `name()`. Arregle `Dog` como sea necesario, usando la definición de `name()` que se encuentra en la clase base.
15. Escriba un pequeño programa para mostrar la diferencia entre llamar a una función virtual dentro de una función miembro normal y llamar a una función virtual dentro de un constructor. El programa de probar que las dos llamadas producen diferentes resultados.
16. Modifique `VirtualsInDestructors.cpp` por heredando una clase de `Derived` y sobreescribiendo `f()` y el destructor. En `main()`, cree y haga un upcast de un objeto de su nuevo tipo, después borrelo.
17. Use el Ejercicio 16 y añada llamadas a `f()` en cada destructor. Explique que ocurre.
18. Cree un clase que tenga un dato miembro y una clase derivada que añada otro dato miembro. Escriba una función no miembro que use un objeto de la clase base *por valor* e imprima el tamaño del objeto usando `sizeof`. En el `main()` cree un objeto de la clase derivada, imprima su tamaño, y llame a su función. Explique lo que ocurre.
19. Cree un ejemplo sencillo de una llamada a una función virtual y genere su salida en ensamblador. Localize el código en ensamblador para la llamada a la función virtual y explique el código.
20. Escriba una clase con una función virtual y una función no virtual. Herede una nueva clase, haga un objeto de esa clase, y un upcast a un puntero del tipo de la clase base. Use la función `clock()` que se encuentra en `<ctime>` (necesitará echar un vistazo a su librerí C) para medir la diferencia entre una llamada

virtual y una llamada no virtual. Será necesario realizar multiples llamadas a cada función para poder ver la diferencia.

21. Modifique `C14:Order.cpp` añadiendo una función virtual en la clase base de la macro `CLASS` (que pinte algo) y haciendo el destructor virtual. Cree objetos de las distintas subclases y hagales un upcast a la clase base. Verifique que el comportamiento virtual funciona y que se realiza de forma correcta la construcción y la destrucción del objeto.

22. Escriba una clase con tres funciones virtuales sobrecargadas. Herede una nueva clase y sobreescriba una de las funciones. Cree un objeto de la clase derivada. ¿Se puede llamar a todas las funciones de la clase base a través del objeto derivado? Haga un *upcast* de la dirección del objeto a la base. ¿Se pueden llamar a las tres funciones a través de la base? Elimine la definición sobreescrita en la clase derivada. Ahora ¿Se puede llamar a todas las funciones de la clase base a través del objeto derivado?.

23. Modifique `VariantReturn.cpp` para que muestre que su comportamiento funciona con referencias igual que con punteros.

24. En `Early.cpp`, ¿Cómo se le puede indicar al compilador que haga la llamada usando ligadura estática o ligadura dinámica? Determine el caso para su propio compilador.

25. Cree una clase base que contenga una función `clone()` que devuelva un puntero a una *copia* del objeto actual. Derive dos subclases que sobreescriban `clone()` para devolver copias de sus tipos específicos. En el `main()`, cree y haga *upcast* de sus dos tipos derivados, y llame a `clone()` para cada uno y verifique que las copias clonadas son de los subtipos correctos. Experimente con su función `clone()` para que se pueda ir al tipo base, y después intente regresar al tipo exacto derivado. ¿Se le ocurre alguna situación en la que sea necesario esta aproximación?

26. Modifique `OStackTest.cpp` creando su propia clase, después haga multiple herencia con `Object` para crear algo que pueda ser introducido en la pila. Pruebe su clase en el `main()`.

27. Añada un tipo llamado `Tensor` a `OperartorPolymorphism.cpp`.

28. (Intermedio) Cree una clase base `X` sin datos miembro y sin constructor, pero con una función virtual. Cree una `Y` que herede de `X`, pero sin un constructor explícito. Genere código ensamblador y examinelo para deteriminar si se crea y se llama un constructor de `X` y, si eso ocurre, qué código lo hace. Explique lo que haya descubierto. `X` no tiene constructor por defecto, entonces ¿por qué no se queja el compilador?

29. (Intermedio) Modifique el Ejercicio 28 escribiendo constructores para ambas clases de tal forma que cada constructor llame a una función virtual. Genere el código ensamblador. Determine donde se encuentra asignado el VPTR dentro del constructor. ¿El compilador está usando el mecanismo virtual dentro del constructor? Explique por qué se sigue usando la version local de la función.

30. (Avanzado) Si una función llama a un objeto pasado por valor si ligadura estática, una llamada virtual accede a partes que no existen. ¿Es posible? Escriba un código para forzar una llamada virtual y vea si se produce un cuelgue de la aplicación. Para explicar el comportamiento, observe que ocurre si se pasa un objeto por valor.

31. (Avanzado) Encuentre exactamente cuanto tiempo más es necesario para una llamada a una función virtual buscando en la información del lenguaje ensamblador de su procesador o cualquier otro manual técnico y encontrando los pulsos de reloj necesarios para una simple llamada frente al número necesario de las instrucciones de las funciones virtuales.

32. Determine el tamaño del VPTR (usando `sizeof`) en su implementación. Ahora herede de dos clases (herencia múltiple) que contengan funciones virtuales. ¿Se tiene una o dos VPTR en la clase derivada?

33. Cree una clase con datos miembros y funciones virtuales. Escriba una función que mire en la memoria de un objeto de su clase y que imprima sus distintos fragmentos. Para hacer esto será necesario experimentar y de forma iterativa descubrir donde se encuentra alojado el VPTR del objeto.

34. Imagine que las funciones virtuales no existen, y modifique `Instrument4.cpp` para que use *moldeado dinámico* para hacer el equivalente de las llamadas virtuales. Esplique porque es una mala idea.

35. Modifique `StaicHierarchyNavigation.cpp` para que en vez de usar el RTTI de C++ use su propio RTTI via una función virtual en la clase base llamada `whatAmI()` y un `enum type { Circulos, Cuadrados };`.

36. Comience con `PointerToMemberOperator.cpp` del capítulo 12 y demuestre que el polimorfismo todavía funciona con punteros a miembros, incluso si `operator->*` está sobrecargado.

16: Introducción a las Plantillas

La herencia y la composición proporcionan una forma de retilizar código objeto. Las plantillas de C++ proporcionan una manera de reutilizar el código *fuente*.

Aunque las plantillas (o templates) son una herramienta de programación de propósito general, cuando fueron introducidos en el lenguaje, parecían oponerse al uso de las jerarquías de clases contenedoras basadas en objetos (demostrado al final del Capítulo 15). Además, los contenedores y algoritmos del C++ Standard (explicados en dos capítulos del Volumen 2 de este libro, que se puede bajar de www.BruceEckel.com) están construidos exclusivamente con plantillas y son relativamente fáciles de usar por el programador.

Este capítulo no sólo muestra los fundamentos de los templates, también es una introducción a los contenedores, que son componentes fundamentales de la programación orientada a objetos lo cual se evidencia a través de los contenedores de la librería estándar de C++. Se verá que este libro ha estado usando ejemplos contenedores - `Stash` y `Stack`- para hacer más sencillo el concepto de los contenedores; en este capítulo se sumará el concepto del `iterator`. Aunque los contenedores son el ejemplo ideal para usarlos con las plantillas, en el Volumen 2 (que tiene un capítulo con plantillas avanzadas) se aprenderá que también hay otros usos para los templates.

16.1. Contenedores

Supóngase que se quiere crear una pila, como se ha estado haciendo a través de este libro. Para hacerlo sencillo, esta clase manejará enteros.

```
//: C16:IntStack.cpp
// Simple integer stack
//{L} fibonacci
#include "fibonacci.h"
#include "../require.h"
#include <iostream>
using namespace std;

class IntStack {
  enum { ssize = 100 };
  int stack[ssize];
  int top;
public:
  IntStack() : top(0) {}
  void push(int i) {
```

```
    require(top < ssize, "Too many push()es");
    stack[top++] = i;
  }
  int pop() {
    require(top > 0, "Too many pop()s");
    return stack[--top];
  }
};

int main() {
  IntStack is;
  // Add some Fibonacci numbers, for interest:
  for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
  // Pop & print them:
  for(int k = 0; k < 20; k++)
    cout << is.pop() << endl;
} ///:~
```

La clase `IntStack` es un ejemplo trivial de una pila. Para mantener la simplicidad ha sido creada con un tamaño fijo, pero se podría modificar para que automáticamente se expanda usando la memoria del montón, como en la clase `Stack` que ha sido examinada a través del libro.

`main()` añade algunos enteros a la pila, y posteriormente los extrae. Para hacer el ejemplo más interesante, los enteros son creados con la función `fibonacci()`, que genera los tradicionales números de la reproducción del conejo. Aquí está el archivo de cabecera que declara la función:

```
//: C16:fibonacci.h
// Fibonacci number generator
int fibonacci(int n); ///:~
```

Aquí está la implementación:

```
//: C16:fibonacci.cpp {O}
#include "../require.h"

int fibonacci(int n) {
  const int sz = 100;
  require(n < sz);
  static int f[sz]; // Initialized to zero
  f[0] = f[1] = 1;
  // Scan for unfilled array elements:
  int i;
  for(i = 0; i < sz; i++)
    if(f[i] == 0) break;
  while(i <= n) {
    f[i] = f[i-1] + f[i-2];
    i++;
  }
  return f[n];
} ///:~
```

Esta es una implementación bastante eficiente, porque nunca se generan los números más de una vez. Se usa un array `static` de int, y se basa en el hecho de que el compilador inicializará el array estático a cero. El primer bucle `for` mueve el índice `i` a la primera posición del array que sea cero, entonces un bucle `while` añade números Fibonacci al array hasta que se alcance el elemento deseado. Hay que hacer notar que si los números Fibonacci hasta el elemento `n` ya están inicializados, entonces también se salta el bucle `while`.

16.1.1. La necesidad de los contenedores

Obviamente, una pila de enteros no es una herramienta crucial. La necesidad real de los contenedores viene cuando se empizan a crear objetos en el montón (heap) usando `new` y se destruyen con `delete`. En un problema general de programación no se saben cuantos objetos van a ser necesarios cuando se está escribiendo el programa. Por ejemplo, en un sistema de control de tráfico aéreo no se quiere limitar el número de aviones que el sistema pueda gestionar. No puede ser que el programa se aborte sólo porque se excede algún número. En un sistema de diseño asistido por computadora, se están manejando montones de formas, pero únicamente el usuario determina (en tiempo de ejecución) cuantas formas serán necesarias. Una vez apreciemos estas tendencias, se descubrirán montones de ejemplos en otras situaciones de programación.

Los programadores de C que dependen de la memoria virtual para manejar su "gestión de memoria" encuentran a menudo como perturbantentes las ideas del `new`, `delete` y de los contenedores de clases. Aparentemente, una práctica en C es crear un enorme array global, más grande que cualquier cosa que el programa parezca necesitar. Para esto no es necesario pensar demasiado (o hay que meterse en el uso de `malloc()` y `free()`), pero se producen programas que no se pueden portar bien y que esconden sutiles errores.

Además, si se crea un enorme array global de objetos en C++, la sobrecarga de los constructores y de los destructores pueden enlentecer las cosas de forma significativa. La aproximación de C++ funciona mucho mejor: Cuando se necesite un objeto, se crea con `new`, y se pone su puntero en un contenedor. Más tarde, se saca y se hace algo con él. De esta forma, sólo se crean los objetos cuando sea necesario. Y normalmente no se dan todas las condiciones para la inicialización al principio del programa. `new` permite esperar hasta que suceda algo en el entorno para poder crear el objeto.

Así, en la situación más común, se creará un contenedor que almacene los punteros de algunos objetos de interés. Se crearán esos objetos usando `new` y se pondrá el puntero resultante en el contenedor (potencialmete haciendo upcasting en el proceso), más tarde el objeto se puede recuperar cuando sea necesario. Esta técnica produce el tipo de programas más flexible y general.

16.2. Un vistazo a las plantillas

Ahora surge un nuevo problema. Tenemos un `IntStack`, que maneja enteros. Pero queremos una pila que maneje formas, o flotas de aviones, o plantas o cualquier otra cosa. Reinventar el código fuente cada vez no parece una aproximación muy inteligente con un lenguaje que propugna la reutilización. Debe haber un camino mejor.

Hay tres técnicas para reutilizar código en esta situación: el modo de C, presen-

tado aquí como contraste; la aproximación de Smalltalk, que afectó de forma significativa a C++, y la aproximación de C++: los templates.

La solución de C. Por supuesto hay que escapar de la aproximación de C porque es desordenada y provoca errores, al mismo tiempo que no es nada elegante. En esta aproximación, se copia el código de una `Stack` y se hacen modificaciones a mano, introduciendo nuevos errores en el proceso. Esta no es una técnica muy productiva.

La solución de Smalltalk. Smalltalk (y Java siguiendo su ejemplo) optó por una solución simple y directa: Se quiere reutilizar código, pues utilicese la herencia. Para implementarlo, cada clase contenedora maneja elementos de una clase base genérica llamada `Object` (similar al ejemplo del final del capítulo 15). Pero debido a que la librería de Smalltalk es fundamental, no se puede crear una clase desde la nada. En su lugar, siempre hay que heredar de una clase existente. Se encuentra una clase lo más cercana posible a lo que se desea, se hereda de ella, y se hacen un par de cambios. Obviamente, esto es un beneficio porque minimiza el trabajo (y explica porque se pierde un montón de tiempo aprendiendo la librería antes de ser un programador efectivo en Smalltalk).

Pero también significa que todas las clases de Smalltalk acaban siendo parte de un único árbol de herencia. Hay que heredar de una rama de este árbol cuando se está creando una nueva clase. La mayoría del árbol ya esta allí (es la librería de clases de Smalltalk), y la raiz del árbol es una clase llamada `Object` - la misma clase que los contenedores de Smalltalk manejan.

Es un truco ingenioso porque significa que cada clase en la jerarquía de herencia de Smalltalk (y Java[1]) se deriva de `Object`, por lo que cualquier clase puede ser almacenada en cualquier contenedor (incluyendo a los propios contenedores). Este tipo de jerarquía de árbol única basada en un tipo genérico fundamental (a menudo llamado `Object`, como también es el caso en Java) es conocido como "jerarquía basada en objectos". Se puede haber oido este témino y asumido que es un nuevo concepto fundamental de la POO, como el polimorfismo. Sin embargo, simplemente se refiere a la raíz de la jerarquía como `Object` (o algún témino similar) y a contenedores que almacenan `Object`s.

Debido a que la librería de clases de Smalltalk tenía mucha más experiencia e historia detrás de la que tenía C++, y porque los compiladores de C++ originales no tenían librerías de clases contenedoras, parecía una buena idea duplicar la librería de Smalltalk en C++. Esto se hizo como experimento con una de las primeras implementaciónes de C++[2], y como representaba un significativo ahorro de código mucha gente empezo a usarlo. En el proceso de intentar usar las clases contenedoras, descubrieron un problema.

El problema es que en Smalltalk (y en la mayoría de los lenguajes de POO que yo conozco), todas las clases derivan automáticamente de la jerarquía única, pero esto no es cierto en C++. Se puede tener una magnifica jerarquía basada en objetos con sus clases contenedoras, pero entonces se compra un conjunto de clases de figuras, o de aviones de otro vendedor que no usa esa jerarquía. (Esto se debe a que usar una jerarquía supone sobrecarga, rechazada por los programadores de C). ¿Cómo se inserta un árbol de clases independientes en nuestra jerarquía? El problema se parece a lo siguiente:

[1] Con la excepción, en Java, de los tipos de datos primitivos, que se hicieron no `Object`s por eficiencia.
[2] La librería OOPS, por Keith Gorlen, mientras estaba en el NIH.

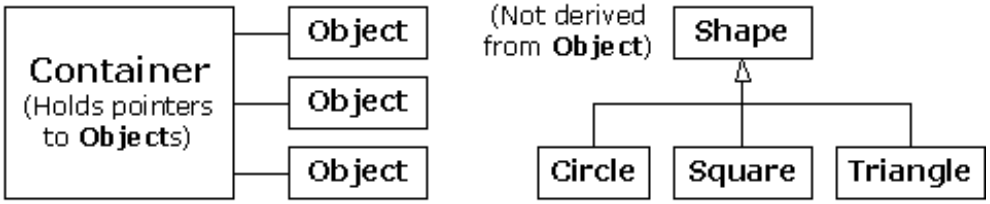

Figura 16.1: Contenedores

Debido a que C++ suporta múltiples jerarquías independientes, la jerarquía basada en objetos de Smalltalk no funciona tan bien.

La solución parace obvia. Si se pueden tener múltiples jerarquías de herencia, entonces hay que ser capaces de heredar de más de una clase: La herencia múltiple resuelve el problema. Por lo que se puede hacer lo siguiente (un ejemplo similar se dió al final del Capítulo 15).

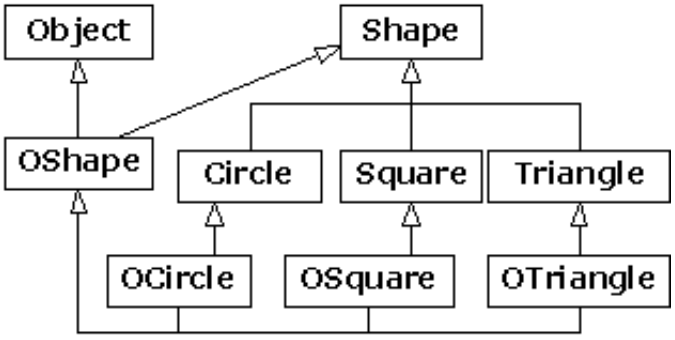

Figura 16.2: Herencia múltiple

Ahora `OShape` tiene las características y el comportamiento de `Shape`, pero como también está derivado de `Object` puede ser insertado en el contenedor. La herencia extra dada a `OCircle`, `OSquare`, etc. es necesaria para que esas clases puedan hacer upcast hacia `OShape` y puedan mantener el comportamiento correcto. Se puede ver como las cosas se están volviendo confusas rápidamente.

Los vendedores de compiladores inventaron e incluyeron sus propias jerarquías y clases contenedoras, muchas de las cuales han sido reemplazadas desde entonces por versiones de templates. Se puede argumentar que la herencia múltiple es necesaria para resolver problemas de programación general, pero como se verá en el Volumen 2 de este libro es mejor evitar esta complejidad excepto en casos especiales.

16.2.1. La solución de la plantilla

Aunque una jerarquía basada en objetos con herencia múltiple es conceptualmente correcta, se vuelve difícil de usar. En su libro[3], Stroustrup demostró lo que el consideraba una alternativa preferible a la jerarquía basada en objetos. Clases contenedoras que fueran creadas como grandes macros del preprocesador con argumentos que pudieran ser sustituidos con el tipo deseado. Cuando se quiera crear un

[3] *The C++ Programming Language* by Bjarne Stroustrup (1ª edición, Addison-Wesley, 1986)

contenedor que maneje un tipo en concreto, se hacen un par de llamadas a macros.

Desafortunadamente, esta aproximación era confusa para toda la literatura existente de Smalltalk y para la experiencia de programación, y era un poco inmanejable. Básicamente, nadie la entendía.

