El lenguaje de programación C++

Bjarne Stroustrup

«... y tú, Marcus, me has dado muchas cosas; ahora voy a darte yo un buen consejo. Sé muchos. Abandona el juego de ser siempre Marcus Cocoza. Te has preocupado demasiado por Marcus Cocoza, y en realidad has sido esclavo y prisionero suyo. No has hecho nada sin considerar antes cómo podía afectar a la felicidad y el prestigio de Marcus Cocoza. Tenías siempre mucho miedo de que Marcus hiciera una tontería o se aburriera. ¿Y qué importaba? Todo el mundo hace tonterías... Me gustaría que fueras espontáneo, que tu corazón se iluminara de nuevo. A partir de ahora debes ser no una, sino muchas personas, tantas como puedas imaginar... »

Karen Blixen
«The Dreamers»

1

Notas para el lector

«Ha llegado el momento», dijo la Morsa,
«de hablar de muchas cosas.»
L. Carroll

Estructura del libro — aprendizaje de C++ — el diseño de C++ — eficiencia y estructura — nota filosófica — nota histórica — uso de C++ — C y C++ — sugerencias para programadores de C — sugerencias para programadores de C++ — reflexiones sobre programación en C++ — consejos —referencias

1.1 Estructura del libro

El presente libro consta de seis partes:

Introducción: Los capítulos 1 a 3 ofrecen una visión de conjunto del lenguaje C++, los estilos clave de programación que soporta y la biblioteca C++ estándar.

Parte I: Los capítulos 4 a 9 proporcionan una introducción didáctica a los tipos predefinidos de C++ y a los recursos básicos para construir programas a partir de ellos.

Parte II: Los capítulos 10 a 15 son una introducción didáctica a la programación orientada a objetos y genérica con C++.

Parte III: En los capítulos 16 a 22 se presenta la biblioteca C++ estándar.

Parte IV: En los capítulos 23 a 25 se estudian temas relacionados con el diseño y el desarrollo de software.

Apéndices: Los apéndices A, B y C contienen detalles sobre los aspectos técnicos del lenguaje.

El capítulo 1 ofrece una visión de conjunto del libro, algunos consejos sobre cómo usarlo e información fundamental sobre C++ y su uso. Animo al lector a que lo hojee, lea lo que le parezca interesante y vuelva a él tras leer otras partes del libro.

Los capítulos 2 y 3 ofrecen una visión de conjunto de los conceptos y características prin-

var al lector para que dedique tiempo a los conceptos fundamentales y las características básicas del lenguaje, mostrándole lo que se puede expresar usando todo el lenguaje C++. Cuando menos, estos capítulos deben convencer al lector de que C++ no es (sólo) C, y de que C++ ha recorrido un largo camino desde la primera y segunda ediciones de este libro. El capítulo 2 sirve para familiarizarse a alto nivel con C++. Se analizan las características del lenguaje que soportan abstracción de datos, programación orientada a objetos y programación genérica. El capítulo 3 introduce los principios básicos y los componentes principales de la biblioteca estándar. Ello me permite usar componentes de la biblioteca estándar en los capítulos siguientes, y al lector le permite usar componentes de la biblioteca en los ejercicios, en lugar de trabajar directamente con características predefinidas, de menor nivel.

Los capítulos introductorios ofrecen un ejemplo de una técnica general que se aplica a lo largo de todo el libro: para posibilitar un análisis más directo y realista de alguna técnica o característica, en ocasiones presento primero brevemente un concepto y más adelante lo analizo en profundidad. Este planteamiento me permite ofrecer ejemplos concretos antes de presentar un tratamiento más general del tema. Así pues, la organización del libro refleja la idea de que habitualmente aprendemos mejor avanzando de lo concreto a lo abstracto, aun cuando, al mirar atrás, lo abstracto parezca sencillo y evidente.

La Parte I describe el subconjunto de C++ que soporta estilos de programación tradicionalmente hechos en C o Pascal. Estudia los tipos, expresiones y estructuras de control fundamentales para los programas C++. También se analiza la modularidad, soportada por espacios de nombres (en inglés, *namespaces*), archivos fuente y manejo de excepciones. Doy por supuesto que el lector está familiarizado con los conceptos fundamentales de programación usados en la Parte I. Por ejemplo, explico los recursos de C++ para expresar recursión e iteración pero no dedico mucho tiempo a explicar la utilidad de esos conceptos.

La Parte II describe los recursos de C++ para definir y usar tipos nuevos. Se presentan en ella las clases concretas y abstractas (interfaces) (capítulos 10 y 12), junto con la sobrecarga de operadores (capítulo 11), el polimorfismo y el uso de jerarquías de clases (capítulos 12 y 15). El capítulo 13 estudia las plantillas, es decir, los recursos de C++ para definir familias de tipos y funciones. En él se explican las técnicas básicas que se usan para proporcionar contenedores, tales como listas, y para soportar programación genérica. El capítulo 14 presenta el manejo de excepciones, analiza las técnicas para manejo de errores y ofrece estrategias de tolerancia a fallos. Parto de la hipótesis de que el lector, o no está muy familiarizado con la programación orientada a objetos y la programación genérica, o podría beneficiarse de una explicación sobre cómo son soportadas por C++ las principales técnicas de abstracción; explico también las técnicas en sí mismas. En la Parte IV se avanza más en esa misma dirección.

La Parte III presenta la biblioteca estándar C++. Su finalidad es facilitar la comprensión del uso de la biblioteca, mostrar técnicas generales de diseño y programación, e indicar cómo extender la biblioteca. La biblioteca proporciona contenedores (como ***list***, ***vector*** y ***map***; capítulos 16 y 17), algoritmos estándar (como ***sort***, ***find*** y ***merge***; capítulos 18 y 19), cadenas[1] (capítulo 20), Entrada/Salida (capítulo 21) y soporte para computación numérica (capítulo 22)

La Parte IV analiza cuestiones que surgen cuando se usa C++ en el diseño e implementación de grandes sistemas de software. El capítulo 23 se centra en problemas relacionados con el diseño y la gestión. En el capítulo 24 se analiza la relación entre el lenguaje de programación C++ y los problemas de diseño. El capítulo 25 presenta algunas formas de usar las clases en el diseño.

El Apéndice A es una gramática de C++ con unas cuantas anotaciones. En el Apéndice B se analiza la relación entre C y C++, y entre el Estándar C++ (también llamado ISO C++ y ANSI C++) y las versiones que lo han precedido. El Apéndice C presenta algunos ejemplos de aspectos técnicos del lenguaje.

[1] A lo largo de todo el libro, se utiliza «cadena» (en inglés, *string*) como sinónimo de «secuencia de caracteres». *(N. de los T.)*

1.1.1 Ejemplos y referencias

En este libro se hace más hincapié en la organización del programa que en la escritura de los algoritmos. En consecuencia, he evitado emplear algoritmos ingeniosos o difíciles de comprender. Habitualmente un algoritmo trivial es más apropiado para ilustrar un aspecto de la definición del lenguaje o una cuestión relacionada con la estructura del programa. Así, por ejemplo, uso el algoritmo de ordenación directa cuando, en código real, sería mejor el método rápido. A menudo forma parte de los ejercicios una reimplementación con un algoritmo más adecuado. En código real, una llamada a una función de biblioteca suele ser más apropiada que el código utilizado en el libro para ilustrar características del lenguaje.

Los ejemplos de manual dan necesariamente una visión deformada del desarrollo de software. Al clarificar y simplificar los ejemplos, desaparecen las complejidades debidas a la escala. No encuentro nada que pueda sustituir a la escritura de programas de tamaño real para formarse una idea exacta de cómo es un lenguaje de programación. Este libro se centra en las características del lenguaje, las técnicas básicas a partir de las cuales se componen todos los programas y las reglas de composición.

La selección de ejemplos refleja mis antecedentes en compiladores, bibliotecas y simuladores. Los ejemplos son versiones simplificadas de lo que se encuentra en el código real. La simplificación es necesaria para evitar que el lenguaje de programación y los aspectos de diseño se pierdan en los detalles. No hay ejemplos «vistosos» sin equivalente en el mundo real. Siempre que ha sido posible, he relegado al Apéndice C los ejemplos relacionados con aspectos técnicos del lenguaje en los que se usan variables llamadas ***x*** e ***y***, tipos ***A*** y ***B***, y funciones ***f***() y ***g***().

En los ejemplos de código se usa un tipo de letra de anchura proporcional para los identificadores. Por ejemplo:

```
#include<iostream>
int main()
{
    std::cout << "Hola, nuevo mundo!\n";
}
```

A primera vista este estilo de presentación puede parecer «antinatural» a los programadores acostumbrados a ver el código en tipos de letra de anchura constante. Sin embargo, los tipos de anchura proporcional se consideran en general mejores que los de anchura constante para la presentación de texto. Usar un tipo de anchura proporcional me permite también presentar el código con menos saltos de línea ilógicos. Además, los experimentos que

he realizado demuestran que la mayor parte de la gente encuentra enseguida el nuevo estilo más fácil de leer.

Siempre que es posible, las características del lenguaje y la biblioteca C++ se presentan en su contexto de uso y no en la forma árida de un manual. Las características del lenguaje presentadas y el detalle con que se describen reflejan mi punto de vista sobre lo que se necesita para un uso efectivo de C++. Un libro complementario, *The Annotated C++ Language Standard,* escrito en colaboración con Andrew Koenig, contiene la definición completa del lenguaje, junto con comentarios cuyo propósito es hacerlo más accesible. Lógicamente, debería haber otro libro complementario, *The Annotated Library C++*. Sin embargo, dadas mis limitaciones en cuanto a tiempo y capacidad para escribir, no puedo prometer que llegue a producirlo.

Las referencias a otras partes de este libro tienen la forma §2.3.4 (capítulo 2, apartado 3, subapartado 4), §B.5.6 (Apéndice B, subapartado 5.6) y §6.6[10] (capítulo 6, ejercicio 10). La cursiva se usa en ocasiones para destacar palabras (por ejemplo, «un literal de cadena *no* es aceptable»), para señalar la primera aparición de conceptos importantes (por ejemplo, *polimorfismo*), para símbolos no terminales de la gramática C++ (por ejemplo, *sentencia-for*) y para los comentarios en los ejemplos de código. Se usa cursiva negrita para representar identificadores, palabras clave y valores numéricos en los ejemplos de código (por ejemplo, ***class***, ***contador*** y ***1712***).

1.1.2 Ejercicios

Los ejercicios se encuentran al final de los capítulos. En su mayoría consisten en escribir un programa. El lector debe escribir siempre código suficiente para que sea compilada y ejecutada una solución con al menos tres casos de prueba. La dificultad de los ejercicios varía considerablemente, por lo que se indica la dificultad estimada para cada uno mediante una escala exponencial, según la cual si un ejercicio (*1) le lleva al lector diez minutos, un ejercicio (*2) puede llevarle una hora y un ejercicio (*3) un día. El tiempo necesario para escribir y probar un programa depende más de la experiencia del lector que del ejercicio en sí. Un ejercicio (*1) llevaría un día si el lector tuviera que familiarizarse primero con un nuevo sistema informático para ejecutarlo. Por el contrario, un ejercicio (*5) podría hacerlo en una hora alguien que casualmente tuviera a mano la colección de programas adecuada.

Como fuente de ejercicios extra para la Parte I se puede usar cualquier libro sobre programación en C, y para las partes II y III se pueden usar libros sobre estructuras de datos y algoritmos.

1.1.3 Nota sobre implementación

El lenguaje usado en este libro es «C++ puro», tal como se define en el estándar C++ [C++, 1997]. En consecuencia, los ejemplos deben funcionar sobre cualquier implementación C++. Los principales fragmentos de programas del libro se probaron usando diversas implementaciones C++. Los ejemplos en los que se usan características adoptadas recientemente en C++ no compilaron en todas las implementaciones. Sin embargo, carece de interés mencionar qué implementación falló en la compilación de qué ejemplos. Semejante información quedaría enseguida anticuada porque los implementadores están trabajando mucho para garantizar que sus implementaciones acepten correctamente todas las características de C++. En el Apéndice B encontrará el lector sugerencias sobre cómo manejarse con compiladores C++ más antiguos y con código escrito para compiladores C.

1.2 Aprendizaje de C++

Lo más importante para aprender C++ es concentrarse en los conceptos y no perderse en los detalles técnicos del lenguaje. La finalidad de aprender un lenguaje de programación es mejorar como programador, es decir, ser más eficaz en el diseño e implementación de los sistemas nuevos y en el mantenimiento de los antiguos. Por ello, es mucho más importante comprender las técnicas de programación y diseño que conocer los detalles; ese conocimiento llega con el tiempo y la práctica.

C++ soporta diversos estilos de programación. Todos se basan en una comprobación estática de tipos rigurosa y la mayoría pretende alcanzar un alto nivel de abstracción y una representación directa de las ideas del programador. Todos ellos pueden lograr sus propósitos de manera eficaz, al tiempo que mantienen la eficiencia de espacio y de tiempo de ejecución. Un programador que proceda de un lenguaje diferente (C, Fortran, Smalltalk, Lisp, ML, Ada, Eiffel, Pascal o Modula-2) debe tener en cuenta que, para obtener el máximo beneficio de C++, debe dedicar tiempo a aprender e interiorizar estilos y técnicas de programación adecuados a C++. Lo mismo puede decirse de los programadores habituados a usar versiones anteriores y menos expresivas de C++.

La aplicación irreflexiva a otro lenguaje de técnicas eficaces en uno conduce generalmente a la obtención de código torpe, de bajo rendimiento y difícil de mantener. Escribirlo resulta además sumamente frustrante, porque cada línea de código y cada mensaje de error del compilador recuerdan al programador que el lenguaje utilizado es diferente del «artiguo». Se puede escribir con el estilo de Fortran, C, Smalltalk, etcétera, en cualquier lenguaje, pero no resulta agradable ni económico cuando la filosofía del lenguaje es diferente. Cualquier lenguaje puede ser una productiva fuente de ideas sobre cómo escribir programas C++. Sin embargo, las ideas deben transformarse en algo que se adapte a la estructura general y el sistema de tipos de C++ para que sean efectivas en ese contexto diferente. Sobre el sistema básico de tipos de un lenguaje sólo se pueden obtener victorias pírricas.

C++ soporta una aproximación gradual al aprendizaje. Cómo se plantee el lector el aprendizaje de un nuevo lenguaje de programación dependerá de lo que ya sepa y de lo que pretenda aprender. No hay un planteamiento que sea válido para todo el mundo. La hipótesis de la que he partido es que el lector está aprendiendo C++ para mejorar como programador y diseñador. Es decir, supongo que el propósito del lector al aprender C++ no es sencillamente aprender una sintaxis nueva para hacer cosas de la forma a la que está acostumbrado, sino aprender formas nuevas y mejores de construir sistemas. Y esto hay que hacerlo gradualmente, porque la adquisición de cualquier nueva habilidad importante requiere tiempo y práctica. Pensemos en el tiempo que lleva aprender bien un idioma nuevo o aprender a tocar bien un instrumento musical. Mejorar como diseñador de sistemas es más fácil y más rápido, pero no tanto como le gustaría a la mayoría de la gente.

Como consecuencia de lo anterior, el lector va a usar C++ —a menudo para crear sistemas reales— antes de comprender todas las características y técnicas del lenguaje. Al soportar diversos paradigmas de programación (capítulo 2), C++ soporta programación productiva con diversos niveles de experiencia. Todo nuevo estilo de programación añade una herramienta más al arsenal del lector, pero cada uno de ellos es eficaz por sí solo y aumenta su eficacia como programador. C++ está organizado de modo que el lector pueda aprender sus conceptos en orden aproximadamente lineal y obtenga beneficios prácticos en el proceso. Esto es importante porque permite obtener unos beneficios más o menos proporcionales al esfuerzo empleado.

En cuanto al eterno debate sobre si es necesario aprender C antes que C++, estoy firmemente convencido de que es mejor ir directamente a C++. C++ es más seguro, más expresivo y reduce la necesidad de centrarse en técnicas de bajo nivel. Es más fácil que el lector aprenda las partes más espinosas de C que son necesarias para compensar su carencia de recursos de nivel superior una vez que ha estado en contacto con el subconjunto común de C y C++, y con algunas de las técnicas de nivel superior soportadas directamente en C++. El Apéndice B es una guía para que los programadores que pasan de C++ a C puedan enfrentarse al, llamémoslo, código heredado.

Hay varias implementaciones de C++ desarrolladas y distribuidas de forma independiente. Existen también abundantes herramientas, bibliotecas y entornos de desarrollo de software. Son incontables los libros, manuales, periódicos, revistas, informes, boletines electrónicos, listas de correo, conferencias y cursos en los que puede informarse el lector sobre los últimos avances en C++, su uso, herramientas, bibliotecas, implementaciones, etcétera. Si el lector piensa utilizar seriamente C++, le recomiendo que acceda a esas fuentes. Cada una de ellas tiene sus sesgos, por lo que conviene usar al menos dos. Vea, por ejemplo, [Barton, 1994], [Booch, 1994], [Henricson, 1997], [Koenig, 1997], [Martin, 1995].

1.3 El diseño de C++

La sencillez fue un criterio de diseño importante: siempre que hubo que elegir entre simplificar la definición del lenguaje y simplificar el compilador, se eligió lo primero. Sin embargo, se concedió gran importancia a conservar la compatibilidad con C; ello imposibilitó limpiar la sintaxis de C.

C++ no tiene tipos predefinidos de datos ni operaciones primitivas de alto nivel. Así, por ejemplo, el lenguaje C++ no proporciona un tipo de matriz con un operador de inversión ni un tipo de cadena con un operador de concatenación. Si un usuario desea un tipo así, puede definirlo en el propio lenguaje. En realidad, definir un tipo nuevo para uso general o específico de la aplicación es la actividad de programación más fundamental en C++. Un tipo definido por el usuario bien diseñado y un tipo predefinido difieren sólo en la forma en que se definen, no en la forma en que se usan. La biblioteca estándar C++ descrita en la Parte III proporciona muchos ejemplos de esos tipos y sus usos. Desde el punto de vista del usuario, hay poca diferencia entre un tipo predefinido y un tipo proporcionado por la biblioteca estándar.

En el diseño de C++ se han evitado las características susceptibles de suponer costes extra en tiempo de ejecución o en memoria aun sin ser utilizadas. Así, por ejemplo, se rechazaron construcciones que hubieran hecho necesario almacenar «información de mantenimiento» en todos los objetos, de modo que si un usuario declara una estructura compuesta por dos cantidades de 16 bits, esa estructura cabrá en un registro de 32 bits.

C++ fue diseñado para su uso en un entorno tradicional de compilación y tiempo de ejecución: el entorno de programación de C en el sistema UNIX. Afortunadamente, C++ nunca se ha restringido a UNIX; simplemente usó UNIX y C como modelo para las relaciones entre lenguaje, bibliotecas, compiladores, enlazadores, entornos de ejecución, etcétera. Ese modelo mínimo ayudó a C++ a salir airoso en prácticamente cualquier plataforma informática. Hay, sin embargo, buenas razones para usar C++ en entornos que proporcionan bastante más soporte. Se puede sacar partido de recursos como la carga dinámica, la compilación incremental y una base de datos de definiciones de tipos sin afectar al lenguaje.

Las características de C++ de comprobación de tipos y ocultación de datos se basan en el análisis de los programas en tiempo de compilación para impedir corrupciones accidentales de los datos. No proporcionan confidencialidad o protección contra alguien que incumpla deliberadamente las normas. Sin embargo, se pueden usar con libertad sin incurrir en costes de tiempo de ejecución ni de espacio. La idea es que, para que sea útil, una característica de lenguaje debe ser no sólo elegante, sino también asequible en el contexto de un programa real.

Para una descripción sistemática y detallada del diseño de C++, consulte el lector [Stroustrup, 1994].

1.3.1 Eficiencia y estructura

C++ se desarrolló a partir del lenguaje de programación C y, con escasas excepciones, conserva C como subconjunto. El lenguaje base, el subconjunto C de C++, está diseñado de forma que haya una estrecha correspondencia entre sus tipos, operadores y sentencias, y los objetos de los que se ocupan directamente los computadores: números, caracteres y direcciones. Salvo para los operadores ***new***, ***delete***, ***typeid***, ***dynamic_cast*** y ***throw***, y el *bloque-try*, las expresiones y sentencias individuales C++ no necesitan soporte de tiempo de ejecución.

C++ puede usar las mismas secuencias de llamada y retorno de funciones que C, u otras más eficientes. Cuando incluso esos mecanismos relativamente eficientes son demasiado costosos, se puede sustituir en línea una función de C++, de modo que podemos disfrutar de la comodidad notacional de las funciones sin coste extra en tiempo de ejecución.

Una de las finalidades originales de C era sustituir a la codificación en lenguaje ensamblador para las tareas más exigentes de programación de sistemas. Cuando se diseñó C++ se prestó gran atención a no comprometer lo ganado en esa área. La diferencia entre C y C++ radica fundamentalmente en el grado de énfasis sobre los tipos y la estructura. C es expresivo y permisivo. C++ es aún más expresivo. Sin embargo, para conseguir ese aumento de expresividad, hay que prestar más atención a los tipos de los objetos. Conociendo los tipos de los objetos, el compilador puede manejar correctamente expresiones en situaciones en las que, de otro modo, habríamos tenido que especificar las operaciones hasta el mínimo detalle. Conocer los tipos de los objetos permite asimismo al compilador detectar errores que, de otro modo, habrían persistido hasta la prueba, o incluso más tarde. Hay que tener en cuenta que el uso del sistema de tipos para comprobar argumentos de funciones, proteger los datos de una corrupción accidental, proporcionar nuevos tipos, proporcionar nue-

vos operadores, etcétera, no aumenta los costes en cuanto a tiempo de ejecución ni a espacio en C++.

La importancia concedida a la estructura en C++ refleja el aumento en la escala de los programas escritos desde que se diseñó C. Podemos hacer funcionar a la fuerza un programa pequeño (1.000 líneas, por ejemplo) aunque incumplamos todas las normas del buen estilo. Pero no ocurre lo mismo con un programa más grande. Si la estructura de un programa de 100.000 líneas es mala, descubriremos que se introducen nuevos errores con la misma velocidad con que se eliminan los antiguos. C++ fue diseñado para hacer posible que los programas más grandes se estructuraran de manera racional, de modo que fuera razonable que una persona se ocupara de cantidades de código mucho mayores. Además, se pretendía que una línea promedio de código C++ expresara mucho más que una línea promedio de código C o Pascal. A estas alturas, C++ ha alcanzado con creces esos objetivos.

No todos los fragmentos de código pueden estar bien estructurados, ser independientes del hardware, fáciles de leer, etcétera. C++ posee características que están ideadas para manipular recursos del hardware de manera directa y eficiente, sin tener en cuenta la seguridad ni la facilidad de comprensión. Posee asimismo recursos para ocultar ese código detrás de interfaces elegantes y seguras.

Naturalmente, el uso de C++ para programas más grandes conduce al uso de C++ por grupos de programadores. El énfasis de C++ en la modularidad, las interfaces fuertemente tipadas y la flexibilidad demuestra entonces su valor. El equilibrio de recursos de C++ para escribir programas grandes no lo tiene ningún lenguaje. Sin embargo, a medida que los programas son más grandes, los problemas asociados con su desarrollo y mantenimiento pasan de ser problemas de lenguaje a ser problemas más globales de herramientas y gestión. En la Parte IV se exploran algunas de estas cuestiones.

En este libro se hace hincapié en técnicas que sirven para proporcionar recursos de propósito general, tipos de gran utilidad, bibliotecas, etcétera. Estas técnicas servirán tanto a los diseñadores de programas pequeños como a los de programas grandes. Además, y debido a que todos los programas no triviales se componen de muchas partes semiindependientes, las técnicas de escritura de esas partes sirven a los programadores de cualquier aplicación.

El lector podría pensar que la especificación de un programa mediante el uso de una estructura de tipos más detallada da lugar a un texto fuente del programa más largo. Con C++ no es así. Un programa C++ que declare los tipos de los argumentos de las funciones, que use clases, etcétera, será habitualmente un poco más corto que el programa C equivalente que no emplee esos recursos. Cuando se usen bibliotecas, un programa C++ resultará mucho más corto que su equivalente C, suponiendo, naturalmente, que se pudiera construir un equivalente C que funcionara.

1.3.2 Nota filosófica

Un lenguaje de programación cumple dos propósitos relacionados entre sí: proporciona un vehículo para que el programador especifique acciones que deben ejecutarse y proporciona un conjunto de conceptos para que el programador los use cuando piense en lo que se puede hacer. El primer propósito requiere, idealmente, un lenguaje que esté «próximo a la máquina», de modo que todos los aspectos importantes de una máquina sean manejados con sencillez y eficiencia, en una forma que sea razonablemente obvia para el programador. El lenguaje C fue diseñado partiendo de esta idea. El segundo propósito requiere, en condiciones ideales, un lenguaje que esté «próximo al problema a resolver», de modo que sea posible expresar de forma directa y concisa los conceptos de una solución. Los recursos añadidos a C para crear C++ fueron diseñados partiendo de esta idea.

La conexión entre el lenguaje en el que pensamos/programamos y los problemas y soluciones que podemos imaginar es muy estrecha. Por esta razón, restringir las características del lenguaje con el fin de eliminar errores del programador es, como mínimo, peligroso. Como ocurre con los lenguajes naturales, tiene grandes ventajas ser al menos bilingüe. Un lenguaje proporciona a un programador un conjunto de herramientas conceptuales, si son inadecuadas para una tarea, simplemente no se usarán. Es imposible garantizar el buen diseño y la ausencia de errores por la mera presencia o ausencia de características específicas del lenguaje.

El sistema de tipos debe ser de especial utilidad para las tareas no triviales. El concepto de clase de C++ ha demostrado ser una potente herramienta conceptual.

1.4 Nota histórica

Inventé C++, escribí sus primeras definiciones y produje su primera implementación. Elegí y formulé los criterios de diseño para C++, diseñé todos sus recursos principales y me hice responsable del procesamiento de propuestas de extensiones en el comité de estándares C++.

Sin duda, C++ debe mucho a C [Kernighan, 1978]. C se conserva como subconjunto. También he conservado el énfasis de C en recursos que son de nivel suficientemente bajo para afrontar las tareas más exigentes de programación de sistemas. C, por su parte, debe mucho a su predecesor, BCPL [Richards, 1980]; de hecho, la convención de comentarios // de BCPL fue reintroducida en C++. La otra fuente importante de inspiración de C++ fue Simula67 [Dahl, 1972]; el concepto de clase (con clases derivadas y funciones virtuales) se tomó de él. La capacidad de C++ para sobrecarga de operadores y la libertad de colocar una declaración en cualquier lugar en el que pueda aparecer una sentencia recuerda a Algol68 [Woodward, 1974].

Desde la edición original de este libro el lenguaje ha sido revisado y perfeccionado ampliamente. Las áreas principales de revisión fueron los recursos de resolución de sobrecarga, enlace y gestión de memoria. Se hicieron además diversos cambios menores para aumentar la compatibilidad con C. Se añadieron varias generalizaciones y unas cuantas extensiones principales, entre las que se incluyen: herencia múltiple, funciones miembro ***static***, funciones miembro ***const***, miembros ***protected***, plantillas, manejo de excepciones, identificación del tipo en tiempo de ejecución y espacios de nombres. El motivo común de estas extensiones y revisiones fue hacer de C++ un lenguaje mejor para escribir y usar bibliotecas. La evolución de C++ se describe en [Stroustrup, 1994].

El recurso de plantillas fue diseñado originalmente para soportar contenedores de tipificación estática (tales como listas, vectores y mapas) y para soportar un uso elegante y eficiente de esos contenedores (programación genérica). Era objetivo fundamental reducir el uso de macros y moldes (*casts*, conversiones explícitas de tipo). Las plantillas se inspiraron en parte en los genéricos de Ada (tanto en sus puntos fuertes como en los débiles) y en parte en los módulos parametrizados de Clu. De forma semejante, el mecanismo para manejo de excepciones de C++ se inspiró parcialmente en Ada [Ichbiah, 1979], Clu [Liskov,

1979] y ML [Wikström, 1987]. Otros avances introducidos entre 1985 y 1995 —como la herencia múltiple, las funciones virtuales puras y los espacios de nombres— fueron sobre todo generalizaciones impulsadas por la experiencia en el uso de C++ y no ideas importadas de otros lenguajes.

Desde 1980 se han usado versiones iniciales del lenguaje, conocidas en conjunto como «C con clases» [Stroustrup, 1994]. La invención del lenguaje se debió originalmente a mi deseo de escribir programas de simulación dirigida por eventos para los que habría sido ideal Simula67, salvo por cuestiones de eficiencia. Se usó «C con clases» para proyectos importantes en los que los recursos para escribir programas con un uso mínimo de tiempo y espacio fueron sometidos a pruebas rigurosas. Faltaban la sobrecarga de operadores, las referencias, las funciones virtuales, las plantillas, las excepciones y numerosos detalles. El primer uso de C++ fuera de una organización de investigación comenzó en julio de 1983.

El nombre C++ (se pronuncia «ce más más»; *«see plus plus»*, en inglés) fue acuñado por Rick Mascitti en el verano de 1983. Alude a la naturaleza evolutiva de los cambios a partir de C; «++» es el operador de incremento de C. El nombre «C+», ligeramente más corto, es un error de sintaxis; se ha usado también como nombre de un lenguaje sin ninguna conexión con el que nos ocupa. Quienes conocen la semántica de C encuentran que C++ es inferior a ++C. El lenguaje no se llama D porque es una extensión de C y no intenta solucionar problemas suprimiendo características. Para otra interpretación más del nombre C++, consulte el lector el apéndice de [Orwell, 1949].

C++ fue diseñado originalmente para que el autor y sus amigos no tuvieran que programar en ensamblador, en C ni en diversos lenguajes modernos de alto nivel. Su finalidad principal era conseguir que fuera más fácil y más agradable para el programador individual escribir buenos programas. En los primeros años no hubo diseño en papel de C+–; el diseño, la documentación y la implementación fueron simultáneos. No hubo tampoco un «proyecto C++» ni un «comité de diseño C++». C++ evolucionó siempre para resolver problemas encontrados por los usuarios y como consecuencia de conversaciones entre el autor, sus amigos y sus colegas.

Más adelante, el crecimiento explosivo de C++ causó algunos cambios. Durante 1987 se hizo evidente que era inevitable la estandarización formal de C++ y que debíamos empezar a preparar el terreno para la labor de estandarización [Stroustrup, 1994]. El resultado fue un esfuerzo consciente por mantener el contacto entre los implementadores de compiladores de C++ y los principales usuarios mediante correo convencional y electrónico, así como en encuentros personales en conferencias sobre C++ y en otras ocasiones.

Los laboratorios AT&T Bell hicieron una contribución importante a ese esfuerzo al permitirme compartir los borradores de las versiones revisadas del manual de referencia de C++ con implementadores y usuarios. Debido a que muchas de esas personas trabajan para compañías que podrían haber sido consideradas competidoras de AT&T, no se debe subestimar la importancia de esta contribución. Una compañía con menos amplitud de miras habría originado importantes problemas de fragmentación del lenguaje sencillamente no haciendo nada. Lo que ocurrió fue que casi un centenar de individuos de decenas de organizaciones leyeron y comentaron lo que sería el manual de referencia generalmente aceptado y el documento base para el proyecto de estandarización ANSI de C++. Sus nombres aparecen en *The Annotated C++ Reference Manual* [Ellis, 1989]. Por último, en diciembre de 1989 se reunió el comité X3J16 del ANSI por iniciativa de Hewlett-Packard. En junio de 1991 la estandarización de C++ del ANSI (de ámbito estadounidense) pasó a formar parte de un esfuerzo de estandarización ISO (internacional) de C++. Desde 1990, estos comités conjuntos de estándares C++ han sido el principal foro para la evolución de C++ y el perfeccionamiento de su definición. He participado a fondo en esos comités. Concretamente, como presidente del grupo de trabajo para extensiones, fui directamente responsable de tratar las propuestas de cambios importantes de C++ y la adición de nuevas características al lenguaje. En abril de 1995 se publicó un borrador de estándar para su examen público. Para 1998 se prevé la aprobación formal de un estándar C++ internacional.

C++ ha evolucionado conjuntamente con algunas de las clases fundamentales que se presentan en este libro. Así, por ejemplo, diseñé las clases ***complex***, ***vector*** y ***stack*** junto con los mecanismos de sobrecarga de operadores. Las clases de cadena (en inglés, *string*) y de lista fueron desarrolladas por Jonathan Shopiro y por mí como parte del mismo esfuerzo. Las clases de cadena y de lista de Jonathan fueron las primeras en usarse de modo extensivo como parte de una biblioteca. La clase de cadena de la biblioteca del estándar C++ tiene sus raíces en estos esfuerzos iniciales. La biblioteca de tareas descrita en [Stroustrup, 1987] y en §12.7[11] formó parte del primer programa «C con clases» que se escribió. La escribí, así como sus clases asociadas, para soportar simulaciones de estilo Simula. La biblioteca de tareas ha sido revisada y reimplementada, sobre todo por Jonathan Shopiro, y se sigue usando mucho. La biblioteca de flujos (en inglés, *streams*) que se describe en la primera edición de este libro fue diseñada e implementada por mí. Jerry Schwarz la transformó en biblioteca de flujos de E/S (capítulo 21) usando la técnica de manipulador de Andrew Koenig (§21.4.6) y otras ideas. La biblioteca de flujos de E/S se perfeccionó luego durante el proceso de estandarización, realizando el grueso del trabajo Jerry Schwarz, Nathan Myers y Norihiro Kumagai. El desarrollo del recurso de plantillas estuvo influido por las plantillas ***vector***, ***map***, ***list*** y ***sort***, ideadas por Andrew Koenig, Alex Stepanov, otros autores y yo mismo. A su vez, el trabajo de Alex Stepanov sobre programación genérica con uso de plantillas desembocó en las partes de contenedores y algoritmos de la biblioteca estándar de C++ (§16.3, capítulo 17, capítulo 18, §19.2). La biblioteca ***valarray*** para computación numérica (capítulo 22) es fundamentalmente trabajo de Kent Budge.

1.5 Uso de C++

Centenares de miles de programadores usan C++ en prácticamente cualquier dominio de aplicación. Este uso está soportado por alrededor de una decena de implementaciones independientes, centenares de bibliotecas, centenares de manuales, diversas publicaciones técnicas, muchas conferencias e innumerables consultores. Hay formación y enseñanza a una amplia variedad de niveles.

Las primeras aplicaciones tenían generalmente un fuerte aroma a programación de sistemas. Así, por ejemplo, varios sistemas operativos importantes han sido escritos en C++ [Campbell, 1987], [Rozier, 1988], [Hamilton, 1993], [Berg, 1995], [Parrington, 1995] y muchos más tienen partes clave hechas en C++. Consideré que para C++ era esencial una rigurosa eficiencia a bajo nivel. Esto nos permite usar C++ para escribir controladores de dispositivos y software que se basa en la manipulación directa de hardware con restricciones de tiempo real. En código de ese tipo la predecibilidad del rendimiento es, como mínimo,

tan importante como la pura velocidad. A menudo lo es también la compacidad del sistema resultante. C++ fue diseñado de modo que todas las características del lenguaje sean utilizables en el código en condiciones de fuertes limitaciones de tiempo y espacio [Stroustrup, 1994, §4.5].