Mientras tanto, Stroustrup y el equipo de C++ de los Laboratorios Bell habían modificado su aproximación de las macros, simplificándola y moviéndola del dominio del preprocesador al compilador. Este nuevo dispositivo de sustitución de código se conoce como `template` [4] (plantilla), y representa un modo completamente diferente de reutilizar el código. En vez de reutilizar código objeto, como en la herencia y en la composición, un template reutiliza *código fuente*. El contenedor no maneja una clase base genérica llamada `Object`, si no que gestiona un parámetro no especificado. Cuando se usa un template, el parámetro es sustituido *por el compilador*, parecido a la antigua aproximación de las macros, pero más claro y fácil de usar.

Ahora, en vez de preocuparse por la herencia o la composición cuando se quiera usar una clase contenedora, se usa la versión en plantilla del contenedor y se crea una versión específica para el problema, como lo siguiente:

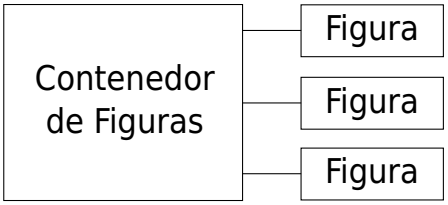

Figura 16.3: Contenedor de objetos `Figura`

El compilador hace el trabajo por nosotros, y se obtiene el contenedor necesario para hacer el trabajo, en vez de una jerarquía de herencia inmanejable. En C++, el template implementa el concepto de *tipo parametrizado*. Otro beneficio de la aproximación de las plantillas es que el programador novato que no tenga familiaridad o esté incómodo con la herencia puede usar las clases contenedoras de manera adecuada (como se ha estado haciendo a lo largo del libro con el `vector`).

16.3. Sintaxis del Template

La palabra reservada `template` le dice al compilador que la definición de clases que sigue manipulará uno o más tipos no especificados. En el momento en que el código de la clase actual es generado, los tipos deben ser especificados para que el compilador pueda sustituirlos.

Para demostrar la sintaxis, aquí está un pequeño ejemplo que produce un array con límites comprobados:

```
//: C16:Array.cpp
#include "../require.h"
#include <iostream>
using namespace std;

template<class T>
```

[4] La inspiración de los templates parece venir de los `generics` de ADA

```
class Array {
  enum { size = 100 };
  T A[size];
public:
  T& operator[](int index) {
    require(index >= 0 && index < size,
      "Index out of range");
    return A[index];
  }
};

int main() {
  Array<int> ia;
  Array<float> fa;
  for(int i = 0; i < 20; i++) {
    ia[i] = i * i;
    fa[i] = float(i) * 1.414;
  }
  for(int j = 0; j < 20; j++)
    cout << j << ": " << ia[j]
         << ", " << fa[j] << endl;
} ///:~
```

Se puede ver que parece una clase normal excepto por la línea.

```
    template<class T>
```

que indica que `T` es un parámetro de sustitución, y que representa un nombre de un tipo. Además, se puede ver que `T` es usado en todas las partes de la clase donde normalmente se vería al tipo específico que el contenedor gestiona.

En `Array` los elementos son insertados y extraidos con la misma función: el operador sobrecargado `operator[]`. Devuelve una referencia, por lo que puede ser usado en ambos lados del signo igual (es decir, tanto como `lvalue` como `rvalue`). Hay que hacer notar que si el índice se sale de los límites se usa la función `require()` para mostrar un mensaje. Como `operator[]` es `inline`, se puede usar esta aproximación para garantizar que no se producen violaciones del límite del array para entonces eliminar el `require()`.

En el `main()`, se puede ver lo fácil que es crear `Array`s que manejen distintos tipos de objetos. Cuando se dice:

```
    Array<int> ia;
    Array<float> fa;
```

el compilador expande dos veces la plantilla del `Array` (que se conoce como `instantiation` o crear una instancia), para crear dos nuevas *clases generadas*, las cuales pueden ser interpretadas como `Array_int` y `Array_float`. Diferentes compiladores pueden crear los nombres de diferentes maneras. Estas clases son idénticas a las que hubieran producido de estar hechas a mano, excepto que el compilador las crea por nosotros cuando se definen los objetos `ia` y `fa`. También hay que notar que las definiciones de clases duplicadas son eludidas por el compilador.

16.3.1. Definiciones de función no inline

Por supuesto, hay veces en las que se querrá tener definición de funciones no inline. En ese caso, el compilador necesita ver la declaración del `template` antes que la definición de la función miembro. Aquí está el ejemplo anterior, modificado para mostrar la definición del miembro no inline.

```
//: C16:Array2.cpp
// Non-inline template definition
#include "../require.h"

template<class T>
class Array {
  enum { size = 100 };
  T A[size];
public:
  T& operator[](int index);
};

template<class T>
T& Array<T>::operator[](int index) {
  require(index >= 0 && index < size,
    "Index out of range");
  return A[index];
}

int main() {
  Array<float> fa;
  fa[0] = 1.414;
} ///:~
```

Cualquier referencia al nombre de una plantilla de clase debe estar acompañado por la lista de argumentos del template, como en `Array<T>operator[]`. Se puede imaginar que internamente, el nombre de la clase se rellena con los argumentos de la lista de argumentos de la plantilla para producir un nombre identificador único de la clase for cada instanciación de la plantilla.

Archivos cabecera

Incluso si se crean definiciones de funciones no inline, normalmente se querrá poner todas las declaraciones y definiciones de un template en un archivo cabecera. Esto parece violar la regla usual de los archivos cabecera de «No poner nada que asigne almacenamiento», (lo cual previene múltiples errores de definición en tiempo de enlace), pero las definiciones de plantillas son especial. Algo precedido por `template<...>` significa que el compilador no asignará almacenamiento en ese momento, sino que se esperará hasta que se lo indiquen (en la instanciación de una plantilla), y que en algún lugar del compilador y del enlazador hay un mecanismo para eliminar las múltiples definiciones de una plantilla idéntica. Por lo tanto casi siempre se pondrá toda la declaración y definición de la plantilla en el archivo cabecera por facilidad de uso.

Hay veces en las que puede ser necesario poner las definiciones de la plantilla en un archivo `cpp` separado para satisfacer necesidades especiales (por ejemplo, forzar las instanciaciones de las plantillas para que se encuentren en un único archivo `dll` de Windows). La mayoría de los compiladores tienen algún mecanismo para

permitir esto; hay que investigar la documentación del compilador concreto para usarlo.

Algunas personas sienten que poner el código fuente de la implementación en un archivo cabecera hace posible que se pueda robar y modificar el código si se compra la librería. Esto puede ser una característica, pero probablemente dependa del modo de mirar el problema: ¿Se está comprando un producto o un servicio? Si es un producto, entonces hay que hacer todo lo posible por protegerlo, y probablemente no se quiera dar el código fuente, sino sólo el código compilado. Pero mucha gente ve el software como un servicio, incluso más, como un servicio por suscripción. El cliente quiere nuestra pericia, quieren que se mantenga ese fragmento de código reutilizable para no tenerlo que hacer él - para que se pueda enfocar en hacer su propio trabajo. Personalmente creo que la mayoría de los clientes le tratarán como una fuente de recursos a tener en cuenta y no querrán poner en peligro su relación con usted. Y para los pocos que quieran robar en vez de comprar o hacer el trabajo original, de todas formas probablemante tampoco se mantendrían con usted.

16.3.2. IntStack como plantilla

Aquí está el contenedor y el iterador de `IntStack.cpp`, implementado como una clase contenedora genérica usando plantillas:

```
//: C16:StackTemplate.h
// Simple stack template
#ifndef STACKTEMPLATE_H
#define STACKTEMPLATE_H
#include "../require.h"

template<class T>
class StackTemplate {
  enum { ssize = 100 };
  T stack[ssize];
  int top;
public:
  StackTemplate() : top(0) {}
  void push(const T& i) {
    require(top < ssize, "Too many push()es");
    stack[top++] = i;
  }
  T pop() {
    require(top > 0, "Too many pop()s");
    return stack[--top];
  }
  int size() { return top; }
};
#endif // STACKTEMPLATE_H ///:~
```

Hay que darse cuenta que esta plantilla asume ciertas características de los objetos que está manejando. Por ejemplo, `StackTemplate` asume que hay alguna clase de operación de asignación a `T` dentro de la función `push()`. Se puede decir que una plantilla «implica una interfaz» para los tipos que es capaz de manejar.

Otra forma de decir esto es que las plantillas proporcionan una clase de mecanismo de *tipado débil* en C++, lo cual es típico en un lenguaje fuertemente tipado. En vez de insistir en que un objeto sea del mismo tipo para que sea aceptable, el tipado

débil requiere únicamente que la función miembro a la que se quiere llamar esté *disponible* para un objeto en particular. Es decir, el código débilmente tipado puede ser aplicado a cualquier objeto que acepte esas llamadas a funciones miembro, lo que lo hace mucho más flexible[5].

Aquí tenemos el objeto revisado para comprobar la plantilla:

```
//: C16:StackTemplateTest.cpp
// Test simple stack template
//{L} fibonacci
#include "fibonacci.h"
#include "StackTemplate.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  StackTemplate<int> is;
  for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
  for(int k = 0; k < 20; k++)
    cout << is.pop() << endl;
  ifstream in("StackTemplateTest.cpp");
  assure(in, "StackTemplateTest.cpp");
  string line;
  StackTemplate<string> strings;
  while(getline(in, line))
    strings.push(line);
  while(strings.size() > 0)
    cout << strings.pop() << endl;
} ///:~
```

La única diferencia está en la creación de `is`. Dentro de la lista de argumentos del template hay que especificar el tipo de objeto que la pila y el iterador deberán manejar. Para demostrar la genericidad de la plantilla, se crea un `StackTemplate` para manejar string. El ejemplo lee las líneas del archivo con el código fuente.

16.3.3. Constantes en los Templates

Los argumentos de los templates no restrigen su uso a tipos class; se pueden también usar tipos empotrados. Los valores de estos argumentos se convierten en constantes en tiempo de compilación para una instanciación en particular de la plantilla. Se pueden usar incluso valores por defecto para esos argumentos. El siguiente ejemplo nos permite indicar el tamaño de la clase `Array` durante la instanciación, pero también proporciona un valor por defecto:

```
//: C16:Array3.cpp
// Built-in types as template arguments
#include "../require.h"
#include <iostream>
using namespace std;
```

[5] Todos los métodos en Smalltalk y Python están débilmente tipados, y ese es el motivo por lo que estos lenguajes no necesitan el mecanismo de los templates. En efecto, se consiguen plantillas sin templates.

```
template<class T, int size = 100>
class Array {
  T array[size];
public:
  T& operator[](int index) {
    require(index >= 0 && index < size,
      "Index out of range");
    return array[index];
  }
  int length() const { return size; }
};

class Number {
  float f;
public:
  Number(float ff = 0.0f) : f(ff) {}
  Number& operator=(const Number& n) {
    f = n.f;
    return *this;
  }
  operator float() const { return f; }
  friend ostream&
    operator<<(ostream& os, const Number& x) {
      return os << x.f;
  }
};

template<class T, int size = 20>
class Holder {
  Array<T, size>* np;
public:
  Holder() : np(0) {}
  T& operator[](int i) {
    require(0 <= i && i < size);
    if(!np) np = new Array<T, size>;
    return np->operator[](i);
  }
  int length() const { return size; }
  ~Holder() { delete np; }
};

int main() {
  Holder<Number> h;
  for(int i = 0; i < 20; i++)
    h[i] = i;
  for(int j = 0; j < 20; j++)
    cout << h[j] << endl;
} ///:~
```

Como antes, `Array` es un array de objetos que previene de rebasar los límites. La clase `Holder` es muy parecida a `Array` excepto que tiene un puntero a `Array` en vez de un tener incrustrado un objeto del tipo `Array`. Este puntero no se inicializa en el constructor; la inicialización es retrasada hasta el primer acceso. Esto se conoce como *inicialización perezosa*; se puede usar una técnica como esta si se están creando un montón de objetos, pero no se está accediendo a todos ellos y se quiere ahorrar

almacenamiento.

Hay que resaltar que nunca se almacena internamente el valor de `size` en la clase, pero se usa como si fuera un dato interno dentro de las funciones miembro.

16.4. Stack y Stash como Plantillas

Los problemas recurrentes de «propiedad» con las clases contenedoras `Stack` y `Stash` (Pila y Cola respectivamente) que han sido usadas varias veces a través del libro, vienen del hecho de que estos contenedores no son capaces de saber exactamente que tipo manejan. Lo más cerca que han estado es en el «contenedor» de objectos `Stack` que se vio al final del capítulo 15 en `OStackTest.cpp`.

Si el programador cliente no elimina explícitamente todos los punteros a objeto que están almacenados en el contenedor, entonces el contenedor debería ser capaz de eliminar esos punteros de manera adecuada. Es decir, el contenedor «posee» cualquiera de los objetos que no hayan sido eliminados, y es el responsable de limpiarlos. La dificultad radica en que el limpiado requiere conocer el tipo del objeto, y crear un contenedor genérico *no* requiere conocer el tipo de ese objeto. Con los templates, sin embargo, podemos escribir código que no conozcan el tipo de objeto, y fácilmente instanciar una nueva versión del contenedor por cada tipo que queramos que contenga. La instancia contenedora individual *conoce* el tipo de objetos que maneja y puede por tanto llamar al destructor correcto (asumiendo que se haya proporcionado un destructor virtual).

Para la pila es bastante sencillo debido a todas las funciones miembro pueden ser introducidas en línea:

```
//: C16:TStack.h
// The Stack as a template
#ifndef TSTACK_H
#define TSTACK_H

template<class T>
class Stack {
  struct Link {
    T* data;
    Link* next;
    Link(T* dat, Link* nxt):
      data(dat), next(nxt) {}
  }* head;
public:
  Stack() : head(0) {}
  ~Stack(){
    while(head)
      delete pop();
  }
  void push(T* dat) {
    head = new Link(dat, head);
  }
  T* peek() const {
    return head ? head->data : 0;
  }
  T* pop(){
    if(head == 0) return 0;
    T* result = head->data;
```

```
    Link* oldHead = head;
    head = head->next;
    delete oldHead;
    return result;
  }
};
#endif // TSTACK_H ///:~
```

Si se compara esto al ejemplo de `OStack.h` al final del capítulo 15, se verá que `Stack` es virtualmente idéntica, excepto que `Object` ha sido reemplazado con *`T`*. El programa de prueba también es casi idéntico, excepto por la necesidad de múltiple herencia de string y `Object` (incluso por la necesidad de `Object` en sí mismo) que ha sido eliminada. Ahora no tenemos una clase `MyString` para anunciar su destrucción por lo que añadimos una pequeña clase nueva para mostrar como la clase contenedora `Stack` limpia sus objetos:

```
//: C16:TStackTest.cpp
//{T} TStackTest.cpp
#include "TStack.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

class X {
public:
  virtual ~X() { cout << "~X " << endl; }
};

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // File name is argument
  ifstream in(argv[1]);
  assure(in, argv[1]);
  Stack<string> textlines;
  string line;
  // Read file and store lines in the Stack:
  while(getline(in, line))
    textlines.push(new string(line));
  // Pop some lines from the stack:
  string* s;
  for(int i = 0; i < 10; i++) {
    if((s = (string*)textlines.pop())==0) break;
    cout << *s << endl;
    delete s;
  } // The destructor deletes the other strings.
  // Show that correct destruction happens:
  Stack<X> xx;
  for(int j = 0; j < 10; j++)
    xx.push(new X);
} ///:~
```

El destructor de `X` es virtual, no porque se sea necesario aquí, sino porque `xx` podría ser usado más tarde para manejar objetos derivados de `X`.

Note lo fácil que es crear diferentes clases de `Stack`s para string y para `X`. Debido a la plantilla, se consigue lo mejor de los dos mundos: la facilidad de uso de la `Stack` junto con un limpiado correcto.

16.4.1. Cola de punteros mediante plantillas

Reorganizar el código de `PStash` en un template no es tan simple porque hay un número de funciones miembro que no deben estar en línea. Sin embargo, como buena plantilla aquellas definiciones de función deben permanecer en el archivo cabecera (el compilador y el enlazador se preocuparán por los problemas de múltiples definiciones). El código parece bastante similar al `PStash` ordinario excepto que el tamaño del incremento (usado por `inflate()`) ha sido puesto en el template como un parámetro no de clase con un valor por defecto, para que el tamaño de incremento pueda ser modificado en el momento de la instanciación (esto significa que el tamaño es fijo aunque se podría argumentar que el tamaño de incremento debería ser cambiable a lo largo de la vida del objeto):

```
//: C16:TPStash.h
#ifndef TPSTASH_H
#define TPSTASH_H

template<class T, int incr = 10>
class PStash {
  int quantity; // Number of storage spaces
  int next; // Next empty space
  T** storage;
  void inflate(int increase = incr);
public:
  PStash() : quantity(0), next(0), storage(0) {}
  ~PStash();
  int add(T* element);
  T* operator[](int index) const; // Fetch
  // Remove the reference from this PStash:
  T* remove(int index);
  // Number of elements in Stash:
  int count() const { return next; }
};

template<class T, int incr>
int PStash<T, incr>::add(T* element) {
  if(next >= quantity)
    inflate(incr);
  storage[next++] = element;
  return(next - 1); // Index number
}

// Ownership of remaining pointers:
template<class T, int incr>
PStash<T, incr>::~PStash() {
  for(int i = 0; i < next; i++) {
    delete storage[i]; // Null pointers OK
    storage[i] = 0; // Just to be safe
  }
  delete []storage;
}
```

```
template<class T, int incr>
T* PStash<T, incr>::operator[](int index) const {
  require(index >= 0,
    "PStash::operator[] index negative");
  if(index >= next)
    return 0; // To indicate the end
  require(storage[index] != 0,
    "PStash::operator[] returned null pointer");
  // Produce pointer to desired element:
  return storage[index];
}

template<class T, int incr>
T* PStash<T, incr>::remove(int index) {
  // operator[] performs validity checks:
  T* v = operator[](index);
  // "Remove" the pointer:
  if(v != 0) storage[index] = 0;
  return v;
}

template<class T, int incr>
void PStash<T, incr>::inflate(int increase) {
  const int psz = sizeof(T*);
  T** st = new T*[quantity + increase];
  memset(st, 0, (quantity + increase) * psz);
  memcpy(st, storage, quantity * psz);
  quantity += increase;
  delete []storage; // Old storage
  storage = st; // Point to new memory
}
#endif // TPSTASH_H ///:~
```

El tamaño del incremento por defecto es muy pequeño para garantizar que se produzca la llamada a `inflate()`. Esto nos asegura que funcione correctamente.