La mayor parte de las aplicaciones tienen tramos de código que son esenciales para un rendimiento aceptable. Sin embargo, la mayor cantidad de código no está en esos tramos. Para la mayor parte del código, la mantenibilidad, la facilidad de extensión y la facilidad de prueba es clave. El soporte de C++ para estas cuestiones ha conducido a su amplio uso siempre que la fiabilidad es indispensable y en áreas en las que las necesidades cambian significativamente a lo largo del tiempo. Son ejemplos de ello la banca, el comercio, los seguros, las telecomunicaciones y las aplicaciones militares. Durante años el control central del sistema telefónico estadounidense de larga distancia ha confiado en C++, y todas las llamadas 800 (sin coste para el abonado que llama) han sido encaminadas por un programa C++ [Kamath, 1993]. Muchas de estas aplicaciones son grandes y de larga vida. En consecuencia, la estabilidad, compatibilidad y posibilidad de cambio de escala han sido preocupaciones constantes en el desarrollo de C++. No es infrecuente encontrar programas C++ de un millón de líneas.

Al igual que C, C++ no se diseñó específicamente para computación numérica. Sin embargo, se hace con C++ mucha computación numérica, científica y de ingeniería. Una razón importante para ello es que el trabajo numérico tradicional debe combinarse a menudo con gráficos y con cálculos que se basan en estructuras de datos y no encajan en el molde Fortran tradicional [Budge, 1992][Barton, 1994]. Los gráficos y las interfaces de usuario son áreas en las que se usa mucho C++. Cualquiera que haya usado alguna vez un Apple Macintosh o un PC con Windows ha usado indirectamente C++ porque las interfaces primarias de usuario de esos sistemas son programas C++. Además, algunas de las más populares bibliotecas que soportan X para UNIX están escritas en C++. Así pues, C++ es una opción habitual para el vasto número de aplicaciones en las que la interfaz de usuario es una parte importante.

Todo lo anterior señala el que puede ser el principal punto fuerte de C++: su capacidad para ser usado con eficacia en aplicaciones que requieren trabajo en diversas áreas de aplicación. Es bastante frecuente encontrar aplicaciones que entrañan integración en red local y de gran amplitud, análisis numérico, gráficos, interacción del usuario y acceso a bases de datos. Tradicionalmente estas áreas de aplicación han sido consideradas distintas y han sido atendidas casi siempre por comunidades técnicas diferentes que usan diversos lenguajes de programación. Sin embargo, C++ se ha usado ampliamente en todas esas áreas. Además, puede coexistir con fragmentos de código y programas escritos en otros lenguajes.

C++ es ampliamente utilizado en la enseñanza y la investigación. Es algo que ha sorprendido a algunas personas que —acertadamente— señalaban que C++ no es el lenguaje más pequeño ni más limpio de los que se han diseñado. Es, sin embargo:

- suficientemente limpio para la enseñanza de los conceptos básicos,
- suficientemente realista, eficiente y flexible para proyectos exigentes,
- suficientemente accesible para organizaciones y colaboraciones que se apoyan en distintos entornos de desarrollo y ejecución,
- suficientemente amplio como vehículo para la enseñanza de conceptos y técnicas avanzados, y
- suficientemente comercial como vehículo para poner en uso profesional lo aprendido.

C++ es un lenguaje con el que el lector puede crecer.

1.6 C y C++

C fue elegido como lenguaje base para C++ porque:

[1] es versátil, conciso y de nivel relativamente bajo;
[2] es adecuado para la mayor parte de las tareas de programación de sistemas;
[3] se ejecuta en todas partes y con todas las máquinas; y
[4] se adapta al entorno de programación UNIX.

C tiene sus problemas, pero un lenguaje diseñado de la nada también los tendría y los problemas de C los conocemos. Algo importante: trabajar con C hizo posible que «C con clases» fuera una herramienta útil (aunque torpe) a los pocos meses de pensar por primera vez en añadir a C clases como las de Simula.

A medida que se generalizó el uso de C++ y se hicieron más significativos los recursos que ofrecía más allá y por encima de C, surgió una y otra vez la pregunta de si era conveniente o no mantener la compatibilidad. Es evidente que podrían haberse evitado algunos problemas si se hubiera rechazado parte de la herencia de C (véase, por ejemplo, [Sethi, 1981]). No se hizo porque:

[1] hay millones de líneas de código C que se beneficiarían de C++ siempre que no fuera necesario reescribir por completo C en C++;
[2] hay millones de líneas de código de funciones de biblioteca y software de utilidades escritas en C que podrían usarse desde/con programas C++ siempre que C++ fuera compatible con C en cuanto a enlazado de programas y tuviera una sintaxis muy similar;
[3] hay centenares de miles de programadores que conocen C y, por tanto, sólo necesitan aprender a usar las características nuevas de C++, en lugar de volver a aprender los fundamentos; y
[4] C++ y C serán usados durante años en los mismos sistemas por las mismas personas, por lo que las diferencias deben ser muy grandes o muy pequeñas, para reducir al mínimo los errores y confusiones.

La definición de C++ se ha revisado para garantizar que una construcción que es legal en C y legal en C++ tiene el mismo significado en ambos lenguajes (§B.2).

El lenguaje C también ha evolucionado, en parte por la influencia del desarrollo de C++ [Rosler, 1984]. El estándar ANSI C [C, 1990] contiene una sintaxis de declaración de funciones tomada de «C con clases». El préstamo funciona en ambos sentidos. Así, por ejemplo, el tipo de puntero ***void*** fue inventado para ANSI C e implementado por primera vez en C++. Como se prometió en la primera edición de este libro, se ha revisado la definición de C++ para eliminar incompatibilidades gratuitas; C++ es ahora más compatible con C de lo que lo era originalmente. Lo ideal era que C++ estuviera tan próximo a ANSI C como fuera posible, pero no más [Koenig, 1989]. La compatibilidad total nunca fue una meta porque ello comprometería la seguridad con respecto a los tipos, y la integración fluida de tipos definidos por el usuario y tipos predefinidos.

Conocer C no es requisito previo para aprender C++. Programar en C favorece muchas técnicas y trucos que las características del lenguaje C++ han hecho innecesarios. Así, por

ejemplo, la conversión explícita de tipos se necesita con menos frecuencia en C++ que en C (§1.6.1). Sin embargo, los *buenos* programas C tienden a ser programas C++. Por ejemplo, todos los programas de *The C Programming Language (2nd Edition)*, de Kernighan y Ritchie [Kernighan, 1988], son programas C++. La experiencia con cualquier lenguaje de tipificación estática será de ayuda para aprender C++.

1.6.1 Sugerencias para programadores de C

Cuanto mejor se conoce C, más difícil parece que es no escribir C++ en estilo C, con lo que se pierden parte de los beneficios potenciales de C++. Ruego al lector que consulte en el Apéndice B las diferencias entre C y C++. Veamos algunas de las áreas en las que C++ tiene formas mejores de hacer algo que C:

[1] Las macros casi nunca son necesarias en C++. Use el lector ***const*** (§5.4) o ***enum*** (§4.8) para definir constantes manifiestas, ***inline*** (§7.1.1) para evitar los costes extra de las llamadas a función, ***template***s (capítulo 13) para especificar familias de funciones y tipos, y ***namespace***s (§8.2) para evitar conflictos de nombres.

[2] No declare el lector una variable hasta que la necesite, de modo que pueda inicializarla inmediatamente. Una declaración puede aparecer en cualquier lugar en el que pueda aparecer una sentencia (§6.3.1), en inicializadores de la *sentencia-for* (§6.3.3) y en condiciones (§6.3.2.1).

[3] No hay que usar ***malloc***(). El operador ***new*** (§6.2.6) hace mejor el mismo trabajo y, en lugar de ***realloc***(), se debe intentar un ***vector*** (§3.8).

[4] Procure el lector evitar ***void****, la aritmética de punteros, las uniones y los moldes (en inglés, *casts*), excepto en las profundidades de la implementación de alguna función o clase. En la mayoría de los casos un molde es una indicación de error de diseño. Si el lector tiene que usar una conversión explícita de tipo, pruebe a usar uno de los «nuevos moldes» (§6.2.7) para un enunciado más preciso de lo que intenta hacer.

[5] Hay que reducir al mínimo el uso de arrays y cadenas de estilo C. A menudo se pueden usar las clases ***string*** (§3.5) y ***vector*** (§3.7.1) de la biblioteca estándar C++ para simplificar la programación con respecto al estilo tradicional de C. En general, el lector debe tratar de no construir lo que ya proporciona la biblioteca estándar.

Para cumplir las convenciones de enlazado de programas con C, debe declararse que una función C++ tiene enlace con C (§9.2.4).

Lo más importante es que el lector trate de concebir el programa como un conjunto de conceptos en interacción representados como clases y objetos, en lugar de como un manojo de estructuras de datos con funciones que juegan con sus bits.

1.6.2 Sugerencias para programadores de C++

En este momento hay muchas personas que llevan ya una década usando C++. Son muchas más las que usan C++ en un solo entorno y han aprendido a convivir con las restricciones impuestas por los primeros compiladores y las bibliotecas de primera generación. A menudo lo que se le ha escapado en estos años a un programador experimentado de C++ no es la introducción de las nuevas características como tales, sino los cambios de relaciones entre esas características que hacen viables técnicas nuevas y fundamentales de programación. Dicho de otro modo, cosas que el lector no pensó o no encontró prácticas cuando aprendió C++ podrían constituir ahora un planteamiento superior. Es algo que sólo se puede descubrir reexaminando los fundamentos del lenguaje.

Recomiendo al lector que lea los capítulos por orden. Si ya conoce el contenido de un capítulo, podrá repasarlo en unos minutos. Si no lo conoce, aprenderá algo con lo que no contaba. Por mi parte, he aprendido bastante escribiendo este libro, y sospecho que apenas ningún programador de C++ conoce todas las características y técnicas presentadas. Además, para usar bien un lenguaje hace falta una perspectiva que imponga orden en el conjunto de características y técnicas. Mediante su organización y sus ejemplos, este libro ofrece esa perspectiva

1.7 Reflexiones sobre la programación en C++

En condiciones ideales, la tarea de diseñar un programa se plantea en tres fases. Primero se llega a una comprensión clara del problema (análisis), luego se identifican los conceptos clave que intervienen en su solución (diseño) y por último se expresa esa solución en un programa (programación). Sin embargo, es frecuente que los detalles del problema y los conceptos de la solución sólo se comprendan con claridad tras el esfuerzo que supone expresarlos en un programa e intentar que éste funcione aceptablemente. Por esta razón es importante la elección del lenguaje de programación.

En la mayoría de las aplicaciones hay conceptos que no es fácil representar como uno de los tipos fundamentales o como una función sin datos asociados. Dado un concepto así, hay que declarar una clase para representarlo en el programa. Una clase C++ es un tipo. Es decir, especifica cómo se comportan los objetos de su clase: cómo se crean, cómo se pueden manipular y cómo se destruyen. Una clase puede especificar también cómo se representan los objetos, aunque en las etapas iniciales del diseño de un programa esto no debe ser una preocupación importante. La clave para escribir buenos programas es diseñar las clases de modo que cada una de ellas represente con limpieza un solo concepto. A menudo esto significa que hay que centrarse en cuestiones del tipo: ¿Cómo se crean los objetos de esta clase? ¿Pueden copiarse y/o destruirse los objetos de esta clase? ¿Qué operaciones pueden aplicarse a esos objetos? Si no hay respuestas satisfactorias para estas preguntas, es probable que el concepto no estuviera «limpio» desde el principio. Sería entonces buena idea pensar más en el problema y la solución propuesta para él, en lugar de empezar inmediatamente a «codificar» los problemas.

Los conceptos más fáciles de abordar son los que tienen una formalización matemática tradicional: números de todo tipo, conjuntos, formas geométricas, etcétera. E/S orientada a texto, cadenas, contenedores básicos, los algoritmos fundamentales sobre esos contenedores y algunas clases matemáticas forman parte de la biblioteca estándar C++ (capítulo 3, §.6.1.2). Existe, además, una enorme variedad de bibliotecas que soportan conceptos generales y específicos de dominio.

Los conceptos no existen en el vacío; siempre hay agrupamientos de conceptos relacionados entre sí. Organizar la relación entre las clases de un programa —es decir, determinar la relación exacta que hay entre los diferentes conceptos que intervienen en una solución— suele ser más difícil que establecer inicialmente las distintas clases. El resultado

no debe ser un embrollo en el que cada clase (concepto) dependa de todas las demás. Consideremos dos clases, A y B. Relaciones como «A llama a funciones de B», «A crea B» y «A tiene un miembro B» rara vez causan problemas importantes, mientras que relaciones como «A usa datos de B» habitualmente pueden ser eliminadas.

Una de las herramientas intelectuales más potentes para manejar la complejidad es el ordenamiento jerárquico, es decir, la organización de conceptos relacionados entre sí como una estructura de árbol con el concepto más general en la raíz. En C++ las clases derivadas constituyen organizaciones de ese tipo. A menudo se puede organizar un programa como un conjunto de árboles o gráficos acíclicos dirigidos de clases. Es decir, el programador especifica una serie de clases base, cada una con su conjunto de clases derivadas. A menudo se pueden usar funciones virtuales (§2.5.5, §12.2.6) para definir operaciones para la versión más general de un concepto (clase base). En caso necesario, se puede refinar la interpretación de esas operaciones para casos especiales concretos (clases derivadas).

A veces incluso un gráfico acíclico dirigido parece insuficiente para organizar los conceptos de un programa; algunos conceptos parecen tener una dependencia mutua intrínseca. En ese caso tratamos de localizar dependencias cíclicas, de modo que no afecten a la estructura general del programa. Si el lector no puede eliminar o localizar tales dependencias mutuas, lo más probable es que se encuentre en un atolladero del que ningún lenguaje de programación puede ayudarle a salir. Si no es posible concebir relaciones fáciles de establecer entre los conceptos básicos, es probable que el programa resulte inmanejable.

Una de las mejores herramientas para desenmarañar gráficos de dependencia es separar claramente interfaz e implementación. Las clases abstractas (§2.5.4, §12.3) son la herramienta fundamental de C++ para hacerlo.

Otra forma de expresar aspectos comunes es por medio de plantillas (§2.7, capítulo 13). Una plantilla de clase especifica una familia de clases. Así, por ejemplo, una plantilla de listas especifica «lista de T», pudiendo ser «T» cualquier tipo. Así pues, una plantilla es un mecanismo para especificar cómo se genera un tipo, dado otro tipo como argumento. Las plantillas más frecuentes son clases contenedor como listas, arrays y arrays asociativos, y los algoritmos fundamentales que usan esos contenedores. Habitualmente es un error expresar la parametrización de una clase y sus funciones asociadas con un tipo que use la herencia. Se hace mejor con plantillas.

Recordemos que gran parte de las tareas de programación se pueden realizar con sencillez y claridad usando sólo tipos primitivos, estructuras de datos, funciones simples y unas cuantas clases de biblioteca. No se debe usar todo el aparato que interviene en la definición de tipos nuevos a menos que sea realmente necesario.

La pregunta: «¿Cómo se escriben buenos programas en C++?» es muy parecida a la pregunta: «¿Cómo se escribe buena prosa?». Hay dos respuestas: «Sabiendo lo que se quiere decir» y «Con la práctica, imitando la buena escritura». Ambas parecen tan apropiadas para C++ como para la prosa, y no menos difíciles de seguir.

1.8 Consejos

A continuación se ofrecen una serie de «reglas» que el lector puede tener en cuenta para aprender C++. A medida que gane dominio del lenguaje podrá adaptarlas para hacerlas más apropiadas a sus aplicaciones y su estilo de programación. Son, de manera deliberada, muy sencillas, por lo que no entran en detalles. No hay que tomarlas al pie de la letra. Para escribir un buen programa hace falta inteligencia, gusto y paciencia. No espere el lector hacerlo bien a la primera; experimente.

[1] Cuando el lector programa, crea una representación concreta de sus ideas para la solución de algún problema. Hay que dejar que la estructura del programa refleje esas ideas lo más directamente posible:
  [a] Si puede pensar en algo como una idea independiente, convierta ese «algo» en una clase.
  [b] Si puede pensar en algo como una entidad independiente, convierta ese «algo» en un objeto de alguna clase.
  [c] Si dos clases tienen una interfaz común, convierta ésta en una clase abstracta.
  [d] Si la implementación de dos clases tiene algo significativo en común, convierta esos aspectos comunes en una clase base.
  [e] Si una clase es un contenedor de objetos, conviértala en una plantilla.
  [f] Si una función implementa un algoritmo para un contenedor, conviértala en una función plantilla implementando el algoritmo para una familia de contenedores.
  [g] Si un conjunto de clases, plantillas, etcétera, guardan relación lógica, colóquelas en un espacio de nombres común.

[2] Cuando defina una clase que no implemente una entidad matemática como una matriz o un número complejo, o un tipo de bajo nivel como una lista vinculada:
  [a] No use datos globales (use miembros).
  [b] No use funciones globales.
  [c] No use miembros de datos públicos.
  [d] No use funciones amigas, salvo para evitar [a] o [c].
  [e] No ponga un «campo de tipo» en una clase; use funciones virtuales.
  [f] No use funciones en-línea, salvo como optimización significativa.

En el apartado «Consejos» de cada capítulo encontrará el lector normas prácticas más específicas o detalladas. Recuerde que esos consejos son sólo recomendaciones prácticas, no leyes inmutables. Como cualquier consejo, sólo deben aplicarse «cuando sea razonable». La inteligencia, la experiencia, el sentido común y el buen gusto son insustituibles.

Opino que las recomendaciones negativas («nunca haga tal cosa») son inútiles. En consecuencia, la mayor parte de los consejos están formulados como sugerencias sobre lo que se debe hacer, mientras que las sugerencias negativas no suelen estar formuladas como prohibiciones absolutas. No conozco ninguna característica importante de C++ que no haya visto bien utilizada. Los consejos no contienen explicaciones, sino que cada uno de ellos va acompañado de una referencia al apartado correspondiente del libro. Cuando el consejo es negativo, generalmente el apartado de la referencia proporciona una alternativa recomendada.

1.8.1 Referencias

Aunque en el texto hay unas cuantas referencias directas, se ha reunido aquí una breve relación de libros y otras publicaciones mencionadas, directa o indirectamente.

2

Recorrido por C++

Antes de nada, matemos
a todos los abogados del lenguaje.
Enrique VI, parte II

Qué es C++ — paradigmas de programación — programación por procedimientos — programación modular — compilación independiente — manejo de excepciones — abstracción de datos — tipos definidos por el usuario — tipos concretos — tipos abstractos — funciones virtuales — programación orientada a objetos — programación genérica — contenedores — algoritmos — lenguaje y programación — consejos.

2.1 Qué es C++

C++ es un lenguaje de programación de uso general con un sesgo hacia la programación de sistemas que:

- es un C mejorado,
- soporta abstracción de datos,
- soporta programación orientada a objetos y
- soporta programación genérica.

El presente capítulo explica lo que significa esto sin entrar en los detalles más sutiles de la definición del lenguaje. Su finalidad es ofrecer al lector un panorama general de C++ y las técnicas clave para usarlo, *no* proporcionarle la información detallada necesaria para comenzar a programar en C++.

Si el lector encuentra pasajes difíciles de seguir debe saltárselos y seguir leyendo. Todo se explica con detalle en capítulos posteriores. Sin embargo, si omite alguna parte de este capítulo, debe hacerse a sí mismo el favor de volver sobre ella más adelante.

El conocimiento detallado de las características del lenguaje —aunque sea de *todas* ellas— no compensa la falta de una visión de conjunto del lenguaje y las técnicas fundamentales para usarlo.

2.2 Paradigmas de programación

La programación orientada a objetos es una técnica para programar: un paradigma para escribir programas «buenos» para un conjunto de problemas. Si la denominación «lenguaje de programación orientada a objetos» significa algo, ha de referirse a un lenguaje de programación que proporciona mecanismos que soportan bien el estilo de programación orientada a objetos.

Hay aquí una distinción importante. Se dice que un lenguaje *soporta* un estilo de programación si proporciona recursos que hagan cómodo (razonablemente fácil, seguro y eficiente) usar ese estilo. Un lenguaje no soporta una técnica si hace falta un esfuerzo o una habilidad excepcional para escribir tales programas; simplemente *permite* el uso de esa técnica. Así, por ejemplo, es posible escribir programas estructurados en Fortran y programas orientados a objetos en C, pero es innecesariamente difícil hacerlo porque no son lenguajes que soporten de manera directa esas técnicas.

El soporte para un paradigma adopta no sólo la forma evidente de recursos del lenguaje que permitan el uso directo del paradigma, sino también la forma más sutil de comprobaciones en tiempo de ejecución y/o en tiempo de compilación para detectar la desviación inintencionada del mismo. La comprobación de tipos es el ejemplo más evidente; también se usan detección de la ambigüedad y comprobaciones en tiempo de ejecución para ampliar el soporte lingüístico de los paradigmas. Recursos extralingüísticos como las bibliotecas y los entornos de programación pueden aumentar el soporte de los paradigmas.

Un lenguaje no es necesariamente mejor que otro porque posea una característica que el otro no tiene. Hay muchos ejemplos de lo contrario. Lo importante no son tanto las características que posea un lenguaje, como que esas características sean suficientes para soportar los estilos de programación deseados en las áreas de aplicación deseadas:

[1] Todas las características deben estar limpia y elegantemente integradas en el lenguaje.

[2] Debe ser posible usar las características en combinación para lograr soluciones que, de otro modo, exigirían características independientes adicionales.

[3] Debe haber tan pocas características espurias y «para fines específicos» como sea posible.

[4] La implementación de una característica no debe imponer costes extra importantes a los programas que no la necesiten.

[5] Un usuario debe necesitar conocer sólo el subconjunto del lenguaje usado explícitamente para escribir un programa.

El primer principio es una llamada a la estética y la lógica. Los dos siguientes son expresión del ideal del minimalismo. Los dos últimos pueden resumirse en una expresión: «que lo que no sepas no te haga daño».

C++ fue diseñado para soportar abstracción de datos, programación orientada a objetos y programación genérica, además de técnicas tradicionales de C con esas reservas. *No* se pretendía que impusiera un estilo de programación concreto a todos los usuarios.

En los apartados siguientes se estudian algunos estilos de programación y los mecanismos clave del lenguaje que los soportan. La presentación avanza por una serie de técnicas que van desde la programación por procedimientos hasta el uso de jerarquías de clases en programación orientada a objetos y programación genérica con el uso de plantillas. Cada paradigma se construye sobre los que lo han precedido, añade algo nuevo al arsenal de herramientas del programador C++ y refleja un planteamiento de diseño demostrado.

La presentación de las características del lenguaje no es exhaustiva. Se hace más hincapié en los planteamientos de diseño y las formas de organizar los programas que en los detalles del lenguaje. En esta etapa es mucho más importante formarse una idea de lo que se puede hacer usando C++ que comprender exactamente cómo se puede lograr.

2.3 Programación por procedimientos

El paradigma original de programación es:

> *Decidir qué procedimientos se desean;*
> *usar los mejores algoritmos que se puedan encontrar.*

El centro de atención está en el procesamiento: el algoritmo necesario para realizar el cálculo deseado. Los lenguajes soportan este paradigma proporcionando recursos para pasar argumentos a las funciones y devolviendo valores de las funciones. Lo publicado en relación con esta forma de pensamiento está repleto de análisis sobre formas de pasar los argumentos, formas de distinguir diferentes tipos de argumentos, diferentes tipos de funciones (procedimientos, rutinas y macros), etcétera.

Un ejemplo típico de «buen estilo» es una función de raíz cuadrada (***sqrt***). Dado un argumento en coma flotante de doble precisión, produce un resultado. Para ello, realiza un cálculo matemático bien conocido:

```
double sqrt(double arg)
{
  // código para calcular una raíz cuadrada
}
void f()
{
  double root2 = sqrt(2);
  //...
}
```

Las llaves expresan agrupamiento en C++. En este caso indican el comienzo y el final de los cuerpos de las funciones. La doble barra inclinada abre un comentario que llega hasta el final de la línea. La palabra clave ***void*** indica que una función no devuelve ningún valor.

Desde el punto de vista de la organización del programa, las funciones se usan para crear orden en un laberinto de algoritmos. Los propios algoritmos se escriben usando llamadas a función y otros recursos del lenguaje. En los subapartados siguientes se presenta una versión reducida de los recursos fundamentales de C++ para expresar cálculos.

2.3.1 Variables y aritmética

Todos los nombres y todas las expresiones tienen un tipo que determina las operaciones que se pueden realizar con ellos. Por ejemplo, la declaración

```
int inch;
```

especifica que ***inch*** es de tipo ***int***; es decir, ***inch*** es una variable entera

Una *declaración* es una sentencia que introduce un nombre en el programa. Especifica un tipo para ese nombre. Un *tipo* define el uso adecuado de un nombre o una expresión.

C++ ofrece una serie de tipos fundamentales, que se corresponden directamente con recursos del hardware. Por ejemplo:

```
bool    // booleano, los valores posibles son verdadero y falso
char    // carácter, por ejemplo, 'a', 'z' y '9'
int     // entero, por ejemplo, 1, 42, y 1216
double  // número en coma flotante de doble precisión, por ejemplo, 3.14 y 299793.0
```

Una variable ***char*** es del tamaño natural para contener un carácter en una máquina determinada (típicamente, un byte) y una variable ***int*** es del tamaño natural para aritmética de enteros en una máquina determinada (típicamente, una palabra).

Se pueden usar los operadores aritméticos para cualquier combinación de estos tipos:

```
+    // más, tanto monario como binario
-    // menos, tanto monario como binario
*    // multiplicar
/    // dividir
%    // resto
```

Lo mismo ocurre con los operadores de comparación:

```
==   // igual
!=   // distinto
<    // menor que
>    // mayor que
<=   // menor o igual que
>=   // mayor o igual que
```

En las asignaciones y en las operaciones aritméticas C++ realiza todas las conversiones que tienen sentido entre los tipos básicos, de forma que se pueden mezclar libremente:

```
void una_funcion()      // función que no devuelve ningún valor
{
  double d = 2.2;       // inicializar número en coma flotante
  int i = 7;            // inicializar entero
  d = d+i;              // asignar suma a d
  i = d*i;              // asignar producto a i
}
```

Como en C, = es el operador de asignación y == comprueba la igualdad.

2.3.2 Pruebas y bucles

C++ proporciona un conjunto convencional de sentencias para expresar selección y bucles. Veamos, por ejemplo, una función sencilla que da un mensaje al usuario y devuelve un booleano que indica la respuesta:

```
bool accept()
{
  cout << "Desea continuar (s o n)?\n" ;    //escribir la pregunta
  char answer = 0;
  cin >> answer;                            // leer la respuesta
  if (answer == 's') return true;
  return false;
}
```

El operador << («poner en») se usa como operador de salida; ***cout*** es el flujo de salida estándar. El operador >> («obtener de») se usa como operador de entrada; ***cin*** es el flujo de entrada estándar. El tipo del operando de la derecha de >> determina qué entrada es aceptada y es el destino de la operación de entrada. El carácter \n al final de la cadena de salida representa un salto de línea.

El ejemplo anterior se podría mejorar ligeramente teniendo en cuenta una respuesta 'n':

```
bool accept2()
{
  cout << "Desea continuar (s o n)?\n" ;    //escribir la pregunta
  char answer = 0;
  cin >> answer;                            // leer la respuesta
  switch (answer) {
  case 's' :
    return true;
  case 'n' :
    return false;
  default:
    cout << "Tomo su respuesta como un no.\n" ;
    return false;
  }
}
```

Una *sentencia-switch* contrasta un valor con un conjunto de constantes. Las constantes de los casos deben ser distintas y, si el valor probado no concuerda con ninguna de ellas, se elige ***default***. No es necesario que el programador proporcione un ***default***.

Son pocos los programas que se escriben sin bucles. En este caso, nos gustaría facilitar al usuario unos cuantos intentos:

```
bool accept3()
{
  int try = 1;
  while (try < 4) {
    cout << "Desea continuar (s o n)?\n" ;    //escribir la pregunta
    char answer = 0;
    cin >> answer;                            // leer la respuesta
    switch (answer) {
    case 's' :
      return true;
    case 'n' :
      return false;
    default:
```

```
            cout << "Perdon, no lo entiendo.\n" ;
            try = try +1;
        }
    }
    cout << "Tomo su respuesta como un no. \n" ;
    return false;
}
```

La *sentencia-while* se ejecuta hasta que su condición se hace ***false***.

2.3.3 Punteros y arrays

Un array se puede declarar de la forma siguiente:

```
char v[10] ; // array de 10 caracteres
```

De forma similar, un puntero se puede declarar así:

```
char* p;  // puntero a un carácter
```

En las declaraciones, [] significa «array de» y * significa «puntero a». Todos los arrays tienen **0** como límite inferior, por lo que **v** tiene diez elementos, **v[0]...v[9]**. Una variable puntero puede contener la dirección de un objeto del tipo adecuado:

```
p = &v[3];      // p apunta al cuarto elemento de v
```

El signo & monario es el operador «dirección de».

Consideremos la copia de diez elementos de un array a otro:

```
void otra_funcion()
{
  int v1[10];
  int v2[10];
  //...
  for (int i=0; i<10; ++i) v1[i]=v2[i];
}
```

Esta *sentencia-for* se puede leer como «poner ***i*** a cero, mientras ***i*** sea menor que **10**, copiar el ***i***ésimo elemento e incrementar ***i***». Cuando se aplica a una variable entera, el operador de incremento ++ simplemente añade **1**.

2.4 Programación modular

A lo largo de los años, en el diseño de programas el énfasis se ha desplazado del diseño de los procedimientos a la organización de los datos. Esto refleja, entre otras cosas, un aumento en el tamaño de los programas. Un conjunto de procedimientos afines junto con los datos que manipulan se denomina a menudo *módulo*. El paradigma de programación se convierte en:

> *Decidir qué módulos se desean;*
> *partir el programa de modo que los datos queden ocultos en los módulos.*

Este paradigma se conoce también como *principio de ocultación de datos*. Cuando no hay agrupamiento de los procedimientos con los datos relacionados, el estilo de programación por procedimientos es suficiente. Las técnicas para diseñar «buenos procedimientos» se aplican ahora también a cada uno de los procedimientos de un módulo. El ejemplo más frecuente de módulo es la definición de una pila. Los principales problemas que hay que resolver son:

[1] Proporcionar una interfaz de usuario para la pila (funciones ***push*** ( ) y ***pop*** ( )).

[2] Asegurarse de que sólo se pueda acceder a la representación de la pila (por ejemplo, un array de elementos) por medio de esta interfaz de usuario.

[3] Asegurarse de que la pila sea inicializada antes de ser usada por primera vez.

C++ proporciona un mecanismo para agrupar datos, funciones, etcétera, relacionados en espacios de nombres independientes. Por ejemplo, la interfaz de usuario de un módulo ***Stack*** podría declararse y usarse así:

```
namespace Stack {          // interfaz
  void push(char);
  char pop();
}
void f()
{
  Stack::push('c');
  if (Stack::pop() != 'c') error("imposible");
}
```

La cualificación ***Stack***:: indica que ***push*** ( ) y ***pop*** ( ) son los del espacio de nombres ***Stack***. No interferirán ni causarán confusión otros usos de esos nombres .

La definición de ***Stack*** se puede proporcionar en una parte del programa que se compile aparte:

```
namespace Stack {       // implementación
  const int max_size = 200;
  char v[max_size];
  int top = 0;
  void push(char c) { /* comprobar que no hay desbordamiento y meter c */ }
  char pop() { /* comprobar que no hay subdesbordamiento y extraer */ }
```

El punto clave de este módulo ***Stack*** es que el código de usuario está aislado de la representación de los datos de ***Stack*** por el código que implementa ***Stack***::***push*** ( ) y ***Stack***::***pop*** ( ). El usuario no necesita saber que ***Stack*** se implementa usando un array y se puede cambiar la implementación sin que ello afecte al código de usuario.

Debido a que los datos son sólo una de las cosas que se podría desear «ocultar», la noción de ocultación de datos, utilizada de forma poco cuidadosa, abarca la de *ocultación de información;* es decir, también los nombres de las funciones, los tipos, etcétera, se pueden hacer locales de un módulo. En consecuencia, C++ permite que cualquier declaración sea colocada en un espacio de nombres (§8.2).

Este módulo ***Stack*** es una forma de representar una pila. En los apartados siguientes se utilizan diversas pilas para ilustrar distintos estilos de programación

2.4.1 Compilación separada

C++ soporta la noción de C de compilación separada, que se puede usar para organizar un programa en un conjunto de fragmentos semiindependientes.

Habitualmente colocamos las declaraciones que especifican la interfaz para un módulo en un archivo con un nombre que indica su uso previsto. Así,

```
namespace Stack {          // interfaz
    void push(char);
    char pop();
}
```

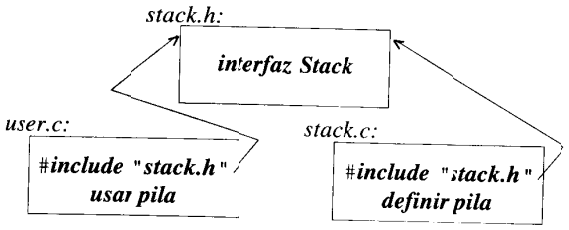
se colocaría en un archivo **stack.h** y los usuarios *incluirán* ese archivo, llamado *archivo de cabecera,* de la forma siguiente:

```
#include "stack.h"         // obtener la interfaz
void f()
{
    Stack::push('c');
    if (Stack::pop() != 'c') error("imposible");
}
```

Para ayudar al compilador a garantizar la coherencia, el archivo que proporciona la implementación del módulo **Stack** incluirá también la interfaz:

```
#include "stack.h"         // obtener la interfaz
namespace Stack {          // representación
    const int max_size = 200;
    char v[max_size];
    int top = 0;
}
void Stack::push(char c) { /* comprobar que no hay desbordamiento y meter c */ }
char Stack::pop() { /* comprobar que no hay subdesbordamiento y sacar */ }
```

El código de usuario va en un tercer archivo, por ejemplo, **user.c**. El código de **user.c** y **stack.c** comparte la información de interfaz de pilas presentada en **stack.h**, pero los dos archivos son por lo demás independientes y pueden ser compilados por separado. Los fragmentos del programa se pueden representar gráficamente así:



La compilación separada es un problema en todos los programas reales. No es simplemente una preocupación en programas que presentan como módulos recursos como una **Stack**. El uso de compilación separada no es, en sentido estricto, un problema del lenguaje; es un problema de cómo aprovechar mejor una implementación concreta del lenguaje. Sin embargo, tiene gran importancia práctica. La mejor forma de resolverlo es maximizando la modularidad, representando esa modularidad lógicamente por medio de características del lenguaje y aprovechando entonces físicamente la modularidad por medio de archivos para una compilación separada efectiva (capítulos 8 y 9).

2.4.2 Manejo de excepciones

Cuando un programa está diseñado como un conjunto de módulos, el manejo de excepciones debe considerarse teniendo en cuenta esos módulos. ¿Qué módulo es responsable del manejo de qué errores? A menudo el módulo que detecta un error no sabe qué acción emprender. La acción de recuperación depende más del módulo que ha invocado la operación que del módulo que ha encontrado el error mientras intentaba realizarla. A medida que los programas crecen, y especialmente cuando se hace un amplio uso de las bibliotecas, adquieren importancia los estándares para manejar los errores (o, más en general, las «circunstancias excepcionales»).