Para comprobar el control de propiedad de `PStack` en template, la siguiente clase muestra informes de creación y destrucción de elementos, y también garantiza que todos los objetos que hayan sido creados sean destruidos. `AutoCounter` permitirá crear objetos en la pila sólo a los objetos de su tipo:

```
//: C16:AutoCounter.h
#ifndef AUTOCOUNTER_H
#define AUTOCOUNTER_H
#include "../require.h"
#include <iostream>
#include <set> // Standard C++ Library container
#include <string>

class AutoCounter {
  static int count;
  int id;
  class CleanupCheck {
    std::set<AutoCounter*> trace;
  public:
    void add(AutoCounter* ap) {
      trace.insert(ap);
```

```
    }
    void remove(AutoCounter* ap) {
      require(trace.erase(ap) == 1,
        "Attempt to delete AutoCounter twice");
    }
    ~CleanupCheck() {
      std::cout << "~CleanupCheck()"<< std::endl;
      require(trace.size() == 0,
       "All AutoCounter objects not cleaned up");
    }
  };
  static CleanupCheck verifier;
  AutoCounter() : id(count++) {
    verifier.add(this); // Register itself
    std::cout << "created[" << id << "]"
              << std::endl;
  }
  // Prevent assignment and copy-construction:
  AutoCounter(const AutoCounter&);
  void operator=(const AutoCounter&);
public:
  // You can only create objects with this:
  static AutoCounter* create() {
    return new AutoCounter();
  }
  ~AutoCounter() {
    std::cout << "destroying[" << id
              << "]" << std::endl;
    verifier.remove(this);
  }
  // Print both objects and pointers:
  friend std::ostream& operator<<(
    std::ostream& os, const AutoCounter& ac){
    return os << "AutoCounter " << ac.id;
  }
  friend std::ostream& operator<<(
    std::ostream& os, const AutoCounter* ac){
    return os << "AutoCounter " << ac->id;
  }
};
#endif // AUTOCOUNTER_H ///:~
```

La clase `AutoCounter` hace dos cosas. Primero, numera cada instancia de `AutoCounter` de forma secuencial: el valor de este número se guarda en `id`, y el número se genera usando el dato miembro `count` que es `static`.

Segundo, y más complejo, una instancia estática (llamada `verifier`) de la clase `CleanupCheck` se mantiene al tanto de todos los objetos `AutoCounter` que son creados y destruidos, y nos informa si no se han limpiado todos (por ejemplo si existe un agujero en memoria). Este comportamiento se completa con el uso de la clase `set` de la Librería Estándar de C++, lo cual es un magnífico ejemplo de cómo las plantillas bien diseñadas nos pueden hacer la vida más fácil (se podrá aprender más de los contenedores en el Volumen 2 de este libro).

La clase `set` está instanciada para el tipo que maneja; aquí hay una instancia que maneja punteros a `AutoCounter`. Un `set` permite que se inserte sólo una instancia de cada objeto; en `add()` se puede ver que esto sucede con la función

`set::insert().insert()` nos informa con su valor de retorno si se está intentando añadir algo que ya se había incluido; sin embargo, desde el momento en que las direcciones a objetos se inserten podemos confiar en C++ para que garantice que todos los objetos tengan direcciones únicas.

En `remove()`, se usa `set::erase()` para eliminar un puntero a `AutoCounter` del `set`. El valor de retorno indica cuantas instancias del elemento se han eliminado; en nuestro caso el valor puede ser únicamente uno o cero. Si el valor es cero, sin embargo, significa que el objeto ya había sido borrado del conjunto y que se está intentando borrar por segunda vez, lo cual es un error de programación que debe ser mostrado mediante `require()`.

El destructor de `CleanupCheck` hace una comprobación final asegurándose de que el tamaño del `set` es cero - Lo que significa que todos los objetos han sido eliminados de manera adecuada. Si no es cero, se tiene un agujero de memoria, lo cual se muestra mediante el `require()`.

El constructor y el destructor de `AutoCounter` se registra y desregistra con el objeto `verifier`. Hay que resaltar que el constructor, el constructor de copia, y el operador de asignación son `private`, por lo que la única forma de crear un objeto es con la función miembro `static create()` - esto es un ejemplo sencillo de una `factory`, y garantiza que todos los objetos sean creados en el montón (heap), por lo que `verifier` no se verá confundido con sobreasignaciones y construcciones de copia.

Como todas las funciones miembro han sido definidas inline, la única razón para el archivo de implementación es que contenga las definiciones de los datos miembro:

```
//: C16:AutoCounter.cpp {O}
// Definition of static class members
#include "AutoCounter.h"
AutoCounter::CleanupCheck AutoCounter::verifier;
int AutoCounter::count = 0;
///:~
```

Con el `AutoCounter` en la mano, podemos comprobar las facilidades que proporciona el `PStash`. El siguiente ejemplo no sólo muestra que el destructor de `PStash` limpia todos los objetos que posee, sino que también muestra como la clase `AutoCounter` detecta a los objetos que no se han limpiado.

```
//: C16:TPStashTest.cpp
//{L} AutoCounter
#include "AutoCounter.h"
#include "TPStash.h"
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  PStash<AutoCounter> acStash;
  for(int i = 0; i < 10; i++)
    acStash.add(AutoCounter::create());
  cout << "Removing 5 manually:" << endl;
  for(int j = 0; j < 5; j++)
    delete acStash.remove(j);
  cout << "Remove two without deleting them:"
```

```
        << endl;
  // ... to generate the cleanup error message.
  cout << acStash.remove(5) << endl;
  cout << acStash.remove(6) << endl;
  cout << "The destructor cleans up the rest:"
       << endl;
  // Repeat the test from earlier chapters:
  ifstream in("TPStashTest.cpp");
  assure(in, "TPStashTest.cpp");
  PStash<string> stringStash;
  string line;
  while(getline(in, line))
    stringStash.add(new string(line));
  // Print out the strings:
  for(int u = 0; stringStash[u]; u++)
    cout << "stringStash[" << u << "] = "
         << *stringStash[u] << endl;
} ///:~
```

Cuando se eliminan los elementos `AutoCounter` 5 y 6 de la `PStash`, se vuelve responsabilidad del que los llama, pero como el cliente nunca los borra se podrín producir agujeros de memoria, que serín detectados por `AutoCounter` en tiempo de ejecución.

Cuando se ejecuta el programa, se verá que el mensaje de error no es tan específico como podría ser. Si se usa el esquema presentado en `AutoCounter` para descubrir agujeros de memoria en nuestro sistema, probablemente se quiera imprimir información más detallada sobre los objetos que no se hayan limpiado. El Volumen 2 de este libro muestra algunas formas más sofisticadas de hacer esto.

16.5. Activando y desactivando la propiedad

Volvamos al problema del propietario. Los contenedores que manejan objetos por valor normalmente no se preocupan por la propiedad porque claramente poseen los objetos que contienen. Pero si el contenedor gestiona punteros (lo cual es común en C++, especialmente con el polimorfismo), entonces es bastante probable que esos punteros sean usados en algún otro lado del programa, y no necesariamente se quiere borrar el objeto porque los otros punteros del programa estarán referenciando a un objeto destruido. Para prevenir que esto ocurra, hay que considerar al propietario cuando se está diseñando y usando un contenedor.

Muchos programas son más simples que este, y no se encuentran con el problema de la propiedad: Un contenedor que maneja punteros a objetos y que son usados sólo por ese contenedor. En este caso el propietario es evidente: El contenedor posee sus objetos.

La mejor aproximación para gestionar quién es el propietario es dar al programador cliente una elección. Esto se puede realizar con un argumento en el constructor que por defecto defina al propietario (el caso más sencillo). Además habrá que poner las funciones «get» y «set» para poder ver y modificar al propietario del contenedor. Si el contenedor tiene funciones para eliminar un objeto, el estado de propiedad normalmente afecta a la función de eliminación, por lo que se deberían encontrar opciones para controlar la destrucción en la función de eliminación. Es concebible que se añadan datos propietarios por cada elemento que contenga el contenedor, por lo

que cada posición debería saber cuando es necesario ser destruido; esto es una variante del conteo de referencias, excepto en que es el contenedor y no el objeto el que conoce el número de referencias a un objeto.

```
//: C16:OwnerStack.h
// Stack with runtime conrollable ownership
#ifndef OWNERSTACK_H
#define OWNERSTACK_H

template<class T> class Stack {
  struct Link {
    T* data;
    Link* next;
    Link(T* dat, Link* nxt)
      : data(dat), next(nxt) {}
  }* head;
  bool own;
public:
  Stack(bool own = true) : head(0), own(own) {}
  ~Stack();
  void push(T* dat) {
    head = new Link(dat,head);
  }
  T* peek() const {
    return head ? head->data : 0;
  }
  T* pop();
  bool owns() const { return own; }
  void owns(bool newownership) {
    own = newownership;
  }
  // Auto-type conversion: true if not empty:
  operator bool() const { return head != 0; }
};

template<class T> T* Stack<T>::pop() {
  if(head == 0) return 0;
  T* result = head->data;
  Link* oldHead = head;
  head = head->next;
  delete oldHead;
  return result;
}

template<class T> Stack<T>::~Stack() {
  if(!own) return;
  while(head)
    delete pop();
}
#endif // OWNERSTACK_H ///:~
```

El comportamiento por defecto del contenedor consiste en destruir sus objetos pero se puede cambiar o modificando el argumento del constructor o usando las funciones miembro de `owns()`.

Como con la mayoría de las plantillas que se verán, la implementación entera se

encuentra en el archivo de cabecera. Aquí tenemos un pequeño test que muestra las capacidades de la propiedad:

```
//: C16:OwnerStackTest.cpp
//{L} AutoCounter
#include "AutoCounter.h"
#include "OwnerStack.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  Stack<AutoCounter> ac; // Ownership on
  Stack<AutoCounter> ac2(false); // Turn it off
  AutoCounter* ap;
  for(int i = 0; i < 10; i++) {
    ap = AutoCounter::create();
    ac.push(ap);
    if(i % 2 == 0)
      ac2.push(ap);
  }
  while(ac2)
    cout << ac2.pop() << endl;
  // No destruction necessary since
  // ac "owns" all the objects
} ///:~
```

El objeto `ac2` no posee los objetos que pusimos en él, sin embargo `ac` es un contenedor «maestro» que tiene la responsabilidad de ser el propietario de los objetos. Si en algún momento de la vida de un contenedor se quiere cambiar el que un contenedor posea a sus objetos, se puede hacer usando `owns()`.

También sería posible cambiar la granularidad de la propiedad para que estuviera en la base, es decir, objeto por objeto. Esto, sin embargo, probablemente haría a la solución del problema del propietario más complejo que el propio problema.

16.6. Manejando objetos por valor

Actualmente crear una copia de los objetos dentro de un contenedor genérico sería un problema complejo si no se tuvieran plantillas. Con los templates las cosas se vuelven relativamente sencillas - sólo hay que indicar que se están manejando objetos en vez de punteros:

```
//: C16:ValueStack.h
// Holding objects by value in a Stack
#ifndef VALUESTACK_H
#define VALUESTACK_H
#include "../require.h"

template<class T, int ssize = 100>
class Stack {
  // Default constructor performs object
```

```
  // initialization for each element in array:
  T stack[ssize];
  int top;
public:
  Stack() : top(0) {}
  // Copy-constructor copies object into array:
  void push(const T& x) {
    require(top < ssize, "Too many push()es");
    stack[top++] = x;
  }
  T peek() const { return stack[top]; }
  // Object still exists when you pop it;
  // it just isn't available anymore:
  T pop() {
    require(top > 0, "Too many pop()s");
    return stack[--top];
  }
};
#endif // VALUESTACK_H ///:~
```

El constructor de copia de los objetos contenidos hacen la mayoría del trabajo pasando y devolviendo objetos por valor. Dentro de `push()`, el almacenamiento del objeto en el array `Stack` viene acompañado con `T::operator=`. Para garantizar que funciona, una clase llamada `SelfCounter` mantiene una lista de las creaciones y construcciones de copia de los objetos.

```
//: C16:SelfCounter.h
#ifndef SELFCOUNTER_H
#define SELFCOUNTER_H
#include "ValueStack.h"
#include <iostream>

class SelfCounter {
  static int counter;
  int id;
public:
  SelfCounter() : id(counter++) {
    std::cout << "Created: " << id << std::endl;
  }
  SelfCounter(const SelfCounter& rv) : id(rv.id){
    std::cout << "Copied: " << id << std::endl;
  }
  SelfCounter operator=(const SelfCounter& rv) {
    std::cout << "Assigned " << rv.id << " to "
              << id << std::endl;
    return *this;
  }
  ~SelfCounter() {
    std::cout << "Destroyed: "<< id << std::endl;
  }
  friend std::ostream& operator<<(
    std::ostream& os, const SelfCounter& sc){
    return os << "SelfCounter: " << sc.id;
  }
};
#endif // SELFCOUNTER_H ///:~
```

```
//: C16:SelfCounter.cpp {O}
#include "SelfCounter.h"
int SelfCounter::counter = 0; ///:~
```

```
//: C16:ValueStackTest.cpp
//{L} SelfCounter
#include "ValueStack.h"
#include "SelfCounter.h"
#include <iostream>
using namespace std;

int main() {
  Stack<SelfCounter> sc;
  for(int i = 0; i < 10; i++)
    sc.push(SelfCounter());
  // OK to peek(), result is a temporary:
  cout << sc.peek() << endl;
  for(int k = 0; k < 10; k++)
    cout << sc.pop() << endl;
} ///:~
```

Cuando se crea un contenedor `Stack`, el constructor por defecto del objeto a contener es ejecutado por cada objeto en el array. Inicialmente se verán 100 objetos `SelfCounter` creados sin ningún motivo aparente, pero esto es justamente la inicialización del array. Esto puede resultar un poco caro, pero no existe ningún problema en un diseño simple como este. Incluso en situaciones más complejas si se hace a `Stack` más general permitiendo que crezca dinámicamente, porque en la implementación mostrada anteriormente esto implicaría crear un nuevo array más grande, copiando el anterior al nuevo y destruyendo el antiguo array (de hecho, así es como lo hace la clase `vector` de la Librería Estándar de C++).

16.7. Introducción a los iteradores

Un `iterator` es un objeto que se mueve a través de un contenedor de otros objetos y selecciona a uno de ellos cada vez, sin porporcionar un acceso directo a la implementación del contenedor. Los iteradores proporcionan una forma estándar de acceder a los elementos, sin importar si un contenedor proporciona alguna marnera de acceder a los elementos directamente. Se verán a los iteradores usados frecuentemente en asociación con clases contenedoras, y los iteradores son un concepto fundamental en el diseño y el uso de los contenedores del Standard C++, los cuales son descritos en el Volumen 2 de este libro (que se puede bajar de www.BruceEckel.com. Un iterador es también un tipo de *patrón de diseño*, lo cual es materia de un capítulo del Volumen 2.

En muchos sentidos, un iterador es un «puntero elegante», y de hecho se verá que los iteradores normalmente ocultan la mayoría de las operaciones de los punteros. Sin embargo, al contrario que un puntero, el iterador es diseñado para ser seguro por lo que es mucho menos probable de hacer el equivalente de avanzar atravesando el

final de un array (o si se hace, se encontrará más fácilmente).

Considere el primer ejemplo de este capítulo. Aquí está pero añadiendo un iterador sencillo:

```
//: C16:IterIntStack.cpp
// Simple integer stack with iterators
//{L} fibonacci
#include "fibonacci.h"
#include "../require.h"
#include <iostream>
using namespace std;

class IntStack {
  enum { ssize = 100 };
  int stack[ssize];
  int top;
public:
  IntStack() : top(0) {}
  void push(int i) {
    require(top < ssize, "Too many push()es");
    stack[top++] = i;
  }
  int pop() {
    require(top > 0, "Too many pop()s");
    return stack[--top];
  }
  friend class IntStackIter;
};

// An iterator is like a "smart" pointer:
class IntStackIter {
  IntStack& s;
  int index;
public:
  IntStackIter(IntStack& is) : s(is), index(0) {}
  int operator++() { // Prefix
    require(index < s.top,
      "iterator moved out of range");
    return s.stack[++index];
  }
  int operator++(int) { // Postfix
    require(index < s.top,
      "iterator moved out of range");
    return s.stack[index++];
  }
};

int main() {
  IntStack is;
  for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
  // Traverse with an iterator:
  IntStackIter it(is);
  for(int j = 0; j < 20; j++)
    cout << it++ << endl;
} ///:~
```

El `IntStackIter` ha sido creado para trabajar solo con un `IntStack`. Hay que resaltar que `IntStackIter` es un `friend` de `IntStack`, lo que lo da un acceso a todos los elementos privados de `IntStack`.

Como un puntero, el trabajo de `IntStackIter` consiste en moverse a través de un `IntStack` y devolver valores. En este sencillo ejemplo, el objeto `IntStackIter` se puede mover sólo hacia adelante (usando la forma prefija y sufija del `operador++`). Sin embargo, no hay límites de la forma en que se puede definir un iterador a parte de las restricciones impuestas por el contenedor con el que trabaje. Esto es totalmente aceptable (incluido los límites del contenedor que se encuentre por debajo) para un iterador que se mueva de cualquier forma por su contenedor asociado y para que se puedan modificar los valores del contenedor.

Es usual el que un iterador sea creado con un constructor que lo asocie a un único objeto contenedor, y que ese iterador no pueda ser asociado a otro contenedor diferente durante su ciclo de vida. (Los iteradores son normalemente pequeños y baratos, por lo que se puede crear otro fácilmente).

Con el iterador, se puede atravesar los elementos de la pila sin sacarlos de ella, como un puntero se mueve a través de los elementos del array. Sin embargo, el iterador conoce la estructura interna de la pila y como atravesar los elementos, dando la sensación de que se está moviendo a través de ellos como si fuera «incrementar un puntero», aunque sea más complejo lo que pasa por debajo. Esta es la clave del iterador: Abstrae el proceso complicado de moverse de un elemento del contenedor al siguiente y lo convierte en algo parecido a un puntero. La meta de cada iterador del programa es que tengan la misma interfaz para que cualquier código que use un iterador no se preocupe de a qué está apuntando - sólo se sabe que todos los iteradores se tratan de la misma manera, por lo que no es importante a lo que apunte el iterador. De esta forma se puede escribir código más genérico. Todos los contenedores y algoritmos en la Librería Estándar de C++ se basan en este principio de los iteradores.

Para ayudar a hacer las cosas más genéricas, sería agradable decir «todas las clases contenedoras tienen una clase asociada llamada `iterator`», pero esto causará normalmente problemas de nombres. La solución consite en añadir una clase anidada para cada contenedor (en este caso, «`iterator`» comienza con una letra minúscula para que esté conforme al estilo del C++ estándar). Aquí está el `InterIntStack.cpp` con un `iterator` anidado:

```
//: C16:NestedIterator.cpp
// Nesting an iterator inside the container
//{L} fibonacci
#include "fibonacci.h"
#include "../require.h"
#include <iostream>
#include <string>
using namespace std;

class IntStack {
  enum { ssize = 100 };
  int stack[ssize];
  int top;
public:
  IntStack() : top(0) {}
  void push(int i) {
    require(top < ssize, "Too many push()es");
    stack[top++] = i;
```

```
  }
  int pop() {
    require(top > 0, "Too many pop()s");
    return stack[--top];
  }
  class iterator;
  friend class iterator;
  class iterator {
    IntStack& s;
    int index;
  public:
    iterator(IntStack& is) : s(is), index(0) {}
    // To create the "end sentinel" iterator:
    iterator(IntStack& is, bool)
      : s(is), index(s.top) {}
    int current() const { return s.stack[index]; }
    int operator++() { // Prefix
      require(index < s.top,
        "iterator moved out of range");
      return s.stack[++index];
    }
    int operator++(int) { // Postfix
      require(index < s.top,
        "iterator moved out of range");
      return s.stack[index++];
    }
    // Jump an iterator forward
    iterator& operator+=(int amount) {
      require(index + amount < s.top,
        "IntStack::iterator::operator+=() "
        "tried to move out of bounds");
      index += amount;
      return *this;
    }
    // To see if you're at the end:
    bool operator==(const iterator& rv) const {
      return index == rv.index;
    }
    bool operator!=(const iterator& rv) const {
      return index != rv.index;
    }
    friend ostream&
    operator<<(ostream& os, const iterator& it) {
      return os << it.current();
    }
  };
  iterator begin() { return iterator(*this); }
  // Create the "end sentinel":
  iterator end() { return iterator(*this, true);}
};

int main() {
  IntStack is;
  for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
  cout << "Traverse the whole IntStack\n";
  IntStack::iterator it = is.begin();
  while(it != is.end())
```

```
    cout << it++ << endl;
  cout << "Traverse a portion of the IntStack\n";
  IntStack::iterator
    start = is.begin(), end = is.begin();
  start += 5, end += 15;
  cout << "start = " << start << endl;
  cout << "end = " << end << endl;
  while(start != end)
    cout << start++ << endl;
} ///:~
```

Cuando se crea una clase `friend` anidada, hay que seguir el proceso de primero declarar el nombre de la clase, después declararla como `friend`, y después definir la clase. De otra forma, se confundirá el compilador.