Volvamos al ejemplo de **Stack**. ¿Qué *habría* que hacer cuando intentamos aplicar **push**() a demasiados caracteres? Quien ha escrito el módulo **Stack** no sabe qué es lo que al usuario le gustaría que se hiciera en ese caso, y el usuario no puede detectar el problema de manera continuada (si pudiera, no se produciría desbordamiento). La solución está en que el programador de **Stack** detecte el desbordamiento y se lo comunique al usuario (desconocido). El usuario podrá entonces emprender la acción apropiada. Por ejemplo:

```
namespace Stack {          // interfaz
    void push(char);
    char pop();
    class Overflow { };    // tipo que representa las excepciones de desbordamiento
}
```

Al detectar un desbordamiento, **Stack::push**() puede invocar el código de manejo de excepciones; es decir, «lanzar una excepción **Overflow**»:

```
void Stack::push(char c)
{
    if (top == max_size) throw Overflow();
    // meter c
}
```

El **throw** transfiere el control a un manejador de las excepciones de tipo **Stack::Overflow** en alguna función que directa o indirectamente ha llamado **Stack::push**(). Para ello, la implementación desenredará la pila de llamadas de función para volver al contexto de esa llamada. Así pues, **throw** actúa como un **return** multinivel. Por ejemplo:

```
void f()
{
    // ...
```

```
try { // excepciones tratadas por el manejador definido más adelante
    while (true) Stack::push('c');
}
catch (Stack::Overflow) {
    // vaya: desbordamiento de la pila; emprender la acción adecuada
}
// ...
}
```

El bucle **while** intentará iterar indefinidamente. Por tanto, se introducirá la cláusula *catch* que proporcione un manejador para *Stack::Overflow* después de que alguna llamada de *Stack::push()* cause un **throw**.

El uso de mecanismos de manejo de excepciones puede hacer el manejo de errores más regular y legible. Véase §8.3 y el capítulo 14 para un análisis más pormenorizado.

2.5 Abstracción de datos

La modularidad es un aspecto fundamental de todos los buenos programas grandes. Se mantiene como centro de atención en todos los comentarios sobre diseño realizados a lo largo de este libro. Sin embargo, los módulos, tal como se han descrito anteriormente, no son suficientes para expresar con limpieza sistemas complejos. Voy a presentar ahora una forma de usar los módulos que proporciona una forma de tipos definidos por el usuario y a continuación mostraré cómo superar algunos problemas de ese planteamiento definiendo directamente tipos definidos por el usuario.

2.5.1 Módulos de definición de tipos

La programación con módulos conduce a una centralización de todos los datos de un tipo bajo el control de un módulo gestor de ese tipo. Si, por ejemplo, quisiéramos muchas pilas —en lugar de la única que proporciona el módulo *Stack* anterior— podríamos definir un gestor de pilas con una interfaz como la siguiente:

```
namespace Stack {
    struct Rep;                 // definición de la estructura de la pila en otro lugar
    typedef Rep& stack;
    stack create();             // hacer una nueva pila
    void destroy(stack s);      // borrar s
    push(stack s, char c);      // meter c en s
    char pop(stack s);          // sacar de s
}
```

La declaración

```
struct Rep;
```

dice que *Rep* es el nombre de un tipo, pero deja la definición del tipo para después (§5.7). La declaración

```
typedef Rep& stack;
```

da el nombre *stack* a una «referencia a *Rep*» (detalles en §5.5). La idea es que una pila se identifica por su *Stack::stack* y que los demás detalles están ocultos para los usuarios.

Una *Stack::stack* actúa en gran medida como una variable de un tipo predefinido:

```
struct Bad_pop { };
void f()
{
    Stack::stack s1 = Stack::create();      // hacer una nueva pila
    Stack::stack s2 = Stack::create();      // hacer otra nueva pila
    Stack::push(s1,'c');
    Stack::push(s2,'k');
    if (Stack::pop(s1) != 'c') throw Bad_pop();
    if (Stack::pop(s2) != 'k') throw Bad_pop();
    Stack::destroy(s1);
    Stack::destroy(s2);
}
```

Podríamos implementar esta *Stack* de diversas formas. Es importante que los usuarios no necesiten saber cómo lo hacemos. Siempre que mantengamos la interfaz sin modificar, al usuario no le afectará que nosotros decidamos reimplementar *Stack*.

Una implementación podría preasignar memoria para unas cuantas representaciones de pilas y dejar que *Stack::create()* diera una referencia a una no utilizada. *Stack::destroy()* podría marcar entonces una representación «no utilizada» de modo que *Stack::create()* pudiera reciclarla:

```
namespace Stack {                   // representación
    const int max_size = 200;
    struct Rep {
        char v[max_size];
        int top;
    };
    const int max = 16;             // número máximo de pilas
    Rep stacks[max];                // representaciones de pila preasignadas
    bool used[max];                 // used[i] es verdadero si stacks[i] está en uso
}
void Stack::push(stack s, char c) { /* comprobar desbordamiento de s y meter c */ }
char Stack::pop(stack s) { /* comprobar subdesbordamiento de s y sacar */ }
Stack::stack Stack::create()
{
    // elegir Rep no usada, marcarla como usada, inicializarla y devolver referencia a ella
}
void Stack::destroy(stack s) /* marcar s como no usada */ }
```

Lo que hemos hecho es envolver un conjunto de funciones de interfaz en torno al tipo de la representación. Cómo se comporte el «tipo de la pila» resultante dependerá en parte de

cómo definamos esas funciones de interfaz, en parte de cómo presentemos el tipo de la representación a los usuarios de ***Stack***s y en parte del diseño del tipo de la representación en sí mismo.

A menudo lo anterior está lejos de ser idóneo. Un problema importante es que la presentación al usuario de «tipos falsos» como el descrito puede variar considerablemente dependiendo de los detalles del tipo de la representación —y los usuarios deberían estar aislados del conocimiento del tipo de la representación—. Si, por ejemplo, hubiéramos elegido usar una estructura de datos más elaborada para identificar una pila, las normas para asignación e inicialización de ***Stack*** : : ***stack***s habrían cambiado de manera espectacular. Esto puede ser incluso deseable en ocasiones. Sin embargo, demuestra que simplemente hemos desplazado el problema de proporcionar ***Stack***s cómodas desde el módulo ***Stack*** al tipo de la representación ***Stack*** : : ***stack***.

Lo que es más importante, los tipos definidos por el usuario e implementados por medio de un módulo que proporciona acceso a un tipo de implementación no se comportan como los tipos predefinidos y reciben un soporte menor y diferente que ellos. Así, por ejemplo, el momento en que se puede usar una ***Stack*** : : ***rep*** está controlado por medio de ***Stack*** : : ***create*** ( ) y ***Stack*** : : ***destroy*** ( ), y no por las normas usuales del lenguaje.

2.5.2 Tipos definidos por el usuario

C++ aborda este problema permitiendo que el usuario defina directamente tipos que se comporten (casi) de la misma forma que los tipos predefinidos. Esos tipos se denominan a menudo *tipos de datos abstractos*. Prefiero hablar de *tipos definidos por el usuario*. Una definición más razonable del tipo de datos abstracto requeriría una especificación matemática «abstracta». Dada una especificación semejante, lo que aquí denominamos *tipos* serían ejemplos concretos de esas entidades verdaderamente abstractas. El paradigma de programación se convierte en:

> *Decidir qué tipos se desean;*
> *proporcionar un conjunto completo de operaciones para cada tipo.*

Cuando no hay necesidad de más de un objeto de un tipo, basta con el estilo de programación de ocultación de datos que usa módulos.

Tipos aritméticos tales como los números racionales y complejos son ejemplos habituales de tipos definidos por el usuario. Consideremos:

```
class complex {
  double re, im;
public:
  complex(double r, double i) { re=r; im=i; }  // construir complejo
                                                // a partir de dos escalares
  complex(double r) { re=r; im=0; }             // construir complejo
                                                // a partir de un escalar
  complex() { re=im=0; }                        //complejo por defecto: (0,0)
  friend complex operator+(complex, complex);
  friend complex operator-(complex, complex);   // binario
  friend complex operator-(complex);            // monario
  friend complex operator*(complex, complex);
  friend complex operator/(complex, complex);
  friend complex operator==(complex, complex);  // igual
  friend complex operator!=(complex, complex);  // distinto
  //...
};
```

La declaración de la clase (es decir, el tipo definido por el usuario) ***complex*** especifica la representación de un número complejo y el conjunto de operaciones con un número complejo. La representación es *privada:* ***re*** e ***im*** sólo son accesibles para las funciones especificadas en la declaración de la clase ***complex***. Tales funciones pueden definirse así:

```
complex operator+(complex a1, complex a2)
{
  return complex(a1.re+a2.re, a1.im+a2.im);
}
```

Una función miembro con el mismo nombre que su clase se denomina *constructor*. Un constructor define una forma de inicializar un objeto de su clase. La clase ***complex*** proporciona tres constructores. Uno hace un ***complex*** a partir de un ***double***, otro toma un par de ***doubles*** y el tercero hace un ***complex*** con un valor por defecto.

La clase ***complex*** se puede usar de la forma siguiente:

```
void f(complex z)
{
  complex a = 2.3;
  complex b = 1/a;
  complex c = a+b*complex(1,2.3);
  //...
  if (c != b) c = -(b/a)+2*b;
}
```

El compilador convierte los operadores en los que intervienen números ***complex*** en llamadas de función adecuadas. Por ejemplo, ***c*** != ***b*** significa ***operator***!=(***c***,***b***) y ***1/a*** significa ***operator/***(***complex***(***1***),***a***).

La mayor parte de los módulos, pero no todos, se expresan mejor como tipos definidos por el usuario.

2.5.3 Tipos concretos

Es posible diseñar tipos definidos por el usuario que satisfagan una amplia variedad de necesidades. Consideremos un tipo ***Stack*** definido por el usuario de la misma manera que el tipo ***complex***. Para que el ejemplo sea un poco más realista, el tipo ***Stack*** está definido para tomar como argumento su número de elementos:

```
class Stack {
  char* v;
  int top;
```

```
    int max_size;
  public:
    class Underflow { };        // usada como excepción
    class Overflow { };         // usada como excepción
    class Bad_size { };         // usada como excepción
    Stack (int s);              // constructor
    ~Stack ();                  // destructor
    void push (char c);
    char pop ();
};
```

El constructor *Stack (int)* será llamado siempre que se cree un objeto de la clase. Esto tiene en cuenta la inicialización. Si se necesita realizar alguna acción de limpieza cuando un objeto de la clase sale fuera de su ámbito, se puede declarar un complemento del constructor, denominado *destructor*[1]:

```
Stack::Stack (int s)    // constructor
{
  top = 0;
  if (10000<s) throw Bad_size ();
  max_size = s;
  v = new char [s];     // asignar elementos en la memoria libre (montículo,
                        // almacenamiento dinámico)
}
Stack::~Stack ()        // destructor
{
  delete [] v;  // liberar los elementos para posible reutilización de su espacio (§6.2.6)
}
```

El constructor inicializa una nueva variable *Stack*. Para ello, asigna memoria en la zona de almacenamiento libre (también llamada montículo o zona de almacenamiento dinámico) usando el operador *new*. El destructor limpia liberando esa memoria. Todo esto se hace sin intervención de los usuarios de *Stack*s. Los usuarios sencillamente crean y usan *Stack*s en gran medida como si fueran variables de tipos predefinidos. Por ejemplo:

```
Stack s_var1 (10);              // pila global con 10 elementos
void f (Stack& s_ref, int i)    // referencia a Stack
{
  Stack s_var2 (i);             // pila local con i elementos
  Stack* s_ptr = new Stack (20); // puntero a Stack asignada en la memoria libre
  s_var1.push ('a');
  s_var2.push ('b');
  s_ref.push ('c');
  s_ptr->push ('d');
  // ...
}
```

[1] Cuando el objeto sale de su ámbito deja de existir; es entonces cuando se llama al destructor. *(N. de los T.)*

Este tipo *Stack* se atiene a las mismas reglas en cuanto a nombre, ámbito, asignación, duración, copia, etcétera, que un tipo predefinido como *int* o *char*.

Naturalmente, las funciones miembro *push ()* y *pop ()* deben estar también definidas en algún lugar:

```
void Stack::push (char c)
{
  if (top == max_size) throw Overflow;
  v [top] = c;
  top = top + 1;
}
char Stack::pop ()
{
  if (top == 0) throw Underflow ();
  top = top - 1;
  return v [top];
}
```

Los tipos como *complex* y *Stack* se denominan *tipos concretos*, en contraste con los *tipos abstractos*, en los que la interfaz aísla más completamente al usuario de los detalles de la implementación.

2.5.4 Tipos abstractos

En la transición de *Stack* como «tipo falso» implementado por un módulo (§2.5.1) a tipo correcto (§2.5.3) se ha perdido una propiedad. La implementación no está desacoplada de la interfaz de usuario, sino que forma parte de lo que se incluiría en un fragmento de programa que usara *Stack*s. La representación es privada y, por tanto, accesible sólo a través de las funciones miembro, pero está presente. Si cambia de forma sigrificativa, el usuario tendrá que volver a compilar. Éste es el precio que se paga por tener tipos concretos que se comporten exactamente igual que tipos predefinidos. En especial, no podemos tener variables locales auténticas de un tipo sin conocer el tamaño de la representación del tipo.

Para tipos que no cambian a menudo, y cuando las variables locales proporcionan la necesaria claridad y eficiencia, lo anterior es aceptable y a menudo idóneo. Sin embargo, si deseamos aislar completamente a los usuarios de una pila de los cambios en su implementación, esta última *Stack* es insuficiente. La solución está entonces en desacoplar la interfaz de la representación y renunciar a las variables locales auténticas.

En primer lugar, definimos la interfaz:

```
class Stack {
public:
    class Underflow { };        // usada como excepción
    class Overflow { };         // usada como excepción
    virtual void push (char c) = 0;
    virtual char pop () = 0;
};
```

La palabra ***virtual*** significa «que puede ser redefinido posteriormente en una clase derivada de ésta» en Simula y C++. Una clase derivada de ***Stack*** proporciona una implementación para la interfaz ***Stack***. La curiosa sintaxis *=0* dice que alguna clase derivada de ***Stack*** *debe* definir la función. Así pues, esta ***Stack*** puede servir como interfaz para cualquier clase que implemente sus funciones ***push*** ( ) y ***pop*** ( ).

Esta ***Stack*** se podría usar así:

```
void f(Stack& s_ref)
{
  s_ref.push('c');
  if (s_ref.pop() != 'c') throw bad_stack();
}
```

Observe el lector cómo usa *f* ( ) la interfaz ***Stack*** en completa ignorancia de los detalles de implementación. Una clase que proporciona la interfaz a otras diversas clases se denomina a menudo *tipo polimórfico*.

Como era de esperar, la implementación podría constar de todo lo de la clase concreta ***Stack*** que ha quedado fuera de la interfaz ***Stack***:

```
class Array_stack : public Stack {    // Array_stack implementa Stack
  char* p;
  int max_size;
  int top;
public:
  Array_Stack(int s);
  ~Array_stack();
  void push(char c);
  char pop();
};
```

Aquí, «*:public*» puede leerse como «se deriva de», «implementa» y «es un subtipo de».

Para que una función como *f* ( ) use una ***Stack*** ignorando por completo los detalles de implementación es necesario que alguna otra función haga un objeto sobre el cual pueda operar. Por ejemplo:

```
void g()
{
  Array_stack as(200);
  f(as);
}
```

Puesto que *f* ( ) no sabe nada de ***Array_stack***s, sino que sólo conoce la interfaz ***Stack***, funcionará igualmente bien para una implementación diferente de una ***Stack***. Por ejemplo:

```
class List_stack : public Stack {   // List_stack implementa Stack
  list<char> lc;                     // lista de caracteres (biblioteca estándar) (§3.7.3)
public:
  List_stack() { }
  void push(char c) { lc.push_front(c); }
  char pop();
};

char List_stack::pop()
{
  char x = lc.front();               // obtener primer elemento
  lc.pop_front();                    // eliminar primer elemento
  return x;
}
```

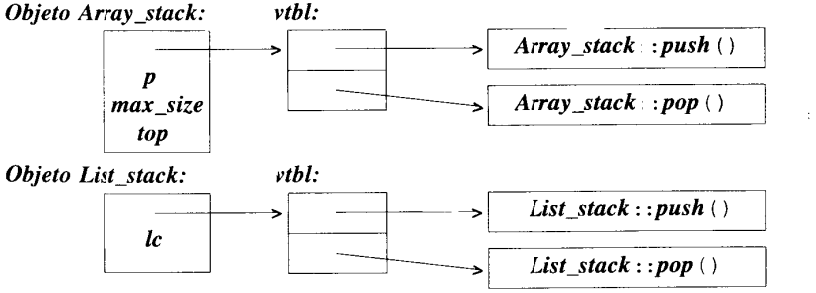
Aquí la representación es una lista de caracteres. ***lc.push_front*** (***c***) añade ***c*** como primer elemento de ***lc***, la llamada ***lc.pop_front*** ( ) elimina el primer elemento y ***lc.front*** ( ) denota el primer elemento de ***lc***.

Una función puede crear una ***List_stack*** y hacer que *f* ( ) la use:

```
void h()
{
  List_stack ls;
  f(ls);
}
```

2.5.5 Funciones virtuales

¿Cómo se resuelve la llamada ***s_set.pop*** ( ) en *f* ( ) para la definición de función adecuada? Cuando *f* ( ) es llamada desde ***h*** ( ), debe ser llamada ***List_stack*** : :***pop*** ( ). Cuando *f* ( ) es llamada desde ***g*** ( ), debe ser llamada ***Array_stack*** : :***pop*** ( ). Para lograr esta resolución, un objeto ***Stack*** debe contener información que indique la función que debe ser llamada en tiempo de ejecución. Una técnica de implementación habitual es que el compilador convierta la llamada de una función ***virtual*** en un índice de una tabla de punteros a funciones. Esa tabla se denomina habitualmente «tabla de funciones virtuales» o, sencillamente, ***vtbl***. Cada una de las clases con funciones virtuales tiene una ***vtbl*** propia que identifica sus funciones virtuales. Se puede representar gráficamente así:



Las funciones de la ***vtbl*** permiten que el objeto sea usado correctamente aun cuando su tamaño y la distribución de sus datos sean desconocidos para el autor de la llamada. Todo lo que éste necesita conocer es la localización de la ***vtbl*** en una ***Stack*** y el índice usado para cada función virtual. Este mecanismo de llamada virtual se puede hacer esencialmente tan eficiente como el mecanismo de «llamada de función normal». Su coste en cuanto a espacio es un puntero en cada objeto de una clase con funciones virtuales más una ***vtbl*** para cada una de las clases.

2.6 Programación orientada a objetos

La abstracción de datos es fundamental para el buen diseño y se mantendrá como centro de atención del diseño a lo largo del presente libro. Sin embargo, los tipos definidos por el usuario no son en sí mismos suficientemente flexibles para atender nuestras necesidades. En este apartado se va a plantear primero un problema con tipos de datos sencillos definidos por el usuario y a continuación se va a mostrar cómo resolverlo usando jerarquías de clases.

2.6.1 Problemas de los tipos concretos

Un tipo concreto, como un «tipo falso» definido por medio de un módulo, es una especie de caja negra. Una vez definida esa caja negra, no interactúa realmente con el resto del programa. No hay manera de adaptarla a nuevos usos si no es modificando su definición. Esta situación puede ser ideal, pero puede también conducir a una grave inflexibilidad. Consideremos la definición de un tipo ***Forma*** para su uso en un sistema gráfico. Supongamos por el momento que ese sistema tiene que soportar círculos, triángulos y cuadrados. Supongamos también que tenemos

```
class Punto { /* ... */ };
class Color { /* ... */ };
```

/* y */ especifican, respectivamente, el comienzo y el final de un comentario. Esta notación se puede usar para comentarios de varias líneas y para comentarios que terminan antes del final de una línea.

Podríamos definir una forma así:

```
enum Variedad { circulo, triangulo, cuadrado };      // enumeración (§4.8)
class Forma {
    Variedad k;        // campo del tipo
    Punto centro;
    Color col;
    // ...
public:
    void dibujar ( );
    void girar (int);
    // ...
};
```

El «campo del tipo» ***k*** es necesario para permitir que operaciones como ***dibujar*** ( ) y ***girar*** ( ) determinen de qué forma se trata (en un lenguaje semejante a Pascal, habría que usar un registro variante con etiqueta ***k***). La función ***dibujar*** ( ) se definiría así:

```
void Forma::dibujar ( )
{
    switch (k) {
    case circulo:
        // dibujar un círculo
        break;
    case triangulo:
        // dibujar un triángulo
        break;
    case cuadrado:
        // dibujar un cuadrado
        break;
    }
}
```

Esto es un embrollo. Funciones como ***dibujar*** ( ) tienen que «conocer» todos los tipos de formas que hay. En consecuencia, el código para cualquier función semejante aumenta cada vez que se añade una nueva forma al sistema. Si definimos una forma nueva, habrá que examinar y (posiblemente) modificar todas las operaciones con las formas. No podemos añadir una forma nueva a un sistema a menos que tengamos acceso al código fuente para todas las operaciones. Dado que añadir una forma nueva entraña «tocar» el código de todas las operaciones importantes con las formas, hacerlo requiere gran habilidad e introduce defectos potenciales en el código que maneja las demás formas (antiguas). Las posibilidades de representación de formas concretas pueden quedar muy recortadas por la necesidad de que (al menos parte de) su representación encaje en la estructura habitualmente de tamaño fijo que presenta la definición del tipo general ***forma***.

2.6.2 Jerarquías de clases

El problema está en que no hay distinción entre las propiedades generales de todas las formas (es decir, una forma tiene un color, puede dibujarse, etcétera) y las propiedades de un tipo específico de forma (un círculo es una forma que tiene un radio, es dibujado por una función de dibujo de círculos, etcétera). La expresión de esta distinción y su aprovechamiento es lo que define la programación orientada a objetos. Los lenguajes con construcciones que permiten expresar y usar esta distinción soportan programación orientada a objetos. Los demás lenguajes no la soportan.

El mecanismo de herencia (tomado por C++ de Simula) proporciona una solución. En primer lugar, especificamos una clase que define las propiedades generales de todas las formas:

```
class Forma {
    Punto centro;
    Color col;
    // ...
public:
    Punto donde ( ) { return centro; }
    void mover (Punto hacia) { centro = hacia; /* ... */ dibujar ( ); }
    virtual void dibujar ( ) = 0;
    virtual void girar (int angulo) = 0;
    // ...
};
```

Como en el tipo abstracto ***Stack*** de §2.5.4, las funciones para las que se puede definir la interfaz de llamada —pero en las que la implementación no puede ser definida todavía— son ***virtual***. En particular, las funciones ***dibujar*** ( ) y ***girar*** ( ) se pueden definir sólo para formas específicas, por lo que son declaradas ***virtual***.

Dada esta definición, podemos escribir funciones generales que manipulen vectores o punteros a formas:

```
void girar_todo (vector<Forma*>& v, int angulo)    // girar los elementos
                                                   // de v 'ángulo' grados
{
  for (int i = 0; i<v.size( ); ++i) v[i] ->girar(angulo);
}
```

Para definir una forma concreta, debemos decir que es una forma y especificar sus propiedades concretas (incluidas las funciones virtuales):

```
class Circulo : public Forma {
  int radio;
public:
  void dibujar( ) { /* ... */ };
  void girar(int) { }      // sí, la función nula
};
```

En C++ se dice que la clase ***Circulo*** es *derivada* de la clase ***Forma*** y se dice que la clase ***Forma*** es una *base* de la clase ***Circulo***. Una terminología alternativa llama a ***Circulo*** y ***Forma*** subclase y superclase, respectivamente. Se dice que la clase derivada hereda los miembros de su clase base, por lo que el uso de clases base y derivadas se denomina habitualmente *herencia*.

El paradigma de programación es:

> *Decidir qué clases se desean;*
> *proporcionar un conjunto completo de operaciones para cada clase;*
> *hacer explícitos los aspectos comunes mediante el uso de la herencia.*

Cuando no existen aspectos comunes es suficiente la abstracción de datos. La cantidad de aspectos comunes entre los tipos que se puede aprovechar usando la herencia y las funciones virtuales es la prueba definitiva de la aplicabilidad a un problema de la programación orientada a objetos. En áreas como los gráficos interactivos existe claramente un enorme ámbito para la programación orientada a objetos. En otras, como los tipos aritméticos clásicos y los cálculos basados en ellos, aparentemente casi no hay ámbito más que para abstracción de datos y los recursos necesarios para el soporte de programación orientada a objetos parecen innecesarios.

Encontrar aspectos comunes entre los tipos de un sistema no es un proceso trivial. La cantidad de aspectos comunes a explotar está afectada por la forma en que está diseñado el sistema. Cuando se diseña un sistema —e incluso cuando se escriben los requisitos del sistema— hay que buscar activamente aspectos comunes. Las clases se pueden diseñar específicamente como bloques constructivos para otros tipos y se pueden examinar las clases existentes para ver si muestran semejanzas susceptibles de ser aprovechadas en una clase base común.

Para intentos de explicar qué es la programación orientada a objetos sin recurrir a construcciones de un lenguaje de programación específico, consulte el lector [Kerr, 1987] y [Booch, 1994] en §23.6.

Las jerarquías de clases y las clases abstractas (§2.5.4) se complementan entre sí, en lugar de excluirse mutuamente (§12.5). En general, los paradigmas reunidos aquí tienden a ser complementarios y a menudo se soportan mutuamente. Así, por ejemplo, las clases y los módulos contienen funciones, mientras que los módulos contienen clases y funciones. El diseñador con experiencia aplica diversos paradigmas de acuerdo con las necesidades.

2.7 Programación genérica

Es poco probable que alguien que quiera una pila quiera siempre una pila de caracteres. Una pila es un concepto general, independiente de la noción de carácter. En consecuencia, debe representarse de manera independiente.

En general, si un algoritmo se puede expresar con independencia de los detalles de la representación, y si se puede hacer esto sin excesivo coste y sin contorsiones lógicas, hay que hacerlo.

El paradigma de programación es:

> *Decidir qué algoritmos se desean;*
> *parametrizarlos de forma que funcionen*
> *para diversos tipos y estructuras de datos adecuados.*

2.7.1 Contenedores

Podemos generalizar un tipo pila-de-caracteres y convertirlo en pila-de-lo-que-sea haciendo de él una *plantilla* y sustituyendo el tipo específico ***char*** por un parámetro de plantilla. Por ejemplo:

```
template<class T> class Stack {
  T* p;
  int max_size;
  int top;
public:
  class Underflow { };
  class Overflow { };
  Stack(int s);      // constructor
  ~Stack( );         // destructor
  void push(T);
  T pop( );
};
```

El prefijo ***template<class T>*** hace de ***T*** un parámetro de la declaración a la que sirve de prefijo.

Las funciones miembro se podrían definir de forma similar:

```
template<class T> void Stack<T>::push(T c)
{
  if (top == max_size) throw Overflow();
  v[top] = c;
  top = top + 1;
}
template<class T> T Stack<T>::pop()
{
  if (top == 0) throw Underflow();
  top = top - 1;
  return v[top];
}
```

Dadas estas definiciones, podemos usar pilas como las siguientes:

```
Stack<char>sc;          // pila de caracteres
Stack<complex>scplx;    // pila de números complejos
Stack< list<int> > sli; // pila de lista de enteros
void f()
{
  sc.push('c');
  if (sc.pop() != 'c') throw Bad_pop();
  scplx.push(complex(1,2));
  if (scplx.pop() != complex(1,2)) throw Bad_pop();
}
```

De modo similar, podemos definir listas, vectores, mapas (es decir, arrays asociativos), etcétera, como plantillas. Una clase que contiene una colección de elementos de algún tipo se denomina habitualmente *clase contenedor* o simplemente *contenedor*.

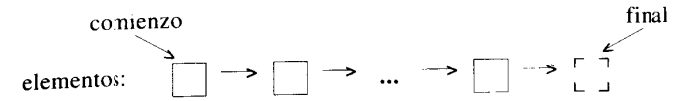
Las plantillas son un mecanismo en tiempo de compilación, por lo que su uso no supone coste de tiempo de ejecución en relación con el «código escrito a mano».

2.7.2 Algoritmos genéricos

La biblioteca estándar C++ proporciona diversos contenedores, y los usuarios pueden escribir los suyos (capítulos 3, 17 y 18). Así pues, nos encontramos con que podemos aplicar el paradigma de la programación genérica una vez más para parametrizar algoritmos por contenedores. Queremos, por ejemplo, clasificar, copiar y buscar *vector*s, *list*s y arrays sin tener que escribir funciones *sort*(), *copy*() y *search*() para cada contenedor. Al mismo tiempo, no queremos convertir a una estructura de datos específica aceptada por una única función de clasificación. En consecuencia, tenemos que encontrar una forma generalizada de definir nuestros contenedores que nos permita manipular uno de ellos sin saber exactamente qué tipo de contenedor es.

Un planteamiento, el adoptado para los contenedores y los algoritmos no numéricos en la biblioteca estándar C++ (§3.18, capítulo 18) consiste en centrarse en la noción de secuencia y manipular secuencias por medio de iteradores.

Veamos una representación gráfica de la noción de secuencia:



Una secuencia tiene un comienzo y un final. Un iterador se refiere a un elemento y proporciona una operación que hace que el iterador se refiera al siguiente elemento de la secuencia. El final de una secuencia es un iterador que se refiere a un elemento situado más allá del último de la secuencia. La representación física del «final» puede ser un elemento centinela, pero no tiene por qué serlo. En realidad, lo importante es que esta noción de secuencia cubre una amplia variedad de representaciones, incluidas listas y arrays.

Necesitamos alguna notación estándar para operaciones como «acceder a un elemento por medio de un iterador» y «hacer que el iterador se refiera al elemento siguiente». Las posibilidades obvias (una vez que se tiene la idea) son usar el operador de desreferencia * para indicar «acceder a un elemento por medio de un iterador» y el operador de incremento ++ para indicar «hacer que el iterador se refiera al elemento siguiente».

Teniendo en cuenta lo anterior, podemos escribir código así:

```
template<class Entrada, class Salida> void copy(Entrada desde,
                                                Entrada demasiado_lejos, Salida a)
{
  while (desde != demasiado_lejos) {
  *a = *desde;          // copiar el elemento al que se apunta
  ++a;                  // entrada siguiente
  ++desde;              // salida siguiente
  }
}
```

Esto copia cualquier contenedor para el que podamos definir iteradores con la sintaxis y la semántica adecuadas.

Los tipos predefinidos en C++ de array y puntero de bajo nivel tienen las operaciones adecuadas para lo anterior, por lo que podemos escribir

```
char vc1[200];  // array de 200 caracteres
char vc2[500];  // array de 500 caracteres
void f()
{
  copy(&vc1[0], &vc1[200], &vc2[0]);
}
```

Esto copia *vc1* desde su primer hasta su último elemento en *vc2* empezando en el primer elemento de *vc2*.

Todos los contenedores de la biblioteca estándar (§16.3, capítulo 17) soportan esta noción de iteradores y secuencias.

Se usan dos parámetros de plantilla, *In* y *Out*, para indicar los tipos de la fuente y el destino en lugar de un solo argumento. Esto se ha hecho porque a menudo queremos copiar de un tipo de contenedor a otro. Por ejemplo:

```
complex ac[200];
void g(vector<complex>& vc, list<complex>& lc)
{
    copy(&ac[0], &ac[200], lc.begin());
    copy(lc.begin(), lc.end(), vc.begin());
}
```

Esto copia el array en la ***list*** y la ***list*** en el ***vector***. Para un contenedor estándar, ***begin***() es un iterador que apunta al primer elemento.

2.8 Posdata

Ningún lenguaje de programación es perfecto. Por fortuna, no es necesario que un lenguaje de programación sea perfecto para que constituya una buena herramienta para la construcción de grandes sistemas. En realidad, un lenguaje de programación de uso general no puede ser perfecto para todas las innumerables tareas a que se le destina. Lo que es perfecto para una tarea suele ser gravemente defectuoso para otra, porque la perfección en un área implica especialización. Así pues, C++ fue diseñado para ser ura buena herramienta para la creación de una amplia variedad de sistemas y para permitir que una amplia variedad de ideas sean expresadas directamente.

No todo puede expresarse directamente usando las características predefinidas del lenguaje. En realidad, tampoco es eso lo ideal. Existen características del lenguaje que soportan diversos estilos y técnicas de programación. En consecuencia, la tarea de aprender un lenguaje debe centrarse en el dominio de los sistemas nativos y naturales de ese lenguaje, no en la comprensión hasta el mínimo detalle de todas las características del lenguaje.

En la programación práctica ofrece muy pocas ventajas conocer hasta las características más oscuras del lenguaje o utilizar el mayor número posible de características. Una característica del lenguaje aislada tiene poco interés. Sólo adquiere sentido e interés en el contexto que proporcionan las técnicas y las demás características. Así pues, en los siguientes capítulos, no pierda de vista el lector que la finalidad real de examinar los detalles de C++ es ser capaz de usarlos conjuntamente para soportar un buen estilo de programación en el contexto de diseños sensatos.

2.9 Consejos

[1] Que no cunda el pánico. Todo se aclarará en su momento; §2.1.

[2] No es necesario que el lector conozca todos los detalles de C++ para que escriba buenos programas; §1.7.

[3] Hay que concentrarse en las técnicas de programación, no en las características del lenguaje; §2.1.

3

Recorrido por la biblioteca estándar

¿Por qué perder el tiempo apreadiendo
cuando la ignorancia es instantánea?
Hobbes

Bibliotecas estándar — salida — cadenas — entrada — vectores — comprobación de rangos — contenedores: generalidades — algoritmos — iteradores — iteradores de E/S — transversales y predicados — algoritmos que usan funciones miembro — algoritmos: generalidades — números complejos — aritmética de vectores — biblioteca estándar: generalidades — consejos.

3.1 Introducción

Ningún programa importante se escribe única y exclusivamente con lenguaje de programación. Primero se desarrolla un conjunto de bibliotecas de apoyo que constituyen la base para el trabajo posterior.

Como continuación del anterior, el presente capítulo ofrece un recorrido rápido por los recursos básicos de biblioteca, para dar al lector una idea de lo que se puede hacer usando C++ y su biblioteca estándar. Se presentan tipos útiles de biblioteca, tales como ***string***, ***vector***, ***list*** y ***map***, así como formas útiles de usarlos. Ello me permite ofrecer ejemplos y ejercicios mejores en los capítulos siguientes. Como en el capítulo 2, animo vivamente al lector a que no se distraiga ni se desanime si no entiende completamente los detalles. La finalidad de este capítulo es ofrecer al lector una idea de lo que va a venir y ayudarle a entender los usos más sencillos de los recursos de biblioteca más útiles. En §16.12 se facilita una introducción más detallada a la biblioteca estándar.

Los recursos de biblioteca estándar descritos en este libro forman parte de cualquier implementación C++ completa. Además de la biblioteca estándar C++, la mayor parte de las implementaciones ofrecen sistemas de «interfaz gráfica de usuario», a menudo denominados GUI (*graphical user interface*) o sistemas de ventanas, para la interacción entre el usuario y el programa. De modo semejante, la mayor parte de los entornos de desarrollo de aplicaciones proporcionan «bibliotecas fundamentales» que soportan entornos de de-

sarrollo y/o ejecución «estándar» de la empresa o sector de que se trate. Tales sistemas y bibliotecas no se describen aquí. Lo que se pretende es proporcionar una descripción autónoma de C++ tal como lo define el estándar y mantener la portabilidad de los ejemplos, excepto cuando se señala expresamente lo contrario. Naturalmente, animo a los programadores a explorar todos los recursos de que dispongan en todos los sistemas a su alcance, pero esa tarea se deja a los ejercicios.

3.2 ¡Hola, mundo!

El programa C++ mínimo,

```
int main() { }
```

define una función denominada *main* que no toma argumentos y no hace nada.

Todos los programas C++ han de tener una función llamada *main*() . El programa comienza ejecutando esa función. El valor *int* devuelto por *main*() es, en su caso, el valor devuelto por el programa «al sistema». Si no se devuelve ningún valor, el sistema recibirá un valor que indique que se ha completado con éxito. Un valor distinto de cero procedente de *main*() indica fallo.

Lo habitual es que un programa produzca alguna salida. Veamos un programa que escribe *Hola, mundo*!:

```
#include <iostream>
int main()
{
    std::cout << "Hola, mundo!\n";
}
```

La línea #*include* <*iostream*> da al compilador la instrucción de que *incluya* las declaraciones de los recursos estándar de E/S que se encuentran en *iostream*. Sin esas declaraciones, la expresión

```
std::cout << "Hola, mundo!\n";
```

no tendría sentido. El operador << («poner en») escribe su segundo argumento sobre su primer argumento. En este caso, se escribe el literal de cadena "*Hola, mundo*!\n" en el flujo de salida estándar *std*::*cout*. Un literal de cadena es una secuencia de caracteres rodeada por comillas dobles. En un literal de cadena, el carácter de barra invertida \ seguido de otro carácter denota un único carácter especial. En este caso, \ es el carácter de salto de línea, de modo que los caracteres escritos son *Hola, mundo*!, seguidos de un salto de línea.

3.3 El espacio de nombres de la biblioteca estándar

La biblioteca estándar está definida en un espacio de nombres (§2.4, §8.2) llamado *std*. Es por esta razón por la que he escrito *std*::*cout*, en lugar de sólo *cout*. He indicado explícitamente que se use *standard cout*, y no otra *cout*.

Todos los recursos de biblioteca estándar son proporcionados por medio de alguna cabecera estándar semejante a <*iostream*>. Por ejemplo:

```
#include<string>
#include<list>
```

Esto da acceso a las *string* y *list* estándar. Para usarlas, se puede emplear el prefijo *std*::

```
std::string s = "No le busques tres pies al gato";
std::list<std::string>slogans;
```

Para simplificar, rara vez usaré explícitamente el prefijo *std*:: en los ejemplos. Tampoco incluiré siempre explícitamente las cabeceras necesarias. Para compilar y ejecutar los fragmentos de programa del libro, el lector debe incluir las cabeceras adecuadas (enumeradas en §3.7.5, §3.8.6 y capítulo 16). Además, debe usar el prefijo *std*:: o hacer globales todos los nombres de *std* (§8.2.3). Por ejemplo:

```
#include<string>          // hacer accesibles los recursos estándar de cadena
using namespace std;      // hacer accesibles los nombres std sin el prefijo std::
string s = "Bienaventurado el ignorante!"; // bien: la cadena es std::string
```

En general es de mal gusto descargar todos los nombres de un espacio de nombres en el espacio de nombres global. Sin embargo, en aras de la brevedad, en los fragmentos de programa usados para ilustrar características del lenguaje y de la biblioteca omito repetir las cualificaciones #*include* y *std*::. A lo largo del libro uso casi exclusivamente la biblioteca estándar, por lo que, si se utiliza un nombre de la biblioteca estándar, se trata de un uso de lo que ofrece el estándar o forma parte de una explicación sobre cómo podría definirse el recurso estándar.

3.4 Salida

La biblioteca de flujos de entrada y salida define la salida para todos los tipos predefinidos. Además, es fácil definir la salida de un tipo definido por el usuario. Por defecto, los valores de salida a *cout* son convertidos en una secuencia de caracteres. Por ejemplo,

```
void f()
{
    cout << 10;
}
```

colocará el carácter *1* seguido por el carácter *0* en el flujo de salida estándar. También lo hará

```
void (g)
{
    int i = 10;
    cout << i;
}
```

Se puede combinar salida de tipos diferentes de la forma evidente:

```
void h(int i)
{
    cout << "el valor de i es ";
    cout << i;
```

```
    cout << '\n';
}
```

Si *i* tiene el valor *10*, la salida será

```
el valor de i es 10
```

Una constante de carácter es un carácter encerrado entre comillas sencillas. Observe el lector que una constante de carácter se presenta en la salida como un carácter y no como un valor numérico. Por ejemplo,

```
void k()
{
    cout << 'a';
    cout << 'b';
    cout << 'c';
}
```

dará como salida *abc*.

La gente se cansa en seguida de repetir el nombre del flujo de salida cuando dirige a la salida elementos relacionados entre sí. Por suerte, el resultado de una expresión de salida puede ser utilizado para salidas posteriores. Por ejemplo:

```
void h2(int i)
{
    cout << "el valor de i es " << i << '\n';
}
```

Esto es equivalente a *h*(). Los flujos se explican con mayor detalle en el capítulo 21.

3.5 Cadenas

La biblioteca estándar proporciona un tipo *string* para complementar los literales de cadena usados anteriormente. El tipo *string* proporciona una serie de operaciones de cadena útiles, tales como la concatenación. Por ejemplo:

```
string s1 = "Hola";
string s2 = "mundo";
void m1()
{
    string s3 = s1 + "," + s2 + "!\n";
    cout << s3;
}
```

En este caso, *s3* se inicializa a la secuencia de caracteres

```
Hola, mundo!
```

seguida por un salto de línea. La adición de cadenas implica concatenación. Se pueden añadir cadenas, literales de cadena y caracteres a una cadena.

En muchas aplicaciones la forma de concatenación más frecuente es la adición de algo al final de una cadena. Es lo que soporta directamente la operación +=. Por ejemplo:

```
void m2(string& s1, string& s2)
{
    s1 = s1 + '\n';    // adjuntar salto de línea
    s2 += '\n';        // adjuntar salto de línea
}
```

Las dos formas de añadir algo al final de una cadena son equivalentes desde el punto de vista semántico, pero prefiero la segunda porque es más concisa y probablemente será implementada con mayor eficiencia.

Naturalmente se pueden comparar *string*s entre sí y con literales de cadena. Por ejemplo:

```
string conjuro;
void responder(const string& respuesta)
{
    if (respuesta == conjuro) {
        // hacer magia
    }
    else if (respuesta == "si") {
        // ...
    }
    // ...
}
```

La clase de cadenas de la biblioteca estándar se describe en el capítulo 20. Entre otras características útiles, proporciona la posibilidad de manipular subcadenas. Por ejemplo:

```
string name = "Niels Stroustrup";
void m3()
{
    string s = name.substr(6, 10);          // s = "Stroustrup"
    name.replace(0, 5, "Nicholas");         // el nombre se convierte
                                            // en "Nicholas Stroustrup"
}
```

La operación *substr*() devuelve una cadena que es copia de la subcadena indicada por sus argumentos. El primer argumento es un índice a la cadena (una posición) y el segundo argumento es la longitud de la subcadena deseada. Puesto que la indexación parte de *0*, *s* toma el valor *Stroustrup*.

La operación *replace*() sustituye un valor por una subcadena. En este caso, la subcadena que comienza en *0* y tiene longitud *5* es *Niels*; es sustituida por *Nicholas*. Así pues, el valor final de *name* es *Nicholas Stroustrup*. Observe el lector que no es necesario que la cadena sustitutoria tenga el mismo tamaño que la sustituida.

3.5.1 Cadenas de estilo C

Una cadena de estilo C es un array de caracteres terminado en cero (§5.2.2). Como veremos, podemos introducir fácilmente una cadena de estilo C en una *string*. Para llamar a las funciones que toman cadenas de estilo C, es necesario que podamos extraer el valor de

una ***string*** en forma de cadena de estilo C. Es lo que hace la función ***c_str*** ( ) (§20.4.1). Por ejemplo, podemos imprimir el ***name*** usando la función de salida de C ***printf*** ( ) (§21.8) de la forma siguiente:

```
void f()
{
  printf("nombre: %s\n", name.c_str());
}
```

3.6 Entrada

La biblioteca estándar ofrece ***istream***s para la entrada. Al igual que los ***ostream***s, los ***istream***s se ocupan de representaciones de cadenas de caracteres de los tipos predefinidos y pueden ampliarse fácilmente para manejar tipos definidos por el usuario.

Como operador de entrada se usa el operador >> («obtener de»); ***cin*** es el flujo de entrada estándar. El tipo del operando situado a la derecha de >> determina qué entrada es aceptada y cuál es el destino de la operación de entrada. Por ejemplo,

```
void f()
{
  int i;
  cin >> i;      // lee un entero a i
  double d;
  cin >> d;      // lee un número en coma flotante de doble precisión a d
}
```

lee un número, tal como ***1234***, de la entrada estándar a la variable entera ***i*** y un número en coma flotante, tal como ***12.34e5***, a la variable en coma flotante de doble precisión ***d***.

Veamos un ejemplo que realiza conversiones de pulgadas a centímetros y de centímetros a pulgadas. El lector introduce un número seguido por un carácter que indica la unidad de medida: centímetros o pulgadas. El programa da el valor correspondiente en la otra unidad:

```
int main()
{
  const float factor = 2.54;     // 1 pulgada es igual a 2.54 cm
  float x, in, cm;
  char ch = 0;
  cout << "introducir longitud: ";
  cin >> x;          // leer un número en coma flotante
  cin >> ch;         // leer un sufijo
  switch (ch) {
  case 'i':          // pulgada
    in = x;
    cm = x*factor;
    break;
  case 'c':          // cm
    in = x/factor;
    cm = x;
    break;
  default:
    in = cm = 0;
    break;
  }
  cout << in << " in = " << cm << " cm\n";
}
```

La *sentencia-switch* contrasta un valor con un conjunto de constantes. Para salir de la *sentencia-switch* se usan *sentencias-break*. Las constantes de caso deben ser distintas. Si el valor contrastado no coincide con ninguna de ellas, se elige ***default***. No es necesario que el programador proporcione un ***default***.

A menudo queremos leer una secuencia de caracteres. Una forma cómoda de hacerlo es leerla a una ***string***. Por ejemplo:

```
int main()
{
  string str;
  cout << "Por favor, introduzca su nombre\n";
  cin >> str;
  cout << "Hola, " << str << "!\n";
}
```

Si el lector teclea

Eric

la respuesta es

Hola, Eric!

Se puede leer una línea completa usando la función ***getline*** ( ). Por ejemplo:

```
int main()
{
  string str;
  cout << "Por favor, introduzca su nombre\n";
  cin.getline(cin, str);
  cout << "Hola, " << str << "!\n";
}
```

Con este programa, la entrada

Eric Bloodaxe

da la salida deseada:

Hola, Eric Bloodaxe!

Las cadenas estándar tienen la bonita propiedad de expandirse para contener lo que pongamos en ellas, de modo que si introducimos un par de megabytes de puntos y comas, el

programa nos devolverá páginas y páginas de puntos y comas, a menos que a nuestra máquina o sistema operativo se le agote antes algún recurso esencial.

3.7 Contenedores

La computación suele entrañar la creación de colecciones de distintas formas de objetos y la posterior manipulación de esas colecciones. Leer caracteres a una cadena e imprimir la cadena es un ejemplo sencillo. Una clase cuya finalidad principal es contener objetos se conoce comúnmente como *contenedor*. Proporcionar los contenedores adecuados para una tarea determinada y soportarlos con operaciones fundamentales útiles son pasos importantes en la construcción de cualquier programa.

Para ilustrar los contenedores más útiles de la biblioteca estándar, veamos un programa sencillo para guardar nombres y números de teléfono. Es el tipo de programa para el cual hay diferentes planteamientos que parecen «sencillos y evidentes» a personas de distinta formación.

3.7.1 *vector*

A muchos programadores de C un array predefinido de pares (nombre, número de teléfono) les parecería un punto de partida apropiado:

```
struct Entry {
    string name;
    int number;
};
Entry phone_book[1000];
void print_entry(int i)      // uso simple
{
    cout << phone_book[i].name << ' ' << phone_book[i].number << '\n';
}
```

Sin embargo, un array predefinido tiene un tamaño fijo. Si elegimos un tamaño grande, malgastamos espacio; si elegimos un tamaño más pequeño, el array se desbordará. En cualquier caso, tendremos que escribir código de gestión de memoria de bajo nivel. La biblioteca estándar proporciona un ***vector*** (§16.3) que se ocupa de eso:

```
vector<Entry> phone_book(1000);
void print_entry(int i)      // uso simple, exactamente como para un array
{
    cout << phone_book[i].name << ' ' << phone_book[i].number << '\n';
}
void add_entries(int n)      // aumentar el tamaño en n
{
    phone_book.resize(phone_book.size()+n);
}
```

La función miembro ***size***() de ***vector*** da el número de elemertos.

Observe el lector el uso del paréntesis en la cefinición de ***phone_book***. Hemos hecho un único objeto del tipo ***vector<Entry>*** y proporcionado su tamaño inicial como inicializador. Es algo muy diferente a declarar un array predefinido:

```
vector<Entry>books(1000);    // vector de 1000 elementos
vector<Entry>books[1000];    // 1000 vectores vacíos
```

Si el lector comete el error de usar [] por () al declarar un ***vector***, casi con toda seguridad su compilador capturará el error y emitirá ur mensaje de error cuando el lector intente usar el ***vector***.

Un ***vector*** es un objeto único que puede ser asignado. Por ejemplo:

```
void f(vector<Entry>& v)
{
    vector<Entry> v2 = phone_book;
    v = v2;
    // ...
}
```

Asignar un ***vector*** implica copiar sus elementos. Así pues, tras la inicialización y asignación en ***f***(), ***v*** y ***v2*** contienen una copia de cada uno de los elementos ***Entry*** de la guía telefónica. Cuando un ***vector*** contiene muchos elementos, estas asignaciones e inicializaciones, en apariencia inocentes, pueden suponer unos costes prohibitivos. Siempre que la copia no sea deseable deben utilizarse referencias o punteros.

3.7.2 Comprobación de rango

El ***vector*** de la biblioteca estándar no proporciona comprobación de rango por defecto (§16.3.3). Por ejemplo:

```
void f()
{
    int i = phone_book[1001].number;    // 1001 está fuera de rango
    // ...
}
```

Es probable que la inicialización coloque algún valor aleatorio en ***i*** en lugar de dar error. Como esto no es deseable, voy a usar una adaptación de comprobación de rango sencilla de ***vector***, denominada ***Vec***, en los capítulos siguientes. Un ***Vec*** es como un ***vector***, salvo porque lanza una excepción de tipo ***out_of_range*** si un subíndice está fuera de rango.

Las técnicas para implementar tipos como ***Vec*** y para usar eficazmente las excepciones se analizan en §11.12, §8.3 y el capítulo 14. Sin embargo, esta definición es suficiente para los ejemplos del libro:

```
template<class T> class Vec : public vector<T> {
public:
    Vec() : vector<T>() { }
    Vec(int s) : vector<T>(s) { }
    T& operator[](int i) { return at(i); }              // comprobado el rango
const T& operator[](int i) const { return at(i); }      // comprobado el rango
};
```

La operación **at**( ) es una operación de subindexación de **vector** que lanza una excepción de tipo **out_of_range** si su argumento está fuera del rango de **vector** (§16.3.3).

Volviendo al problema de cómo guardar los nombres y números de teléfono, ahora podemos usar un **Vec** para asegurarnos de que se capturan los accesos fuera de rango. Por ejemplo:

```
Vec<Entry> phone_book(1000);
void print_entry(int i)        // uso simple, exactamente como para vector
{
    cout << phone_book[i].name << ' ' << phone_book[i].number << '\n';
}
```

Un acceso fuera de rango lanzará una excepción que puede capturar el usuario. Por ejemplo:

```
void f()
{
    try {
        for (int i = 0; i<10000; i++) print_entry(i);
    }
    catch (out_of_range) {
        cout << "error de rango\n";
    }
}
```

La excepción será lanzada, y capturada, cuando se intente **phone_book**[*i*] con *i* = *1000*.

Si el usuario no captura este tipo de excepción, el programa terminará de una manera bien definida, en lugar de seguir adelante o fallar de una manera no definida. Una forma de reducir al mínimo las sorpresas procedentes de las excepciones es usar un **main**( ) con un *bloque-try* como cuerpo:

```
int main()
try {
    // código del lector
}
catch (out_of_range) {
    cerr << "error de rango\n";
}
catch (...) {
    cerr << "lanzada excepcion desconocida\n";
}
```

Esto proporciona manejadores de excepciones por defecto, de modo que si dejamos capturar una excepción se imprime un mensaje de error en el flujo de salida estándar diagnóstico de errores **cerr** (§21.2.1).

3.7.3 *list*

La inserción y el borrado de entradas en la guía de teléfonos será habitual. Por tanto, tal vez una lista sería más apropiada que un vector para representar una guía de teléfonos sencilla. Por ejemplo:

```
list<Entry> phone_book;
```

Cuando usamos una lista tendemos a no acceder a los elementos usando la subindexación como hacemos habitualmente para los vectores. En lugar de ello, examinaríamos la lista buscando un elemento con un valor dado, para lo cual aprovechamos el hecho de que una **list** es una secuencia, como se describe en §3.8:

```
void print_entry(const string& s)
{
    typedef list<Entry>::const_iterator LI;
    for (LI i = phone_book.begin(); i != phone_book.end(); ++i) {
        Entry& e = *i;      // referencia usada como abreviatura
        if (s == e.name) cout << e.name << ' ' << e.number << '\n';
    }
}
```

La búsqueda de **s** comienza al principio de la lista y sigue avanzando hasta que se encuentra **s** o se llega al final. Todos los contenedores de la biblioteca estándar proporcionan las funciones **begin**( ) y **end**( ), que devuelven un iterador hasta los elementos primero y siguiente al último, respectivamente (§16.3.2). Dado un iterador *i*, el elemento siguiente es ++*i*. Dado un iterador *i*, el elemento al que se refiere es **i*.

No es necesario que el usuario conozca el tipo exacto de iterador para un contenedor estándar. Ese tipo de iterador forma parte de la definición del contenedor y es posible referirse a él por el nombre. Cuando no necesitamos modificar un elemento del contenedor, **const_iterator** es el tipo que queremos. En caso contrario, usamos el tipo simple **iterator** (§16.3.1).

Es fácil añadir elementos a una **list**:

```
void add_entry(Entry& e, list<Entry>::iterator i)
{
    phone_book.push_front(e);    // añadir al principio
    phone_book.push_back(e);     // añadir al final
    phone_book.insert(i, e);     // añadir antes del elemento al que se refiere 'i'
}
```

3.7.4 *map*

Escribir código para buscar un nombre en una lista de pares (nombre,número) es realmente tedioso. Además, una búsqueda lineal es bastante poco eficiente para listas que no sean muy cortas. Otras estructuras de datos soportan directamente inserción, borrado y búsqueda basados en valores. En especial, la biblioteca estándar proporciona el tipo **map** (§17.4.1). Un **map** es un contenedor de pares de valores. Por ejemplo:

```
map<string, int> phone_book;
```

En otros contextos son conocidos los **map** como arrays asociativos o diccionarios.

Cuando está indexado por un valor de su primer tipo (denominado *clave*), un **map** devuelve el valor correspondiente del segundo tipo (denominado *valor* o *tipo mapeado*). Por ejemplo:

```
void print_entry (const string& s)
{
    if (int i = phone_book[s]) cout << s << ' ' << i << '\n';
}
```

Si no se ha encontrado concordancia para la clave *s*, se devuelve un valor por defecto desde *phone_book*. El valor por defecto para un tipo entero en un *map* es *0*. En nuestro caso, suponemos que *0* no es un número de teléfono válido.

3.7.5 Contenedores estándar

Para representar una guía de teléfonos se puede usar un *map*, una *list* y un *vector*. Sin embargo, cada uno de ellos tiene ventajas e inconvenientes. Por ejemplo, subindexar un *vector* es fácil y barato. Por otra parte, insertar un elemento entre otros dos tiende a ser costoso. Una *list* tiene exactamente las propiedades contrarias. Un *map* se parece a una *list* de pares (clave,valor) pero está optimizada para encontrar los valores a partir de las claves.

La biblioteca estándar proporciona algunos de los tipos de contenedores más generales y útiles, para que el programador pueda seleccionar el contenedor que mejor se adapte a las necesidades de una aplicación:

Resumen de contenedores estándar

vector<T>	Vector de tamaño variable (§16.3)
list<T>	Lista doblemente enlazada (§17.2.2)
queue<T>	Cola (§17.3.2)
stack<T>	Pila(§17.3.1)
deque<T>	Cola de dos extremos (§17.2.3)
priority_queue<T>	Cola clasificada por valor (§17.3.3)
set<T>	Conjunto (§17.4.3)
multiset<T>	Conjunto en el que un valor puede aparecer muchas veces (§17.4.3)
map<key, val>	Array asociativo (§17.4.1)
multimap<key, val>	Mapa en el que un valor puede aparecer muchas veces (§17.4.2)

Los contenedores estándar se presentan en §16.2, §16.3 y el capítulo 17. Los contenedores se definen en el espacio de nombres *std* y se presentan en las cabeceras <*vector*>, <*list*>, <*map*>, etcétera (§16.2).

Los contenedores estándar y sus operaciones básicas están diseñados para que sean similares desde el punto de vista notacional. Además, los significados de las operaciones son equivalentes en los distintos contenedores. En general, las operaciones básicas son aplicables a todos los tipos de contenedores. Por ejemplo, *push_back* puede usarse (con razonable eficiencia) para añadir elementos al final de un *vector*, así como de una *list*, y todos los contenedores tienen una función miembro *size* () que devuelve su número de elementos.

Esta uniformidad notacional y semántica permite a los programadores proporcionar nuevos tipos de contenedores que se pueden usar de formas muy semejantes a los estándar. Ejemplo de ello es el vector de rango comprobado *Vec* (§3.7.2). En el capítulo 17 se muestra cómo se puede añadir a la estructura un *hash_map*. La uniformidad de las interfaces de contenedor nos permite asimismo especificar algoritmos con independencia de los tipos individuales de contenedores.

3.8 Algoritmos

Una estructura de datos, como una lista o un vector, no es muy útil por sí misma. Para usarla, necesitamos operaciones para acceso básico tales como añadir y suprimir elementos. Además, rara vez nos limitamos a almacenar objetos en un contenedor. Los clasificamos, los imprimimos, extraemos subconjuntos, suprimimos elementos, buscamos objetos, etcétera. En consecuencia, la biblioteca estándar proporciona los algoritmos más comunes para contenedores además de proporcionar los tipos de contenedores más frecuentes. Lo siguiente, por ejemplo, clasifica un *vector* y coloca una copia de cada uno de los elementos únicos de *vector* en una *list*:

```
void f(vector<Entry>& ve, list<Entry>& le)
{
    sort(ve.begin(), ve.end());
    unique_copy(ve.begin(), ve.end(), le.begin());
}
```

Los algoritmos estándar se describen en el capítulo 18. Se expresan en términos de secuencias de elementos (§2.7.2). Una secuencia está representada por un par de iteradores que especifican los elementos primero y siguiente al último. En el ejemplo, *sort* () clasifica la secuencia desde *ve.begin* () hasta *ve.end* (), que casualmente son todos los elementos de un *vector*. A la hora de escribir, sólo hay que especificar el primer elemento a escribir. Si se escribe más de un elemento, todos los elementos que sigan a ese inicial serán sobrescritos.

Si deseábamos añadir los elementos nuevos al final de un contenedor, podríamos haber escrito:

```
void f(vector<Entry>& ve, list<Entry>& le)
{
    sort(ve.begin(), ve.end());
    unique_copy(ve.begin(), ve.end(), back_inserter(le));    // adjuntar a le
}
```

Un *back_inserter* () añade elementos al final de un contenedor, ampliando éste para hacerles hueco (§19.2.4). Los programadores de C agradecerán que los contenedores estándar más *back_inserter* () eliminen la necesidad de utilizar la gestión de memoria explícita de estilo C, propensa a errores, usando *realloc* () (§16.3.5). Olvidarse de utilizar un *back_inserter* () a la hora de adjuntar puede conducir a errores. Por ejemplo:

```
void f(list<Entry>& ve, vector<Entry>& le
{
    copy(ve.begin(), ve.end(), le);          // error: le no es un iterador
    copy(ve.begin(), ve.end(), le.end());    // mal: escribe más allá del final
    copy(ve.begin(), ve.end(), le.begin());  // sobrescribir elementos
}
```

3.8.1 Uso de los iteradores

Cuando se encuentra por primera vez un contenedor se pueden obtener unos cuantos iteradores que hacen referencia a elementos útiles; ***begin*** ( ) y ***end*** ( ) son los mejores ejemplos de ello. Además, muchos algoritmos devuelven iteradores. Así, por ejemplo, el algoritmo estándar ***find*** busca un valor dentro de una secuencia y devuelve un iterador al elemento encontrado. Usando ***find*** podemos escribir una función que cuente el número de apariciones de un carácter en una ***string***:

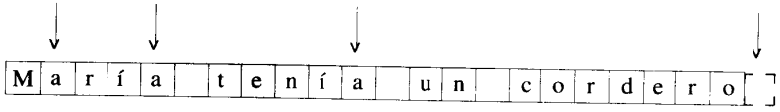
```
int count(const string& s, char c)
{
  string::const_iterator i=find(s.begin(),s.end(),c);
  int n = 0;§
  while (i != s.end()) {
    ++n;
    i = find(i+1,s.end(),c);
  }
  return n;
}
```

El algoritmo ***find*** devuelve un iterador a la primera aparición de un valor dentro de una secuencia o el iterador siguiente al final. Consideremos lo que ocurre con una simple llamada de ***count***:

```
void f()
{
  string m = "Maria tenia un cordero";
  int a_count = count(m,'a');
}
```

La primera llamada a ***find*** ( ) encuentra la primera '***a***' de ***Maria***. Así pues, el iterador apunta a ese carácter y no a ***s.end*** ( ), por lo que entramos en el bucle. En él, comenzamos la búsqueda en ***i+1***; es decir, empezamos un carácter después de donde hemos encontrado la '***a***'. Luego seguimos en el bucle hasta encontrar las otras dos '***a***'. Hecho esto, ***find*** ( ) alcanza el final y devuelve ***s.end*** ( ), por lo que la condición ***i*** != ***s.end*** ( ) no se cumple y salimos del bucle.

La llamada de ***count*** ( ) podría representarse gráficamente de la forma siguiente:



Las flechas indican los valores inicial, intermedios y final del iterador ***i***.

Naturalmente, el algoritmo ***find*** funcionará de modo equivalente en todos los contenedores estándar. En consecuencia, podríamos generalizar la función ***count*** ( ) de la misma forma:

```
template<class C, class T>int count(const C& v, T val)
{
  typename C::const_iterator i = find(v.begin(),v.end(),val);   // ver §C.13.5
  int n = 0;;
  while (i != v.end()) {
    ++n;
    ++i;        // saltar después del elemento que acabamos de encontrar
    i = find(i,v.end(),val);
  }
  return n;
}
```

Esto funciona, por lo que podemos decir:

```
void f(list<complex>& lc, vector<string>& vc, string s)
{
  int i1 = count(lc,complex(1,3));
  int i2 = count(vc, "Crisipo");
  int i3 = count(s,'x');
}
```

Sin embargo, no tenemos que definir una plantilla ***count***. Contar las apariciones de un elemento es de tanta utilidad general que la biblioteca estándar proporciona ese algoritmo. Para ser totalmente general, el ***count*** de la biblioteca estándar toma una secuencia como argumento en lugar de un contenedor, por lo que diríamos:

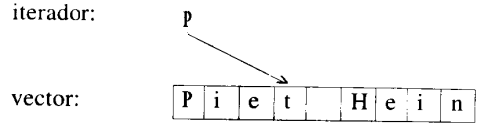
```
void f(list<complex>& lc, vector<string>& vc, string s)
{
  int i1 = count(lc.begin(),lc.end(),complex(1,3));
  int i2 = count(vc.begin(),vc.end(),"Diogenes");
  int i3 = count(s.begin(),s.end(),'x');
}
```

El uso de una secuencia nos permite usar ***count*** para un array predefinido y también para contar partes de un contenedor. Por ejemplo:

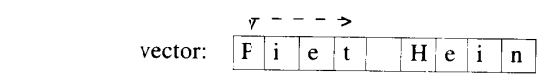
```
void g(char cs[], int sz)
{
  int i1 = count(&cs[0],&cs[sz],'z');     // zetas del array
  int i2 = count(&cs[0],&cs[sz/s],'z');   // zetas de la primera mitad del array
}
```

3.8.2 Tipos de iteradores

¿Qué son en realidad los iteradores? Cualquier iterador concreto es un objeto de algún tipo Hay, sin embargo, muchos tipos diferentes de iteradores porque un iterador tiene que contener la información necesaria para hacer su trabajo para un tipo de contenedor determinado. Estos tipos de iteradores pueden ser tan diferentes como los contenedores y las necesidades especializadas que atienden. Así, por ejemplo, un iterador de ***vector*** es con toda probabilidad un puntero ordinario, porque un puntero es una forma bastante razonable de referirse a un elemento de un ***vector***:

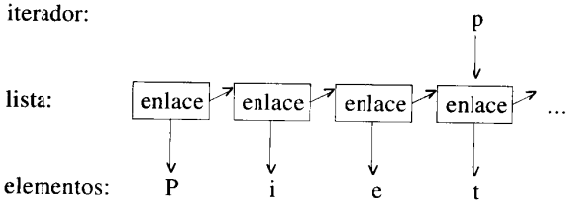


Alternativamente, un iterador ***vector*** podría implementarse como un puntero al ***vector*** más un índice:



El uso de un iterador así haría posible la comprobación de rango (§19.3).

Un iterador de lista sería algo más complicado que un simple puntero a un elemento, porque un elemento de una lista en general no sabe dónde está el siguiente elemento de esa lista. Así pues, un iterador de lista podría ser un puntero a un enlace:



Lo que es común a todos los iteradores es su semántica y el nombre de sus operaciones. Así, por ejemplo, al aplicar ++ a cualquier iterador se obtiene un iterador que hace referencia al siguiente elemento. De forma similar, * da el elemento al que se refiere el iterador. En realidad, cualquier objeto que cumpla unas cuantas reglas sencillas como éstas es un iterador (§19.2.1). Además, los usuarios rara vez necesitan conocer el tipo de un iterador específico; cada contenedor «conoce» sus tipos de iterador y los hace accesibles bajo los nombres convencionales ***iterator*** y ***const_iterator***. Por ejemplo, ***list<Entry>::iterator*** es el tipo de iterador general para ***list<Entry>***. Rara vez tengo que preocuparme de los detalles sobre cómo se define ese tipo.

3.8.3 Iteradores y E/S

Los iteradores son un concepto general y útil para tratar con secuencias de elementos en contenedores. Sin embargo, los contenedores no son el único lugar donde encontramos secuencias de elementos. Por ejemplo, un flujo de entrada produce una secuencia de valores y escribimos una secuencia de valores en un flujo de salida. En consecuencia, la noción de iteradores se puede aplicar a la entrada y salida.

Para hacer un ***ostream_iterator***, hemos de especificar qué flujo se va a usar y el tipo de objetos escritos en él. Podemos definir, por ejemplo, un iterador que se refiera al flujo de salida estándar, ***cout***:

```
ostream_iterator<string>oo(cout);
```

El efecto de la asignación a ***oo** es escribir el valor asignado en ***cout***. Por ejemplo:

```
int main()
{
    *oo = "Hola, ";    // significa cout<<"Hola,"
    ++oo;
    *oo = "mundo!\n";  // significa cout<<"mundo!\"
}
```

Ésta es otra forma más de escribir el mensaje canónico en la salida estándar. El ***++oo*** se hace para imitar la escritura en un array por medio de un puntero. Esta forma no sería mi primera opción para una tarea tan sencilla, pero la utilidad de tratar la salida como un contenedor de sólo escritura resultará en seguida evidente, si no lo es ya.

De forma semejante, un ***istream_iterator*** es algo que nos permite tratar un flujo de entrada como un contenedor de sólo lectura. Nuevamente debemos especificar el flujo que se va a usar y el tipo de valores previsto:

```
istream_iterator<string>ii(cin);
```

Como los iteradores de entrada aparecen siempre en pares que representan una secuencia, debemos proporcionar un ***input_iterator*** para indicar el final de la entrada. El ***input_iterator*** por defecto es:

```
istream_iterator<string>eos;
```

Podríamos ahora leer ***Hola, mundo!*** desde la entrada y escribirlo de nuevo así:

```
int main()
{
    string s1 = *ii;
    ++ii;
    string s2 = *ii;
    cout << s1 << ' ' << s2 << '\n';
}
```

En realidad, no se pretende que ***input_iterator***s y ***output_iterator***s sean usados directamente. Por el contrario, habitualmente son proporcionados como argumentos de algoritmos. Podemos escribir, por ejemplo, un programa sencillo para leer un archivo, clasificar las palabras leídas, eliminar duplicaciones y escribir el resultado en otro archivo:

```
int main()
{
    string from, to;
    cin >> from >> to;              // obtener nombres de los archivos fuente y destino
    ifstream is(from.c_str());      // input stream(c_str(); véase §3.5)
    istream_iterator<string>ii(is); // iterador de entrada para el flujo
    istream_iterator<string>eos;    // centinela de entrada
    vector<string>b(ii,eos);        // b es un vector inicializado desde la entrada
```

```
    sort(b.begin(),b.end());          // clasificar el búfer
    ofstream os(to.c_str());          // flujo de salida
    ostream_iterator<string>oo(os,"\n");   // iterador de salida para el flujo
    unique_copy(b.begin(),b.end(),oo);     // copiar el búfer en la salida
                                           // descartar valores reiterados
    return !is.eof() && !os;          // devolver estado de error (§3.2, §21.3.3)
}
```

Un *ifstream* es un *istream* que puede conectarse a un archivo y un *ofstream* es un *ostream* que puede conectarse a un archivo. El segundo argumento de *ostream_iterator* se usa para delimitar los valores de la salida.

3.8.4 Transversales y predicados

Los iteradores nos permiten escribir bucles para iterar a lo largo de una secuencia. Sin embargo, escribir bucles puede ser tedioso, por lo que la biblioteca estándar proporciona formas para que sea llamada una función para cada elemento de una secuencia.

Supongamos que estamos escribiendo un programa que lee palabras de la entrada y registra su frecuencia de aparición. La representación obvia de las cadenas y sus frecuencias asociadas es un *map*:

```
map<string,int>histogram;
```

La acción evidente a realizar para que cada cadena registre su frecuencia es:

```
void record(string& s)
{
    histogram[s]++;        // registrar la frecuencia de "s"
}
```

Una vez leída la entrada, nos gustaría extraer los datos que hemos recopilado. El *map* se compone de una secuencia de pares (*string*,*int*). En consecuencia, nos gustaría llamar

```
void print(pair<string,int>& r)
{
    cout << r.first << ' ' << r.second << '\n';
}
```

para cada elemento del mapa (el primer elemento de un par se llama *first* y el segundo elemento *second*). El primer elemento del *pair* es una *const string* porque todas las claves de *map* son constantes. Así pues, el programa principal se convierte en:

```
int main()
{
    istream_iterator<string>ii(cin);
    istream_iterator<string>eos;
    for_each(ii,eos,record);
    for_each(histogram.begin(),histogram.end(),print);
}
```

A señalar que no es necesario clasificar el *map* para obtener salida en orden. Un *map* mantiene ordenados sus elementos de modo que una iteración lo atraviesa en orden (creciente).