Al iterador se le han dado algunas vueltas de tuerca más. La función miembro `current()` produce el elemento que el iterador está seleccionando actualmente en el contenedor. Se puede «saltar» hacia adelante un número arbitrario de elementos usando el `operator+=`. También, se pueden ver otros dos operadores sobrecargados: == y != que compararán un iterador con otro. Estos operadores pueden comparar dos `IntStack::iterator`, pero su intención primordial es comprobar si el iterador está al final de una secuencia de la misma manera que lo hacen los iteradores «reales» de la Librería Estándar de C++. La idea es que dos iteradores definan un rango, incluyendo el primer elemento apuntado por el primer iterador pero *sin* incluir el último elemento apuntado por el segundo iterador. Por esto, si se quiere mover a través del rango definido por los dos iteradores, se dirá algo como lo siguiente:

```
    while (star != end)
    cout << start++ << endl;
```

Donde `start` y `end` son los dos iteradores en el rango. Note que el iterador `end`, al cual se le suele referir como el `end sentinel`, no es desreferenciado y nos avisa que estamos al final de la secuencia. Es decir, representa el que «otro sobrepasa el final».

La mayoría del tiempo se querrá mover a través de la secuencia entera de un contenedor, por lo que el contenedor necesitará alguna forma de producir los iteradores indicando el principio y el final de la secuencia. Aquí, como en la Standard C++ Library, estos iteradores se producen por las funciones miembro del contenedor `begin()` y `end()`. `begin()` usa el primer constructor de `iterator` que por defecto apunta al principio del contenedor (esto es el primer elemento que se introdujo en la pila). Sin embargo, un segundo constructor, usado por `end()`, es necesario para crear el iterador final. Estar «al final» significa apuntar a lo más alto de la pila, porque `top` siempre indica el siguiente espacio de la pila que esté disponible pero sin usar. Este constructor del `iterator` toma un segundo argumento del tipo bool, lo cual es útil para distinguir los dos constructores.

De nuevo se usan los números Fibonacci para rellenar la `IntStack` en el `main()`, y se usan iteradores para moverse completamente a través de la `IntStack` así como para moverse en un reducido rango de la secuencia.

El siguiente paso, por supuesto, es hacer el código general transformándolo en un template del tipo que maneje, para que en vez ser forzado a manejar enteros se pueda gestionar cualquier tipo:

```
//: C16:IterStackTemplate.h
// Simple stack template with nested iterator
#ifndef ITERSTACKTEMPLATE_H
#define ITERSTACKTEMPLATE_H
#include "../require.h"
#include <iostream>

template<class T, int ssize = 100>
class StackTemplate {
  T stack[ssize];
  int top;
public:
  StackTemplate() : top(0) {}
  void push(const T& i) {
    require(top < ssize, "Too many push()es");
    stack[top++] = i;
  }
  T pop() {
    require(top > 0, "Too many pop()s");
    return stack[--top];
  }
  class iterator; // Declaration required
  friend class iterator; // Make it a friend
  class iterator { // Now define it
    StackTemplate& s;
    int index;
  public:
    iterator(StackTemplate& st): s(st),index(0){}
    // To create the "end sentinel" iterator:
    iterator(StackTemplate& st, bool)
      : s(st), index(s.top) {}
    T operator*() const { return s.stack[index];}
    T operator++() { // Prefix form
      require(index < s.top,
        "iterator moved out of range");
      return s.stack[++index];
    }
    T operator++(int) { // Postfix form
      require(index < s.top,
        "iterator moved out of range");
      return s.stack[index++];
    }
    // Jump an iterator forward
    iterator& operator+=(int amount) {
      require(index + amount < s.top,
        " StackTemplate::iterator::operator+=() "
        "tried to move out of bounds");
      index += amount;
      return *this;
    }
    // To see if you're at the end:
    bool operator==(const iterator& rv) const {
      return index == rv.index;
    }
    bool operator!=(const iterator& rv) const {
      return index != rv.index;
    }
```

```
    friend std::ostream& operator<<(
      std::ostream& os, const iterator& it) {
      return os << *it;
    }
  };
  iterator begin() { return iterator(*this); }
  // Create the "end sentinel":
  iterator end() { return iterator(*this, true);}
};
#endif // ITERSTACKTEMPLATE_H ///:~
```

Se puede ver que la transformación de una clase regular en un `template` es razonablemente transparente. Esta aproximación de primero crear y depurar una clase ordinaria, y después transformarla en plantilla, está generalmente considerada como más sencilla que crear el template desde la nada.

Dese cuenta que en vez de sólo decir:

```
    friend iterator;  // Hacerlo amigo
```

Este código tiene:

```
    friend class iterator;  // Hacerlo amigo
```

Esto es importante porque el nombre «`iterator`» ya existe en el ámbito de resolución, por culpa de un archivo incluido.

En vez de la función miembro `current()`, el `iterator` tiene un `operator*` para seleccionar el elemento actual, lo que hace que el `iterator` se parezca más a un puntero lo cual es una práctica común.

Aquí está el ejemplo revisado para comprobar el template.

```
//: C16:IterStackTemplateTest.cpp
//{L} fibonacci
#include "fibonacci.h"
#include "IterStackTemplate.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  StackTemplate<int> is;
  for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
  // Traverse with an iterator:
  cout << "Traverse the whole StackTemplate\n";
  StackTemplate<int>::iterator it = is.begin();
  while(it != is.end())
    cout << it++ << endl;
  cout << "Traverse a portion\n";
  StackTemplate<int>::iterator
    start = is.begin(), end = is.begin();
  start += 5, end += 15;
  cout << "start = " << start << endl;
```

```
  cout << "end = " << end << endl;
  while(start != end)
    cout << start++ << endl;
  ifstream in("IterStackTemplateTest.cpp");
  assure(in, "IterStackTemplateTest.cpp");
  string line;
  StackTemplate<string> strings;
  while(getline(in, line))
    strings.push(line);
  StackTemplate<string>::iterator
    sb = strings.begin(), se = strings.end();
  while(sb != se)
    cout << sb++ << endl;
} ///:~
```

El primer uso del iterador simplemente lo recorre de principio a fin (y muestra que el límite final funciona correctamente). En el segundo uso, se puede ver como los iteradores permite fácilmente especificar un rango de elementos (los contenedores y los iteradores del Standard C++ Library usan este concepto de rangos casi en cualquier parte). El sobrecargado `operator+=` mueve los iteradores `start` y `end` a posiciones que están en el medio del rango de elementos de `is`, y estos elementos son imprimidos. Hay que resaltar, como se ve en la salida, que el elemento final no está incluido en el rango, o sea que una vez llegado al elemento final (end sentinel) se sabe que se ha pasado el final del rango - pero no hay que desreferenciar el elemento final o si no se puede acabar desreferenciando un puntero nulo. (Yo he puesto un guardian en el `StackTemplate::iterator`, pero en la Librería Estándar de C++ los contenedores y los iteradores no tienen ese código - por motivos de eficiencia - por lo que hay que prestar atención).

Por último para verificar que el `StackTemplate` funciona con objetos clase, se instancia uno para strings y se rellena con líneas del código fuente, las cuales son posteriormente imprimidas en pantalla.

16.7.1. Stack con iteradores

Podemos repetir el proceso con la clase de tamaño dinámico `Stack` que ha sido usada como un ejemplo a lo largo de todo el libro. Aquí está la clase `Stack` con un iterador anidado en todo el medio:

```
//: C16:TStack2.h
// Templatized Stack with nested iterator
#ifndef TSTACK2_H
#define TSTACK2_H

template<class T> class Stack {
  struct Link {
    T* data;
    Link* next;
    Link(T* dat, Link* nxt)
      : data(dat), next(nxt) {}
  }* head;
public:
  Stack() : head(0) {}
  ~Stack();
```

```
  void push(T* dat) {
    head = new Link(dat, head);
  }
  T* peek() const {
    return head ? head->data : 0;
  }
  T* pop();
  // Nested iterator class:
  class iterator; // Declaration required
  friend class iterator; // Make it a friend
  class iterator { // Now define it
    Stack::Link* p;
  public:
    iterator(const Stack<T>& tl) : p(tl.head) {}
    // Copy-constructor:
    iterator(const iterator& tl) : p(tl.p) {}
    // The end sentinel iterator:
    iterator() : p(0) {}
    // operator++ returns boolean indicating end:
    bool operator++() {
      if(p->next)
        p = p->next;
      else p = 0; // Indicates end of list
      return bool(p);
    }
    bool operator++(int) { return operator++(); }
    T* current() const {
      if(!p) return 0;
      return p->data;
    }
    // Pointer dereference operator:
    T* operator->() const {
      require(p != 0,
        "PStack::iterator::operator->returns 0");
      return current();
    }
    T* operator*() const { return current(); }
    // bool conversion for conditional test:
    operator bool() const { return bool(p); }
    // Comparison to test for end:
    bool operator==(const iterator&) const {
      return p == 0;
    }
    bool operator!=(const iterator&) const {
      return p != 0;
    }
  };
  iterator begin() const {
    return iterator(*this);
  }
  iterator end() const { return iterator(); }
};

template<class T> Stack<T>::~Stack() {
  while(head)
    delete pop();
}
```

```
template<class T> T* Stack<T>::pop() {
  if(head == 0) return 0;
  T* result = head->data;
  Link* oldHead = head;
  head = head->next;
  delete oldHead;
  return result;
}
#endif // TSTACK2_H ///:~
```

Hay que hacer notar que la clase ha sido cambiada para soportar la posesión, que funciona ahora debido a que la clase conoce ahora el tipo exacto (o al menos el tipo base, que funciona asumiendo que son usados los destructores virtuales). La opción por defecto es que el contenedor destruya sus objetos pero nosotros somos responsables de los objetos a los que se haga `pop()`.

El iterador es simple, y físicamente muy pequeño - el tamaño de un único puntero. Cuando se crea un `iterator`, se inicializa a la cabeza de la lista enlazada, y sólo puede ser incrementado avanzando a través de la lista. Si se quiere empezar desde el principio, hay que crear un nuevo iterador, y si se quiere recordar un punto de la lista, hay que crear un nuevo iterador a partir del iterador existente que está apuntando a ese elemento (usando el constructor de copia del iterador).

Para llamar a funciones del objeto referenciado por el iterador, se puede usar la función `current()`, el `operator*`, o la desreferencia de puntero `operator->` (un elemento común en los iteradores). La última tiene una implementación que *parece* idéntica a `current()` debido a que devuelve un puntero al objeto actual, pero es diferente porque el operador desreferencia del puntero realiza niveles extra de desreferenciación (ver Capítulo 12).

La clase `iterator` sigue el formato que se vio en el ejemplo anterior. `class iterator` está anidada dentro de la clase contenedora, contiene constructores para crear un iterador que apunta a un elemento en el contenedor y un iterador «marcador de final», y la clase contenedora tiene los métodos `begin()` y `end()` para producir estos iteradores. (Cuando aprenda más de la Librería Estándar de C++, verá que los nombres `iterator`, `begin()` y `end()` que se usan aquí tienen correspondecia en las clases contenedoras. Al final de este capítulo, se verá que esto permite manejar estas clases contenedoras como si fueran clases de la STL).

La implementación completa se encuentra en el archivo cabecera, por lo que no existe un archivo `cpp` separado. Aquí tenemos un pequeño test que usa el iterador.

```
//: C16:TStack2Test.cpp
#include "TStack2.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  ifstream file("TStack2Test.cpp");
  assure(file, "TStack2Test.cpp");
  Stack<string> textlines;
  // Read file and store lines in the Stack:
  string line;
```

```
  while(getline(file, line))
    textlines.push(new string(line));
  int i = 0;
  // Use iterator to print lines from the list:
  Stack<string>::iterator it = textlines.begin();
  Stack<string>::iterator* it2 = 0;
  while(it != textlines.end()) {
    cout << it->c_str() << endl;
    it++;
    if(++i == 10) // Remember 10th line
      it2 = new Stack<string>::iterator(it);
  }
  cout << (*it2)->c_str() << endl;
  delete it2;
} ///:~
```

Una pila `Stack` es instanciada para gestionar objetos string y se rellena con líneas de un fichero. Entonces se crea un iterador y se usa para moverse a través de la secuencia. La décima línea es recordada mediante un segundo iterador creado con el constructor de copia del primero; posteriormente esta línea es imprimida y el iterador - crado dinámicamente - es destruido. Aquí la creación dinámica de objetos es usada para controlar la vida del objeto.

16.7.2. PStash con iteradores

Para la mayoría de los contenedores tiene sentido tener un iterador. Aquí tenemos un iterador añadido a la clase `PStash`:

```
//: C16:TPStash2.h
// Templatized PStash with nested iterator
#ifndef TPSTASH2_H
#define TPSTASH2_H
#include "../require.h"
#include <cstdlib>

template<class T, int incr = 20>
class PStash {
  int quantity;
  int next;
  T** storage;
  void inflate(int increase = incr);
public:
  PStash() : quantity(0), storage(0), next(0) {}
  ~PStash();
  int add(T* element);
  T* operator[](int index) const;
  T* remove(int index);
  int count() const { return next; }
  // Nested iterator class:
  class iterator; // Declaration required
  friend class iterator; // Make it a friend
  class iterator { // Now define it
    PStash& ps;
    int index;
  public:
```

```
  iterator(PStash& pStash)
    : ps(pStash), index(0) {}
  // To create the end sentinel:
  iterator(PStash& pStash, bool)
    : ps(pStash), index(ps.next) {}
  // Copy-constructor:
  iterator(const iterator& rv)
    : ps(rv.ps), index(rv.index) {}
  iterator& operator=(const iterator& rv) {
    ps = rv.ps;
    index = rv.index;
    return *this;
  }
  iterator& operator++() {
    require(++index <= ps.next,
      "PStash::iterator::operator++ "
      "moves index out of bounds");
    return *this;
  }
  iterator& operator++(int) {
    return operator++();
  }
  iterator& operator--() {
    require(--index >= 0,
      "PStash::iterator::operator-- "
      "moves index out of bounds");
    return *this;
  }
  iterator& operator--(int) {
    return operator--();
  }
  // Jump interator forward or backward:
  iterator& operator+=(int amount) {
    require(index + amount < ps.next &&
      index + amount >= 0,
      "PStash::iterator::operator+= "
      "attempt to index out of bounds");
    index += amount;
    return *this;
  }
  iterator& operator-=(int amount) {
    require(index - amount < ps.next &&
      index - amount >= 0,
      "PStash::iterator::operator-= "
      "attempt to index out of bounds");
    index -= amount;
    return *this;
  }
  // Create a new iterator that's moved forward
  iterator operator+(int amount) const {
    iterator ret(*this);
    ret += amount; // op+= does bounds check
    return ret;
  }
  T* current() const {
    return ps.storage[index];
  }
  T* operator*() const { return current(); }
```

```
    T* operator->() const {
      require(ps.storage[index] != 0,
        "PStash::iterator::operator->returns 0");
      return current();
    }
    // Remove the current element:
    T* remove(){
      return ps.remove(index);
    }
    // Comparison tests for end:
    bool operator==(const iterator& rv) const {
      return index == rv.index;
    }
    bool operator!=(const iterator& rv) const {
      return index != rv.index;
    }
  };
  iterator begin() { return iterator(*this); }
  iterator end() { return iterator(*this, true);}
};

// Destruction of contained objects:
template<class T, int incr>
PStash<T, incr>::~PStash() {
  for(int i = 0; i < next; i++) {
    delete storage[i]; // Null pointers OK
    storage[i] = 0; // Just to be safe
  }
  delete []storage;
}

template<class T, int incr>
int PStash<T, incr>::add(T* element) {
  if(next >= quantity)
    inflate();
  storage[next++] = element;
  return(next - 1); // Index number
}

template<class T, int incr> inline
T* PStash<T, incr>::operator[](int index) const {
  require(index >= 0,
    "PStash::operator[] index negative");
  if(index >= next)
    return 0; // To indicate the end
  require(storage[index] != 0,
    "PStash::operator[] returned null pointer");
  return storage[index];
}

template<class T, int incr>
T* PStash<T, incr>::remove(int index) {
  // operator[] performs validity checks:
  T* v = operator[](index);
  // "Remove" the pointer:
  storage[index] = 0;
  return v;
}
```

```
template<class T, int incr>
void PStash<T, incr>::inflate(int increase) {
  const int tsz = sizeof(T*);
  T** st = new T*[quantity + increase];
  memset(st, 0, (quantity + increase) * tsz);
  memcpy(st, storage, quantity * tsz);
  quantity += increase;
  delete []storage; // Old storage
  storage = st; // Point to new memory
}
#endif // TPSTASH2_H ///:~
```

La mayoría de este archivo es un traducción prácticamente directa del anterior `PStash` y el iterador anidado dentro de un template. Esta vez, sin embargo, el operador devuelve referencias al iterador actual, la cual es una aproximación más típica y flexible.

El destructor llama a `delete` para todos los punteros que contiene, y como el tipo es obtenido de la plantilla, se ejecutará la destrucción adecuada. Hay que estar precavido que si el contenedor controla punteros al tipo de la clase base, este tipo debe tener un destructor `virtual` para asegurar un limpiado adecuado de los objetos derivados que hayan usado un upcast cuando se los alojó en el contenedor.

El `PStash::iterator` mantiene el modelo de engancharse a un único objeto contenedor durante su ciclo de vida. Además, el constructor de copia permite crear un nuevo iterador que apunte a la misma posición del iterador desde el que se le creo, creando de esta manera un marcador dentro del contenedor. Las funciones miembro `operator+=` y el `operator-=` permiten mover un iterador un número de posiciones, mientras se respeten los límites del contenedor. Los operadores sobrecargados de incremento y decremento mueven el iterador una posición. El `operator+` produce un nuevo iterador que se mueve adelante la cantidad añadida. Como en el ejemplo anterior, los operadores de desreferencia de punteros son usados para manejar el elemento al que el iterador está referenciando, y `remove()` destruye el objeto actual llamando al `remove()` del contenedor.