En muchas tareas de programación hay que buscar algo en un contenedor en lugar de sólo hacer algo a cada uno de los elementos. Así, por ejemplo, el algoritmo *find* (§18.5.2) proporciona una forma cómoda de buscar un valor específico. Una variante más general de la misma idea busca un elemento que cumple un requisito específico. Por ejemplo, podríamos querer buscar en un *map* el primer valor mayor que *42*. Un *map* es una secuencia de pares (clave,valor), por lo que buscamos en esa lista un *pair*<*const string*,*int*> donde el *int* sea mayor que *42*:

```
bool gt_42(pair<const string,int>& r)
{
    return r.second>42;
}
void f(map<string,int>& m)
{
    typedef map<string,int>::const_iterator MI;
    MI i = find_if(m.begin(),m.end(),gt_42);
    //...
}
```

Como alternativa, podríamos contar el número de palabras con frecuencia superior a 42:

```
void g(const map<string,int>& m)
{
    int c42 = count_if(m.begin(),m.end(),gt_42);
    //...
}
```

Una función como *gt_42*(), que se usa para controlar el algoritmo, se denomina *predicado*. Para cada elemento se llama a un predicado y éste devuelve un valor booleano que es usado por el algoritmo para realizar la acción prevista. Por ejemplo, *find_if*() busca hasta que su predicado devuelve *true* para indicar que se ha encontrado un elemento de interés. De forma semejante, *count_if*() cuenta el número de veces que su predicado es *true*.

La biblioteca estándar proporciona unos cuantos predicados útiles y algunas plantillas que sirven para crear más (§18.4.2).

3.8.5 Algoritmos que usan funciones miembro

Muchos algoritmos aplican una función a los elementos de una secuencia. Por ejemplo, en §3.8.4

```
for_each(ii,eos,record);
```

llama a *record*() para las cadenas leídas de la entrada.

A menudo trabajamos con contenedores de punteros y nos gustaría llamar a una función miembro del objeto apuntado en lugar de a una función global sobre el puntero. Así, por ejemplo, podríamos querer llamar a la función miembro *shape*::*draw*() para cada elemento de una *list*<*shape**>. Para manejar este ejemplo concreto, sencillamente escribimos una función no miembro que invoque a la función miembro. Por ejemplo:

```
void draw(Shape* p)
{
  p->draw();
}
void f(list<Shape*>& sh)
{
  for_each(sh.begin(), sh.end(), draw);
}
```

Generalizando esta técnica, podemos escribir así el ejemplo:

```
void g(list<shape*>& sh)
{
  for_each(sh.begin(), sh.end(), mem_fun(&Shape::draw));
}
```

La plantilla ***mem_fun*** ( ) (§18.4.4.2) de la biblioteca estándar toma un puntero a una función miembro (§15.5) como argumento y produce algo que puede ser llamado para un puntero a la clase del miembro. El resultado de ***mem_fun*** (&***shape***::***draw***) toma un argumento ***shape**** y devuelve lo que devuelva ***shape***::***draw***.

El mecanismo ***mem_fun*** ( ) es importante porque hace posible que los algoritmos estándar sean usados para contenedores de objetos polimórficos.

3.8.6 Algoritmos de la biblioteca estándar

¿Qué es un algoritmo? Una definición general de algoritmo dice que es «un conjunto finito de reglas que da una secuencia de operaciones para resolver un conjunto específico de problemas [y] tiene cinco características importantes: Finitud... Definición... Entrada... Salida... Efectividad» [Knuth, 1968, §1.1]. En el contexto de la biblioteca estándar C++, un algoritmo es un conjunto de plantillas que operan sobre secuencias de elementos.

La biblioteca estándar proporciona muchos algoritmos, que se definen en el espacio de nombres ***std*** y se presentan en la cabecera <***algo***>; algunos son especialmente útiles:

Algunos algoritmos estándar

for_each ( )	Invocar la función para cada elemento (§18.5.1)
find ( )	Encontrar la primera aparición de los argumentos (§18.5.2)
find_if ( )	Encontrar la primera concordancia del predicado (§18.5.2)
count ( )	Contar las apariciones de un elemento (§18.5.3)
count_if ( )	Contar las concordancias del predicado (§18.5.3)
replace ( )	Sustituir un elemento por un valor nuevo (§18.6.4)
replace_if ( )	Sustituir elemento que concuerda con predicado por valor nuevo (§18.6.4)
copy ( )	Copiar elementos (§18.6.1)
unique_copy ( )	Copiar elementos que no son duplicados (§18.6.1)
sort ( )	Clasificar elementos (§18.7.1)
equal_range ( )	Encontrar todos los elementos con valores equivalentes (§18.7.2)
merge ( )	Fusionar secuencias clasificadas (§18.7.3)

Estos algoritmos, y muchos más (véase el capítulo 18), se pueden aplicar a elementos de contenedores, ***strings*** y arrays predefinidos.

3.9 Matemáticas

Al igual que C, C++ no se diseñó pensando primordialmente en el cálculo numérico. Sin embargo, en C++ se hace gran cantidad de trabajo numérico y la biblioteca estándar refleja esta realidad.

3.9.1 Números complejos

La biblioteca estándar soporta una familia de tipos de números complejos que sigue las líneas de la clase ***complex*** descrita en 2.5.2. Para soportar números complejos en los que los escalares son de precisión sencilla, números en coma flotante (***float***s), números de doble precisión (***double***s), etcétera, la clase ***complex*** de la biblioteca estándar es una plantilla:

```
template<class scalar>class complex {
public:
  complex(scalar, re, scalar im);
  // ...
};
```

Las operaciones aritméticas habituales y las funciones matemáticas más frecuentes están soportadas para números complejos. Por ejemplo:

```
// función exponencial estándar de <complex>:
template<class C, class T> complex<C>pow(const complex<C>&, int);
void f(complex<float>fl, complex<double>db)
{
  complex<long double> ld = fl+sqrt(db);
  db += fl+3;
  fl = pow(1/fl, 2);
  // ...
}
```

Para más detalles, véase §22.5.

3.9.2 Aritmética de vectores

El ***vector*** descrito en 3.7.1 estaba diseñado para ser un mecanismo general para contener valores, ser flexible y encajar en la arquitectura de contenedores, iteradores y algoritmos. Sin embargo, no soporta operaciones matemáticas de vectores. Añadir esas operaciones a ***vector*** sería fácil, pero su generalidad y flexibilidad imposibilita optimizaciones que suelen ser consideradas esenciales para el trabajo numérico serio. En consecuencia, la biblioteca estándar proporciona un vector, llamado ***valarray***, que es menos general y más susceptible de optimización para cálculo numérico

```
template<class V>class valarray {
  //...
  T& operator[] (size_t);
  //...
};
```

El tipo *size_t* es el tipo entero sin signo que usa la implementación para índices de array.

Las operaciones aritméticas habituales y las funciones matemáticas más frecuentes están soportadas por *valarray*. Por ejemplo:

```
// función de valor absoluto estándar de <valarray>:
template<class T> valarray<T> abs (const valarray<T>&);

void f (valarray<double>& a1, valarray<double>& a2)
{
  valarray<double>a = a1*3.14+a2/a1;
  a2 += a1*3.14;
  a = abs(a);
  double d = a2[7];
  //...
}
```

Para más detalles, véase §22.4.

3.9.3 Soporte numérico básico

Naturalmente, la biblioteca estándar contiene las funciones matemáticas más comunes, como *log*(), *pow*() y *cos*(), para tipos en coma flotante; véase §22.3. Además, se proporcionan clases que describen las propiedades de los tipos predefinidos, como el exponente máximo de un *float*; véase §22.2.

3.10 Recursos de la biblioteca estándar

Los recursos que proporciona la biblioteca estándar pueden clasificarse así:

[1] Soporte básico del lenguaje en tiempo de ejecución (para asignación e información sobre tipos en tiempo de ejecución); véase §16.1.3.

[2] La biblioteca C estándar (con modificaciones muy poco importantes para reducir al mínimo las infracciones del sistema de tipos); véase §16.1.2.

[3] Cadenas y flujos de E/S (con soporte para juegos de caracteres internacionales y localización); véanse los capítulos 20 y 21.

[4] Una estructura de contenedores (como *vector*, *list* y *map*) y algoritmos que usan contenedores (como transversales generales, clasificaciones y fusiones); véanse los capítulos 16, 17, 18 y 19.

[5] Soporte para cálculo numérico (números complejos más vectores con operaciones aritméticas, porciones BLAS y generalizadas, y semántica diseñada para facilitar la optimización); véase el capítulo 22.

El criterio principal para incluir una clase en la biblioteca fue que la usaran casi todos los programadores de C++ (tanto principiantes como expertos), que pudiera proporcionarse en una forma general que no supusiera costes significativos adicionales con respecto a una versión más sencilla del mismo recurso, y que los usos simples fueran fáciles de aprender. En esencia, la biblioteca estándar C++ proporciona las estructuras de datos fundamentales más frecuentes, junto con los algoritmos fundamentales usados en ellas.

Todos los algoritmos funcionan con todos los contenedores sin usar conversiones. Esta estructura, denominada convencionalmente STL [Stepanov, 1994] es extensible en el sentido de que los usuarios pueden proporcionar fácilmente contenedores y algoritmos además de los proporcionados como parte del estándar y hacer que funcionen directamente con los contenedores y algoritmos estándar.

4

Tipos y declaraciones

No aceptes nada que no sea la perfección.
Anónimo

La perfección sólo se logra
al borde del colapso.
C. N. Parkinson

Tipos — tipos fundamentales — booleanos — caracteres — literales de carácter — enteros — literales enteros — tipos en coma flotante — literales en coma flotante — tamaños — ***void*** — enumeraciones — declaraciones — nombres — ámbito — inicialización — objetos — ***typedef***s — consejos — ejercicios.

4.1 Tipos

Consideremos

```
x = y+f(2);
```

Para que esto tenga sentido en un programa C++, los nombres ***x***, ***y*** y ***f*** deben estar adecuadamente declarados. Es decir, el programador debe especificar que las entidades llamadas ***x***, ***y*** y ***f*** existen y son de tipos para los cuales = (asignación), + (suma) y () (llamada de función), respectivamente, tienen sentido.

Todo nombre (identificador) de un programa C++ tiene un tipo asociado a él. Ese tipo determina qué operaciones se pueden aplicar al nombre (es decir, a la entidad referenciada por el nombre) y cómo se interpretan tales operaciones. Por ejemplo, las declaraciones

```
float x;          // x es una variable en coma flotante
int y = 7;        // y es una variable entera con valor inicial 7
float f(int);     // f es una función que toma un argumento de tipo int y
                  // devuelve un número en coma flotante
```

harían que el ejemplo tuviera sentido. Como se declara que *y* es un ***int***, puede ser asigna-

da, usada en expresiones aritméticas, etcétera. Por otra parte, se declara que *f* es una función que toma un ***int*** como argumento, por lo que se la puede llamar cado un argumento adecuado.

En este capítulo se presentan los tipos fundamentales (§4.1.1) y las declaraciones (§4.9). Los ejemplos se limitan a demostrar características del lenguaje; no se pretende que hagan nada útil. Los ejemplos más amplios y realistas se han reservado para capítulos posteriores, cuando se haya descrito más de C++. El presente capítulo proporciona simplemente los elementos más elementales a partir de los que se construyen programas C++. El lector debe conocer esos elementos, más la terminología y la sintaxis sencilla que los acompaña, para poder llevar a cabo un proyecto real en C++ y especialmente para leer código escrito por otros. Sin embargo, no es necesario comprender por completo todos los detalles mencionados en este capítulo para entender los capítulos siguientes. En consecuencia, tal vez el lector prefiera hojear este capítulo, observar los conceptos principales y volver más tarde sobre él, cuando surja la necesidad de comprender más detalles.

4.1.1 Tipos fundamentales

C++ tiene un conjunto de tipos fundamentales que corresponden a las unidades de almacenamiento básico más comunes de un computador y a las formas más comunes de usarlas para contener datos:

§4.2 Tipo booleano (***bool***)
§4.3 Tipos de caracteres (como ***char***)
§4.4 Tipos enteros (como ***int***)
§4.5 Tipos en coma flotante (como ***double***)

Además, el usuario puede definir:

§4.8 Tipos de enumeración para representar conjuntos específicos de valores (***enum***)

Hay también

§4.7 Un tipo, ***void***, usado para denotar la ausencia de información.

A partir de estos tipos podemos construir otros:

§5.1 Tipos de puntero (como ***int****)
§5.2 Tipos de array (como ***char***[])
§5.5 Tipos de referencia (como ***double***&)
§5.7 Estructuras de datos y clases (capítulo 10)

Los tipos booleano, de caracteres y enteros se denominan colectivamente *tipos integrales*. Los tipos integrales y en coma flotante se denominan colectivamente *tipos aritméticos*. Las enumeraciones y las clases (capítulo 10) se denominan *tipos definidos por el usuario* porque deben ser definidos por los usuarios y no están disponibles para usarlos sin declaración previa, como lo están los tipos fundamentales. En contraste, los otros tipos se denominan *tipos predefinidos*.

Los tipos integrales y en coma flotante se proporcionan en una variedad de tamaños para dar al programador la posibilidad de elegir la cantidad de almacenamiento consumido, la precisión y el rango disponible para cálculos (§4.6). Se parte de la hipótesis de que un computador proporciona bytes para contener caracteres, palabras para contener y calcular valores enteros, alguna entidad más adecuada para cálculo en coma flotante y direcciones para referir a esas entidades. Los tipos fundamentales de C++, junto con los punteros y los arrays, presentan estas nociones a nivel de máquina al programador con una razonable independencia de la implementación.

Para la mayor parte de las aplicaciones se podría usar sencillamente ***bool*** para valores lógicos, ***char*** para caracteres, ***int*** para valores enteros y ***double*** para valores en coma flotante. Los restantes tipos fundamentales son variaciones para optimizaciones y necesidades especiales que es mejor no tener en cuenta hasta que surgen tales necesidades. Deben conocerse, sin embargo, para leer código C y C++ antiguo.

4.2 Booleanos

Un booleano, ***bool***, puede tener uno de dos valores: ***true*** (verdadero) o ***false*** (falso). Los booleanos se usan para expresar el resultado de operaciones lógicas. Por ejemplo:

```
void f(int a, int b)
{
  bool b1 = a==b;   // = es asignación, == es igualdad
  // ...
}
```

Si ***a*** y ***b*** tienen el mismo valor, ***b1*** se convierte en ***true***; en caso contrario, ***b1*** se convierte en ***false***.

Un uso habitual de ***bool*** es como tipo del resultado de una función que prueba alguna condición (predicado). Por ejemplo:

```
bool is_open(File*);
bool greater(int a, int b) { return a>b; }
```

Por definición, ***true*** tiene el valor ***1*** cuando se convierte en entero y ***false*** tiene el valor ***0***. A la inversa, los enteros se pueden convertir implícitamente en valores ***bool***: los enteros distintos de cero se convierten en ***true*** y ***0*** se convierte en ***false***. Por ejemplo:

```
bool b = 7;      // bool(7) es verdadero; por tanto, b se convierte en true
int i = true;    // int(true) es 1; por tanto, i se convierte en 1
```

En las expresiones aritméticas y lógicas, los valores ***bool*** se convierten en ***int*** y aritmética de enteros, y las operaciones lógicas se realizan con los valores convertidos. Si el resultado se convierte de nuevo en ***bool***, un ***0*** se convierte en ***false*** y un valor distinto de cero se convierte en ***true***.

```
void g()
{
  bool a = true;
  bool b = true;
  bool x = a+b;     // a+b es 2; por tanto, x se convierte en true
  bool y = a|b;     // a|b es 1; por tanto, y se convierte en true
}
```

Un puntero puede convertirse implícitamente en ***bool*** (§6.2.5). Un puntero distinto de cero se convierte en ***true***; los punteros con valor cero se convierten en ***false***.

4.3 Tipos de caracteres

Una variable de tipo ***char*** puede contener un carácter del juego de caracteres de la implementación. Por ejemplo:

```
char ch = 'a';
```

De manera casi universal, un ***char*** tiene 8 bits, de modo que puede contener uno de 256 valores diferentes. Habitualmente el juego de caracteres es una variante del ISO-646 (por ejemplo, ASCII) y proporciona los caracteres que aparecen en el teclado del lector. Son muchos los problemas que surgen por el hecho de que este juego de caracteres esté sólo parcialmente estandarizado (§C.3).

Hay importantes variaciones entre los juegos de caracteres que soportan diferentes lenguajes naturales y también entre juegos de caracteres diferentes que soportan el mismo lenguaje natural de diferentes maneras. Sin embargo, aquí nos interesa únicamente la forma en que esas diferencias afectan a las reglas de C++. El problema, más amplio y más interesante, de cómo programar en un entorno multilingüe y multijuego de caracteres queda fuera del alcance de este libro, aunque se hace referencia a él en distintos lugares (§20.2, §21.7, §C3.3).

Se puede presuponer con confianza que el juego de caracteres de la implementación incluye los dígitos decimales, los 26 caracteres alfabéticos del inglés y algunos de los caracteres básicos de puntuación. No se puede presuponer con confianza que no haya más de 127 caracteres en un juego de caracteres de 8 bits (algunos juegos proporcionan 256 caracteres), que no haya más caracteres alfabéticos que los del inglés (la mayoría de las lenguas europeas tienen más), que los caracteres alfabéticos sean contiguos (EBCDIC deja un hueco entre '*i*' y '*j*') o que todos los caracteres usados para escribir C++ estén disponibles (algunos juegos de caracteres nacionales no cuentan con { }, [ ], | ; §C3.1). Siempre que sea posible debemos evitar hacer hipótesis sobre la representación de objetos. Esta norma general se aplica incluso a los caracteres.

Cada constante de carácter tiene un valor entero. Por ejemplo, el valor de '*b*' es *98* en el juego de caracteres ASCII. Veamos un pequeño programa que le dirá al lector el valor entero de cualquier carácter que introduzca:

```
#include<iostream>
int main()
{
    char c;
    cin>>c;
    cout<< "el valor de '" << c << "' es" << int(c) << '\n';
}
```

La notación ***int***(*c*) da el valor entero para un carácter *c*. La posibilidad de convertir un ***char*** en entero suscita la pregunta de si un ***char*** tiene signo o no lo tiene. Los 256 valores representados por un byte de 8 bits pueden interpretarse como los valores que van de *0* a *255* o como los valores que van de *-127* a *127*. Por desgracia, la opción tomada para un ***char*** sin más está definida por la implementación (§C.1, §C.3.4). C++ proporciona dos tipos para los cuales está definida la respuesta: ***signed char***, que puede contener como mínimo los valores que van de *-127* a *128*, y ***unsigned char***, que puede contener como mínimo los valores que van de *0* a *255*. Afortunadamente, la diferencia importa sólo para los valores que quedan fuera del rango *0-127*, y los caracteres más comunes están dentro de ese rango.

Los valores situados fuera del rango anterior almacenados en un ***char*** simple pueden dar lugar a delicados problemas de portabilidad. Vea el lector §C.3.4 si necesita usar más de un tipo de ***char*** o si almacena enteros en variables ***char***.

Se proporciona un tipo ***wchar_t*** para contener caracteres de un juego de caracteres mayor, como Unicode. Es un tipo distinto. El tamaño de ***wchar_t*** está definido por la implementación y es suficiente para contener el mayor juego de caracteres soportado por el locale de la implementación (véase 21.7, §C.3.3). La rareza del nombre se explica porque es un resto de C. En C, ***wchar_t*** es un ***typedef*** (§4.9.7) en lugar de un tipo predefinido. El sufijo ***_t*** se añadió para distinguirlo de los ***typedef*** estándar.

Observe el lector que los tipos de caracteres son tipos integrales (§4.1.1), por lo que les son aplicables operaciones aritméticas y lógicas (§6.2).

4.3.1 Literales de carácter

Un literal de carácter, denominado a menudo constante de carácter, es un carácter encerrado entre comillas sencillas; por ejemplo, '*a*' y '*0*'. El tipo de un literal de carácter es ***char***. Los literales de carácter son en realidad constantes simbólicas para el valor entero de los caracteres del juego de caracteres de la máquina en la que se va a ejecutar el programa C++. Si, por ejemplo, el lector está en una máquina que usa el juego de caracteres ASCII, el valor de '*0*' es *48*. El uso de literales de carácter en lugar de la notación decimal hace que los programas sean más portables. Unos cuantos caracteres tienen además nombres estándar que usan la barra invertida y el carácter de escape. Por ejemplo, ***n** es un salto de línea y ***t** es una tabulación horizontal. Véase en §C.3.2 los detalles sobre los caracteres de escape.

Los literales de carácter extendidos tienen la forma ***L'ab'***, estando el número de caracteres entrecomillados y su significado definidos por la implementación para concordar con el tipo ***wchar_t***. Un literal de carácter extendido tiene el tipo ***wchar_t***.

4.4 Tipos enteros

Al igual que ***char***, cada tipo entero puede tener tres formas: ***int*** «simple», ***signed int*** y ***unsigned int***. Además los enteros pueden tener tres tamaños: ***short int***, ***int*** «simple» y ***long int***. A un ***long int*** es posible referirse simplemente como ***long***. De forma semejante, ***short*** es sinónimo de ***short int***, ***unsigned*** de ***unsigned int*** y ***signed*** de ***signed int***.

Los tipos de enteros ***unsigned*** son idóneos para usos en los que se trata el almacenamiento como un array de bits. Casi nunca es buena idea usar un ***unsigned*** en lugar de un ***int*** con el fin de ganar un bit más para representar enteros positivos. Los intentos de garantizar que algunos valores sean positivos declarando variables ***unsigned*** serán desbaratados habitualmente por las reglas de conversión implícita (§C.6.1, §C.6.2.1).

A diferencia de los ***char*** simples, los ***int*** simples tienen siempre signo. Los tipos ***int*** con signo son sencillamente sinónimos más explícitos de sus equivalentes ***int*** simples.

4.4.1 Literales enteros

Los literales enteros se presentan de cuatro formas: decimales, octales, hexadecimales y literales de carácter. Los literales decimales son los que se usan más habitualmente y tienen el aspecto que sería de esperar:

0 1234 976 12345678901234567890

El compilador debe advertir sobre los literales demasiado largos para ser representados.

Un literal que comienza por cero seguido de ***x*** (***0x***) es un número hexadecimal (base 16). Un literal que comienza por cero seguido de un dígito es un número octal (base 8). Por ejemplo:

decimal:	***0***	***2***	***63***	***83***
octal:	***00***	***02***	***077***	***0123***
hexadecimal:	***0x0***	***0x2***	***0x3f***	***0x53***

Las letras ***a***, ***b***, ***c***, ***d***, ***e*** y ***f***, o sus equivalentes en mayúsculas, se usan para representar ***10***, ***11***, ***12***, ***13***, ***14*** y ***15***, respectivamente. Los notaciones octal y hexadecimal son de máxima utilidad para expresar patrones de bits. El uso de estas notaciones para expresar números auténticos puede provocar sorpresas. Así, por ejemplo, en una máquina en la que un ***int*** se represente como un entero de 16 bits de complemento a dos ***0xffff*** es el número decimal ***-1***. Si se hubieran usado más bits para representar un entero, habría sido ***65535***.

El sufijo ***U*** se puede usar para escribir explícitamente literales ***unsigned***. De forma semejante, se puede usar el sufijo ***L*** para escribir explícitamente literales ***long***. Así, por ejemplo, ***3*** es un ***int***, ***3U*** es un ***unsigned int*** y ***3L*** es un ***long int***. Si no se proporciona sufijo, el compilador da a un literal entero un tipo adecuado de acuerdo con su valor y con los tamaños de enteros de la implementación (§C.4).

Es buena idea limitar el uso de constantes no obvias a unos pocos inicializadores ***const*** (§5.4) o enumerador (§4.8) bien comentados.

4.5 Tipos en coma flotante

Los tipos en coma flotante representan números en coma flotante. Al igual que los enteros, los tipos en coma flotante pueden tener tres tamaños: ***float*** (precisión sencilla), ***double*** (precisión doble) y ***long double*** (precisión extendida).

El sentido exacto de precisión «sencilla», «doble» y «extendida» viene definido por la implementación. Elegir la precisión adecuada para un problema en el que la elección de la precisión es importante requiere unos conocimientos considerables de cálculo de coma flotante. Si el lector carece de esos conocimientos, busque consejo, dedique tiempo a adquirirlos o use ***double*** y espere que ocurra lo mejor.

4.5.1 Literales en coma flotante

Por defecto, un literal en coma flotante es de tipo ***double***. También en este caso, el compilador debe advertir sobre literales en coma flotante que son demasiado largos para ser representados. Veamos algunos literales en coma flotante:

1.23 .23 0.23 1. 1.0 1.2e10 1.23e-15

Observe el lector que no puede haber espacio en medio de un literal en coma flotante. Por ejemplo, ***65.43 e-21*** no es un literal en coma flotante sino cuatro componentes léxico independientes (que causan un error de sintaxis):

65.43 e - 21

Si se desea un literal en coma flotante de tipo ***float*** se puede definir usando el sufijo ***f*** o ***F***:

3.14159265f 2.0f 2.997925F

4.6 Tamaños

Algunos aspectos de los tipos fundamentales de C++, como el tamaño de un ***int***, vienen definidos por la implementación (§C.2). Suelo señalar esas dependencias y recomendar que se eviten o se tomen medidas para reducir su impacto. ¿Por qué preocuparse? Las personas que programan en diversos sistemas o usan diversos compiladores se preocupan mucho porque, si no lo hacen, se ven obligadas a perder tiempo buscando y subsanando oscuros defectos. Las personas que afirman no preocuparse de la portabilidad habitualmente usan un único sistema y pueden permitirse mantener la actitud de que «el lenguaje es lo que mi compilador implementa». Pero esto denota estrechez de miras e imprudencia. Si el programa del lector es un éxito, probablemente se llevará a otros sistemas, y alguien tendrá que buscar y resolver los problemas relacionados con características dependientes de la implementación. Además, a menudo es necesario compilar los programas con otros compiladores para el mismo sistema, e incluso es posible que una versión futura de nuestro compilador preferido haga las cosas de un modo algo distinto al actual. Es mucho más fácil conocer y limitar el impacto de las dependencias de la implementación cuando se escribe un programa, que intentar deshacer el embrollo más adelante.

Es relativamente fácil limitar el impacto de las características del lenguaje dependientes de la implementación. Limitar el impacto de los recursos de biblioteca dependientes del sistema es mucho más difícil. Un planteamiento es usar los recursos de la biblioteca estándar siempre que sea posible.

La razón para proporcionar más de un tipo entero, más de un tipo sin signo y más de un tipo en coma flotante es permitir que el programador aproveche características del hardware. En muchas máquinas hay diferencias significativas en cuanto a necesidades de memoria, tiempos de acceso a la memoria y velocidad de cálculo entre las diferentes variedades de los tipos fundamentales. Cuando se conoce una máquina habitualmente es fácil elegir, por ejemplo, el tipo adecuado de entero para una variable concreta. Escribir código de bajo nivel realmente transportable es más difícil.

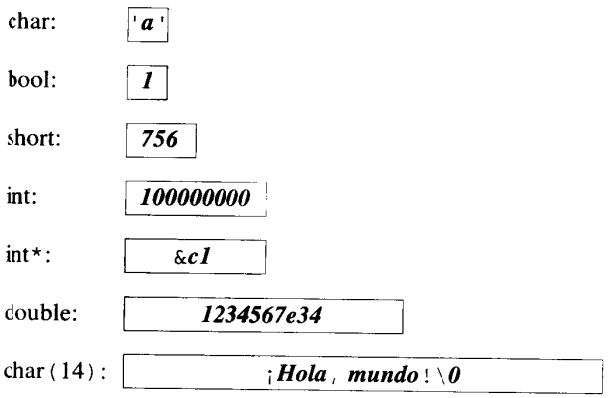
Los tamaños de los objetos C++ se expresan en múltiplos del tamaño de un ***char***; por definición, el tamaño de un ***char*** es ***1***. El tamaño de un objeto o tipo se puede obtener usando el operador ***sizeof*** (§6.2). Esto es lo garantizado sobre los tamaños de los tipos fundamentales:

$$1 \equiv sizeof(char) \le sizeof(short) \le sizeof(int) \le sizeof(long)$$
$$1 \le sizeof(bool) \le sizeof(long)$$
$$sizeof(char) \le sizeof(wchar\_t) \le sizeof(long)$$

sizeof(***float***) ≤ *sizeof*(***double***) ≤ *sizeof*(***long double***)
sizeof(*N*) ≡ *sizeof*(***signed*** *N*) ≡ *sizeof*(***unsigned*** *N*)

donde *N* puede ser ***char***, ***short***, ***int*** o ***long int***. Además, está garantizado que un ***char*** tiene al menos 8 bits, un ***short*** tiene al menos 16 bits y un ***long*** tiene al menos 32 bits. Un ***char*** puede contener un carácter del juego de caracteres de la máquina.

Veamos una representación gráfica de un conjunto plausible de tipos fundamentales y una cadena de muestra:



En la misma escala (.2 pulgadas para un byte), un megabyte de memoria se extendería alrededor de tres millas (cinco kilómetros) a la derecha.

Se supone que el tipo ***char*** es elegido por la implementación para que sea el tipo más adecuado para contener y manipular caracteres en un computador dado; habitualmente es un byte de 8 bits. De forma semejante, se supone que el tipo ***int*** es elegido para que sea el más adecuado para contener y manipular enteros en un computador dado; habitualmente es una palabra de 4 bytes (32 bits). No es aconsejable dar por supuesto más. Por ejemplo, hay máquinas con ***char*** de 32 bits.

Cuando se necesitan, se pueden encontrar los aspectos dependientes de la implementación en <***limits***> (§22.2). Por ejemplo:

```
#include <limits>
int main()
{
  cout << "largest float == " << numeric_limits<float>::max()
       << " , char is signed == " << numeric_limits<char>::is_signed << '\n';
}
```

Los tipos fundamentales se pueden mezclar libremente en las asignaciones y expresiones. Siempre que es posible, los valores se convierten de modo que no se pierda información (§C.6).

Si un valor ***v*** se puede representar exactamente en una variable de tipo ***T***, una conversión de ***v*** a ***T*** conserva el valor y no plantea problemas. Es mejor evitar los casos en los que las conversiones no conservan el valor (§C.6.2.6).

Es necesario conocer la conversión implícita con cierto detalle para llevar a cabo un proyecto importante y, sobre todo, para entender código real escrito por otros. Sin embargo, no son conocimientos necesarios para leer los capítulos siguientes.

4.7 *void*

El tipo ***void*** es fundamental desde el punto de vista sintáctico. Sin embargo, sólo se puede usar como parte de un tipo más complicado; no hay objetos de tipo ***void***. Se usa para especificar que una función no devuelve ningún valor o como tipo base para apuntadores a objetos de tipo desconocido. Por ejemplo:

```
void x;          // error: no hay objetos void
void f();        // la función f no devuelve ningún valor (§7.3)
void* pv;        // puntero a un objeto de tipo desconocido (§5.6)
```

Cuando se declara una función hay que especificar el tipo del valor devuelto. Sería lógico que se pudiera indicar que una función no devuelve ningún valor omitiendo el tipo devuelto. Sin embargo, ello haría que la gramática (Apéndice A) fuera menos regular y entraría en conflicto con el uso en C. En consecuencia, ***void*** se utiliza como «tipo seudodevuelto» para indicar que una función no devuelve ningún valor.

4.8 Enumeraciones

Una *enumeración* es un tipo que puede contener un conjunto de valores especificados por el usuario. Una vez definidas, las enumeraciones se usan de modo muy parecido a los tipos enteros.

Se pueden definir constantes enteras con nombre como miembros de una enumeración. Por ejemplo,

```
enum { ASM, AUTO, BREAK };
```

define tres constantes enteras, denominadas enumeradores, y les asigna valores. Por defecto, los valores de los enumeradores se asignan en orden creciente a partir de ***0***, de modo que ***ASM***==***0***, ***AUTO***==***1*** y ***BREAK***==***2***. A las enumeraciones se les puede dar nombre. Por ejemplo:

```
enum keyword { ASM, AUTO, BREAK };
```

Cada enumeración es un tipo distinto. El tipo de un enumerador es su enumeración. Por ejemplo, ***AUTO*** es de tipo ***keyword***.

Declarar una variable ***keyword*** en lugar de simplemente ***int*** puede dar tanto al usuario como al compilador una indicación sobre su uso previsto. Por ejemplo:

```
void f(keyword key)
{
  switch (key) {
  case ASM:
    // hacer algo
    break;
```

```
    case BREAK:
        // hacer algo
        break;
    }
}
```

Un compilador puede emitir una advertencia (en inglés, *warning*) porque sólo se manejan dos de los tres valores ***keyword***.

Un enumerador puede ser inicializado por una *expresión-constante* (§C.5) de tipo integral (§4.1.1). El rango de una enumeración contiene todos los valores de enumerador de la enumeración redondeados a la potencia binaria mayor más próxima menos *1*. El rango desciende a *0* si el enumerador más pequeño no es negativo y a la potencia binaria menor más próxima más *1* si el enumerador más pequeño es negativo. Esto define el menor campo de bits capaz de contener los valores del enumerador. Por ejemplo:

```
enum e1 { dark, light };                    // rango 0:1
enum e2 { a = 3, b = 9 };                   // rango 0:15
enum e3 { min = -10, max = 1000000 };       // rango -1048576:1048575
```

Un valor de tipo integral puede ser convertido explícitamente a un tipo de enumeración. El resultado de tal conversión es no definido a menos que el valor esté dentro del rango de la enumeración. Por ejemplo:

```
enum flag { x=1, y=2, z=4, e=8 }; // rango 0:15
flag f1 = 5;            // error de tipo: 5 no es de tipo flag
flag f2 = flag(5);      // bien: flag(5) es de tipo flag y está en el rango de flag
flag f3 = flag(12e);    // bien: flag(12) es de tipo flag y está en el rango de flag
flag f4 = flag(99);     // no definido: 99 no está en el rango de flag
```

La última asignación muestra por qué no hay conversión implícita de un entero a una enumeración; la mayor parte de los valores enteros no están representados en una enumeración concreta.

La noción de rango de valores para una enumeración difiere de la noción de enumeración en la familia de lenguajes Pascal. Sin embargo, los ejemplos de manipulación de bits que requieren valores situados fuera del conjunto de enumeradores para estar bien definidos tienen una larga historia en C y C++.

El ***sizeof*** de una enumeración es el ***sizeof*** de algún tipo integral que pueda contener su rango y no sea mayor que ***sizeof(int)***, a menos que un enumerador no pueda ser representado como ***int*** o como ***unsigned int***. Por ejemplo, ***sizeof(e1)*** podría ser *1* o tal vez *4*, pero no *8*, en una máquina en la que ***sizeof(int)*** ==*4*.

Por defecto, las enumeraciones se convierten a enteros para las operaciones aritméticas (§6.2). Una enumeración es un tipo definido por el usuario, por lo que los usuarios pueden definir sus propias operaciones, tales como ++ y << para una enumeración (§11.2.3).

4.9 Declaraciones

Para que se pueda usar un nombre (identificador) en un programa C++ es necesario declararlo antes. Es decir, debe especificarse su tipo para informar al compilador sobre a qué entidad se refiere el nombre. Veamos unos ejemplos que ilustran la diversidad de las declaraciones:

```
char ch;
string s;
int count = 1;
const double pi = 3.1415926535897932385;
extern int error_number;

char* name = "Njal";
char* season[] = { "spring", "summer", "fall", "winter" };

struct Date { int d, m, y; };
int day(Date* p) { return p->d; }
double sqrt(double);
template<class T>T abs(T a) { return a<0 ? -a : a; }

typedef complex<short> Point;
struct User;
enum Beer { Carlsber, Tuborg, Thor };
namespace NS { int a; }
```

Como se puede ver en estos ejemplos, una declaración puede hacer más que asociar simplemente un tipo con un nombre. La mayor parte de las *declaraciones* anteriores son también *definiciones;* es decir, definen también una entidad para el nombre al que se refieren. Para ***ch***, esa entidad es la cantidad adecuada de memoria a usar como variable: se asignará esa memoria. Para ***day***, es la función especificada. Para la constante ***pi***, es el valor ***3.1415926535897932385***. Para ***Date***, la entidad es un tipo nuevo. Para ***point***, es el tipo ***complex<short>***, de modo que ***point*** se convierte en sinónimo de ***complex<short>***. De las declaraciones anteriores, sólo

```
double sqrt(double);
extern int error_number;
struct User;
```

no son también definiciones; es decir, la entidad a la que se refieren tiene que ser definida en otro lugar. El código (cuerpo) para la función ***sqrt*** debe ser especificado por alguna otra declaración, la memoria para la variable ***int error_number*** debe ser asignada por alguna otra declaración de ***error_number*** y alguna otra declaración del tipo ***User*** debe definir qué aspecto tiene el tipo. Por ejemplo:

```
double sqrt(double d) { /* ... */ }
int error_number = 1;
struct User { /* ... */ };
```

En un programa C++ tiene que haber exactamente una definición para cada nombre. Sin embargo, puede haber muchas declaraciones. Todas las declaraciones de una entidad deben coincidir en el tipo de entidad a que se refieren. Por tanto, el fragmento siguiente tiene dos errores:

```
int count;
int count;      // error: redefinición
```

```
extern int error_number;
extern short error_number;      // error: incongruencia de tipo
```

y éste no tiene ninguno (para el uso de ***extern***, véase §9.2):

```
extern int error_number;
extern int error_number;
```

Algunas definiciones especifican un «valor» para las entidades que definen. Por ejemplo:

```
struct Date { int d, m, y; };
typedef complex<short>point;
int day(Date* p) { return p->d; }
const double pi = 3.1415926535897932385;
```

Para tipos, plantillas, funciones y constantes el «valor» es permanente. Para los tipos de datos no constantes el valor inicial puede ser cambiado más adelante. Por ejemplo:

```
void f()
{
    int count = 1;
    char* name = "Bjarne";
    // ...
    count = 2;
    name = "Marian";
}
```

De las definiciones, sólo

```
char ch;
string s;
```

no especifican valores. Véanse en §4.9.5 y §10.4.2 explicaciones sobre cómo y cuándo se asigna a una variable un valor por defecto. Toda declaración que especifica un valor es una definición.

4.9.1 Estructura de las declaraciones

Una declaración se compone de cuatro partes: un «especificador» opcional, un tipo base, un declarador y un inicializador opcional. Salvo las definiciones de función y espacio de nombres, las declaraciones terminan con punto y coma. Por ejemplo:

```
char* reyes[] = { "Antigono", "Seleuco", "Ptolomeo" };
```

Aquí el tipo base es ***char***, el declarador es ****reyes**[]* y el inicializador es = { . . . }.

Un especificador es una palabra clave inicial, como ***virtual*** (§2.5.5, §12.2.6) y ***extern*** (§9.2), que especifica algún atributo no de tipo de lo que es declarado.

Un declarador consta de un nombre y, opcionalmente, de algunos operadores de declarador. Los operadores de declarador más frecuentes son (§A.7.1):

*	***puntero***	***prefijo***
*const	***puntero constante***	***prefijo***
&	***referencia***	***prefijo***
[]	***array***	***sufijo***
()	***funcion***	***sufijo***

Su uso sería sencillo si todos fueran prefijos o sufijos. Sin embargo, *, [] y () fueron diseñados para uso especular er las expresiones (§6.2). Así pues, * es prefijo, y [] y () son sufijos. Los operadores de declarador sufijos vinculan más que los prefijos. En consecuencia, ****reyes**[]* es un vector de punteros a algo y tenemos que usar paréntesis para expresar tipos como «puntero a función»; véanse ejemplos en 5.1. Para todos los detalles, consulte el lector la gramática del Apéndice A.

Hay que señalar que el tipo no se puede dejar fuera de una declaración. Por ejemplo:

```
const c = 7;        // error: no hay tipo
gt(int a, int b) { return (a>b) ? a : b; }      // error: no hay tipo devuelto
unsigned ui;        // bien: 'unsigned' es el tipo 'unsigned int'
long li;            // bien: 'long' es el tipo 'long int'
```

En esto el estándar C++ difiere de versiones anteriores de C y C++ que permitían los dos primeros ejemplos al considerar que ***int*** era el tipo cuando no se especificaba ninguno (§B.2). Esta regla del «***int*** implícito» era una fuente de errores y confusión.

4.9.2 Declaración de múltiples nombres

Es posible declarar varios nombres en una sola declaración. La declaración contiene sencillamente una lista de declaradores separados por comas. Por ejemplo, podemos declarar así dos enteros:

```
int x, y;     // int x; int y;
```

Observe el lector que los operadores se aplican individualmente a cada nombre, no a nombres contiguos de la misma declaración. Por ejemplo:

```
int* p, y;      // int* p;  int y;    NO int* y;
int x, *q;      // int x;   int* q;
int v[10], *pv; // int v[10];  int* pv;
```

Construcciones como éstas restan legibilidad a los programas y deben ser evitadas.

4.9.3 Nombres

Un nombre (identificador) se compone de una secuencia de letras y dígitos. El primer carácter debe ser una letra. El carácter de subrayado _ se considera una letra. C++ no impone límite al número de caracteres de un nombre. Sin embargo, hay partes de las implementaciones (en especial el enlazador) que no están bajo el control de quien escribe el compilador y esas partes, por desgracia, imponen a veces límites. Algunos entornos de tiempo de ejecución hacen asimismo necesario ampliar o restringir el conjunto de caracteres aceptado en un identificador. Las ampliaciones (por ejemplo, permitir el carácter $ en un nombre) dan lugar a programas no portables. Las palabras clave de C++ (Apéndice A), como ***int*** o ***new***, no pueden ser usadas como nombre de una entidad definida por el usuario. Ejemplos de nombres son:

```
hola            este_nombre_es_extraordinariamente_largo
DEFINIDO        foO     bAr         u_name      HorseSense
var0            var1    CLASE       _clase      ___
```

Ejemplos de secuencias de caracteres que no pueden ser usadas como identificadores son:

```
012             un loco     $sys        class       3var
pay.due         foo~bar     .nombre     if
```

Los nombres que comienzan con un guión de subrayado están reservados para recursos especiales de la implementación y el entorno de ejecución, por lo que no deben ser usados en programas de aplicaciones.

Al leer un programa, el compilador busca siempre la cadena más larga de caracteres que podría componer un nombre. Por tanto, ***var10*** es un solo nombre, no el nombre ***var*** seguido del número ***10***. Igualmente, ***elseif*** es un solo nombre, no la palabra clave ***else*** seguida de la palabra clave ***if***.

Las letras mayúsculas y minúsculas son distintas, por lo que ***Count*** y ***count*** son nombres diferentes, pero no es aconsejable elegir nombres que difieran sólo en el uso de las mayúsculas. En general, es mejor evitar los nombres que presentan sólo pequeñas diferencias. Por ejemplo, la o mayúscula (***O***) y el cero (***0***) pueden ser difíciles de distinguir, como ocurre con la ele minúscula (***l***) y el uno (***1***). En consecuencia, ***l0***, ***lO***, ***l1*** y ***ll*** son malas opciones para nombres identificadores.

Los nombres de un ámbito grande deben ser más bien largos y razonablemente obvios, como ***vector***, ***Ventana_con_borde*** y ***Department_number***. Sin embargo, el código es más claro si los nombres que se usan sólo en un ámbito pequeño son cortos y convencionales, como ***x***, ***i*** y ***p***. Se pueden usar clases (capítulo 10) y espacios de nombres (§8.2) para mantener los ámbitos pequeños. A menudo es útil hacer que los nombres que se usan con frecuencia sean más bien cortos y reservar los nombres muy largos para entidades que se utilizan poco. Hay que elegir nombres que reflejen el significado de una entidad más que su implementación. Por ejemplo, ***listin_telefonico*** es mejor que ***lista_de_numeros***, aunque los números de teléfono se almacenen en una ***list*** (§3.7). Elegir bien los nombres es un arte.

El lector debe procurar mantener un estilo coherente en la elección de nombres. Puede, por ejemplo, usar mayúscula inicial en los tipos definidos por el usuario de biblioteca no estándar y comenzar los nombres no de tipo con minúscula (por ejemplo, ***Forma*** y ***current_token***). Asimismo puede usar mayúsculas para las macros (si tiene que usar macros; por ejemplo, ***HACK***) y guiones de subrayado para separar las palabras de un identificador. Sin embargo, la coherencia es difícil de lograr porque en general los programas se componen de fragmentos de distinto origen y suele haber varios estilos diferentes en uso, todos razonables. El lector debe ser coherente en el uso de abreviaturas y acrónimos.

4.9.4 Ámbito

Una declaración introduce un nombre en un ámbito; es decir, los nombres sólo se pueden usar en partes concretas del texto del programa. Para un nombre declarado en una función (a menudo denominado *nombre local*), ese ámbito se extiende desde el punto en que es declarado hasta el final del bloque en que se produce su declaración. Un *bloque* es una sección de código limitada por un par { }.

Los nombres se denominan *globales* si se definen fuera de funciones, clases (capítulo 10) o espacios de nombres (§8.2). El ámbito de un nombre global se extiende desde el punto de declaración hasta el final del archivo en el que se produce la declaración. Una declaración de un nombre en un bloque puede ocultar una declaración en el bloque que lo incluye o un nombre global. Es decir, un nombre puede ser redefinido para referirse a una entidad diferente dentro de un bloque. Cuando se sale del bloque, el nombre recobra su significado anterior. Por ejemplo:

```
int x;              // x global
void f()
{
    int x;          // x local oculta a x global
    x = 1;          // asignar a x local
    {
        int x;      // oculta la primera x local
        x = 2;      // asignar a la segunda x local
    }
    x = 3;          // asignar a la primera x local
}
int* p = &x;        // tomar la dirección de la x global
```

Es inevitable ocultar nombres cuando se escriben programas grandes. Sin embargo, un lector humano puede fácilmente no darse cuenta de que un nombre ha sido ocultado. Como son errores relativamente raros, pueden ser muy difíciles de encontrar. En consecuencia, hay que minimizar la ocultación de nombres. Usar nombres como ***i*** y ***x*** para variables globales o para variables locales en una función grande es una forma de buscarse problemas.

Es posible referirse a un nombre global oculto usando el operador de resolución de ámbito ::. Por ejemplo:

```
int x;
void f2()
{
    int x = 1;      // ocultar x global
    ::x = 2;        // asignar a x global
}
```

No hay forma de usar un nombre local oculto.

El ámbito de un nombre comienza en el punto donde se declara; es decir, después del declarador completo y antes del inicializador. Esto implica que un nombre se puede usar incluso para especificar su propio valor inicial. Por ejemplo:

```
int x;
void f3()
{
    int x = x;      // perverso: inicializar x con su propio valor (no inicializado)
}
```

No se trata de algo ilegal, sino simplemente estúpido. Un buen compilador advertirá si se usa una variable antes de haber sido fijada (véase también §5.9[9]).

Es posible usar un solo nombre para referirse a dos objetos diferentes de un bloque sin utilizar el operador : : . Por ejemplo:

```
int x = 11;
void f4 ( )          // perverso:
{
  int y = x;         // usar x global: y = 11
  int x = 22;
  y = x;             // usar x local: y = 22
}
```

Los nombres de argumento de función se consideran declarados en el bloque más destacado de la definición de la función, por lo que

```
void f5 (int x)
{
  int x;             // error
}
```

es un error porque *x* se define dos veces en el mismo ámbito. Hacer que esto sea un error permite capturar una equivocación sutil y bastante frecuente.

4.9.5 Inicialización

Si se especifica un inicializador para un objeto, ese inicializador determina el valor inicial del objeto. Si no se especifica inicializador, un objeto estático local (§7.12, §10.2.4), de espacio de nombres (§8.2) o global (§4.9.4) (denominados colectivamente *objetos estáticos)* se inicializa a **0** del tipo apropiado. Por ejemplo:

```
int a;               // significa "int a = 0;"
double d;            // significa "double d = 0.0;"
```

Las variables locales (denominadas a veces *objetos automáticos*) y los objetos creados en la zona de memoria libre (a veces denominados *objetos dinámicos* u *objetos de montículo*) no se inicializan por defecto. Por ejemplo:

```
void f ( )
{
  int x;             // x no tiene un valor bien definido
  // ...
}
```

Los miembros de arrays y estructuras son inicializados o no por defecto dependiendo de si el array o estructura es estático. Los tipos definidos por el usuario pueden tener definida inicialización por defecto (§10.4.2).

Los objetos más complicados requieren más de un valor como inicializador. Esto es manejado por listas de inicializadores delimitadas por { y } para la inicialización al estilo C de arrays (§5.2.1) y estructuras (§5.7). Para tipos definidos por el usuario con constructores se usan listas de argumentos al estilo de las funciones (§2.5.2, §10.2.3).

Observe el lector que un paréntesis vacío ( ) en una declaración significa siempre «función» (§7.1). Por ejemplo:

```
int a [ ] = { 1, 2 };  // inicializador de array
Point z (1, 2);        // inicializador de estilo función (inicialización por constructor)
int f ( );             // declaración de función
```

4.9.6 Objetos y valores-l

Podemos asignar y usar «variables» que no tienen nombre, y también es posible asignar a expresiones de aspecto extraño (por ejemplo, `*p[a+10]=7`). En consecuencia, es necesario un nombre para «algo que está en la memoria». Ésta es la noción más sencilla y fundamental de objeto. Es decir, un *objeto* es una zona de almacenamiento contigua, un *valor-l* es una expresión que se refiere a un objeto. El término *valor-l*[1] se acuñó originalmente con el significado de «algo que puede estar a la izquierda de una asignación». Sin embargo, no todos los valores-l se pueden usar a la izquierda de una asignación; un valor-l se puede referir a una constante (§5.5). Un valor-l que no ha sido declarado ***const*** se denomina a menudo *valor-l modificable*. Esta noción de objeto, sencilla y de bajo nivel, no debe ser confundida con la noción de objeto de clase y objeto de tipo polimórfico (§15.4.3).

A menos que el programador especifique otra cosa (§7.1.2, §10.4.8), un objeto declarado en una función se crea cuando se encuentra su definición y se destruye cuando su nombre sale del ámbito (§10.4.4). Tales objetos se denominan objetos automáticos. Los objetos declarados en ámbito global o de espacio de nombres, y los ***static*** declarados en funciones o clases, son creados e inicializados una vez (sólo) y «viven» hasta que el programa termina (§10.4.9). Tales objetos se denominan objetos estáticos. Los miembros no estáticos de una estructura, array o clase tienen su duración determinada por el objeto del que son miembros.

Usando los operadores ***new*** y ***delete*** se pueden crear objetos cuya duración sea controlada directamente (§6.2.6).

4.9.7 *typedef*

Una declaración con la palabra ***typedef*** como prefijo declara un nombre nuevo para el tipo, en lugar de una variable nueva del tipo dado. Por ejemplo:

```
typedef char* Pchar;
Pchar p1, p2;          // p1 y p2 son char*
char* p3 = p1;
```

Un nombre definido así, habitualmente denominado «***typedef***», puede ser una abreviatura cómoda para un tipo con un nombre poco manejable. Por ejemplo, ***unsigned char*** es demasiado largo para usarlo muy a menudo, por lo que podríamos definir un sinónimo, ***uchar***:

```
typedef unsigned char uchar;
```

Otro uso de un ***typedef*** es para limitar a un lugar la referencia directa a un tipo. Por ejemplo

```
typedef int int32;
typedef short int16;
```

[1] En inglés, *lvalue*, de *left* (izquierda) y *value* (valor). *(N. de los T.)*

Si ahora usamos ***int32*** siempre que necesitemos un entero potencialmente largo, podemos trasladar nuestro programa a una máquina en la que ***sizeof(int)*** sea ***2*** redefiniendo la única aparición de ***int*** en nuestro código:

```
typedef long int32;
```

Para bien y para mal, los ***typedef*** son sinónimos de otros tipos y no tipos distintos. En consecuencia, se mezclan libremente con los tipos para los que son sinónimos. Quienes deseen tener tipos distintos con semántica o representación idéntica deben volver la vista hacia las enumeraciones (§4.8) o las clases (capítulo 10).

4.10 Consejos

[1] Haga que los ámbitos sean pequeños; §4.9.4.
[2] No use el mismo nombre en un ámbito y en el ámbito que lo contiene; 4.9.4.
[3] Declare (sólo) un nombre por declaración; §4.9.2.
[4] Haga que los nombres frecuentes y locales sean cortos, y los nombres poco frecuentes y no locales más largos; §4.9.3.
[5] Evite los nombres de aspecto parecido; §4.9.3.
[6] Mantenga la coherencia en el estilo de los nombres; §4.9.3.
[7] Elija cuidadosamente los nombres para que reflejen más el significado que la implementación; §4.9.3.
[8] Use un ***typedef*** para definir un nombre que tenga sentido para un tipo predefinido en los casos en que el tipo predefinido usado para representar un valor pueda cambiar; §4.9.7.
[9] Use ***typedef*** para definir sinónimos de los tipos; use enumeraciones y clases para definir tipos nuevos; §4.9.7.
[10] Recuerde que toda declaración debe especificar un tipo (no hay «***int*** implícito»); §4.9.1.
[11] Evite suposiciones innecesarias sobre el valor numérico de los caracteres; §4.3.1, §C.6.2.1.
[12] Evite suposiciones innecesarias sobre el tamaño de los enteros; §4.6.
[13] Evite suposiciones innecesarias sobre el rango de los tipos en coma flotante; §4.6.
[14] Prefiera un ***int*** sin más a un ***short int*** o un ***long int***; §4.6.
[15] Prefiera un ***double*** a un ***float*** o un ***long double***; §4.5.
[16] Prefiera los ***char*** sin más a los ***signed char*** o ***unsigned char***; §C.3.4.
[17] Evite hacer suposiciones innecesarias sobre el tamaño de los objetos; §4.6.
[18] Evite la aritmética sin signo; §4.4.
[19] Desconfíe de las conversiones de ***signed*** a ***unsigned*** y de ***unsigned*** a ***signed***; §C.6.2.6.
[20] Desconfíe de las conversiones de coma flotante a entero; §6.2.6.
[21] Desconfíe de las conversiones a un tipo más pequeño, como de ***int*** a ***char***; §C.6.2.6

4.11 Ejercicios

1. (*2) Haga que se ejecute el programa "Hola, mundo!" (§3.2). Si no compila como está escrito, consulte §B.3.1.
2. (*1) Haga lo siguiente para cada una de las declaraciones de §4.9: si la declaración no es una definición, escriba una definición para ella. Si la declaración es una definición, escriba una declaración para ella que no sea también una definición.
3. (*1,5) Escriba un programa que imprima los tamaños de los tipos fundamentales, unos cuantos tipos de puntero y unas cuantas enumeraciones de su elección. Use el operador ***sizeof***.
4. (*1,5) Escriba un programa que imprima las letras '***a***'... '***z***' y los dígitos '***0***'... '***9***' y sus valores enteros. Haga lo mismo para otros caracteres imprimibles. Repita lo mismo pero usando la notación hexadecimal.
5. (*2) ¿Cuáles son, en su sistema, los valores mayores y menores de los tipos siguientes: ***char***, ***short***, ***int***, ***long***, ***float***, ***double***, ***long double*** y ***unsigned***.
6. (*1) ¿Cuál es el nombre local más largo que puede usar en un programa C++ en su sistema? ¿Cuál es el nombre externo más largo que puede usar en un programa C++ en su sistema? ¿Hay restricciones a los caracteres que puece usar en un nombre?
7. (*2) Dibuje un gráfico de los tipos entero y fundamentales en el que un tipo apunte a otro tipo si todos los valores del primero pueden ser representados como valores del segundo en todas las implementaciones conformes al estándar. Dibuje el mismo gráfico para los tipos en su implementación preferida.

5

Punteros, arrays y estructuras

Lo sublime y lo ridículo están a menudo
tan estrechamente relacionados que es difícil separarlos.
Tom Paine

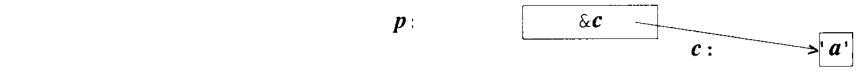
Punteros — cero — arrays — literales de cadena — punteros a arrays — constantes — punteros y constantes — referencias —***void**** — estructuras de datos — consejos — ejercicios.

5.1 Punteros

Para un tipo ***T***, ***T**** es el tipo «puntero a ***T***». Es decir, una variable de tipo ***T**** puede contener la dirección de un objeto de tipo ***T***. Por ejemplo:

```
char c = 'a';
char* p = &c;      //p contiene la dirección de c
```

o, gráficamente:



Por desgracia, los punteros a arrays y los punteros a funciones necesitan una notación más complicada:

```
int* pi;           // puntero a int
char** ppc;        // puntero a puntero a char
int* ap[15];       // array de 15 punteros a ints
int (*fp)(char*);  // puntero a función que toma argumento char*; devuelve int
int* f(char*);     // función que toma argumento char*; devuelve puntero a int
```

Véase en §4.9.1 una explicación de la sintaxis de declaración y en el Apéndice A la gramática completa.

La operación fundamental de un puntero es *desreferenciar*, es decir, referir al objeto al que apunta el puntero. Esta operación se denomina también *indirección*. El operador de desreferenciación es el * monario (prefijo). Por ejemplo:

```
char c = 'a';
char* p = &c;       // p contiene la dirección de c
char c2 = *p;       // c2 == 'a'
```

La variable a la que apunta **p** es **c** y el valor almacenado en **c** es '**a**', por lo que el valor de ***p** asignado a **c2** es '**a**'.

Es posible realizar algunas operaciones aritméticas con punteros a elementos de array (§5.3). Los punteros a funciones pueden ser de extremada utilidad; se estudian en §7.7.

La implementación de los punteros tiene la finalidad de mapear directamente a los mecanismos de direccionamiento de la máquina en la que se ejecuta el programa. La mayor parte de las máquinas pueden direccionar un byte. Las que no pueden, suelen tener hardware que extrae los bytes de las palabras. Por el contrario, hay pocas máquinas que puedan direccionar directamente un bit individual. En consecuencia, el objeto más pequeño que puede ser asignado de forma independiente y al que se puede apuntar usando un tipo de puntero predefinido es un ***char***. Observe el lector que un ***bool*** ocupa al menos tanto espacio como un ***char*** (§4.6). Para almacenar valores menores de manera más compacta se pueden usar operaciones lógicas (§6.2.4) o campos de bits en estructuras (§8.1).

5.1.1 Cero

El cero (**0**) es un ***int***. Debido a las conversiones estándar (§C.6.2.3), se puede usar **0** como constante de cualquier tipo integral (§4.1.1), de coma flotante, de puntero y de puntero a miembro. El tipo de cero estará determinado por el contexto. El cero estará representado habitual pero no necesariamente por el patrón de bits *todo-ceros* del tamaño apropiado.

Ningún objeto se asigna con la dirección **0**. En consecuencia, **0** actúa como literal de puntero, indicando que un puntero no se refiere a un objeto.

En C se ha hecho popular definir una macro ***NULL*** para representar el puntero cero. Debido a la más estricta comprobación de tipos en C++, el uso de **0** en lugar de una macro ***NULL*** produce menos problemas. Si el lector cree que tiene que definir ***NULL***, use

```
const int NULL = 0;
```

El cualificador ***const*** (§5.4) impide la redefinición accidental de ***NULL*** y garantiza que se puede usar ***NULL*** donde se necesite una constante.

5.2 Arrays

Para un tipo **T**, **T**[***size***] es el tipo «array de ***size*** (tamaño) elementos de tipo **T**». Los elementos se indexan de **0** a ***size*-1**. Por ejemplo:

```
float v[3];          // array de tres float: v[0], v[1], v[2]
char* a[32];         // array de 32 punteros a char: a[0]... a[31]
```

El número de elementos de un array, el límite del array, debe ser una expresión constante (§C.5). Si el lector necesita límites variables, use un ***vector*** (§3.7.1, §15.3). Por ejemplo:

```
void f(int i)
{
    int v1[i];              // error: el tamaño de array no es una expresión constante
    vector<int> v2(10);     // bien
}
```

Los arrays multidimensionales se representan como arrays de arrays. Por ejemplo:

```
int d2[10][20];      // d2 es un array de 10 arrays de 20 enteros
```

Usar la notación de coma como se usa para los límites de array en algunos otros lenguajes da lugar a errores en tiempo de compilación porque la coma (,) es un operador de secuencia y no está permitido en las expresiones constantes (§C.5). Por ejemplo, pruebe el lector lo siguiente:

```
int bad[5, 2];       // error: en una expresión constante no está permitida la coma
```

Los arrays multidimensionales se describen en §C.7. Es mejor evitarlos fuera del código de bajo nivel.

5.2.1 Inicializadores de array

Un array puede ser inicializado por una lista de valores. Por ejemplo:

```
int v1[] = { 1, 2, 3, 4 };
char v2[] = { 'a', 'b', 'c', 0 };
```

Cuando se declara un array sin un tamaño específico pero con una lista de inicializadores, se calcula el tamaño contando los elementos de la lista de inicializadores. En consecuencia, **v1** y **v2** son de tipo ***int***[**4**] y ***char***[**4**], respectivamente. Si se especifica explícitamente un tamaño, es un error dar elementos de más en una lista de inicializadores. Por ejemplo:

```
char v3[2] = { 'a', 'b', 0 };       // error: demasiados inicializadores
char v4[3] = { 'a', 'b', 0 };       // bien
```

Si el inicializador no proporciona todos los elementos, se presupone **0** para los elementos vector restantes. Por ejemplo:

```
int v5[8] = { 1, 2, 3, 4 };
```

es equivalente a

```
int v5[] = { 1, 2, 3, 4, 0, 0, 0, 0 };
```

Observe el lector que no hay asignación de array que concuerde con la inicialización:

```
void f()
{
    v4 = { 'c', 'd', 0 }        // error: no hay asignación de array
}
```

Cuando el lector necesite asignaciones así, debe usar un ***vector*** (§16.3) o un ***valarray*** (§22.4).

Un array de caracteres lo puede inicializar adecuadamente un literal de cadena (§5.2.2).

5.2.2 Literales de cadena

Un *literal de cadena* es una secuencia de caracteres encerrada entre comillas dobles:

```
"esto es una cadena"
```

Un literal de cadena contiene un carácter más de los que aparenta; es finalizado por el carácter nulo '\0' con el valor 0. Por ejemplo:

```
sizeof( "Bohr" ) ==5
```

El tipo de un literal de cadena es el «array del número adecuado de caracteres *const*»; por tanto, "*Bohr*" es de tipo *const char*[5].

Un literal de cadena puede ser asignado a un *char**. Esto se permite porque en definiciones anteriores de C y C++ el tipo de un literal de cadena era *char**. Permitir la asignación de un literal de cadena a un *char** garantiza que millones de líneas de C y C++ siguen siendo válidas. Sin embargo, es un error intentar modificar un literal de cadena por medio de un puntero semejante:

```
void f()
{
  char* p = "Platon";
  p[4] = 'e';          // error: asignación a const; el resultado es no definido
}
```

Esta forma de error en general no puede ser capturada hasta el tiempo de ejecución y hay diferencias entre las implementaciones en cuanto a la aplicación de esta norma. Hacer constantes los literales de cadena no es sólo obvio, sino que además permite que las implementaciones hagan optimizaciones significativas en la forma en que se almacenan los literales de cadena y se accede a ellos.

Si queremos una cadena con la que tengamos la garantía de que vamos a poder modificarla, debemos copiar los caracteres en un array:

```
void f()
{
  char[] p = "Zenon";      // p es un array de 6 char
  p[0] = 'R';              // bien
}
```

Un literal de cadena es asignado estáticamente de modo que sea seguro devolverlo desde una función. Por ejemplo:

```
const char* mensaje_de_error(int i)
{
  // ...
  return "error de rango";
}
```

La memoria que contiene *error de rango* no desaparecerá después de una llamada de *mensaje_de_error*().

Que dos literales de carácter idénticos sean asignados en memoria como uno solo es algo definido por la implementación (§C.1). Por ejemplo:

```
const char* p = "Heraclito";
const char* q = "Heraclito";
void g()
{
  if (p == q) cout << "uno!\n";      // resultado definido por la implementación
}
```

Observe el lector que == compara las direcciones (valores de puntero) cuando se aplica a punteros y no los valores a los que se apunta.

La cadena vacía se escribe como un par de comillas dobles adyacentes, "", (y tiene el tipo *const char*[1]).

La convención de la barra inclinada invertida para representar caracteres no gráficos (§C.3.2) se puede usar también con una cadena. Esto hace posible representar las comillas dobles (") y el carácter de escape barra inclinada invertida (\) en una cadena. El más habitual de esos caracteres es, con mucho, el carácter de salto de línea, '\n'. Por ejemplo:

```
cout<<"tono audible al final del mensaje\a\n";
```

El carácter de escape '\a' es el carácter ASCII *BEL* (conocido también como *alerta*) que provoca la emisión de un sonido.

No es posible tener un salto de línea «real» en una cadena:

```
"esto no es una cadena
sino un error de sintaxis"
```

Las cadenas largas se pueden interrumpir con espacios en blanco para hacer más limpio el texto del programa. Por ejemplo:

```
char alfa[] =  "abcdefghijklmnopqrstuvwxyz"
               "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

El compilador concatenará las cadenas adyacentes, por lo que habría sido equivalente inicializar *alfa* con una sola cadena:

```
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

Es posible tener el carácter nulo en una cadena, pero la mayoría de los programas no sospecharán que hay caracteres después de él. Por ejemplo, la cadena "*Jens\000Munk*" será tratada como "*Jens*" por funciones de la biblioteca estándar como *strcpy*() y *strlen*(); véase §20.4.1.

Una cadena con el prefijo *L*, como *L"angst"*, es una cadena de caracteres extendidos (§4.3, §C.3.3). Su tipo es *const wchar_t*[].

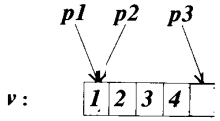
5.3 Punteros en arrays

Punteros y arrays están estrechamente relacionados en C++. Se puede usar el nombre de un array como puntero a su elemento inicial. Por ejemplo:

```
int v[] = { 1, 2, 3, 4 };
int* p1 = v;          // puntero al elemento inicial (conversión implícita)
```

```
int* p2 = &v[0];    // puntero al elemento inicial
int* p3 = &v[4];    // puntero al elemento siguiente al último
```

o, gráficamente:



Tomar un puntero al elemento siguiente al último de un array está garantizado que funciona. Esto es importante para muchos algoritmos (§2.7.2, §18.3). Sin embargo, como un puntero así no apunta en realidad a ningún elemento del array, no se puede usar para leer o escribir. El resultado de tomar la dirección del elemento anterior al elemento inicial es no definido y debe ser evitado. En algunas arquitecturas de máquina los arrays se asignan a menudo en los límites de direccionamiento de la máquina, por lo que «elemento anterior al inicial» carece simplemente de sentido.

La conversión implícita de un nombre de array en puntero al elemento inicial del array se usa ampliamente en llamadas de función en código de estilo C. Por ejemplo:

```
extern "C" int strlen(const char*);    // de <string.h>
void f()
{
  char v[] = "Annemarie";
  char* p = v;                         // conversión implícita de char[] a char*
  strlen(p);
  strlen(v);                           // conversión implícita de char[] a char*
  v = p;                               // error: no se puede asignar al array
}
```

En ambas llamadas se pasa el mismo valor a la función **strlen**() de la biblioteca estándar. La dificultad está en que es imposible evitar la conversión implícita. Dicho de otro modo, no hay manera de declarar una función de forma que se copie el array **v** cuando se llama a la función. Afortunadamente, no hay conversión implícita ni explícita de puntero a array.

La conversión implícita del argumento del array en puntero implica que se pierde el tamaño del array para la función llamada. Sin embargo, la función llamada debe determinar de algún modo el tamaño para realizar una operación que tenga sentido. Como otras funciones de la biblioteca C estándar que toman punteros a caracteres, **strlen**() confía en el cero para indicar el fin de la cadena; **strlen**(**p**) devuelve el número de caracteres que hay hasta la terminación **0**, sin incluirla. Todo muy de bajo nivel. El **vector** (§16.3) y el **string** (capítulo 20) de la biblioteca estándar no adolecen de este problema.

5.3.1 Navegación por arrays

Un acceso eficiente y elegante a los arrays (y otras estructuras de datos parecidas) es la clave para muchos algoritmos (véase §3.8, capítulo 18). El acceso se puede lograr mediante un puntero a un array más un índice o mediante un puntero a un elemento. Por ejemplo, atravesar una cadena de caracteres usando un índice,

```
void fi(char v[])
{
  for (int i = 0; v[i]!=0; i++) use(v[i]);
}
```

es equivalente a un transversal que use un puntero:

```
void fp(char v[])
{
  for (char* p = v; *p!=0; p++) use(*p);
}
```

El operador * prefijo desreferencia un puntero de modo que ***p** sea el carácter al que apunta **p** y ++ incrementa el puntero de modo que se refiera al siguiente elemento del array.

No hay razón intrínseca por la que una versión deba ser más rápida que la otra. Con compiladores modernos ambos ejemplos deben generar idéntico código (véase §5.9[8]). Los programadores pueden elegir entre ambas versiones siguiendo criterios lógicos y estéticos.

El resultado de aplicar los operadores aritméticos +, −, ++ o −− a punteros depende del tipo del objeto apuntado. Cuando se aplica un operador aritmético a un puntero **p** de tipo **T***, se supone que **p** apunta a un elemento de un array de objetos de tipo **T**; **p+1** apunta al siguiente elemento de ese array y **p−1** apunta al elemento anterior. Esto implica que el valor entero de **p+1** será **sizeof**(**T**) mayor que el valor entero de **p**. Por ejemplo, al ejecutar

```
#include<iostream.h>
int main()
{
  int vi[10];
  short vs[10];
  cout << &vi[0] << ' ' << &vi[1] << '\n';
  cout << &vs[0] << ' ' << &vs[1] << '\n';
}
```

se ha producido

```
07xfffaef0 0x7fffaef4
0x7fffaedec 0x7fffaede
```

usando una notación hexadecimal por defecto para los valores de puntero. Esto muestra que en mi implementación **sizeof**(**short**) es **2** y **sizeof**(**int**) es **4**.