Se usa la misma clase de código de antes para crear el marcador final: un segundo constructor, la función miembro del contenedor `end()`, y el `operator==` y `operator!=` para comparaciones.

El siguiente ejemplo crea y comprueba dos diferentes clases de objetos `Stash`, uno para una nueva clase llamada `Int` que anuncia su construcción y destrucción y otra que gestiona objetos string de la librería Estándar.

```
//: C16:TPStash2Test.cpp
#include "TPStash2.h"
#include "../require.h"
#include <iostream>
#include <vector>
#include <string>
using namespace std;

class Int {
  int i;
public:
  Int(int ii = 0) : i(ii) {
```

```
    cout << ">" << i << ' ';
  }
  ~Int() { cout << "~" << i << ' '; }
  operator int() const { return i; }
  friend ostream&
    operator<<(ostream& os, const Int& x) {
      return os << "Int: " << x.i;
  }
  friend ostream&
    operator<<(ostream& os, const Int* x) {
      return os << "Int: " << x->i;
  }
};

int main() {
  { // To force destructor call
    PStash<Int> ints;
    for(int i = 0; i < 30; i++)
      ints.add(new Int(i));
    cout << endl;
    PStash<Int>::iterator it = ints.begin();
    it += 5;
    PStash<Int>::iterator it2 = it + 10;
    for(; it != it2; it++)
      delete it.remove(); // Default removal
    cout << endl;
    for(it = ints.begin();it != ints.end();it++)
      if(*it) // Remove() causes "holes"
        cout << *it << endl;
  } // "ints" destructor called here
  cout << "\n--------------------\n";
  ifstream in("TPStash2Test.cpp");
  assure(in, "TPStash2Test.cpp");
  // Instantiate for String:
  PStash<string> strings;
  string line;
  while(getline(in, line))
    strings.add(new string(line));
  PStash<string>::iterator sit = strings.begin();
  for(; sit != strings.end(); sit++)
    cout << **sit << endl;
  sit = strings.begin();
  int n = 26;
  sit += n;
  for(; sit != strings.end(); sit++)
    cout << n++ << ": " << **sit << endl;
} ///:~
```

Por conveniencia `Int` tiene asociado un `ostream operator<<` para `Int&` y `Int*`.

El primer bloque de código en `main()` está rodeado de llaves para forzar la destrucción de `PStash<Int>` que produce un limpiado automático por este destructor. Unos cuantos elementos son sacados y borrados a mano para mostrar que `PStash` limpia el resto.

Para ambas instancias de `PStash`, se crea un iterador y se usa para moverse a través del contenedor. Note la elegancia generada por el uso de estos constructores; no hay que preocuparse por los detalles de implementación de usar un array. Se le dice al contenedor y al iterador *qué* hacer y no *cómo* hacerlo. Esto produce una solución más sencilla de conceptualizar, construir y modificar.

16.8. Por qué usar iteradores

Hasta ahora se han visto los mecanismos de los iteradores, pero entender el por qué son tan importantes necesita un ejemplo más complejo.

Es normal ver el polimorfismo, la creación dinámica de objetos, y los contenedores en un programa orientado a objetos real. Los contendores y la creación dinámica de objetos resuelven el problema de no saber cuantos o que tipo de objetos se necesitarán. Y si el contenedor está configurado para manejar punteros a la clase base, cada vez que se ponga un puntero a una clase derivada hay un upcast (con los beneficios que conlleva de claridad de código y extensibilidad). Como código del final del Volumen 1, este ejemplo reune varios aspectos de todo lo que se ha aprendido - si es capaz de seguir este ejemplo, entonces está preparado para el Volumen 2.

Suponga que esta creando un programa que permite al usuario editar y producir diferentes clases de dibujos. Cada dibujo es un objeto que contiene una colección de objetos `Shape`:

```
//: C16:Shape.h
#ifndef SHAPE_H
#define SHAPE_H
#include <iostream>
#include <string>

class Shape {
public:
  virtual void draw() = 0;
  virtual void erase() = 0;
  virtual ~Shape() {}
};

class Circle : public Shape {
public:
  Circle() {}
  ~Circle() { std::cout << "Circle::~Circle\n"; }
  void draw() { std::cout << "Circle::draw\n";}
  void erase() { std::cout << "Circle::erase\n";}
};

class Square : public Shape {
public:
  Square() {}
  ~Square() { std::cout << "Square::~Square\n"; }
  void draw() { std::cout << "Square::draw\n";}
  void erase() { std::cout << "Square::erase\n";}
};

class Line : public Shape {
public:
  Line() {}
```

```
  ~Line() { std::cout << "Line::~Line\n"; }
  void draw() { std::cout << "Line::draw\n";}
  void erase() { std::cout << "Line::erase\n";}
};
#endif // SHAPE_H ///:~
```

Se usa la estructura clásica de las funciones virtuales en la clase base que son sobreescritas en la clase derivada. Hay que resaltar que la clase `Shape` incluye un destructor virtual, algo que se debería añadir automáticamente a cualquier clase con funciones `virtual`es. Si un contenedor maneja punteros o referencias a objetos `Shape`, entonces cuando los destructores virtuales sean llamados para estos objetos todo será correctamente limpiado.

Cada tipo diferente de dibujo en el siguiente ejemplo hace uso de una plantilla de clase contenedora diferente: el `PStash` y el `Stack` que han sido definido en este capítulo, y la clase `vector` de la Librería Estándar de C++. El «uso» de los contenedores es extremadamente simple, y en general la herencia no es la mejor aproximación (composición puede tener más sentido), pero en este caso la herencia es una aproximación más simple.

```
//: C16:Drawing.cpp
#include <vector> // Uses Standard vector too!
#include "TPStash2.h"
#include "TStack2.h"
#include "Shape.h"
using namespace std;

// A Drawing is primarily a container of Shapes:
class Drawing : public PStash<Shape> {
public:
  ~Drawing() { cout << "~Drawing" << endl; }
};

// A Plan is a different container of Shapes:
class Plan : public Stack<Shape> {
public:
  ~Plan() { cout << "~Plan" << endl; }
};

// A Schematic is a different container of Shapes:
class Schematic : public vector<Shape*> {
public:
  ~Schematic() { cout << "~Schematic" << endl; }
};

// A function template:
template<class Iter>
void drawAll(Iter start, Iter end) {
  while(start != end) {
    (*start)->draw();
    start++;
  }
}

int main() {
  // Each type of container has
```

```
  // a different interface:
  Drawing d;
  d.add(new Circle);
  d.add(new Square);
  d.add(new Line);
  Plan p;
  p.push(new Line);
  p.push(new Square);
  p.push(new Circle);
  Schematic s;
  s.push_back(new Square);
  s.push_back(new Circle);
  s.push_back(new Line);
  Shape* sarray[] = {
    new Circle, new Square, new Line
  };
  // The iterators and the template function
  // allow them to be treated generically:
  cout << "Drawing d:" << endl;
  drawAll(d.begin(), d.end());
  cout << "Plan p:" << endl;
  drawAll(p.begin(), p.end());
  cout << "Schematic s:" << endl;
  drawAll(s.begin(), s.end());
  cout << "Array sarray:" << endl;
  // Even works with array pointers:
  drawAll(sarray,
    sarray + sizeof(sarray)/sizeof(*sarray));
  cout << "End of main" << endl;
} ///:~
```

Los distintos tipos de contenedores manejan punteros a `Shape` y punteros a objetos de clases derivadas de `Shape`. Sin embargo, debido al polimorfismo, cuando se llama a las funcione virtuales ocurre el comportamiento adecuado.

Note que `sarray`, el array de `Shape*`, puede ser recorrido como un contenedor.

16.8.1. Plantillas Función

En `drawAll()` se ve algo nuevo. En este capítulo, únicamente hemos estado usando *plantillas de clases*, las cuales pueden instanciar nuevas clases basadas en uno o más parámetros de tipo. Sin embargo, se puede crear *plantillas de función*, las cuales crean nuevas funciones basadas en parámetros de tipo. La razón para crear una plantilla de función es la misma por la cual se crea una plantilla de clase: intentar crear código más genérico, y se hace retrasando la especificación de uno o más tipos. Se quiere decir que estos parámetros de tipos soportan ciertas operaciones, no qué tipos exactos son.

Se puede pensar sobre la plantilla función `drawAll()` como si fuera un *algoritmo* (y así es como se llaman la mayoría de las plantillas de función de la STL). Sólo dice como hacer algo dado unos iteradores que describen un rango de elementos, mientras que estos iteradores pueden ser desreferenciados, incrementados, y comparados. Estos son exactamente la clase de iteradores que hemos estado desarrollando en este capítulo, y también - y no por casualidad - la clase de iteradores que son producidos por los contenedores de la Librería Estándar de C++, evidenciado por el

uso de `vector` en este ejemplo.

Además nos gustaría que `drawAll()` fuera un *algoritmo genérico*, para que los contenedores pudieran ser de cualquier tipo y que no se tuviera que escribir una nueva versión del algoritmo para cada tipo diferente del contenedor. Aquí es donde las plantillas de funciones son esenciales, porque automáticamente generan el código específico para cada tipo de contenedor diferente. Pero sin la indirección extra proporcionada por los iteradores, estas generalizaciones no serían posibles. Este es el motivo por el que los iteradores son importantes; nos permiten escribir código de propósito general que involucra a contenedores sin conocer la estructura subyacente del contenedor. (Note que los iteradores de C++ y los algoritmos genéricos requieren plantillas de funciones).

Se puede ver el alcance de esto en el `main()`, ya que `drawAll()` funciona sin cambiar cada uno de los diferentes tipos de contenedores. E incluso más interesante, `drawAll()` también funciona con punteros al principio y al final del array `sarray`. Esta habilidad para tratar arrays como contenedores está integrada en el diseño de la Librería Estándar de C++, cuyos algoritmos se parecen mucho a `drawAll()`.

Debido a que las plantillas de clases contenedoras están raramente sujetas a la herencia y al upcast se ven como clases «ordinarias», casi nunca se verán funciones virtuales en clases contenedoras. La reutilización de las clases contenedoras está implementado mediante plantillas, no mediante herencia.

16.9. Resumen

Las clases contenedoras son una parte esencial de la programación orientada a objetos. Son otro modo de simplificar y ocultar los detalles de un programa y de acelerar el proceso de desarrollo del programa. Además, proporcionan un gran nivel de seguridad y flexibilidad reemplazando los anticuados arrays y las relativamente toscas técnicas de estructuras que se pueden encontrar en C.

Como el programador cliente necesita contenedores, es esencial que sean fáciles de usar. Aquí es donde entran los `templates`. Con las plantillas la sintaxis para el reciclaje del código fuente (al contrario del reciclaje del código objeto que proporciona la herencia y la composición) se vuelve lo suficientemente trivial para el usuario novel. De hecho, la reutilización de código con plantillas es notablemente más fácil que la herencia y el polimorfismo.

Aunque se ha aprendido cómo crear contenedores y clases iteradoras en este libro, en la práctica es mucho más útil aprender los contenedores e iteradores que contiene la Librería Estándar de C++, ya que se puede esperar encontrarlas en cualquier compilador. Como se verá en el Volumen 2 de este libro (que se puede bajar de www.BruceEckel.com, los contenedores y algoritmos de la STL colmarán virtualmente sus necesidades por lo que no tendrá que crear otras nuevas.

Las características que implica el diseño con clases contenedoras han sido introducidas a lo largo de todo el capítulo, pero hay que resaltar que van mucho más allá. Una librería de clases contenedoras más complicada debería cubrir todo tipo de características adicionales, como la multitarea, la persistencia y la recolección de basura.

16.10. Ejercicios

Las soluciones a los ejercicios se pueden encontrar en el documento electrónico titulado «The Thinking in C++ Annotated Solution Guide», disponible por poco dinero en www.BruceEckel.com.

1. Implemente la jerarquía de herencia del diagrama de `OShape` de este capítulo.
2. Modifique el resultado del Ejercicio 1 del capítulo 15 para usar la `Stack` y el `iterator` en `TStack2.h` en vez de un array de punteros a `Shape`. Añada destructores a la jerarquía de clases para que se pueda ver que los objetos `Shape` han sido destruidos cuando la `Stack` se sale del ámbito.
3. Modifique `TPStash.h` para que el valor de incremento usado por `inflate()` pueda ser cambiado durante la vida de un objeto contenedor particular.
4. Modifique `TPStash.h` para que el valor de incremento usado por `inflate()` automáticamente cambie de tamaño para que reduzca el número de veces que debe ser llamado. Por ejemplo, cada vez que se llama podría doblar el valor de incremento para su uso en la siguiente llamada. Demuestre la funcionalidad mostrando cada vez que se llama a `inflate()`, y escriba código de prueba en `main()`.
5. Convierta en plantilla la función de `fibonacci()` con los tipos que puede producir (puede generar long, float, etc. en vez de sólo int).
6. Usar el `vector` de la STL como implementación subyacente, para crear una platilla `Set` que acepte solo uno de cada tipo de objeto que se aloje en él. Cree un iterador anidado que soporte el concepto de "marcador final" de este capítulo. Escriba código de prueba para el `Set` en el `main()`, y entonces sustituyalo por la plantilla `set` de la STL para comprobar que el comportamiento es correcto.
7. Modifique `AutoCounter.h` para que pueda ser usado como un objeto miembro dentro de cualquier clase cuya creación y destrucción quiera comprobar. Añada un miembro string para que contenga el nombre de la clase. Compruebe esta herramienta dentro una clase suya.
8. Cree una versión de `OwnerStack.h` que use un `vector` de la Librería Estándar de C++ como su implementación subyacente. Será necesario conocer algunas de las funciones miembro de `vector` para poder hacerlo (sólo hay que mirar en el archivo cabecera <vector>).
9. Modifique `ValueStack.h` para que pueda expandirse dinámicamente según se introduzcan más objetos y se quede sin espacio. Cambie `ValueStackTest.cpp` para comprobar su nueva funcionalidad.
10. Repita el ejercicio 9 pero use el `vector` de la STL como la implementación interna de `ValueStack`. Note lo sencillo que es.
11. Modifique `ValueStackTest.cpp` para que use un `vector` de la STL en vez de un `Stack` en el `main()`. Dése cuenta del comportamiento en tiempo de ejecución: ¿Se genera un grupo de objetos por defecto cuando se crea el `vector`?

12. Modifique `TStack2.h` para que use un `vector` de la STL. Asegurese de que no cambia la interfaz, para que `TStack2Test.cpp` funcione sin cambiarse.

13. Repita el Ejercicio 12 usando una `stack` de la Librería Estándar de C++ en vez de un `vector`.

14. Modifique `TPStash2.h` para que use un `vector` de la STL como su implementación interna. Asegurese que no cambia la interfaz, por lo que `TPStash2Test.cpp` funciona sin modificarse.

15. En `IterIntStack.cpp`, modifique `IntStackIter` para darle un constructor de «marcador final», y añada el `operator==` y el `operator!=`. En el `main()`, use un iterador para moverse a través de los elementos del contenedor hasta que se encuentre el marcador.

16. Use `TStack2.h`, `TPSTash2.h`, y `Shape.h`, instancie los contenedores `PStash` y `Stack` para que contenga `Shape*`, rellene cada uno con punteros a `Shape`, entonces use iteradores para moverse a través de cada contenedor y llame a `draw()` para cada objeto.

17. Cree una plantilla en la clase Int para que pueda alojar cualquier tipo de objetos (Siéntase libre de cambiar el nombre de la clase a algo más apropiado).

18. Cree una plantilla de la clase `IntArray` en `IostreamOperatorOverloading.cpp` del capítulo 12, introduzca en plantilla ambos tipos de objetos que están contenidos y el tamaño del array interno

19. Convierta `ObjContainer` en `NestedSmartPointer.cpp` del Capítulo 12 en una plantilla. Compruebelo con dos clases diferentes.

20. Modifique `C15:OStack.h` y `C15:OStackTest.cpp` consiguiendo que `class Stack` pueda tener múltiple herencia automáticamente de la clase contenida y de `Object`. La `Stack` contenida debe aceptar y producir sólo punteros del tipo contenido.

21. Repita el ejercicio 20 usando `vector` en vez de `Stack`.

22. Herede una clase `StringVector` de `vector<void>` y redefina las funciones miembro `push_back()` y el `operator[]` para que acepten y produzcan únicamente string* (y realizen el moldeado adecuado). Ahora creee una plantilla que haga automáticamente lo mismo a una clase contenedora para punteros de cualquier tipo. Esta técnica es a menudo usada para reducir el código producido por muchas instanciaciones de templates.

23. En `TPStash2.h`, añada y compruebe un `operator-` para `PStash::iterator`, siguiendo la lógica de `operator+`.

24. En `Drawing.cpp`, añada y compruebe una plantilla de función que llame a funciones miembro `erase()`.

25. (Avanzado) Modifique la clase `Stack` en `TStack2.h` para permitir una granularidad de la propiedad: Añada una bandera para cada enlace indicando si el enlace posee el objeto al que apunta, y de soporte a esta información la función `push()` y en el destructor. Añada funciones miembro para leer y cambiar la propiedad de cada enlace.

26. (Avanzado) Modifique `PointerToMemberOperator.cpp` del Capítulo 12 para que la `FunctionObject` y el `operator->*` sean convertidos en plantillas para que funcionen con cualquier tipo de retorno (para `operator->*`, tendrá que usar *plantillas miembro* descritas en el Volumen 2). Añada soporte y compruebe para cero, uno y dos argumentos en las funciones miembro `Dog`.

A: Estilo de codificación

Este apéndice no trata sobre indentación o colocación de paréntesis y llaves, aunque sí que se menciona. Trata sobre las directrices generales que se usan en este libro para la organización de los listados de código.

Aunque muchas de estas cuestiones se han tratado a lo largo del libro, este apéndice aparece al final de manera que se puede asumir que cada tema es FIXME:juego limpio, y si no entiende algo puede buscar en la sección correspondiente.

Todas las decisiones sobre estilo de codificación en este libro han sido consideradas y ejectuadas deliberadamente, a veces a lo largo de períodos de años. Por supuesto, cada uno tiene sus razones para organizar el código en el modo en que lo hace, y yo simplemente intento explicarle cómo llegué a tomar mi postura y las restricciones y factores del entorno que me llevaron a tomar esas decisiones.

A.1. General

En el texto de este libro, los identificadores (funciones, variables, y nombres de clases) aparecen en negrita. Muchas palabras reservadas también son negritas, exceptuando aquellas que se usan tan a menudo que escribirlas en negrita puede resultar tedioso, como «class» o «virtual».

Utilizo un estilo de codificación particular para los ejemplos de este libro. Se desarrolló a lo largo de varios años, y se inspiró parcialmente en el estilo de Bjarne Stroustrup en el *The C++ Programming Language* [1] original. El asunto del estilo de codificación es ideal para horas de acalorado debate, así que sólo diré que no trato de dictar el estilo correcto a través de mis ejemplos; tengo mis propios motivos para usar el estilo que uso. Como C++ es un lenguaje de formato libre, cada uno puede continuar usando el estilo que le resulte más cómodo.