La sustracción de punteros se define sólo cuando ambos punteros apuntan a elementos del mismo array (aunque el lenguaje no tiene una forma rápida de asegurarse de que es así). Cuando se sustrae un puntero de otro, el resultado es el número de elementos de array que hay entre los dos punteros (un entero). Se puede sumar un entero a un puntero o restar un entero de un puntero; en ambos casos el resultado es un valor de puntero. Si ese valor no apunta a un elemento del mismo array que el puntero original o a uno más allá, el resultado de usar ese valor es no definido. Por ejemplo:

```
void f()
{
  int v1[10];
  int v2[10];
```

```
    int i1 = &v1[5]-&v1[3];      // i1 = 2
    int i2 = &v1[5]-&v2[3];      // resultado no definido
    int* p1 = v2+2;              // p1 = &v2[2]
    int* p2 = v2-2;              // *p2 no definido
}
```

Habitualmente es innecesaria la aritmética complicada de punteros y suele ser mejor evitarla. La suma de punteros carece de sentido y no está permitida.

Los arrays no son autodescriptivos porque no está garantizado que el número de elementos de un array esté almacenado en el array. Esto implica que para atravesar un array que no contiene un terminador como lo contienen las cadenas de caracteres tenemos que proporcionar de algún modo el número de elementos. Por ejemplo:

```
void fp(char v[], unsigned int size)
{
    for (int i=0; i<size, i++) use(v[i]);
    const int N = 7;
    char v2[N];
    for (int i=0; i<N, i++) use(v2[i]);
}
```

Hay que señalar que la mayoría de las implementaciones no ofrecen comprobación de rango para los arrays. Este concepto de array es intrínsecamente de bajo nivel. Se puede proporcionar una noción más avanzada de arrays mediante el uso de clases; véase §3.7.1.

5.4 Constantes

C++ ofrece el concepto de constante definida por el usuario, ***const***, para expresar la noción de que un valor no cambia directamente. Esto es útil en varios contextos. Por ejemplo, muchos objetos no cambian sus valores después de la inicialización, las constantes simbólicas conducen a un código más mantenible que los literales incrustados directamente en el código, a menudo se lee a través de los punteros pero nunca se escribe a través de ellos y la mayoría de los parámetros de función se leen pero no se escriben.

La palabra clave ***const*** se puede añadir a la declaración de un objeto para hacer que el objeto declarado sea una constante. Como no pueden ser asignadas, las constantes tienen que ser inicializadas. Por ejemplo:

```
const int model = 90;                // model es una const
const int v[] = { 1, 2, 3, 4 };      // v[i] es una const
const int x;                         // error: no hay inicializador
```

Declarar algo ***const*** garantiza que su valor no cambiará dentro de su ámbito:

```
void f()
{
    model = 200;     // error
    v[2]++;          // error
}
```

Hay que señalar que ***const*** modifica un tipo; es decir, restringe las formas en que se puede usar un objeto, en lugar de especificar cómo hay que asignar la constante. Por ejemplo:

```
void g(const X* p)
{
    // no se puede modificar *p aquí
}
void h()
{
    X val;      // val puede ser modificado
    g(&val);
    // ...
}
```

Dependiendo de lo inteligente que sea, un compilador puede beneficiarse de que un objeto sea una constante de diversas formas. Por ejemplo, el inicializador para una constante es a menudo (pero no siempre) una expresión constante (§C.5); si lo es, puede ser evaluada en tiempo de compilación. Además, si el compilador conoce todos los usos de la ***const*** no necesita asignar espacio para contenerla. Por ejemplo:

```
const int c1 = 1;
const int c2 = 2;
const int c3 = my_f(3);    // no se conoce el valor de c3 en tiempo de compilación
extern const int c4;       // no se conoce el valor de c4 en tiempo de compilación
const int* p = &c2;        // es necesario asignar espacio para c2
```

Dado esto, el compilador conoce los valores de ***c1*** y ***c2*** de modo que puedan ser usados en expresiones constantes. Como los valores de ***c3*** y ***c4*** no se conocen en tiempo de compilación (usando sólo la información disponible en esta unidad de compilación; véase §9.1), debe asignarse memoria para ***c3*** y ***c4***. Como se toma (y presumiblemente se usa en otro sitio) la dirección de ***c2***, hay que asignar memoria para ***c2***. El caso más sencillo y frecuente es aquel en el cual el valor de la constante se conoce en tiempo de compilación y no es necesario asignar memoria; ***c1*** es un ejemplo de ello. La palabra clave ***extern*** indica que ***c4*** se define en otro lugar (§9.2).

Habitualmente es necesario asignar almacenamiento para un array de constantes porque el compilador no puede, en general, averiguar a qué elementos del array hay referencias en las expresiones. Sin embargo, en muchas máquinas se pueden lograr mejoras de eficiencias aun en ese caso colocando los arrays de constantes en almacenamiento de sólo lectura.

Los usos habituales de ***const*** son como límites de arrays y como etiquetas ***case*** en las *sentencias-switch*. Por ejemplo:

```
const int a = 42;
const int b = 99;
const int max = 128;
int v[max];
void f(int i)
{
    switch (i) {
```

```
    case a:
        //...
    case b:
        //...
    }
}
```

Los enumeradores (§4.8) son a menudo una alternativa a ***const*** en casos semejantes.

La forma en que se puede usar ***const*** con funciones miembros de clases se analiza en §10.2.6 y §10.2.7.

Se deben usar sistemáticamente constantes simbólicas para evitar la presencia de «números mágicos» en el código. Si en el código se repite una constante numérica, como un límite de array, resulta demasiado difícil revisarlo porque hay que cambiar todas las apariciones de esa constante para realizar una actualización correcta. El uso de una constante simbólica, en cambio, localiza la información. Habitualmente una constante numérica representa una hipótesis sobre el programa. Por ejemplo, ***4*** puede representar el número de bytes de un entero, ***128*** el número de caracteres necesarios para almacenar una entrada y ***6.24*** el factor de cambio entre la corona danesa y el dólar estadounidense. Si se dejan como constantes numéricas en el código, esos valores son difíciles de localizar y entender para quien se ocupe del mantenimiento. Los valores numéricos de ese tipo suelen pasar inadvertidos y se convierten en errores cuando se porta el programa o algún otro cambio infringe las hipótesis que representan. La representación de hipótesis mediante constantes simbólicas bien comentadas reduce al mínimo los problemas de mantenimiento mencionados.

5.4.1 Punteros y constantes

Cuando se usa un puntero intervienen dos objetos: el propio puntero y el objeto apuntado. Al añadir a una declaración de puntero el prefijo ***const*** se hace que el objeto, no el puntero, sea constante. Para declarar que el puntero, no el objeto apuntado, sea constante usamos el operador declarador ****const*** en lugar de un simple *. Por ejemplo:

```
void f1(char* p)
{
    char s[] = "Gorm";
    const char* pc = s;             // puntero a constante
    pc[3] = 'g';                    // error: pc apunta a constante
    pc = p; // bien
    char *const cp = s;             // puntero constante
    cp[3] = 'a'; // bien
    cp = p;                         // error: cp es constante
    const char *const cpc = s;      // puntero const a const
    cpc[3] = 'a';                   // error: cpc apunta a constante
    cpc = p;                        // error: cpc es constante
}
```

El operador declarador que hace a un puntero constante es ****const***. No hay operador declarador ***const***, por lo que un ***const*** que aparezca delante del * es tomado como parte del tipo base. Por ejemplo:

```
char *const cp;      // puntero const a char
char const* pc;      // puntero a char const
const char* pc2;     // puntero a char const
```

A algunas personas les resulta útil leer estas declaraciones de derecha a izquierda. Por ejemplo, «*cp* es un puntero ***const*** a un ***char***» y «***pc2*** es un puntero a un ***char const***».

Un objeto que sea constante cuando se accede a él por medio de un puntero puede ser variable cuando se accede a él de otras formas. Esto es especialmente útil para los argumentos de función. Al declarar ***const*** un argumento puntero, se prohíbe que la función modifique el objeto apuntado. Por ejemplo:

```
char* strcpy(char* p, const char* q);      // no se puede modificar *q
```

Se puede asignar la dirección de una variable a un puntero a constante porque de ello no se puede derivar daño alguno. Sin embargo, no se puede asignar la dirección de una constante a un puntero sin restricciones porque ello posibilitaría que se cambiara el valor del objeto. Por ejemplo:

```
void f4()
{
    int a = 1;
    const int c = 2;
    const int* p1 = &c;     // bien
    const int* p2 = &a;     // bien
    int* p3 = &c;           // error: inicialización de int* con const int*
    *p3 = 7;                // intento de cambiar el valor de c
}
```

Es posible eliminar explícitamente las restricciones sobre un puntero a ***const*** mediante conversión explícita de tipos (§10.2.7.1 y §15.4.2.1).

5.5 Referencias

Una *referencia* es un nombre alternativo para un objeto. Se usan sobre todo para especificar argumentos y devolver valores para funciones en general y para operadores sobrecargados (capítulo 11) en particular. La notación *X&* significa *referencia a X*. Por ejemplo:

```
void f()
{
    int i = 1;
    int& r = i;     // r e i se refieren ahora al mismo int
    int x = r;      // x = 1
    r = 2;          // i = 2;
}
```

Para garantizar que una referencia sea el nombre de algo (es decir, esté vinculada a un objeto) debemos inicializar la referencia. Por ejemplo:

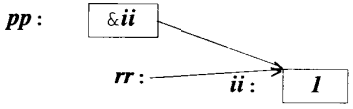
```
int i = 1;
int& r1 = i;          // bien: r1 inicializado
int& r2;              // error: falta el inicializador
extern int& r3;       // bien: r3 inicializado en otro lugar
```

La inicialización de una referencia es bastante diferente de la asignación a ella. A pesar de las apariencias, ningún operador actúa sobre una referencia. Por ejemplo:

```
void g()
{
  int ii = 0;
  int& rr = ii;
  rr++;               // ii es incrementado a 1
  int* pp = &rr;      // pp apunta a ii
}
```

Esto es legal, pero `rr++` no incrementa la referencia *rr*; en lugar de ello, `++` se aplica a un *int* que casualmente es *ii*. En consecuencia, el valor de una referencia no se puede cambiar después de la inicialización; se refiere siempre al objeto para el que fue inicializada. Para obtener un puntero al objeto denotado por una referencia *rr* podemos escribir `&rr`.

La implementación obvia de una referencia es como puntero (constante) que se desreferencia cada vez que se usa. No es demasiado perjudicial concebir las referencias de este modo, siempre que se recuerde que una referencia no es un objeto que pueda ser manipulado como lo es un puntero:



En algunos casos el compilador puede optimizar y descartar una referencia de modo que no haya objeto que la represente en tiempo de ejecución.

La inicialización de una referencia es trivial cuando el inicializador en un valor-l (un objeto cuya dirección podemos tomar; véase §4.9.6). El valor para un *T* «simple» debe ser un valor-l de tipo *T*&.

No es necesario que el inicializador para un `const T&` sea un valor-l ni siquiera de tipo *T*. En tales casos,

[1] se aplica primero conversión implícita de tipo a *T* si es necesario (véase §C.6);

[2] se coloca a continuación el valor resultante en una variable temporal de tipo *T*; y,

[3] por último, se usa esa variable temporal como valor del inicializador.

Consideremos:

```
double& dr = 1;          // error: es necesario valor-l
const double& cdr = 1;   // bien
```

La interpretación de esta última inicialización sería:

```
double temp = double(1);   // crear primero una temporal con el valor adecuado
const double& cdr = temp;  // usar luego la temporal como inicializador para cdr
```

Una temporal creada para contener un inicializador de referencia persiste hasta el final del ámbito de su referencia.

Las referencias a variables y las referencias a constantes se distinguen porque la introducción de una temporal en el caso de la variable es muy propensa a error; una asignación a la variable se convertiría en una asignación a la —temprana en desaparecer—temporal. Este problema no existe para las referencias a constantes, las cuales son a menudo importantes como argumentos de función (§11.6).

Se puede usar una referencia para especificar un argumento de función de modo que la función pueda cambiar el valor de un objeto que se le pase. Por ejemplo:

```
void increment(int& aa) { aa++; }
void f()
{
  int x = 1;
  increment(x); // x = 2
}
```

La semántica del paso de argumentos está definida por la de la inicialización, por lo que, cuando es llamado, el argumento *aa* de *increment* se convierte en otro nombre para *x*. Para mantener la legibilidad del programa a menudo es mejor evitar las funciones que modifican sus argumentos. En lugar de ello se puede devolver un valor explícitamente desde la función o requerir un argumento de puntero:

```
int next(int p) { return p+1; }
void incr(int* p) { (*p)++; }
void g()
{
  int x = 1;
  increment(x); // x = 2
  x = next(x);  // x = 3
  incr(&x);     // x = 4
}
```

La notación *increment*(*x*) no da a quien lee el código una pista de que el valor de *x* está siendo modificado de la misma forma que la da *x*=*next*(*x*) e *incr*(&*x*). En consecuencia, sólo se deben usar argumentos de referencia «simples» cuando el nombre de la función proporciona un fuerte indicio de que se modifica el argumento de la referencia.

Las referencias se usan también para definir funciones que se pueden utilizar tanto en el lado izquierdo como en el derecho de una asignación. Muchos de los usos más interesantes de esto se dan en el diseño de tipos no triviales definidos por el usuario. Como ejemplo, vamos a definir un array asociativo simple. Definimos primero la estructura *Pair*:

```
struct Pair {
  string name;
  double val;
};
```

La idea básica es que un *string* tiene un valor en coma flotante asociado a él. Es fácil definir una función, *value*(), que mantenga una estructura de datos consistente en un *Pair* para cada cadena diferente que se le haya presentado. Para abreviar la presentación, se usa una implementación muy sencilla (e ineficiente):

```
vec<Pair>pairs;
double& value(const string& s)
/*
   mantener un conjunto de Pairs; buscar s, devolver su valor si se encuentra;
   en caso contrario, hacer un Pair nuevo y devolver el valor por defecto 0
*/
{
for (int i = 0; i<pairs.size(); i++)
   if (s == pairs[i].name) return pairs[i].val;
Pair p = { s, 0 };
pairs.push_back(p);      // añadir Pair al final (§3.7.3)
return pairs[pairs.size() -1].val;
}
```

Esta función se puede entender como un array de valores en coma flotante indexado por cadenas de caracteres. Para una cadena de argumentos dada, *value*() encuentra el objeto en coma flotante correspondiente *(no* el valor del objeto en coma flotante correspondiente); devuelve entonces una referencia a él. Por ejemplo:

```
int main()      // contar el número de apariciones de cada palabra en la entrada
{
  string buf;
  while (cin>>buf) value(buf)++;
  for (vector<Pair>::const_iterator p=pairs.begin(); p!=pairs.end(); ++p)
     cout<<p->name<< " : " <<p->val<<'\n';
}
```

Cada vez, el bucle *while* lee una palabra de la cadena de entrada estándar *cin* a *buf* (§3.6) y actualiza el contador asociado con él. Por último, se imprime la tabla resultante con las diferentes palabras de la entrada, cada una con su número de apariciones. Por ejemplo, dada la entrada

```
aa bb bb aa aa bb aa aa
```

el programa producirá:

```
aa: 5
bb: 3
```

Es fácil perfeccionar lo anterior y convertirlo en un tipo de array asociativo adecuado usando una clase de plantilla con el operador de selección [] sobrecargado (§11.8). Es incluso más fácil usar simplemente la biblioteca estándar *map* (§17.4.1).

5.6 Puntero a void

Se puede asignar un puntero de cualquier tipo de objeto a una variable de tipo *void**, se puede asignar un *void** a otro *void**, se pueden hacer comparaciones de igualdad y desigualdad entre *void**s y se puede convertir explícitamente un *void** en otro tipo. Otras operaciones serían poco seguras porque el compilador no puede saber a qué tipo de objeto se apunta realmente. En consecuencia, las demás operaciones dan lugar a errores en tiempo de compilación. Para usar un *void** debemos convertirlo explícitamente en un puntero a un tipo específico. Por ejemplo:

```
void f(int* pi)
{
  void* pv = pi;                    // bien: conversión implícita de int* en void*
  *pv;                              // error: no se puede desreferenciar void*
  pv++;                             // error: no se puede desreferenciar void*
  int* pi2 = static_cast<int*>(pv); // conversión explícita de nuevo a int*
  double* pd1 = pv;                 // error
  double* pd2 = pi;                 // error
  double* pd3 = static_cast<double*>(pv); // inseguro
}
```

En general, no es seguro usar un puntero que ha sido convertido («molde») a un tipo diferente del tipo del objeto apuntado. Por ejemplo, una máquina podría suponer que a todos los *double* se les asigna un límite de 8 bytes. En ese caso, se produciría un comportamiento extraño si *pi* apuntara a un *int* que no hubiera sido asignado así. Esta forma de conversión explícita de tipo es intrínsecamente insegura y peligrosa. En consecuencia, la notación usada, *static_cast*, se diseñó para que fuera peligrosa.

El uso primario de *void** es pasar punteros a funciones a las que no les está permitido hacer hipótesis sobre el tipo del objeto y devolver desde las funciones objetos no tipificados. Para utilizar un objeto así debemos usar conversión explícita de tipos.

Habitualmente las funciones que usan punteros *void** tienen existencia al nivel más bajo del sistema, donde se manipulan recursos reales de hardware. Por ejemplo:

```
void* my_alloc(size_t n);   // asignar n bytes de mi montículo especial
```

La aparición de *void** en niveles superiores del sistema debe considerarse sospechosa, indicador probable de errores de diseño. Cuando se usa para optimización, *void** se puede ocultar tras una interfaz segura con respecto al tipo (§13.5, §24.4.2).

No se pueden asignar a *void** punteros a funciones (§7.7) ni punteros a miembros (§15.5).

5.7 Estructuras

Un array es un agregado de elementos del mismo tipo. Una *struct* es un agregado de elementos de tipos (casi) arbitrarios. Por ejemplo:

```
struct direccion {
   char* nombre;       // "Jim Dandy"
   long int numero;    // 61
   char* calle;        // "South St"
   char* ciudad;       // "New Providence"
   char* estado[2];    // 'N''J'
   int cp;             // 7974
};
```

Esto define un tipo nuevo llamado *direccion* que se compone de las unidades de datos que

son necesarias para enviar correo a alguien. Observe el lector el punto y coma final. Éste es uno de los escasísismos lugares en C++ en los que es necesario un punto y coma después de una llave, por lo que muchas personas se olvidan de él.

Se pueden declarar variables del tipo ***direccion*** exactamente igual que las demás y se puede acceder a cada uno de sus *miembros* con el operador . (punto). Por ejemplo:

```
void f()
{
  direccion jd;
  jd.nombre = "Jim Dandy";
  jd.numero = 61;
}
```

La rotación usada para inicializar arrays se puede usar para inicializar variables de tipo de estructura. For ejemplo:

```
direccion jd = {
  "Jim Dandy",
  61, "South St.",
  "New Providence", { 'N', 'J' }, 7974
};
```

Habitualmente es mejor, no obstante, usar un constructor (§10.2.3). Hay que señalar que ***jd.estado*** no puede ser inicializado por la cadena "***NJ***". Las cadenas son terminadas por el carácter '\0'. Por tanto, "***NJ***" tiene tres caracteres, uno más de los que caben en ***jd.estado***.

A los objetos de tipos de estructura se suele acceder por medio de punteros usando el operador -> (desreferencia de puntero de estructura). Por ejemplo:

```
void print_dir(direccion* p)
{
  cout << p->nombre << '\n'
    << p->numero << ' ' << p->calle << '\n';
    << p->ciudad << '\n'
    << p->estado[0] << p->estado[1] << ' ' << p->cp << '\n';
}
```

Donde ***p*** es un puntero, ***p->m*** es equivalente a (***p**) .m*.

Los objetos de tipos de estructura pueden ser asignados, pasados como argumentos de función y devueltos como el resultado de una función. Por ejemplo:

```
direccion actual;
direccion actualizar(direccion siguiente)
{
  direccion prev = actual;
  actual = siguiente;
  return prev;
}
```

Otras operaciones plausibles, como la comparación (== y !=) no están definidas. Sin embargo, el usuario puede definir esos operadores (capítulo 11).

El tamaño de un objeto de un tipo de estructura no es necesariamente la suma de los tamaños de sus miembros. Esto se debe a que muchas máquinas requieren que los objetos de ciertos tipos sean asignados en límites dependientes de la arquitectura o manejan esos objetos con mucha mayor eficiencia si lo están. For ejemplo, los enteros se asignan a menudo en límites de palabra. En tales máquinas se dice que los objetos tienen que estar *alineados* adecuadamente. Esto da lugar a «agujeros» en las estructuras. Por ejemplo, en muchas máquinas ***sizeof***(***direccion***) es ***24*** y no ***22*** como se podría esperar. El lector puede minimizar el despilfarro de espacio sencillamente ordenando los miembros por tamaño (primero el miembro más grande). Sin embargo, habitualmente es mejor ordenar los miembros teniendo en cuenta la legibilidad y clasificarlos por tamaño sólo si hay una necesidad demostrada de optimizar.

El nombre de un tipo está disponible para su uso inmediatamente después de que ha sido encontrado, sin esperar a que se haya visto toda la declaración. Por ejemplo:

```
struct Link {
  Link* previous;
  Link* successor;
};
```

No es posible declarar objetos nuevos de un tipo de estructura hasta que se ha visto toda la declaración. Por ejemplo:

```
struct No_good {
  No_good member;    // error: definición recursiva
};
```

Esto es un error porque el compilador no es capaz de determinar el tamaño de ***No_good***. Para posibilitar que dos (o más) tipos de estructura se refieran una a otra podemos declarar que un nombre sea el nombre de un tipo de estructura. Por ejemplo:

```
struct List;      // a definir más adelante
struct Link {
  Link* pre;
  Link* suc;
  List* member_of;
};
struct List {
  Link* head;
};
```

Sin la primera declaración de ***List***, el uso de ***List*** en la declaración de ***Link*** hubiera dado lugar a un error de sintaxis

El nombre de un tipo de estructura puede usarse antes de que sea definido el tipo siempre que ese uso no requiera que se conozca el nombre de un miembro o el tamaño de la estructura. Por ejemplo:

```
class S;      // 'S' es el nombre de algún tipo
extern S a;
S f();
void g(S);
```

```
S* h(S*);
```

Sin embargo, muchas de las siguientes declaraciones no se pueden usar a menos que se defina el tipo *S*:

```
void k(S* p)
{
    S a;          // error: S sin definir; es necesario el tamaño para asignar
    f();          // error: S sin definir; es necesario el tamaño para devolver valor
    g(a);         // error: S sin definir; es necesario tamaño para pasar argumentos
    p->m = 7;     // error: S sin definir; nombre de miembro no conocido
    S* q = h(p);  // bien: se pueden asignar y pasar punteros
    q->m = 7;     // error: S sin definir; nombre de miembro no conocido
}
```

Una ***struct*** es una forma sencilla de ***class*** (capítulo 10)

Por razones que se remontan a la prehistoria de C, es posible declarar una ***struct*** y una no-estructura con el mismo nombre en el mismo ámbito. Por ejemplo:

```
struct stat { /* ... */ };
int stat(char* name, struct stat* buf);
```

En este caso, el nombre simple (***stat***) es el nombre de la no-estructura, y a la estructura hay que referirse con el prefijo ***struct***. De forma semejante, se pueden usar como prefijos las palabras clave ***class***, ***union*** (§C.8.2) y ***enum*** (§4.8) para evitar ambigüedades. Sin embargo, es mejor no sobrecargar los nombres y que ello no sea necesario.

5.7.1 Equivalencia de tipo

Dos estructuras son tipos diferentes aunque tengan los mismos miembros. Por ejemplo,

```
struct S1 { int a; };
struct S2 { int a; };
```

son dos tipos diferentes, por tanto

```
S1 x;
S2 y = x;    // error: discordancia de tipo
```

Los tipos de estructuras son también diferentes de los tipos fundamentales, por tanto

```
S1 x;
int i = x;   // error: discordancia de tipo
```

Toda ***struct*** debe tener una única definición en un programa (§9.2.3).

5.8 Consejos

[1] Evite la aritmética no trivial de punteros; §5.3.
[2] Tenga cuidado de no escribir más allá de los límites de un array; §5.3.1.
[3] Use ***0*** en lugar de ***NULL***; §5.1.1.
[4] Use ***vector*** y ***valarray*** en lugar de arrays predefinidos (estilo C); §5.3.1.
[5] Use ***string*** en lugar de arrays de ***char*** terminados por un cero; §5.3.
[6] Reduzca al mínimo el uso de argumentos de referencia simples; §5.5.
[7] Evite ***void**** salvo en código de bajo nivel; §5.6.
[8] Evite los literales no triviales («números mágicos») en el código. En lugar de ellos, defina y use constantes simbólicas; §4.8, §5.4.

5.9 Ejercicios

1. (*1) Escriba declaraciones para lo siguiente: un puntero a un carácter, un arrray de 10 enteros, una referencia a un array de 10 enteros, un puntero a un array de cadenas de caracteres, un puntero a un puntero a un carácter, un entero constante, un puntero a un entero constante y un puntero constante a un entero. Inicialícelos todos.
2. (*1,5) ¿Cuáles son, en su sistema, las restricciones de los tipos de puntero ***char****, ***int**** y ***void****? Por ejemplo, ¿puede un ***int**** tener valor impar? Pista: alineación
3. (*1) Use ***typedef*** para definir los tipos ***unsigned char***, ***cons unsigned char***, puntero a entero, puntero a puntero a ***char***, puntero a arrays de ***char***, array de 7 punteros a ***int***, puntero a un array de 7 punteros a ***int*** y array de 8 arrays de 7 punteros a ***int***.
4. (*1) Escriba una función que permute (intercambie los valores de) dos enteros. Use ***int**** como tipo del argumento. Escriba otra función de permutación usando ***int***& como tipo del argumento.
5. (*1,5) ¿Cuál es el tamaño del array ***str*** en el ejemplo siguiente?:

   ```
   char str[] = "una cadena corta";
   ```

   ¿Cuál es la longitud de la cadena "***una cadena corta***"?
6. (*1) Defina funciones ***f***(***char***), ***g***(***char***&) y ***h***(***const char***&). Llámelas con los argumentos '***a***', ***49***, ***3300***, ***c***, ***uc*** y ***sc***, siendo ***c*** un ***char***, ***uc*** un ***unsigned char*** y ***sc*** un ***signed char***. ¿Qué llamadas son legales? ¿Qué llamadas hacen que el compilador introduzca una variable temporal?
7. (*1,5) Defina una tabla con los nombres de los meses del año y el número de días de cada mes. Escriba esa tabla. Hágalo dos veces: una usando un array de ***char*** para los nombres y un array para el número de días, y otra usando un array de estructuras, cada una de ellas con el nombre de un mes y su número de días.
8. (*2) Ejecute pruebas para ver si su compilador genera realmente código equivalente para iteración usando punteros e iteración usando indexación (§5.3.1). Si es posible solicitar diferentes grados de optimización, vea si ello afecta a la calidad del código generado y en qué modo.
9. (*1,5) Encuentre un ejemplo en el que tendría sentido usar un nombre en su propio inicializador.
10. (*1) Defina un array de cadenas en el que las cadenas contengan los nombres de los meses. Imprima esas cadenas. Pase el array a una función que imprima esas cadenas.
11. (*2) Lea una secuencia de palabras de la entrada. Use ***Quit*** como palabra que finaliza la entrada. Imprima las palabras en el orden en que fueron introducidas. No imprima ninguna palabra dos veces. Modifique el programa para clasificar las palabras antes de imprimirlas.
12. (*2) Escriba una función que cuente el número de apariciones de un par de letras en

una ***string*** y otra que haga lo mismo en un array de ***char*** terminado por un cero (cadena de estilo C). Por ejemplo, el par «ab» aparece dos veces en «xabaacbaxabb».

13. (*1,5) Defina una ***struct Fecha*** para guardar fechas. Proporcione las funciones necesarias para leer ***Fecha***s de la entrada, escribir ***Fecha***s en la salida e inicializar una ***Fecha*** con una fecha.

6

Expresiones y sentencias

La optimización prematura
es la raíz de todos los males.
D. Knuth

Por otra parte,
no podemos olvidarnos de la eficiencia.
Jon Bentley

Ejemplo de calculadora de sobremesa — entrada — argumentos de línea de órdenes — resumen de expresiones — operadores lógicos y relacionales — incremento y decremento — memoria libre — conversión explícita de tipos — resumen de sentencias — declaraciones — sentencias de selección — declaraciones en las condiciones — sentencias de iteración — el denigrado ***goto*** — comentarios y sangría — consejos — ejercicios.

6.1 Una calculadora de sobremesa

Para presentar las sentencias y expresiones se ofrece un programa de calculadora de sobremesa que proporciona las cuatro operaciones aritméticas estándar como operadores infijos sobre números en coma flotante. Por ejemplo, dada la entrada

```
r = 2.5
area = pi * r * r
```

(pi está predefinido) el programa de calculadora escribirá:

```
2.5
19.635
```

siendo ***2.5*** el resultado de la primera línea de entrada y ***19.635*** el resultado de la segunda.

La calculadora se compone de cuatro partes principales: un analizador sintáctico, una

función de entrada, una tabla de símbolos y un controlador. En realidad es un compilador en miniatura en el que el analizador sintáctico hace el análisis sintáctico, la función de entrada maneja la entrada y el análisis léxico, la tabla de símbolos contiene información permanente y el controlador maneja la inicialización, la salida y los errores. Podríamos añadir muchas características a esta calculadora para hacerla más útil (§6.6[20]), pero el código es ya suficientemente largo y la mayoría de las características sólo supondrían aumento de dicho código sin profundizar más en el uso de C+-.

6.1.1 El analizador sintáctico

Veamos una gramática para el lenguaje aceptado por la calculadora:

```
program:
    END                         // END es fin de la entrada
    expr_list END
expr_list:
    expression PRINT            // PRINT es punto y coma
    expression PRINT expr_list
expression:
    term + expression
    term - expression
    term
term:
    primary / term
    primary * term
    primary
primary:
    NUMBER
    NAME
    NAME = expression
    - primary
    (expression)
```

En otras palabras, un programa es una secuencia de expresiones separadas por punto y coma. Las unidades básicas de una expresión son los números, los nombres y los operadores *, /, +, - (tanto monarios como binarios) y =. No es necesario declarar los nombres antes de usarlos.

El estilo de análisis sintáctico usado se denomina habitualmente *descenso recursivo;* es una técnica de arriba abajo conocida y bastante sencilla. En un lenguaje como C+-, en el que las llamadas a función son relativamente poco costosas, es también eficiente. Para cada regla de producción de la gramática hay una función que llama a otras funciones. Los símbolos terminales (por ejemplo, ***END***, ***NUMBER***, + y -) son reconocidos por el analizador léxico ***get_token*** ( ) ; los símbolos no terminales son reconocidos por las funciones del analizador sintáctico ***expr*** ( ), ***term*** ( ) y ***prim*** ( ). En cuanto se conocen los dos operandos de una (sub)expresión se evalúa ésta; en un compilador real se generaría código en ese punto.

El analizador sintáctico usa una función ***get_token*** ( ) para obtener la entrada. El valor de la llamada más reciente a ***get_token*** ( ) puede encontrarse en la variable global ***curr_tok***. El tipo de ***curr_tok*** es la enumeración ***Token_value***:

```
enum Token_value {
    NAME,       NUMBER,     END,
    PLUS='+',   MINUS='-',  MUL='*',   DIV='/',
    PRINT=';',  ASSIGN='=', LP='(',    RP=')'
};
Token_value curr_tok = PRINT;
```

Representar cada componente léxico por el valor entero de su carácter es cómodo y eficiente, y puede ser una ayuda para quienes usan depuradores. Funciona siempre que ningún carácter de los utilizados como entrada tenga un valor usado como enumerador —y ningún juego de caracteres de los que conozco tiene un carácter imprimible con un valor entero de un solo dígito—. Elijo ***PRINT*** como valor inicial para ***curr_tok*** porque ése es el valor que tendrá después de que la calculadora haya evaluado la expresión y mostrado su valor. Así pues, «arranco el sistema» en un estado normal para reducir al mínimo la posibilidad de errores y la necesidad de código especial de arranque.

Cada una de las funciones del analizador sintáctico toma un argumento ***bool*** (§4.2) que indica si la función necesita llamar a ***get_token*** ( ) para obtener el siguiente componente léxico. Cada función del analizador sintáctico evalúa «su» expresión y devuelve el valor. La función ***expr*** ( ) maneja la suma y la resta. Se compone de un solo bucle que busca términos para sumar o restar:

```
double expr(bool get)           // sumar y restar
{
    double left = term(get);
    for(;;)                     // "para siempre"
        switch (curr_tok) {
        case PLUS:
            left += term(true);
            break;
        case MINUS:
            left -= term(true);
            break;
        default:
            return left;
        }
}
```

Esta función no hace gran cosa por sí misma. De una manera típica en las funciones de nivel superior de un programa grande, llama a otras funciones para que hagan el trabajo.

La *sentencia-switch* contrasta el valor de su condición, que es proporcionada entre paréntesis después de la palabra clave ***switch***, con un conjunto de constantes. Para salir de la *sentencia-switch* se usan *sentencias-break*. Las constantes que siguen a las etiquetas ***case*** han de ser distintas. Si el valor que se prueba no concuerda con ninguna etiqueta ***case*** se elige ***default*** (caso por defecto). No es necesario que el programador proporcione un ***default***.

Hay que señalar que una expresión como **2-3+4** es evaluada (**2-3**) +**4**, como se especifica en la gramática.

La curiosa notación ***for*** (; ;) es la forma estándar de especificar un bucle infinito; se podría leer como «para siempre». Es una forma degenerada de una *sentencia-for* (§6.3.3); ***while*** (***true***) es una alternativa. La *sentencia-switch* se ejecuta una y otra vez hasta que se encuentra algo diferente de + y −, ejecutándose entonces la *sentencia-return* en el caso por defecto

Los operadores += y −= se usan para manejar la suma y la resta; se podría haber usado ***left=left+term*** ( ) y ***left=left-term*** ( ) sin que cambiara el significado del programa. Sin embargo, ***left+=term*** ( ) y ***left-=term*** ( ) no sólo son más cortos, sino que además expresan directamente la operación que se pretende. Cada operador de asignación es un componente léxico independiente, por lo que ***a + = 1;*** es un error de sintaxis debido al espacio que hay entre el signo más y el signo igual.

Se proporcionan operadores de asignación para los operadores binarios

```
+   -   *   /   %   &   |   ^   <<   >>
```

de modo que son posibles los siguientes operadores de asignación

```
=   +=   -=   *=   /=   %=   &=   |=   ^=   <<=   >>=
```

El signo % es el operador módulo o resto; &, | y ^ son los operadores lógicos Y, O y O exclusiva a nivel de bits; << y >> son los operadores de desplazamiento a la izquierda y desplazamiento a la derecha; en §6.2 se resumen los operadores y su significado. Para un operador binario @ aplicado a operandos de tipos predefinidos, una expresión ***x@=y*** significa ***x=x@y***, salvo porque ***x*** se evalúa sólo una vez.