Dicho esto, sí haré hincapié en que es importante tener un estilo consistente dentro de un proyecto. Si busca en Internet, encontrará un buen número de herramientas que se pueden utilizar para reformatear todo el código de un proyecto para conseguir esa valiosa consistencia.

Los programas de este libro son ficheros extraídos automáticamentente del texto del libro, lo que permite que se puedan probar para asegurar que funcionan correctamente [2]. De ese modo, el código mostrado en el libro debería funcionar sin

[1] FIXME:Ibid.

[2] (N. de T.) Se refiere al libro original. En esta traducción, los programas son ficheros externos incluidos en el texto.

errores cuando se compile con una implementación conforme al Estándar C++ (no todos los compiladores soportan todas las características del lenguaje). Las sentencias que *deberían* causar errores de compilación están comentadas con `//!` de modo que se pueden descubrir y probar fácilmente de modo automático. Los errores descubiertos por el autor aparecerán primero en la versión electrónica del libro (www.BruceEckel.com) y después en las actualizaciones del libro.

Uno de los estándares de este libro es que todos los programas compilarán y enlazarán sin errores (aunque a veces causarán advertencias). Algunos de los programas, que demuestran sólo un ejemplo de codificación y no representan programas completos, tendrán funciones `main()` vacías, como ésta:

```
int main() {}
```

Esto permite que se pueda enlazar el programa sin errores.

El estándar para `main()` es retornar un int, pero C++ Estándar estipula que si no hay una sentencia `return` en `main()`, el compilador generará automáticamente código para `return 0`. Esta opción (no poner un `return` en `main()`) se usa en el libro (algunos compiladores producen advertencias sobre ello, pero es porque no son conformes con C++ Estándar).

A.2. Nombres de fichero

En C, es tradición nombrar a los ficheros de cabecera (que contienen las declaraciones) con una extensión `.h` y a los ficheros de implementación (que generan alojamiento en memoria y código) con una extensión `.c`. C++ supuso una evolución. Primero fue desarrollado en Unix, donde el sistema operativo distingue entre mayúsculas y minúsculas para nombres de ficheros. Los nombres originales para los ficheros simplemente se pusieron en mayúscula: `.H` y `.C`. Esto, por supuesto, no funcionaba en sistemas operativos que no distinguen entre mayúsculas y minúsculas como DOS. Los vendedores de C++ para DOS usaban extensiones `hxx` y `cxx`, o `hpp` y `cpp`. Después, alguien se dio cuenta que la única razón por la que se puede necesitar un extensión diferente es que el compilador no puede determinar si debe compilarlo como C o C++. Como el compilador nunca compila ficheros de cabecera directamente, sólo el fichero de implementación necesita una distinción. Ahora, en prácticamente todos los sistemas, la costumbre es usar `cpp` para los ficheros de implementación y `.h` para los ficheros de cabecera. Fíjese que cuando se incluye un fichero de cabecera C++, se usa la opción de no poner extensión al nombre del fichero, por ejemplo: `#include <iostream>`

A.3. Marcas comentadas de inicio y fin

Un tema muy importante en este libro es que todo el código que puede ver en el libro ha sido sido verificado (con al menos un compilador). Esto se consigue extrayendo automáticamente los listados del libro. Para facilitar esta tarea, todos los listados de código susceptibles de ser compilados (al contrario que los fragmentos, que hay pocos) tienen unas marcas comentadas al principio y al final. Estas marcas las usa la herramienta de extracción de código `ExtractCode.cpp` del Volumen 2 de este libro (y que se puede encontrar en el sitio web www.BruceEckel.com) para extraer cada listado de código a partir de la versión en texto plano ASCII de este libro.

La marca de fin de listado simplemente le indica a `ExtractCode.cpp` que ese es el final del listado, pero la marca de comienzo incluye información sobre el subdirectorio al que corresponde el fichero (normalmente organizado por capítulos, así que si corresponde al Capítulo 8 debería tener una etiqueta como `C08`), seguido de dos puntos y el nombre del fichero.

Como `ExtractCode.cpp` también crea un `makefile` para cada subdirectorio, la información de cómo construir el programa y la línea de comando que se debe usar para probarlo también se incorpora a los listados. Si un programa es autónomo (no necesita ser enlazado con nada más) no tiene información extra. Esto también es cierto para los ficheros de cabecera. Sin embargo, si no contiene un `main()` y necesita enlazarse con algún otro, aparece un `{O}` después del nombre del fichero. Si ese listado es el programa principal pero necesita ser enlazado con otros componentes, hay una línea adicional que comienza con `//{L}` y continúa con el nombre de todos los ficheros con los que debe enlazarse (sin extensiones, dado que puede variar entre plataformas).

Puede encontrar ejemplos a lo largo de todo el libro.

Cuando un fichero debe extraerse sin que las marcas de inicio y fin deban incluirse en el fichero extraído (por ejemplo, si es un fichero con datos para una prueba) la marca de inicio va seguida de un '!'.

A.4. Paréntesis, llaves e indentación

Habrá notado que el estilo de este libro es diferente a la mayoría de los estilos C tradicionales. Por supuesto, cualquiera puede pensar que su propio estilo es más racional. Sin embargo, el estilo que se emplea aquí tiene una lógica más simple, que se presentará mezclada con las de otros estilos desarrollados.

El estilo está motivado por una cosa: la presentación, tanto impresa como en un seminario. Quizá sus necesidades sean diferentes porque no realiza muchas presentaciones. Sin embargo, el código real se lee muchas más veces de las que se escribe, y por eso debería ser fácil de leer. Mis dos criterios más importantes son la «escaneabilidad» (que se refiere a la facilidad con la que el lector puede comprender el significado de una única línea) y el número de líneas que caben en una página. Lo segundo puede sonar gracioso, pero cuando uno da una charla, distrae mucho a la audiencia que el ponente tenga que avanzar y retroceder diapositivas, y sólo unas pocas líneas de más puede provocar este efecto.

Todo el mundo parece estar de acuerdo en que el código que se pone dentro de llaves debe estar indentado. En lo que la gente no está de acuerdo - y es el sitio donde más inconsistencia tienen los estilos - es: ¿Dónde debe ir la llave de apertura? Esta única cuestión, creo yo, es la que causa la mayoría de las variaciones en los estilos de codificación (Si quiere ver una enumeración de estilos de codificación vea *C++ Programming Guidelines*, de [FIXME:autores] Tom Plum y Dan Saks, Plum Hall 1991), Intentaré convencerle de que muchos de los estilos de codificación actuales provienen de la restricciones previas al C Estándar (antes de los prototipos de función) de manera que no son apropiadas actualmente.

Lo primero, mi respuesta a esa pregunta clave: la llave de apertura debería ir siempre en la misma línea que el «precursor» (es decir «cualquier cosa de la que sea cuerpo: una clase, función, definición de objeto, sentencia `if`, etc». Es una regla única y consistente que aplico a todo el código que escribo, y hace que el formateo de código sea mucho más sencillo. Hace más sencilla la «escaneabilidad» - cuando se lee esta línea:

```
int func(int a);
```

Se sabe, por el punto y coma al final de la línea, que esto es una declaración y no hay nada más, pero al leer la línea:

```
int func(int a) {
```

inmediatamente se sabe que se trata de una definición porque la línea termina con una llave de apertura, y no un punto y coma. Usando este enfoque, no hay diferencia a la hora de colocar el paréntesis de apertura en una definición de múltiples líneas.

```
int func(int a) {
  int b = a + 1;
  return b * 2;
}
```

y para una definición de una sola línea que a menudo se usa para inlines:

```
int func(int a) { return (a + 1) * 2; }
```

Igualmente, para una clase:

```
class Thing;
```

es una declaración del nombre de una clase, y

```
class Thing {
```

es una definición de clase. En todos los casos, se puede saber mirando una sola línea si se trata de una declaración o una definición. Y por supuesto, escribir la llave de apertura en la misma línea, en lugar de una línea propia, permite ahorrar espacio en la página.

Así que ¿por qué tenemos tantos otros estilos? En concreto, verá que mucha gente crea clases siguiente el estilo anterior (que Stroustrup usa en todas las ediciones de su libro *The C++ Programming Language* de Addison-Wesley) pero crean definiciones de funciones poniendo la llave de apertura en una línea aparte (lo que da lugar a muchos estilos de indentación diferentes). Stroustrup lo hace excepto para funciones inline cortas. Con el enfoque que yo describo aquí, todo es consistente - se nombra lo que sea (`class`, función, `enum`, etc) y en la misma línea se pone la llave de apertura para indicar que el cuerpo de esa cosa está debajo. Y también, la llave de apertura se pone en el mismo sitio para funciones inline que para definiciones de funciones ordinarias.

Creo que el estilo de definición de funciones que utiliza mucha gente viene de el antiguo prototipado de funciones de C, en el que no se declaraban los argumentos entre los paréntesis, si no entre el paréntesis de cierre y la llave de apertura (esto demuestra que las raíces de C son el lenguaje ensamblador):

```
void bar()
 int x;
```

```
     float y;
    {
     /* body here */
    }
```

Aquí, quedaría bastante mal poner la llave de apertura en la misma línea, así que nadie lo hacía. Sin embargo, había distintas opiniones sobre si las llaves debían indentarse con el cuerpo del código o debían dejarse a nivel con el «precursor». De modo que tenemos muchos estilos diferentes.

Hay otros argumentos para poner la llave en la línea siguiente a la declaración (de una clase, `struct`, función, etc). Lo siguiente proviene de un lector, y lo presento aquí para que sepa a qué se refiere.

Los usuarios experimentado de **vi** (vim) saben que pulsar la tecla «]» dos veces lleva el cursor a la siguiente ocurrencia de «{» (o ^L) en la columna 0. Esta característica es extremadamente útil para moverse por el código (saltando a la siguiente defición de función o clase). [Mi comentario: cuando yo trabajaba en Unix, GNU Emacs acababa de aparecer y yo me convertí en un fan suyo. Como resultado, **vi** nunca ha tenido sentido para mí, y por eso yo no pienso en términos de «situación de columna 0». Sin embargo, hay una buena cantidad de usuarios de **vi** ahí fuera, a los que les afecta esta característica.]

Poniendo la «{» en la siguiente línea se eliminan algunas confusiones en sentencias condicionales complejas, ayudando a la escaneabilidad.

```
    if (cond1
       && cond2
       && cond3) {
       statement;
    }
```

Lo anterior [dice el lector] tiene una escaneabilidad pobre. Sin embargo,

```
    if (cond1
    && cond2
    && cond3)
    {
    statement;
    }
```

separa el `if` del cuerpo, mejorando la legibilidad. [Sus opiniones sobre si eso es cierto variarán dependiendo para qué lo haya usado.]

Finalmente, es mucho más fácil visualizar llaves emparejadas si están alineadas en la misma columna. Visualmente destacan mucho más. [Fin del comentario del lector]

El tema de dónde poner la llave de apertura es probablemente el asunto en el que hay menos acuerdo. He aprendido a leer ambas formas, y al final cada uno utiliza la que le resulta más cómoda. Sin embargo, he visto que el estándar oficial de codificación de Java (que se puede encontar en la página de Java de Sun) efectivamente es el mismo que yo he presentado aquí - dado que más personas están empezando a programar en ambos lenguajes, la consistencia entre estilos puede ser útil.

Mi enfoque elimina todas las excepciones y casos especiales, y lógicamente produce un único estilo de indentación, Incluso con un cuerpo de función, la consisten-

cia se mantiene, como en:

```
for(int i = 0; i < 100; i++) {
  cout << i << endl;
  cout << x * i << endl;
}
```

El estilo es fácil de enseñar y recordar - use una regla simple y consistente para todo sus formatos, no una para clases, dos para funciones (funciones inline de una línea vs. multi-línea), y posiblemente otras para bucles, sentencias `if`, etc. La consistencia por si sola merece ser tenida en cuenta. Sobre todo, C++ es un lenguaje más nuevo que C, y aunque debemos hacer muchas concesiones a C, no deberíamos acarrear demasiados FIXME:artifacts que nos causen problemas en el futuro. Problemas pequeños multiplicados por muchas líneas de código se convierten en grandes problemas. Para un examen minucioso del asunto, aunque trata de C, vea *C Style: Standards and Guidelines*, de David Straker (Prentice-Hall 1992).

La otra restricción bajo la que debo trabajar es la longitud de la línea, dado que el libro tiene una limitación de 50 caracteres. ¿Qué ocurre si algo es demasiado largo para caber en una línea? Bien, otra vez me esfuerzo en tener una política consistente para las líneas partidas, de modo que sean fácilmente visibles. Siempre que sean parte de una única definición, lista de argumentos, etc., las líneas de continuación deberían indentarse un nivel respecto al comienzo de la definición, lista de argumentos, etc.

A.5. Nombres para identificadores

Aquellos que conozcan Java notarán que yo me he cambiado al estilo estándar de Java para todos los identificadores. Sin embargo, no puedo ser completamente consistente porque los identificadores en C Estándar y en librerías C++ no siguen ese estilo.

El estilo es bastante sencillo. La primera letra de un identificador sólo se pone en mayúscula si el identificador es una clase. Si es una función o variable, la primera letra siempre va en minúscula. El resto del identificador consiste en una o más palabras, todas juntas pero se distinguen porque la primera letra de cada palabra es mayúscula. De modo que una clase es algo parecido a esto:

```
class FrenchVanilla : public IceCream {
```

y un objeto es algo como esto:

```
FrenchVanilla myIceCreamCone(3);
```

y una función:

```
void eatIceCreamCone();
```

(tanto para un método como para un función normal).

La única excepción son las constantes en tiempo de compilación (`const` y `#define`), en las que todas las letras del identificador son mayúsculas.

El valor del estilo es que el uso de mayúsculas tiene significado - viendo la primera letra se puede saber si es una clase o un objeto/método. Esto es especialmente útil cuando se invocan miembros estáticos.

A.6. Orden de los #includes

Los ficheros de cabecera se incluyen en orden «del más específico al más general». Es decir, cualquier fichero de cabecera en el directorio local se incluye primero, después las «herramientas» propias, como `require.h`, luego cabeceras de librerías de terceros, después cabeceras de la librería estándar C++, y finalmente cabeceras de la librería C.

La justificación para esto viene de John Lakos en *Large-Scale C++ Software Design* (Addison-Wesley, 1996):

> FIXME Los errores de uso latentes se puede evitar asegurando que el fichero .h de un componente es coherente en si mismo - sin declaraciones o definiciones externas. Incluir el fichero `.h` como primera línea del fichero `.c` asegura que no falta ninguna pieza de información de la interfaz física del componente en el fichero `.h` (o, si la hay, aparecerá en cuanto intente compilar el fichero `.c`.

Si el orden de inclusión fuese «desde el más específico al más general», entonces es más probable que si su fichero de cabecera no es coherente por si mismo, lo descubrirá antes y prevendrá disgustos en el futuro.

A.7. Guardas de inclusión en ficheros de cabecera

Los guardas de inclusión se usan siempre en los ficheros de cabecera para prevenir inclusiones múltiples durante la compilación de un único fichero `.cpp`. Los guardas de inclusión se implementan usado `#define` y comprobando si el nombre no ha sido definido previamente. El nombre que se usa para el guarda está basado en el nombre del fichero de cabecera, pero con todas las letras en mayúscula y reemplazando el punto por un guión bajo. Por ejemplo:

```
// IncludeGuard.h
#ifndef INCLUDEGUARD_H
#define INCLUDEGUARD_H
// Body of header file here...
#endif // INCLUDEGUARD_H
```

El identificador de la última línea se incluye únicamente por claridad. Algunos preprocesadores ignoran cualquier carácter que aparezca después de un `#endif`, pero no es el comportamiento estándar y por eso el identificador aparece comentado.

A.8. Uso de los espacios de nombres

En los ficheros de cabecera, se debe evitar de forma escrupulosa cualquier contaminación del espacio de nombres. Es decir, si se cambia el espacio de nombres fuera de una función o clase, provocará que el cambio ocurra también en cualquier fichero que incluya ese fichero de cabecera, lo que resulta en todo tipo de problemas.

No están permitidas las declaraciones `using` de ningún tipo fuera de las definiciones de función, y tampoco deben ponerse directivas `using` globales en ficheros de cabecera.

En ficheros `cpp`, cualquier directiva `using` global sólo afectará a ese fichero, y por eso en este libro se usan generalmente para conseguir código más legible, especialmente en programas pequeños.

A.9. Utilización de `require()` y `assure()`

Las funciones `require()` y `assure()` definidas en `requiere.h` se usan constantemente a lo largo de todo el libro, para que informen de problemas. Si se está familiarizado con los conceptos de precondiciones y postcondiciones (introducidos por Bertrand Meyer) es fácil reconocer que el uso de `require()` y `assure()` más o menos proporciona precondiciones (normalmente) y postcondiciones (ocasionalmente). Por eso, al principio de una función, antes de que se ejecute el «núcleo» de la función, se comprueban las precondiciones para estar seguro de que se cumplen todas las condiciones necesarias. Entonces, se ejecuta el «núcleo» de la función, y a veces se comprueban algunas postcondiciones para estar seguro de que el nuevo estado en el que han quedado los datos está dentro de los parámetros correspondientes. Notará que las comprobaciones de postcondición se usan raramente en este libro, y `assure()` se usa principalmente para estar seguro de que los ficheros se abren adecuadamente.

B: Directrices de Programación

Este apéndice es una colección de sugerencias para programación con C++. Se han reunido a lo largo de mi experiencia en como docente y programador y

también de las aportaciones de amigos incluyendo a Dan Saks (co-autor junto a Tom Plum de *C++ Programming Guidelines*, Plum Hall, 1991), Scott Meyers (autor de *Effective C++*, 2ª edición, Addison-Wesley, 1998), and Rob Murray (autor de *C++ Strategies & Tactics*, Addison-Wesley, 1993). También, muchos de los consejos están resumidos a partir del contenido de *Thinking in C++*.

1. Primero haga que funcione, después hágalo rápido. Esto es cierto incluso si se está seguro de que una trozo de código es realmente importante y se sabe que será un cuello de botella es el sistema. No lo haga. Primero, consiga que el sistema tenga un diseño lo más simple posible. Entonces, si no es suficientemente rápido, optimícelo. Casi siempre descubrirá que «su» cuello de botella no es el problema. Guarde su tiempo para lo verdaderamente importante.

2. La elegancia siempre vale la pena. No es un pasatiempo frívolo. No sólo permite que un programa sea más fácil de construir y depurar, también es más fácil de comprender y mantener, y ahí es donde radica su valor económico. Esta cuestión puede requerir de alguna experiencia para creerselo, porque puede parecer que mientras se está haciendo un trozo de código elegante, no se es productivo. La productividad aparece cuando el código se integra sin problemas en el sistema, e incluso cuando se modifica el código o el sistema.

3. Recuerde el principio «divide y vencerás». Si el problema al que se enfrenta es desmasiado confuso, intente imaginar la operación básica del programa se puede hacer, debido a la existencia de una «pieza» mágica que hace el trabajo difícil. Esta «pieza» es un objeto - escriba el código que usa el objeto, después implemente ese objeto encapsulando las partes difíciles en otros objetos, etc.

4. No reescriba automáticamente todo su código C a C++ a menos que necesite un cambiar significativamente su funcionalidad (es decir, no lo arregle si no está roto). *Recompilar C* en C++ es un positivo porque puede revelar errores ocultos. Sim embargo, tomar código C que funciona bien y reescribirlo en C++ no es la mejor forma de invertir el tiempo, a menos que la versión C++ le ofrezca más oportunidad de reutilizarlo como una clase.

5. Si tiene un gran trozo de código C que necesite cambios, primero aisle las partes del código que no se modificará, posiblemente envolviendo esas funciones en una «clase API» como métodos estáticos. Después ponga aténción al código que va a cambiar, recolocandolo dentro de clases para facilitar las modificaciones en el proceso de mantenimiento.

6. Separe al creador de la clase del usuario de la clase (el *programador cliente*). El usuario de la clase es el «consumidor» y no necesita o no quiere conocer que hay dentro de la clase. El creador de la clase debe ser un experto en diseño de clases y escribir la clase para que pueda ser usada por el programador más inexperto posible, y aún así funcionar de forma robusta en la aplicación. El uso de la librería será sencillo sólo is es transparente.

7. Cuando cree una clase, utilice nombres tan claros como sea posible. Eo objetivo debería ser que la interface del programador cliente sea conceptualmente simple. Intente utilizar nombres tan claros que los comentarios sean innecesarios. Luego, use sobrecarga de funciones y argumentos por defecto para crear un interface intuitiva y fácil de usar.

8. El control de acceso permite (al creador de la clase) cambiar tanto como sea posible en el futuro sin afectar al código del cliente en el que se usa la clase. FIXME:Is this light, mantenga todo tan privado como sea posible, y haga pública solamente la interfaz de la clase, usando siempre métodos en lugar de atributos. Ponga atributos públicos sólo cuando se vea obligado. Si una parte de su clase debe quedar expuesta a clases derivadas como protegida, proporcione una interface con funciones en lugar de exponer los datos reales. De este modo, los cambios en la implementación tendrán un impacto mínimo en las clases derivadas.

9. FIXME No caiga en FIXME:analysis paralysis. Hay algunas cosas que no aprenderá hasta que empiece a codificar y consiga algún tipo de sistema. C++ tiene mecanimos de seguridad de fábrica, dejelos trabajar por usted. Sus errores en una clase o conjunto de clases no destruirá la integridad del sistema completo.

10. El análisis y diseño debe producir, como mínimo, las clases del sistema, sus interfaces públicas, y las relaciones con otras clases, especialmente las clases base. Si su metodología de diseño produce más que eso, preguntese a si mismo si todas las piezas producidas por la metodología tiene valor respecto al tiempo de vide del programa. Si no lo tienen, no mantenga nada que no contribuya a su productividad, este es un FIXME:fact of life] que muchos métodos de diseño no tienen en cuenta.

11. Escriba primero el código de las pruebas (antes de escribir la clase), y guardelo junto a la clase. Automatice la ejecución de las pruebas con un `makefile` o herramienta similar. De este modo, cualquier cambio se puede verificar automáticamente ejecutando el código de prueba, lo que permite descubrir los errores inmediatamante. Como sabe que cuenta con esa red de seguridad, puede arriesgar haciendo cambios más grandes cuando descubra la necesidad. Recuerde que las mejoras más importantes en los lenguajes provienen de las pruebas que hace el compilador: chequeo de tipos, gestión de excepciones, etc., pero estas características no puede ir muy lejos. Debe hacer el resto del camino creando un sistema robusto rellenando las pruebas que verifican las características específicas de la clase o programa concreto.

12. Escriba primero el código de las pruebas (antes de escribir la clase) para verificar que el diseño de la clase está completo. Si no puede escribir el código de pruebas, significa que no sabe que aspecto tiene la clases. En resumen, el echo de escribir las pruebas a menudo desvela características adicionales o restricciones que necesita la clase - esas características o restricciones no siempre aparecen durante el análisis y diseño.

13. Recuerde una regla fundamental de la ingeniería del software [1]: *Todos los problemas del diseño de software se puede simplificar introduciendo una nivel más de indirección conceptual*. Esta única idea es la pase de la abstracción, la principal cualidad de la programación orientada a objetos.

14. Haga clases tan atómicas como sea posible: Es decir, dé a cada clase un propósito único y claro. Si el diseño de su clase o de su sistema crece hasta ser demasiado complicado, divida las clases complejas en otras más simples. El indicador más obvio es tamaño total: si una clase es grande, FIXME: chances are it's doing demasiado y debería dividirse.

15. Vigile las definiciones de métodos largos. Una función demasiado larga y complicada es dificil y cara de mantener, y es problema que esté intentado hacer demasiado trabajo por ella misma. Si ve una función así, indica que, al menos, debería dividirse en múltiples funciones. También puede sugerir la creación de una nueva clase.

16. Vigile las listas de argumentos largas. Las llamadas a función se vuelven difíciles de escribir, leer y mantener. En su lugar, intente mover el método a una clase donde sea más apropiado, y/o pasele objetos como argumentos.

17. No se repita. Si un trozo de código se repite en muchos métodos de las clases derivadas, ponga el código en un método de la clase base e invóquelo desde las clases derivadas. No sólo ahorrará código, también facilita la propagación de los cambios. Puede usar una función inline si necesita eficiencia. A veces el descubrimiento de este código común añadirá funcionalidad valiosa a su interface.

18. Vigile las sentencias `switch` o cadenas de `if-else`. Son indicadores típicos de *código dependiente del tipo*, lo que significa que está decidiendo qué código ejecutar basándose en alguna información de tipo (el tipo exacto puede no ser obvio en principio). Normalemente puede reemplazar este tipo de código por herencia y polimorfismo; una llamada a una función polimórfica efectuará la comprobación de tipo por usted, y hará que el código sea más fiable y sencillo de extender.

19. Desde el punto de vista del diseño, busque y distinga cosas que cambian y cosas que no cambian. Es decir, busque elementos en un sistema que podrían cambiar sin forzar un rediseño, después encapsule esos elementos en clases. Puede aprender mucho más sobre este concepto en el capítulo *Dessign Patterns* del Volumen 2 de este libro, disponible en www.BruceEckel.com [2]

20. Tenga cuidado con las FIXME *discrepancia*. Dos objetos semánticamente diferentes puede tener acciones idénticas, o responsabilidades, y hay una tendencia natural a intentar hacer que una sea subclase de la otra sólo como beneficio de la herencia. Ese se llama discrepancia, pero no hay una justificación real para forzar una relación superclase/subclase donde no existe. Un solución mejor es crear una clase base general que produce una herencia para las dos como clases derivadas - eso require un poco más de espacio, pero sigue beneficiandose de la herencia y probablemente hará un importante descubrimiento sobre el diseño.

[1] Que me explicó Andrew Koening.

[2] (N. de T.) Está prevista la traducción del Volumen 2 por parte del mismo equipo que ha traducido este volumen. Visite FIXME

21. Tenga cuidado con la FIXME: *limitación* de la herencia. Los diseños más límpios añaden nuevas capacidades a las heredadas. Un diseño sospechoso elimina capacidades durante la herencia sin añadir otras nuevas. Pero las reglas están hechas para romperse, y si está trabajando con una librería antigua, puede ser más eficiente restringir una clase existente en sus subclases que restructurar la jerarquía de modo que la nueva clase encaje donde debería, sobre la clase antigua.

22. No extienda funcionalidad fundamental por medio de subclases. Si un elemento de la interfaz es esecial para una clase debería estár en la clase base, no añadido en una clase derivada. Si está añadiendo métodos por herencia, quizá debería repensar el diseño.

23. Menos es más. Empiece con una interfaz mínima a una clase, tan pequeña y simple como necesite para resolver el problema que está tratando, pero no intente anticipar todas las formas en las que se *podría* usar la clase. Cuando use la clase, descubrirá formas de usarla y deberá expandir la interface. Sin embargo, una vez que que la clase esté siendo usada, no podrá reducir la interfaz sin causar problemas al código cliente. Si necesita añadir más funciones, está bien; eso no molesta, únicamente obliga a recompilar. Pero incluso si los nuevos métodos reemplazan las funcionalidad de los antiguos, deje tranquila la interfaz existente (puede combinar la funcionalidad de la implementación subyacente si lo desea. Si necesita expandir la interfaz de un método existente añadiendo más argumentos, deje los argumentos existentes en el orden actual, y ponga valores por defecto a todos los argumentos nuevos; de este modo no perturbará ninguna de las llamadas antiguas a esa función.

24. Lea sus clases en voz alta para estar seguro que que suenan lógicas, refiriendose a las relación entre una clase base y una clase derivada com «es-un» y a los objetos miembro como «tiene-un».

25. Cuando tenga que decidir entre herencia y composición, pregunte si necesita hacer upcast al tipo base. Si la respuesta es no, elija composición (objetos miembro) en lugar de herencia. Esto puede eliminar la necesidad de herencia múltiple. Si hereda, los usuarios pensarán FIXME:they are supposed to upcast.

26. A veces, se necesita heredar para acceder a miembros protegidos de una clase base. Esto puede conducir a una necesidad de herencia múltiple. Si no necesita hacer upcast, primero derive una nueva clase para efectuar el acceso protegido. Entonces haga que la nueva clase sea un objeto miembro dentro de cualquier clase que necesite usarla, el lugar de heredar.

27. Típicamente, una clase base se usará principalmente para crear una interface a las clases que hereden de ella. De ese modo, cuando cree una clase base, haga que por defecto los métodos sean virtuales puros. El destructor puede ser también virtual puro (para forzar que los derivadas tengan que anularlo explicitamente), pero recuerde poner al destructor un cuerpo, porque todos destructores de la jerarquía se ejecutan siempre.

28. Cuando pone un método virtual puro en una clase, haga que todos los métodos de la clase sean también viruales, y ponga un constructor virtual. Esta propuesta evita sorpresas en el comportamiento de la interfaz. Empiece a quitar la palabra `virtual` sólo cuando esté intentando optimizar y su perfilador haya apuntado en esta dirección.

29. Use atributos para variaciones en los valores y métodos virtuales para variaciones en el comportamiento. Es decir, si encuentra una clase que usa atributos estáticos con métodos que cambian de comportamiento basandose en esos atributos, probablemente deberia rediseñarla para expresar las diferencias de comportamiento con subclases y métodos virtuales anulados.

30. If debe hacer algo no portable, cree una abstracción para el servicio y póngalo en una clase. Este nivel extra de indirección facilita la portabilidad mejor que si se distribuyera por todo el programa.

31. Evite la herencia múltiple. Estará a salvo de malas situaciones, especialmente cuando repare las interfaces de clases que están fuera de su control (vea el Volumen 2). Debería ser un programador experimentado antes de poder diseñar con herencia múltiple.

32. No use herencia privada. Aunque, está en el lenguaje y parece que tiene una funcionalidad ocasional, ello implica ambigüedades importantes cuando se combina con comprobación dinámica de tipo. Cree un objeto miembro privado en lugar de usar herencia privada.

33. Si dos clases están asociadas entre si de algún modo (como los contenedores y los iteradores). intente hacer que una de ellas sea una clase amiga anidada de la otro, tal como la Librería Estándar C++ hace con los interadores dentro de los contenedores (En la última parte del Capítulo 16 se muestran ejemplos de esto). No solo pone de manifiesto la asociación entre las clases, también permite que el nombre de la clase se pueda reutilizar anidándola en otra clase. La Librería Estándar C++ lo hace definiendo un clase iterador anidada dentro de cada clase contenedor, de ese modo los contenedores tienen una interface común. La otra razón por la que querrá anidar una clase es como parte de la implementación privada. En ese caso, el anidamiento es beneficioso para ocultar la implementación más por la asociación de clases y la prevención de la contaminación del espacio de nombres citada arriba.

34. La sobrecarga de operadores en sólo «azucar sintáctico:» una manera diferente de hacer una llamada a función. Is sobrecarga un operador no está haciendo que la interfaz de la clase sea más clara o fácil de usar, no lo haga. Cree sólo un operador de conversión automática de tipo. En general, seguir las directrices y estilo indicados en el Capítulo 12 cuando sobrecargue operadores.

35. No sea una víctima de la optimización prematura. Ese camino lleva a la locura. In particular, no se preocupe de escribir (o evitar) funciones inline, hacer algunas funciones no virtuales, afinar el código para hacerlo más eficiente cuando esté en las primer fase de contrucción del sistema. El objetivo principal debería ser probar el diseño, a menos que el propio diseño requiera cierta eficiencia.

36. Normalmente, no deje que el compilador cree los constructores, destructores o el `operator=` por usted. Los diseñadores de clases siempre deberían decir qué debe hacer la clase exactamente y mantenerla enteramente bajo su control. Si no quiere costructor de copia u `operator=`, declarelos como privados. Recuerde que si crea algún constructor, el compilador un sintetizará un constructor por defecto.

37. Si su clase contiene punteros, debe crear el constructor de copia, el `operator=` y el destructor de la clase para que funcione adecuadamente.

38. Cuando escriba un constructor de copia para una clase derivada, recuerde llamar explícitamente al constructor de copia de la clase base (también cuando se

usan objetos miembro). (Vea el Capítulo 14.) Si no lo hace, el constructor por defecto será invocado desde la case base (o el objeto miembro) y con mucha probabilidad no hará lo que usted espera. Para invocar el constructor de copia de la clase base, pásele el objeto derivado desde el que está copiando:

```
Derived(const Derived& d) : Base(d) { // ...
```

39. Cuando escriba un operador de asignación para una clase derivada, recuerde llamar explícitamente al operador de asignación de la clase base. (Vea el Capítulo 14.) SI no lo hace, no ocurrirá nada (lo mismo es aplicable a los objetos miembro). Para invocar el operador de asignación de la clase base, use el nombre de la clase base y el operador de resolución de ámbito:

```
    Derived& operator=(const Derived& d) {
    Base::operator=(d);
```

40. Si necesita minimizar las recompilaciones durante el desarrollo de un proyecto largo, use FIXME: demostrada en el Capítulo 5, y eliminela solo si la eficiencia en tiempo de ejecución es un problema.

41. Evite el preprocesador. Use siempre `const` para substitución de valores e inlines para las machos.

42. Mantenga los ámbitos tan pequeños como sea posible de modo que la visibilidad y el tiempo de vidad de los objetos sea lo más pequeño posible. Esto reduce el peligro de usar un objeto en el contexto equivocado y ello supone un bug dificil de encontrar. Por ejemplo, suponga que tiene un contenedor y un trozo de código que itera sobre él. Si copia el código para usarlo otro contenedor, puede que accidentalmente acabe usando el tamaño del primer contenedor como el límite superior del nuevo. Pero, si el primer contendor estuviese fuera del ámbito, podría detectar el error en tiempo de compilación.

43. Evite las variables globales. Esfuercese en pones los datos dentro de clases. En más probable que aparezcan funciones globales de forma natural que variables globales, aunque puede que después descubra que una función global puede encajar como método estático de una clase.

44. Si necesita declara una clase o función de una librería, hágalo siempre incluyendo su fichero de cabecera. Por ejemplo, si quiere crear una función para escribir en un `ostream`, no declare nunca el `ostream` por usted mismo, usando una especificación de tipo incompleta como esta:

```
    class ostream;
```

    Este enfoque hace que su código sea vulnerabla a cambios en la representación. (Por ejmplo, `ostream` podrías ser en realidad un `typedef`.) En lugar de lo anterior, use siempre el ficheor de cabecera:

```
    #include <iostream>
```

    Cuando cree sus propias clases, si una librería es grande, proporciones a sus usuarios una versión abreviada del fichero de cabecera con especificaciones de tipo incompletas (es decir, declaraciones de los nombres de las clases) para

los casos en que ellos puedan necesitar usar únicamente punteros. (eso puede acelerar las compilaciones.)

45. Cuando elija el tipo de retorno de una operador sobrecargado, considere que ocurrirá if se encadenan expresiones. Retorne una copia o referencia al valor (`return *this`) de modo que se pueda usar e una expresión encadenada (`A = B = C`). Cuando defina el `operator=`, recuerde que `x=x`.

46. Cuando escriba una función, pase los argumentos por referencia constante como primera elección. Siempre que no necesite modificar el objeto que está pasando, esta es la mejor práctica porque es tan simple como si lo parasa por valor pero sin pagar el alto precio de construir y destruir un objeto local, que es lo que ocurre cuando se pasa por valor. Normalmente no se querrá preocupar demasiado de las cuestiones de eficiencia cuando esté diseñando y contruyendo su sistema, pero este hábito es una ganancia segura.

47. Tenga cuidado con los temporarios. Cuando esté optimizando, busque creaciones de temporarios, especialmente con sobrecarga de operadores. Si sus constructores y destructores son complicados, el coste de la creació y destrucción de temporarios puede ser muy alto. Cuando devuelva un valor en una función, intente siempre contruir el objeto «en el sitio» (*in place*) con una llamada al constructor en la sentencia de retorno:

```
return MyType(i, j);
```

mejor que

```
MyType x(i, j);
return x;
```

La primera sentencia `return` (también llamada *optimización de valor de retorno*) evita una llamada al constructor de copia y al destructor.

48. Cuando escriba constructores, considere las excepciones. En el mejor caso, el constructor no hará nada que eleve un excepción. En ese escenario, la clasé será compuesta y heredará solo de clases robustas, de modo que ellas se limpiarán automáticamente si se eleva una excepción. Si requiere punteros, usted es responsable de capturar sus propias excepciones y de liberar los recursos antes de elevar una excepción en su constructor. Si un contructor tiene que fallar, la acción apropiada es elevar una excepción.

49. En los constructores, haga lo mínimo necesario. No solo producirá una sobrecarga menor al crear objetos (muchas de las cuales pueden quedar fuera del control del programador), además la probabilidad de que eleven excepciones o causen problemas será menor.

50. La responsabilidad del destructor es la de liberar los recursos solicitados durante la vida del objeto, no sólo durante la construcción.

51. Utilice jerarquías de excepciones, preferiblemente derivadas de la jerarquía de excepción estándar de C++ y anidelas como clases públicas con la clase que eleva la excepción. La persona que captue las excepciónes puede capturar los tipos específicos de excepciones, seguida del tipo base. Si añade una nueva excepción derivada, el código de cliente anterior seguirá capturando la excepción por medio del tipo base.

52. Eleve las excepciones por valor y capturelas por referencia. Deje que el mecanismo de gestión de excepciones haga la gestión de memoria. Si eleva punteros como objetos en la excepción que han sido creados en el montículo, el que capture la excepción debe saber como liberar la excepción, lo cual implica un acoplamiento perjudicial. Si captura las excepciones por valor, causará que se creen temporarios; peor, las partes derivadas de sus objetos-excepción se pueden partir al hacer upcasting por valor.

53. No escriba sus propias clases plantilla a menos que debe. Mire primero en la Librería Estándar de C++, después en librerías de propósito específico. Adquiera habilidad en su uso y conseguirá incrementar mucho su productividad.

54. Cuando cree plantillas, escriba código que no dependa del tipo y ponga ese código en una clase base no-plantilla para evitar que el código aumente de tamaño sin necesidad. Por medio de herencia o composición, puede crear plantillas en las que el volumen de código que contienen es dependiente del tipo y por tanto esencial.

55. No use las funciones de `<stdio>`, como por ejemplo `printf()`. Aprenda a usar iostreams en su lugar; son FIXME:type-safe y type-extensible, y mucho más potentes. El esfuerzo se verá recompensado con regularidad. En general, use siempre librerías C++ antes que librerías C.

56. Evite los tipos predefinidos de C. El soporte de C++ es por compatibilidad con C, pero son tipos mucho menos robustos que las clases C++, de modo que pueden complicar la depuración.

57. Siempre que use tipos predefinidos para variables globales o automáticas, no los defina hasta que pueda inicializarlos. Defina una variable por línea. Cuando defina punteros, ponga el '*' al lado del nombre del tipo. Puede hacerlo de forma segura si define una variable por línea. Este estilo suele resultar menos confuso para el lector.

58. Garantize que tiene lugar la inicialización en todos los aspectos de su programa. Inicialice todos los atributos en la lista de inicialización del constructor, incluso para los tipo predefinidos (usando los pseudo-constructores). Usar la lista de inicialización del constructor es normalmente más eficiente cuando se inicializan subobjetos; si no se hace se invocará el constructor por defecto, y acabará llamando a otros métodos (probablemnte el `operator=`) para conseguir la inicialización que desea.

59. No use la forma `MyType a = b;` para definir un objeto. Esta es una de la mayores fuentes de confusión porque llama a un contructor en lugar de al `operator=`. Por motivos de claridad, sea específico y use mejor la forma `MyType a(b);`. Los resultados son idénticos, pero el lector no se podrá confundir.

60. Use los moldes explícitos descritos en el Capítulo 3. Un molde reemplaza el sistema normal de tipado y es un punto de error. Como los moldes explícitos separan los un-molde-lo hace-todo de C en clases de moldes bien-marcados, cualquiera que depure o mantenga el código podrá encontrar fácilmente todo los sitios en los que es más probable que sucedan errores lógicos.

61. Para que un programa sea robusto, cada componente debe ser robusto. Use todas las herramientas que proporciona C++: control de acceso, excepciones, constantes, comprobación de tipos, etc en cada clase que cree. De ese modo podrá pasar de una forma segura al siguiente nivel de abstracción cuando construya su sistema.

62. Use las constantes con corrección. Esto permite que el compilador advierta de errores que de otro modo serían sutiles y difíciles de encontrar. Esta práctica requiere de cierta disciplina y se debe usar de modo consistente en todas sus clases, pero merece la pena.

63. Use la comprobación de tipos del compilador en su beneficio. Haga todas las compilaciones con todos los avisos habilitados y arregle el código para eliminar todas las advertencias. Escriba código que utilice los errores y advertencias de compilación (por ejemplo, no use listas de argumentos variables, que eliminar todas los comprobaciones de tipos). Use `assert()` para depurar, pero use excepciones para los errores de ejecución.

64. Son preferibles los errores de compilación que los de ejecución. Intente manejar un error tan cerca del punto donde ocurre como sea posible. Es mejor tratar el error en ese punto que elevar una excepción. Capture cualqueir excepción en el manejador más cercano que tenga suficiente información para tratarla. Haga lo que pueda con la excepción en el nivel actual; si no puede resolver el problema, relance la excepción. (Vea el Volumen 2 si necesita más detalles.)

65. Si está usando las especificaciones de excepción (vea el Volumen 2 de este libro, disponible en www.BruceEckel.com, para aprender sobre manejo de excepciones), instale su propia función `unexpected()` usando `set_unexpected()`. Su `unexpected()` debería registrar el error y relanzar la excepción actual. De ese modo, si una función existente es reemplazada y eleva excepciones, dispondrá de un registro de FIXME:culprint y podrá modificar el código que la invoca para manejar la excepción.

66. Cree un `terminate()` definida por el usuario (indicando un error del programador) para registrar el error que causó la excepción, después libere los recursos del sistema, y termine el programa.

67. Si un destructor llama a cualquier función, esas funciones podrían elevar excepciones. Un destructor no puede elevar una excepción (eso podría ocasionar una llamada a `terminate()`, lo que indica un error de programación), así que cualquier destructor que llame a otras funciones debe capturar y tratar sus propias excepciones.

68. No «decore» los nombres de sus atributos privados (poniendo guiones bajos, notación húngara, etc.), a menos que tenga un montón de valores globales ya existentes; en cualquier otro caso, deje que las clases y los espacios de nombres definan el ámbito de los nombres por usted.

69. Ponga atención a la sobrecarga. Una función no debería ejecutar código condicionalmente basandose en el valor de un argumento, sea por defecto o no. En su lugar, debería crear dos o más métodos sobrecargados.

70. Oculte sus punteros dentro de clases contenedor. Dejelos fuera sólo cuando vaya a realizar operaciones con ellos. Los punteros ha sido siempre la mayor fuente de errores. Cuando use `new`, intente colocar el puntero resultante en un contenedor. Es preferible que un contenedor «posea» sus punteros y sea responsable de la limpieza. Incluso mejor, envuelva un puntero dentro de una clase; si aún así quiere que parezca un puntero, sobrecargue `operator->` y `operator*`. Si necesita tener un puntero normal, inicialicelo siempre, preferiblemente con la dirección de un objeto, o cero si es necesario. Asignele un cero cuando le libere para evitar liberaciones múltiples.

71. No sobrecargue los `new` y `delete` globales. Hágalo siempre en cada clase. Sobrecargar las versiones globales affecta la proyecto completo, algo que sólo los creadores del proyecto debería controlar. Cuando sobrecargue `new` y `delete` en las clases, no asume que conoce el tamaño del objeto; alguien puede heredar de esa clase. Use el argumento proporcionado. Si hace algo especial, considere el efecto que podría tener en las clases derivadas.

72. Evite el troceado de objetos. Prácticamente nunca tiene sentido hacer upcast de un objeto por valor. Para evitar el upcast por valor, use métodos virtuales puros en su clase base.

73. A veces la agregación simple resuelve el problema. Un FIXME:«sistema conforme al pasajero» en una línea aérea consta en elementos desconectados: asiento, aire acondicionado, video, etc., y todavía necesita crear muchos más en un avión. ¿Debe crear miembros privados y construir una nueva interfaz completa? No - en este caso, los componentes también son parte de la interfaz pública, así que deberían ser objetos miembros públicos. Esos objetos tienen sus propias implementaciones privadas, que continúan seguras. Sea consciente de que la agregación simple no es una solución usan a menudo, pero que puede ocurrir.

C: Lecturas recomendadas

C.1. Sobre C

Thinking in C: Foundations for Java & C++, por Chuck Allison (un seminario en CDROM de MindView, Inc. , 2000, incluido al final de este libro y disponible también en www.BruceEckel.com). Se trata de un curso que incluye lecciones y transparencias sobre los conceptos básicos del lenguaje C para preparar al lector a aprender Java o C++. No es un curso exhaustivo sobre C; sólo contiene lo necesario para cambiarse a esos otros lenguajes. Unas secciones adicionales sobre esos lenguajes concretos introducen al aspirante a programador en C++ o en Java, a sus características. Requisitos previos: alguna experiencia con un lenguaje de alto nivel, como Pascal, BASIC, Fortran, o LISP (sería posible avanzar por el CD sin ese bagaje, pero el curso no está pensado para servir de introducción básica a la programación).

C.2. Sobre C++ en general

The C++ Programming Language, 3ª edición, por Bjarne Stroustrup (Addison-Wesley 1997).Hasta cierto punto, el objetivo de la obra que tiene en sus manos es permitirle usarel libro de Bjarne a modo de referencia. Dado que contiene la descripción del lenguaje por su propio autor, es típicamente ahí donde se mira para resolver dudas sobre qué se supone que C++ debe o no debe hacer. Cuando empiece a dominar el lenguaje y esté preparado para pasar a las cosas serias, lo necesitará.

C++ Primer, 3ª Edición, por Stanley Lippman y Josee Lajoie (Addison-Wesley 1998). Ha dejado de ser una introducción; se ha convertido en un voluminoso libro muy detallista, y es uno de los que consulto junto con el de Stroustrup cuando intento resolver una cuestión.«Pensar En C++» debe proporcionar una base para entender *C++ Primer* así como el libro de Stroustrup.

C & C++ Code Capsules, por Chuck Allison (Prentice-Hall, 1998).Ese libro presupone unconocimiento de C y C++, y trata cuestiones que ya hayan sido quebraderos decabeza, o que no logró zanjar adecuadamente a la primera. La obra soluciona lagunas tanto en C como en C++.

The C++ Standard.Ese es el documento en el que el comité ha trabajado tanto durante años. No es gratis, desgraciadamente. Pero por lo menos se puede adquirir en formato PDF por sólo $18 en www.cssinfo.com.

C.2.1. Mi propia lista de libros

Aparecen a continuación ordenados por fecha de publicación. No todos están a la venta actualmente.

Computer Interfacing with Pascal & C(publicado por mí, vía Eisys, en 1988. Disponible únicamente a través de www.BruceEckel.com). Es una introducción a la electrónica desde los días en los que CP/M era aun el rey y MSDoS sólo un principiante. Utilicé lenguajes de alto nivel y a menudo el puerto paralelo del ordenador para pilotar varios proyectos electrónicos. Se trata de una adaptación de mis columnas en la primera y mejor revista para la que trabajé, *Micro Cornucopia* (retomando las palabras de Larry o_Brien, editor durante muchos años de *Software Development Magazine*: la mejor revista de electrónica jamás publicada -¡hasta daban los planos para fabricar un robot a partir de una maceta!). Desgraciadamente, *MicroC* dejó de existir mucho antes de que apareciese el Internet. Crear ese libro fue una experiencia editorial muy gratificante para mí.

Using C++ (osborne/McGraw-Hill 1989). Fue uno de los primeros libros publicados acerca de C++. Está agotado y ha sido reemplazado por su segunda edición, bajo el nuevo título «C++ Inside & out.»

C++ Inside & out (osborne/McGraw-Hill 1993).Como se indicó antes, es en realidad la segunda edición de «Using C++». El lenguaje C++ que describe el libro es bastante correcto, pero data de 1992 y «Pensar En C++» está llamado a sustituirlo. Puede saber más acerca de ese libro y conseguir el código fuente en www.BruceEckel.com.

Thinking in C++, 1ª edition (Prentice-Hall 1995).

Black Belt C++, the Master's Collection, Bruce Eckel, editor (M&T Books 1994).Agotado. Está constituido por una serie de capítulos escritos por personas de prestigio sobre la base de sus presentaciones en el coloquio sobre C++ durante la Conferencia sobre Desarrollo de Software que yo presidí. La portada del libro me llevó a ejercer desde entonces más control sobre el diseño de las portadas.

Thinking in Java, 2ª edición (Prentice-Hall, 2000). La primera edición de ese libro ganó el Premio a la Productividad del *Software Development Magazine* y también el *Premio del Editor 1999* del *Java Developer_s Journal*. Se puede descargar desde www.BruceEckel.com.

C.3. Los rincones oscuros

Estos libros profundizan en aspectos del lenguaje, y ayudan a evitar los típicos errores inherentes al desarrollo de programas en C++.

Effective C++(2ª Edición, Addison-Wesley 1998) y «More Effective C++» (Addison-Wesley 1996), por Scott Meyers. La obra clásica e indispensable para resolver los problemas serios y diseñar mejor código en C++. He intentado capturar y plasmar muchos de los conceptos de esos libros en *Pensar en C++*, pero no pretendo haberlo logrado. Cualquiera que dedica tiempo a C++ acaba teniendo esos libros. También disponible en CDRoM.

Ruminations on C++ , por Andrew Koenig y Barbara Moo (Addison-Wesley, 1996).Andrew trabajó personalmente con Stroustrup en muchos aspectos del lenguaje C++ y es por tanto una voz muy autorizada. Me encantaron sus incisivos comentarios y he aprendido mucho con él, tanto por escrito como en persona, a lo largo de los años.

Large-Scale C++ Software Design , por John Lakos(Addison-Wesley, 1996).Trata temas y contesta a preguntas con las que uno se encuentra durante la creación de grandes proyectos, y a menudo de pequeños también.

C++ Gems editor (SIGS Publications, 1996). Una selección de artículos extraídos de *The C++ Report*.

The Design & Evolution of C++ , por Bjarne Stroustrup (Addison-Wesley 1994).Aclaraciones del inventor de C++ acerca de por qué tomó ciertas decisiones durante su diseño. No es esencial, pero resulta interesante.

C.4. Sobre Análisis y Diseño

Extreme Programming Explained por Kent Beck (Addison-Wesley 2000).¡Adoro ese libro! Si,sé que tengo tendencia a tomar posturas radicales, pero siempre había intuido que podía haber un proceso de desarrollo de programas muy diferente, y mucho mejor, y pienso que *XP* se acerca bastante a ello. El único libro que me impactó de forma similar, fue *PeopleWare* (descrito a continuación), que trata de los entornos y la interacción con la cultura de las empresas. *Extreme Programming Explained* habla de programación, y echa abajo la mayoría de las cosas, incluso los recientes «hallazgos». Llega al punto de decir que los dibujos están bien mientras que no se les dedique demasiado tiempo y se esté dispuesto a tirarlos a la basura. (observen que ese libro no lleva el «sello de certificación UML» en su portada). Comprendería que alguien decidiese si quiere trabajar o no para una compañía, basándose sólo en el hecho que usan XP. Es un libro pequeño, con capítulos cortos, fácil de leer, y que da mucho que pensar. Uno empieza a imaginarse trabajando en esa atmósfera y vienen a la mente visiones de un mundo nuevo.

UML Distilled por Martin Fowler (2ª edición, Addison-Wesley, 2000).Cuando se descubre UML por primera vez, resulta intimidante porque hay tantos diagramas y detalles. Según Fowler, la mayoría de esa parafernalia es innecesaria, así que se queda sólo con lo esencial. Para la mayoría de los proyectos, sólo se necesitan unos pocos instrumentos gráficos, y el objetivo de Fowler es llegar a un buen diseño en lugar de preocuparse por todos los artefactos que permiten alcanzarlo. Es un libro corto, muy legible; el primer libro que debería conseguir si necesita entender UML.

The Unified Software Development Process por Ivar Jacobsen, Grady Booch, y James Rumbaugh (Addison-Wesley 1999). Estaba mentalizado para que no me gustase ese libro. Parecía tener todos los ingredientes de un aburrido texto universitario. Me quedé gratamente sorprendido - solo unos islotes dentro del libro contienen explicaciones que dan la impresión que los conceptos no han quedado claros para los propios autores. La mayoría del libro es no solamente claro, sino agradable. Y lo mejor de todo, es que el proceso tiene realmente sentido. Esto no es *Extreme Programming* (y no tiene su claridad acerca de los tests) pero también forma parte del mastodonte UML - incluso si usted no consigue hacer adoptar *XP*, la mayoría de la gente se ha subido al carro de "UML es bueno" (independientemente de su nivel de experiencia real con él) así que podría conseguir que lo adopten. Pienso que ese libro debería ser el buque insignia del UML, y es el que se debe de leer después del *UML Distilled* de Fowler en cuanto se desee tener más nivel de detalle.

Antes de elegir método alguno, es útil enriquecer su perspectiva través de los que no están intentando vender ninguno. Es fácil adoptar un método sin entender realmente lo que se desea conseguir con él o lo que puede hacer por uno. otras personas lo están usando, lo cual parece una buena razón. Sin embargo, los humanos tienen un extraño perfil psicológico: si quieren creer que algo va a solucionar sus problemas, lo van a probar. (Eso se llama experimentación, que es una cosa buena) Pero si eso no les resuelve nada, redoblarán sus esfuerzos y empezarán a anunciar por todo lo alto su fabuloso descubrimiento. (Eso es negación de la realidad, que no es bueno) La idea parece consistir en que si usted consigue meter a más gente en el mismo barco, no se sentirá solo, incluso si no va a ninguna parte (o se hunde). No estoy insinuando que todas las metodologías no llevan a ningún lado, pero hay

que estar armado hasta los dientes con herramientas mentales que ayuden a seguir en el modo de experimentación («Esto no funciona, vamos a probar otra cosa») y no en el de negación («No, no es problema. Todo va maravillosamente, no necesitamos cambiar»). Creo que los libros siguientes, leídos antes de elegir un método, le proporcionarán esas herramientas.

Software Creativity, por Robert Glass (Prentice-Hall, 1995).Ese es el mejor libro que he leído que describa una visión de conjunto sobre el debate de las metodologías. Consta de una serie de ensayos cortos y artículos que Glass ha escrito o comprado (P.J. Plauger es uno de los que contribuyen al libro), que reflejan sus numerosos años dedicados a pensar y estudiar el tema. Son amenos y de la longitud justa para decir lo necesario; no divaga ni aburre al lector. Pero tampoco vende simplemente aire; hay centenares de referencias a otros artículos y estudios. Todos los programadores y jefes de proyecto deberían leer ese libro antes de caer en el espejismo de las metodologías.

Software Runaways: Monumental Software Disasters, por Robert Glass (Prentice-Hall 1997).Lo realmente bueno de ese libro es que expone a la luz lo que nunca contamos: la cantidad de proyectos que no solo fracasan, sino que lo hacen espectacularmente. Veo que la mayoría de nosotros aún piensa «Eso no me va a pasar a mí» (o «Eso no volverá a pasarme») y creo que eso nos desfavorece. Al tener siempre en mente que las cosas pueden salir mal, se está en mejor posición para hacerlas ir bien.

Object Lessons por Tom Love (SIGS Books, 1993). otro buen libro para tener «perspectiva».

Peopleware , por Tom Demarco y Timothy Lister (Dorset House, 2ª edición 1999).A pesar de que tiene elementos de desarrollo de software, ese libro trata de proyectos y equipos de trabajo en general. Pero el énfasis está puesto en las personas y sus necesidades, y no en las tecnologías. Se habla de crear un entorno en el que la gente esté feliz y productiva, en lugar de decidir las reglas que deben seguir para convertirse perfectos engranajes de una máquina. Esta última actitud, pienso yo, es lo que más contribuye a que los programadores sonrían y digan sí con la cabeza cuando un método es adoptado y sigan tranquilamente haciendo lo mismo que siempre.

Complexity, by M. Mitchell Waldrop (Simon & Schuster, 1992). Relata el encuentro entre un grupo de científicos de diferentes disciplinas en Santa Fe, Nuevo Méjico, para discutir sobre problemas reales que como especialistas no podían resolver aisladamente (el mercado bursátil en economía, la formación inicial de la vida en biología, por qué la gente se comporta de cierta manera en sociología, etc.). Al reunir la física, la economía, la química, las matemáticas, la informática, la sociología, y otras ciencias, se está desarrollando un enfoque multidisciplinar a esos problemas. Pero más importante aun, una nueva forma de pensar en esos problemas extremadamente complejos está apareciendo: alejándose del determinismo matemático y de la ilusión de poder escribir una fórmula que prediga todos los comportamientos, hacia la necesidad de observar primero y buscar un patrón para después intentar emularlo por todos los medios posibles. (El libro cuenta, por ejemplo, la aparición de los algoritmos genéticos). Ese tipo de pensamiento, creo yo, es útil a medida que investigamos formas de gestionar proyectos de software cada vez más complejos.