En los capítulos 8 y 9 se analiza cómo organizar un programa como un conjunto de módulos. Con una excepción, las declaraciones para este ejemplo de la calculadora podrían ordenarse de modo que todo se declarara exactamente una vez antes de utilizarse. La excepción es ***expr*** ( ), que llama a ***term*** ( ), que llama a ***prim*** ( ), que a su vez llama a ***expr*** ( ). Este bucle debe romperse de algún modo. Una declaración

```
double expr(bool);
```

antes de la definición de ***prim*** ( ) será idónea para hacerlo.

La función ***term*** ( ) maneja la multiplicación y la división de la misma forma que ***expr*** ( ) maneja la suma y la resta:

```
double term(bool get)      // multiplicar y dividir
{
    double left = prim(get);
    for ( ; ; )
        switch (curr_tok) {
        case MUL:
            left *= prim(true);
            break;
        case DIV:
            if (double d = prim(true)) {
                left /= d;
                break;
            }
            return error("dividir por 0");
        default:
            return left;
        }
}
```

El resultado de dividir por cero es no definido y habitualmente desastroso. En consecuencia, comprobamos si es ***0*** antes de dividir y llamamos a ***error*** ( ) si detectamos un divisor cero. La función ***error*** ( ) se describe en §6.1.4.

La variable ***d*** es introducida en el programa exactamente donde se necesita y es inicializada inmediatamente. El ámbito de un nombre introducido en una condición es la sentencia controlada por esa condición y el valor resultante es el valor de la condición (§6.3.2.1). En consecuencia, la división y la asignación ***left/=d*** se hace si y sólo si ***d*** es distinta de cero.

La función ***prim*** ( ) que maneja un elemento *primario* es muy parecida a ***expr*** ( ) y ***term*** ( ) pero, como estamos descendiendo en la jerarquía de llamadas, se hace ya un poco de trabajo real y no es necesario ningún bucle:

```
double number_value;
string string_value;
double prim(bool get)          // manejar primarios
{
    if (get) get_token();
    switch (curr_tok) {
    case NUMBER:               // constante de coma flotante
    {   double v = number_value;
        get_token();
        return v;
    }
    case NAME:
    {   double& v = table[string_value];
        if (get_token() == ASSIGN)  v = expr(true);
        return v;
    }
    case MINUS:                // menos monario
        return -prim(true);
    case LP:
    {   double e = expr(true);
        if (curr_tok != RP) return error(") previsto");
        get_token();           // comerse ')'
        return e;
    }
    default:
        return error("espera un elemento primario");
    }
}
```

Cuando se ve un ***NUMBER*** (es decir, un entero o un literal en coma flotante) se devuelve su valor. La rutina de entrada ***get_token*** ( ) coloca el valor en la variable global ***number_value***. El uso de una variable global en un programa indica a menudo que la estructura no es muy limpia: que se ha aplicado alguna forma de optimización. Como aquí. En condiciones ideales, un componente léxico consta de dos partes: un valor que especifica la modalidad de componente (un ***Token_value*** en este programa) y (cuando es necesario) el valor del componente. En este caso hay sólo una variable única y sencilla, ***curr_tok***, de modo que es necesaria la variable global ***number_value*** para contener el valor del último ***NUMBER*** leído. La eliminación de esta variable global espuria se ha dejado como ejercicio (§6.6[21]). Realmente no es necesario guardar el valor de ***number_value*** en la variable local ***v*** antes de llamar a ***get_token*** ( ). Para las entradas ilegales la calculadora usa siempre un número en el cálculo antes de leer otro de la entrada. Sin embargo, guardar el valor y mostrarlo correctamente después de un error ayuda al usuario.

De la misma forma que el valor del último ***NUMBER*** se guarda en ***number_value***, la representación de la cadena de caracteres del último ***NAME*** visto se guarda en ***string_value***. Antes de hacer nada con un nombre, la calculadora debe mirar primero hacia delante para ver si está siendo asignado o simplemente leído. En ambos casos se consulta la tabla de símbolos. La tabla de símbolos es un ***map*** (§3.7.4, §17.4.1):

```
map<string,double>table;
```

Es decir, cuando se indexa ***table*** por una ***string***, el valor resultante es el ***double*** correspondiente a la ***string***. Por ejemplo, si el usuario introduce:

```
radio = 6378.388;
```

la calculadora ejecutará

```
double& v = table["radio"];
// ... expr() calcula el valor a asignar ...
v = 6378.388
```

Se usa la referencia ***v*** para retener el ***double*** asociado con ***radio*** mientras ***expr*** ( ) calcula el valor ***6378.388*** a partir de los caracteres de entrada.

6.1.2 La función de entrada

Leer entrada es a menudo la parte más confusa de un programa. Ello se debe a que el programa debe comunicarse con una persona, debe hacer frente a los caprichos de esa persona, las convenciones y los errores aparentemente aleatorios. Tratar de obligar a la persona a comportarse de una manera más adecuada a la máquina se considera a menudo (y con razón) ofensivo. La tarea de una rutina de entrada de bajo nivel es leer caracteres y formar componentes de nivel superior a partir de ellos. Estos componentes son entonces las unidades de entrada para rutinas de nivel superior. En este caso, la rutina de bajo nivel la realiza ***get_token*** ( ). Escribir una rutina de entrada de bajo nivel no debe ser una tarea cotidiana. Muchos sistemas proporcionan funciones estándar para ello.

Construyo ***get_token*** ( ) en dos etapas. En primer lugar, proporciono una versión engañosamente simple que impone una carga al usuario. A continuación la modifico para convertirla en una versión mucho menos elegante pero mucho más fácil de usar.

La idea es leer un carácter, usar ese carácter para decidir qué tipo de componente léxico es necesario formar y devolver entonces el ***Token_value*** que representa el componente léxico leído.

Las sentencias iniciales leen el primer carácter que no sea espacio en blanco a ***ch*** y comprueban que la operación de lectura haya tenido éxito:

```
Token_value get_token()
{
  char ch = 0;
  cin>>ch;
  switch (ch) {
  case 0:
    return curr_tok=END;        // asignar y devolver
```

Por defecto, el operador >> se salta el espacio en blanco (es decir, espacios, tabuladores, saltos de línea, etcétera) y no modifica el valor de ***ch*** si la operación de entrada no ha tenido éxito. En consecuencia, ***ch==0*** indica fin de la entrada.

La asignación es un operador y el resultado de la asignación es el valor de la variable a la que se ha asignado. Esto me permite asignar el valor ***END*** a ***curr_tok*** y devolverlo en la misma sentencia. Tener una sentencia en lugar de dos es útil para mantenimiento. Si la asignación y la devolución vienen separadas en el código, un programador podría actualizar una y olvidarse de actualizar la otra.

Vamos a ver por separado algunos de los casos antes de pasar a considerar la función completa. El terminador de la expresión ';', el paréntesis y los operadores se manejan sencillamente devolviendo sus valores:

```
  case ';':
  case '*':
  case '/':
  case '+':
  case '-':
  case '(':
  case ')':
  case '=':
    return curr_took=Token_value(ch);
```

Los números se manejan así:

```
  case '0': case '1': case '2': case '3': case '4':
  case '5': case '6': case '7': case '8': case '9':
  case '.':
    cin.putback(ch);
    cin>>number_value;
    return curr_tok=NUMBER;
```

Apilar las etiquetas ***case*** en horizontal en lugar de en vertical no es generalmente buena idea, porque se trata de una disposición más difícil de leer. Sin embargo, tener una línea para cada dígito es tedioso. Como el operador >> se ha definido ya para leer constantes en coma flotante a un ***double***, el código es trivial. Primero se devuelve a ***cin*** el carácter inicial (un dígito o un punto). Entonces se puede leer la constante a ***number_value***.

Los nombres se manejan de forma similar:

```
default:                    // NAME, NAME=, o error
    if (isalpha(ch)) {
        cin.putback(ch);
        cin>>string_value;
        return curr_tok=NAME;
    }
    error("bad token");
    return curr_tok=PRINT;
```

La función *isalpha* (§20.4.2) de la biblioteca estándar se usa para no tener que hacer enumerar todos los caracteres como etiqueta *case* aparte. El operador >> aplicado a una cadena (en este caso, *string_value*) lee hasta que choca con espacio en blanco. En consecuencia, el usuario debe terminar un nombre con un espacio delante de un operador que use el nombre como operando. Esto está lejos de ser ideal, por lo que volveremos sobre el problema en §6.1.3.

Veamos, por fin, la función de entrada completa:

```
Token_value get_token()
{
    char ch = 0;
    cin>>ch;
    switch (ch) {
    case 0:
        return curr_tok=END;
    case ';':
    case '*':
    case '/':
    case '+':
    case '-':
    case '(':
    case ')':
    case '=':
        return curr_took=Token_value(ch);
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
    case '.':
        cin.puiback(ch);
        cin>>number_value;
        return curr_tok=NUMBER;
    default:                    // NAME, NAME=, o error
        if(isalpha(ch)) {
            cin.putback(ch);
            cin>>string_value;
            return curr_tok=NAME;
        }
        error("bad token");
        return curr_tok=PRINT;
    }
}
```

La conversión de un operador a su valor de componente léxico es trivial porque el *Token_value* de un operador se ha definido como el valor entero del operador (§4.8).

6.1.3 Entrada de bajo nivel

El uso de la calculadora tal como se ha definido hasta ahora pone de manifiesto unos cuantos inconvenientes. Es tedioso tener que acordarse de añadir un punto y coma después de una expresión para conseguir que su valor se imprima, y que los nombres finalicen sólo con espacio en blanco es una verdadera molestia. Por ejemplo, *x=7* es un identificador, en lugar de ser el identificador *x* seguido del operador = y del número *7*. Ambos problemas se resuelven sustituyendo las operaciones de entrada por defecto, orientadas al tipo, de *get_token*() por código que lea caracteres individuales.

En primer lugar, haremos que un salto de línea equivalga al punto y coma usado para marcar el final de la expresión:

```
Token_value get_token()
{
    char ch;
    do {            // saltar espacio en blanco salvo '\n'
        if(!cin.get(ch)) return curr_tok = END;
    } while (ch!='\n' && isspace(ch));
    switch (ch) {
    case ';':
    case '\n':
        return curr_tok=PRINT;
```

Se usa una *sentencia-do*; es equivalente a una *sentencia-while* salvo porque la sentencia controlada se ejecuta siempre al menos una vez. La llamada *cin.get(ch)* lee un solo carácter del flujo de entrada estándar a *ch*. Por defecto, *get*() no se salta el espacio en blanco como lo hace el operador >>. La prueba *if*(!*cin.get*(*ch*)) falla si no se puede leer ningún carácter en *cin*; en ese caso, se devuelve *END* para finalizar la sesión de calculadora. Se usa el operador ! (NO) porque *get*() devuelve *true* en caso de éxito.

La función *isspace*() de la biblioteca estándar proporciona la prueba estándar para el espacio en blanco (§20.4.2); *isspace*(*c*) devuelve un valor distinto de cero si *c* es un carácter de espacio en blanco y cero en caso contrario. La prueba se implementa como una búsqueda en tabla, por lo que es mucho más rápido usar *isspace*() que probar los distintos caracteres de espacio en blanco. Funciones similares prueban si un carácter es un dígito —*isdigit*()—, una letra —*isalpha*()— o un dígito o letra —*isalnum*()—.

Una vez saltado el espacio en blanco se usa el carácter siguiente para determinar qué modalidad de componente léxico viene a continuación.

El problema causado porque >> lea en una cadena hasta que se encuentra espacio en

blanco se resuelve leyendo los caracteres de uno en uno hasta que se encuentra un carácter que no es una letra ni un dígito:

```
default:            // NAME,NAME=,o error
  if (isalpha(ch)) {
    string_value = ch;
    while (cin.get(ch) && isalnum(ch)) string_value.push_back(ch);
    cin.putback(ch);
    return curr_tok=NAME;
  }
  error("bad token")
  return curr_tok=PRINT;
```

Por fortuna, estas dos mejoras podrían implementarse modificando una sola sección local del código. Construir los programas de modo que se pueden implementar mejoras a través de modificaciones locales es un objetivo importante del diseño.

6.1.4 Manejo de errores

Como el programa es tan sencillo, el manejo de errores no es una preocupación importante. La función de error cuenta simplemente los errores, escribe un mensaje de error y vuelve:

```
int no_of_errors;

double error(const string& s)
{
  no_of_errors++;
  cerr<<"error: " << s << '\n';
  return 1;
}
```

El flujo *cerr* es un flujo de salida no buferado que se usa habitualmente para informar de errores (§21.2.1).

La razón para devolver un valor es que los errores se producen habitualmente en medio de la evaluación de una expresión, por lo que debemos interrumpir por completo esa evaluación o devolver un valor que no sea probable que cause errores posteriores. Lo último es más adecuado para nuestra calculadora sencilla. Si *get_token* hubiera seguido el rastro de los números de línea, *error*() podría haber informado aproximadamente al usuario de dónde se había producido el error. Esto sería útil cuando se usara la calculadora de forma no interactiva (§6.6[19]]).

A menudo un programa debe finalizar después de que se haya producido un error porque no se ha ideado una forma sensata de continuar. Esto se puede hacer llamando a *exit*(), que limpia primero cosas como los flujos de salida y finaliza luego el programa con su argumento como valor devuelto (§9.4.1.1).

Se pueden implementar mecanismos de manejo de errores más estilizados usando excepciones (véase §8.3, capítulo 14) pero el que tenemos aquí es bastante adecuado para una calculadora de 150 líneas.

6.1.5 El controlador

Con todos los elementos del programa en su lugar, sólo necesitamos un controlador que inicie las cosas. En este ejemplo sencillo puede hacerlo *main*():

```
int main()
{
  table["pi"] = 3.1415926535897932385;      // insertar nombres predefinidos
  table["e"] = 2.7182818284590452354;
  while (cin) {
    get_token();
    if (curr_tok == END) break;
    if (curr_tok == PRINT) continue;
    cout << expr(false) << '\n';
  }
  return no_of_errors;
}
```

Convencionalmente, *main*() debe devolver cero si el programa finaliza normalmente y un valor distinto de cero en caso contrario (§3.2). Devolver el número de errores se ajusta muy bien a esto. Como ocurre, la única inicialización que se necesita es insertar los nombres predefinidos en la tabla de símbolos.

La tarea primaria del bucle principal es leer las expresiones y escribir la respuesta. Esto se logra con la línea:

```
cout << expr(false) << '\n';
```

El argumento *false* dice a *expr*() que no es necesario llamar a *get_token*() para obtener un componente léxico actual sobre el que trabajar.

Probar *cin* cada vez en torno al bucle garantiza que el programa finalice si algo va mal con el flujo de entrada y la prueba de *END* garantiza que se salga correctamente del bucle cuando *get_token*() encuentra el final del archivo. Una *sentencia-break* sale de su *sentencia-switch* o bucle (es decir, *sentencia-for*, *sentencia-while* o *sentencia-do*) inclusivo más próximo. La prueba de *PRINT* (es decir, de '\n' y ';') libera a *expr*() de la responsabilidad de manejar expresiones vacías. Una *sentencia-continue* equivale a ir hasta el final de un bucle, por lo que en este caso

```
while (cin) {
  //...
  if (curr_tok == PRINT) continue;
  cout << expr() << '\n';
  :
```

es equivalente a

```
while (cin) {
  //...
  if (curr_tok != PRINT)
    cout << expr() << '\n';
  :
```

6.1.6 Cabeceras

La calculadora usa componentes de la biblioteca estándar. Por tanto, hay que incluir (*#include*) las cabeceras adecuadas para completar el programa:

```
#include<iostream>    // E/S
#include<string>      // cadenas
#include<map>         // mapa
#include<cctype>      // isalpha(), etc.
```

Todas estas cabeceras proporcionan componentes del espacio de nombres ***std***, por lo que para usar los nombres que proporcionan debemos usar cualificación explícita con ***std***:: o llevar los nombres al espacio de nombres global con

```
using namespace std;
```

Para evitar confundir el análisis de las expresiones con problemas de modularidad, he hecho lo último. En los capítulos 8 y 9 se analizan formas de organizar esta calculadora en módulos usando espacios de nombres y cómo organizarla en varios archivos fuente. En muchos sistemas las cabeceras estándar tienen equivalentes con sufijo ***.h*** que declaran clases, funciones, etcétera, y las colocan en el espacio de nombres global (§9.2.1, §9.2.4, §3.3.1).

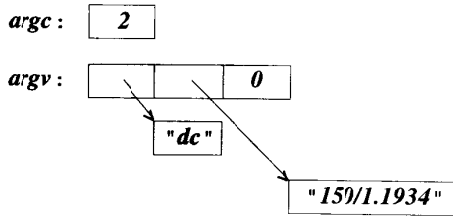
6.1.7 Argumentos de línea de órdenes

Una vez que el programa estuvo escrito y probado, me pareció molesto tener que arrancar primero el programa, teclear después las expresiones y finalmente salir. Mi uso más habitual era para evaluar una sola expresión. Si se podía presentar esa expresión como argumento en la línea de órdenes se ahorrarían unas cuantas pulsaciones.

Los programas comienzan llamando a ***main*** ( ) (§3.2, §9.4). Hecho esto, se dan a ***main*** ( ) dos argumentos que especifican el número de argumentos, habitualmente llamados ***argc***, y un array de argumentos, habitualmente llamado ***argv***. Los argumentos son cadenas de caracteres, por lo que el tipo de ***argv*** es ***char**** [***argc+1***]. El nombre del programa (tal como aparece en la línea de órdenes) es pasado a ***argv***[***0***], por lo que ***argc*** es siempre como mínimo ***1***. La lista de argumentos termina con cero; es decir, ***argv***[***argc***]==***0***. Por ejemplo, para la orden

```
dc 150/1.1934
```

los argumentos tienen los valores siguientes:



Dado que las convenciones para llamar a ***main***() se comparten con C, se usan arrays y cadenas de estilo C.

No es difícil conseguir un argumento de la línea de órdenes. El problema está en cómo usarlo con una reprogramación mínima. La idea es leer de la cadena de órdenes de la misma forma que leemos del flujo de entrada. Un flujo que lee de una cadena se denomina ***istringstream***. Desafortunadamente, no hay ninguna forma elegante de hacer que ***cin*** se refiera a un ***istringstream***. Por tanto, hemos de encontrar una forma de conseguir que las funciones de entrada de la calculadora se refieran a un ***istringstream***. Además, hemos de encontrar una forma de conseguir que las funciones de entrada de la calculadora se refieran a un ***istringstream*** o a ***cin*** dependiendo del tipo de argumento de línea de órdenes que proporcionemos.

Una solución sencilla consiste en introducir un puntero global ***input*** que apunte al flujo de entrada que se va a usar y hacer que todas las rutinas de entrada lo usen:

```
istream* input;      // puntero al flujo de entrada
int main (int argc, char* argv[])
{
  switch (argc) {
  case 1:                                   // leer de la entrada estándar
    input = &cin;
    break;
  case 2:                                   // leer cadena de argumentos
    input = new istringstream(argv[1]);
    break;
  default:
    error("demasiados argumentos");
    return 1;
  }
  table["pi"] = 3.14159265358979323385;     // insertar nombres predefinidos
  table["e"] = 2.71828182845904523354;
  while (*input) {
    get_token();
    if (curr_tok == END) break;
    if (curr_tok == PRINT) continue;
    cout << expr(false) << '\n';
  }
  return no_of_errors;
}
```

Un ***istringstream*** es una modalidad de ***istream*** que lee de su argumento de cadena de caracteres (§21.5.3). Al alcanzar el final de su cadena, un ***istringstream*** falla exactamente igual que lo hacen otros flujos cuando llegan al final de la entrada (§3.6, §21.3.3). Para usar un ***istringstream*** hay que incluir <***stringstream***>.

Sería fácil modificar ***main*** ( ) para que aceptara varios argumentos de la línea de órdenes, pero no parece que sea necesario, especialmente porque se pueden pasar varias expresiones como un solo argumento:

```
dc "rate=1.1934; 150/rate; 19.75/rate; 217/rate"
```

Utilizo comillas porque ; es el separador de órdenes en mis sistemas UNIX. Otros sistemas tienen diferentes convenciones para proporcionar argumentos a un programa en el arranque.

No ha sido elegante modificar todas las rutinas de entrada para usar ***input*** en lugar de ***cin*** y ganar la flexibilidad de usar fuentes alternativas de entrada. Podría haber evitado el cambio si me hubiera mostrado previsor y hubiera introducido algo como ***input*** desde el principio. Un punto de vista más general y útil es señalar que la fuente de entrada debe ser en realidad el parámetro de un módulo calculadora. Es decir, el problema fundamental de este ejemplo de calculadora es que lo que he denominado «la calculadora» es sólo una colección de funciones y datos. No hay módulo (§2.4) ni objeto (§2.5.2) que represente explícitamente a la calculadora. Si hubiera empezado por diseñar un módulo de calculadora o un tipo de calculadora, naturalmente habría considerado cuáles debían ser sus parámetros (§8.5[3], 10.6[16]).

6.1.8 Nota sobre el estilo

A los programadores poco familiarizados con los arrays asociativos el uso del ***map*** de la biblioteca estándar como tabla de símbolos les parece casi un fraude. No lo es. La biblioteca estándar y las demás bibliotecas están pensadas para ser usadas. A menudo una biblioteca ha recibido más atención y cuidado en su diseño e implementación de los que un programador podría dedicar a un fragmento artesanal de código que se va a usar sólo en un programa.

Si examinamos el código para la calculadora, sobre todo en la primera versión, vemos que no presenta demasiado código tradicional de estilo C, de bajo nivel. Muchos de los detalles truculentos tradicionales han sido sustituidos por usos de clases de la biblioteca estándar como ***ostream***, ***string*** y ***map*** (§3.4, §3.5, §3.7.4, capítulo 17).

Hay que señalar la relativa escasez de aritmética, bucles e incluso asignaciones. Así deben ser las cosas en código que no manipula directamente hardware ni implementa abstracciones de bajo nivel.

6.2 Resumen de operadores

En este apartado se presenta un resumen de expresiones y algunos ejemplos. Cada operador va seguido por uno o más nombres usados habitualmente para él y un ejemplo de uso. En estas tablas *nombre_de_clase* es el nombre de una clase, *miembro* es el nombre de un miembro, *objeto* es una expresión que da un objeto de clase, *puntero* es una expresión que da un puntero, *expr* es una expresión y *valor-l* es una expresión que denota un objeto no constante. Un *tipo* sólo puede ser un nombre de tipo totalmente general (con *, ( ), etcétera) cuando aparece entre paréntesis; en caso contrario hay restricciones (§A.5).

La sintaxis de las expresiones es independiente de los tipos de los operandos. Los significados presentados aquí son válidos cuando los operandos son de tipos predefinidos (§4.1.1). Además, el lector puede definir significados para los operadores aplicados a los operandos de tipos definidos por el usuario (§2.5.2, capítulo 11).

Resumen de operadores

resolución de ámbito	*nombre_de_clase* : : *miembro*
resolución de ámbito	*nombre_de_namespace* : : *miembro*
global	: : *nombre*
global	: : *nombre-cualificado*
selección de miembro	*objeto* . *miembro*
selección de miembro	*puntero* -> *miembro*
subindexación	*puntero* [*expr*]
llamada a función	*expr* (*lista_de_expr*)
construcción de valor	*tipo* (*lista_de_expr*)
postincremento	*valor-l*++
posdecremento	*valor-l*--
identificación de tipo	***typeid*** (*tipo*)
identificación de tipo en tiempo de ejecución	***typeid*** (*expr*)
conversión comprobada en tiempo de ejecución	***dynamic_cast*** < *tipo* > (*expr*)
conversión comprobada en tiempo de compilación	***static_cast*** < *tipo* > (*expr*)
conversión no comprobada	***reinterpret_cast*** < *tipo* > (*expr*)
conversión ***const***	***const_cast*** < *tipo* > (*expr*)
tamaño de objeto	***sizeof*** *expr*
tamaño de tipo	***sizeof*** (*tipo*)
preincremento	++*valor-l*
predecremento	--*valor-l*
complemento	~ *expr*
no	! *expr*
menos monario	- *expr*
más monario	+ *expr*
dirección de	& *valor-l*
desreferencia	* *expr*
crear (asignar)	***new*** *tipo*
crear (asignar e inicializar)	***new*** *tipo* (*lista_de_expr*)
crear (colocar)	***new*** (*lista_de_expr*) *tipo*
crear (colocar e inicializar)	***new*** (*lista_de_expr*) *tipo* (*lista_de_expr*)
destruir (desasignar)	***delete*** *puntero*
destruir array	***delete*** [ ] *puntero*
molde (conversión de tipo)	(*tipo*) *expr*
selección de miembro	*objeto* . * *puntero-a-miembro*
selección de miembro	*puntero* -> * *puntero-a-miembro*
multiplicar	*expr* * *expr*
dividir	*expr* / *expr*
módulo (resto)	*expr* % *expr*
sumar (más)	*expr* + *expr*
restar (menos)	*expr* - *expr*

Resumen de operadores *(continuación)*	
desplazamiento a la izquierda	*expr* << *expr*
desplazamiento a la derecha	*expr* >> *expr*
menor que	*expr* < *expr*
menor o igual que	*expr* <= *expr*
mayor que	*expr* > *expr*
mayor o igual que	*expr* >= *expr*
igual	*expr* == *expr*
distinto	*expr* != *expr*
Y a nivel de bits	*expr* & *expr*
O exclusiva a nivel de bits	*expr* ^ *expr*
O inclusiva a nivel de bits	*expr* | *expr*
Y lógica	*expr* && *expr*
O lógica inclusiva	*expr* || *expr*
asignación simple	*valor-l* = *expr*
multiplicar y asignar	*valor-l* *= *expr*
dividir y asignar	*valor-l* /= *expr*
módulo y asignar	*valor-l* %= *expr*
sumar y asignar	*valor-l* += *expr*
restar y asignar	*valor-l* -= *expr*
desplazamiento a la izquierda y asignación	*valor-l* <<= *expr*
desplazamiento a la derecha y asignación	*valor-l* >>= *expr*
Y y asignación	*valor-l* &= *expr*
O inclusiva y asignación	*valor-l* |= *expr*
O exclusiva y asignación	*valor-l* ^= *expr*
expresión condicional	*expr* ? *expr* : *expr*
lanzar excepción	***throw*** *expr*
coma (secuencia)	*expr* , *expr*

Cada recuadro contiene operadores con idéntica precedencia. Los operadores de los recuadros superiores tienen mayor precedencia que los operadores de los recuadros inferiores. Por ejemplo, `a+b*c` significa `a+(b*c)`, en lugar de `(a+b)*c`, porque * tiene mayor precedencia que +.

Los operadores monarios y los de asignación son asociativos por la derecha; todos los demás son asociativos por la izquierda. Por ejemplo `a=b=c` significa `a=(b=c)`, `a+b+c` significa `(a+b)+c` y `*p++` significa `*(p++)`, no `(*p)++`.

Hay unas pocas reglas de gramática que no se pueden expresar en términos de precedencia (conocida también como fuerza de vinculación) y asociatividad. Por ejemplo, `a=b<c?d=e:f=g` significa `(a=b<c)?(d=e):(f=g)`, pero el lector debe consultar la gramática (§A.5) para poder determinarlo.

6.2.1 Resultados

Los tipos del resultado de los operadores aritméticos están determinados por un conjunto de reglas conocidas como «conversiones aritméticas habituales» (§C.6.3). El objetivo general es producir un resultado del tipo del operando «más grande». Por ejemplo, si un operador binario tiene un operando en coma flotante, se hace el cálculo usando aritmética de coma flotante y el resultado es un valor en coma flotante. Si tiene un operando ***long***, se hace el cálculo usando aritmética de enteros largos y el resultado es un ***long***. Los operandos que son más pequeños que un ***int*** (como ***bool*** y ***char***) son convertidos en ***int*** antes de que se aplique el operador.

Los operadores relacionales, ==, <=, etcétera, producen resultados booleanos. El significado y tipo del resultado de los operadores definidos por el usuario están determinados por su declaración (§11.2).

Cuando es viable desde el punto de vista lógico, el resultado de un operador que toma un operando valor-l es un valor-l que denota ese operando valor-l. Por ejemplo:

```
void f(int x, int y)
{
    int j = x = y;          // el valor de x=y es el valor de x después de la asignación
    int* p = &++x;          // p apunta a x
    int* q = &(x++);        // error: x++ no es un valor-l (no es el valor
                            // almacenado en x)
    int* pp = &(x>y:x,y);   // dirección del int con el valor más grande
}
```

Si tanto el segundo como el tercer operando de ?: son valores-l y tienen el mismo tipo, el resultado es de ese tipo y es valor-l. Conservar los valores-l de esta forma permite mayor flexibilidad en el uso de los operadores, algo especialmente útil cuando se escribe código que debe trabajar de manera uniforme y eficiente con tipos tanto predefinidos como definidos por el usuario (es decir, al escribir plantillas o programas que generen código C++).

El resultado de ***sizeof*** es de un tipo entero sin signo denominado ***size_t*** y definido en <***cstddef***>. El resultado de restar punteros es de un tipo entero con signo denominado ***ptrdiff_t*** y definido en <***cstddef***>.

Las implementaciones no tienen que comprobar el desbordamiento aritmético y casi ninguna lo hace. Por ejemplo:

```
void f()
{
    int i = 1;
    while (0<i) i++;
    cout << "i se ha hecho negativo! " << i << '\n';
}
```

Esto intentará (eventualmente) aumentar ***i*** por encima del entero más grande. Lo que ocurra entonces es no definido, pero habitualmente el valor pasa a ser el mayor número negativo (en mi máquina, -2147483648). De forma semejante, el efecto de dividir por cero es no definido, pero hacerlo causa casi siempre la terminación brusca del programa. En concreto, subdesbordamiento, desbordamiento y división por cero no lanzan excepciones estándar (§14.10).

6.2.2 Orden de evaluación

El orden de evaluación de las subexpresiones dentro de una expresión es no definido. En concreto, no se puede dar por supuesto que la expresión se vaya a evaluar de izquierda a derecha. Por ejemplo:

```
int x = f(2) + g(3) ;      // no definido si se llama primero a f() o a g()
```

Se puede generar mejor código en ausencia de restricciones en cuanto al orden de evaluación de la expresión. Sin embargo, la ausencia de restricciones en cuanto al orden de evaluación puede dar lugar a resultados no definidos. Por ejemplo,

```
int i = 1;
v[i] = i++;        // resultado no definido
```

puede ser evaluado como `v[1] = 1` o `v[2] = 1`, o puede causar un comportamiento aún más extraño. Los compiladores pueden advertir sobre tales ambigüedades. Por desgracia, la mayoría no lo hacen.

Los operadores , (coma), && (y lógica) y || (o lógica) garantizan que el operando de su izquierda sea evaluado antes que el de su derecha. Por ejemplo, `b= (a=2, a+1)` asigna **3** a **b**. En §.2.3 puede encontrar el lector ejemplos del uso de || y &&. Para los tipos predefinidos, el segundo operando de && se evalúa sólo si el primer operando es ***true*** y el segundo operando de || se evalúa sólo si el primer operando es ***false***; es lo que se denomina a veces *evaluación en cortocircuito*. Observe el lector que el operador de secuencia , (coma) es diferente desde el punto de vista lógico del que se usa para separar argumentos en una llamada de función. Consideremos:

```
f1 (v[i], i++) ;        // dos argumentos
f2 ( (v[i], i++) ) ;    // un argumento
```

La llamada de *f1* tiene dos argumentos, `v[i]` e `i++`, y el orden de evaluación de las expresiones del argumento es no definido. La dependencia del orden de las expresiones del argumento es de muy mal estilo y tiene un comportamiento no definido. La llamada de *f2* tiene un argumento, la expresión con coma `(v[i], i++)`, que es equivalente a `i++`.

Se puede usar paréntesis para forzar el agrupamiento. Por ejemplo, `a*b/c` puede ser evaluado como `a* (b/c)` o como `(a*b)/c`, pero `a* (b/c)` sólo puede ser evaluado como `(a*b)/c` si el usuario no puede ver la diferencia. En concreto, para muchos cálculos en coma flotante `a* (b/c)` y `(a*b)/c` son significativamente diferentes, por lo que un compilador evaluará esas expresiones exactamente como están escritas.

6.2.3 Precedencia de los operadores

Los niveles de precedencia y las reglas de asociatividad reflejan el uso más habitual. Por ejemplo,

```
if (i<0 || max<i)   // ...
```

significa «si `i` es menor o igual que **0** o si ***max*** es menor que ***i***». Es decir, es equivalente a

```
if ( (i<=0) || (max<i) )   // ...
```

y no lo legal pero sin sentido

```
if (i<= (0||max) <i   // ...
```

Sin embargo, se debe usar el paréntesis siempre que el programador tenga dudas sobre esas reglas. El uso de paréntesis se hace más frecuente a medida que las subexpresiones son más complicadas, pero las subexpresiones complicadas son una fuente de errores. Por tanto, si el lector comienza a notar que necesita usar paréntesis, debe considerar la posibilidad de descomponer la expresión usando una variable extra.

Hay casos en los que la precedencia de operadores no da lugar a la interpretación «obvia». Por ejemplo:

```
if (i&mask == 0)    // ¡vaya! expresión == como operando para &
```

Esto no aplica una máscara a `i` y luego comprueba si el resultado es cero. Como == tiene mayor precedencia que &, la expresión es interpretada como `i& (mask==0)`. Afortunadamente es bastante fácil para un compilador advertir sobre la mayoría de los errores semejantes. En este caso el paréntesis es importante:

```
if ( (i&mask) ==0)    // ...
```

Merece la pena señalar que lo siguiente no funciona como un matemático podría esperar:

```
if (0<= x <=99)    // ...
```

Esto es legal, pero se interpreta como `(0<=x) <=99`, donde el resultado de la primera comparación es ***true*** o ***false***. Ese valor booleano es entonces convertido implícitamente a **1** o **0**, y comparado a continuación con **99**, lo que produce ***true***. Para comprobar si `x` está en el rango **0**. .**99** podríamos usar:

```
if (0<=x&&x<99)    // ...
```

Un error habitual entre los principiantes es el uso de = (asignación) en lugar de == (igual) en una condición:

```
if (a = 7)      // ¡vaya! asignación de constante en condición
```

Esto es natural porque = significa «igual» en muchos lenguajes. Para un compilador es fácil advertir sobre estos errores, y la mayoría lo hacen.

6.2.4 Operadores lógicos a nivel de bits

Los operadores lógicos a nivel de bits &, |, ^, ~, >> y << se aplican a objetos de tipos enteros, es decir, ***bool***, ***char***, ***short***, ***int***, ***long*** y sus equivalentes ***unsigned***. Los resultados son también enteros.

Un uso típico de los operadores lógicos a nivel de bits es para implementar la noción de un conjunto pequeño (como un vector de bits). En este caso, cada bit de un entero sin signo representa a un miembro del conjunto, y el número de bits limita el número de miembros. El operador binario & se interpreta como intersección, | como unión, ^ como diferencia simétrica y ~ como complemento. Se puede usar una enumeración para dar nombre a los miembros de un conjunto así. Veamos un pequeño ejemplo tomado de una implementación de ***ostream***:

```
enum ios_base::iostate {
    goodbit=0, eofbit=01, failbit=010, badbit=0100
};
```

La implementación de un flujo puede establecer y comprobar su estado así:

```
state = goodbit;
// ...
if (state& (badbit|failbit) )     // flujo no bueno
```

El paréntesis extra es necesario porque & tiene mayor precedencia que |.

Una función que alcance el final de la entrada informará de ello así:

```
state |= eofbit;
```

El operador |= se usa para sumar al estado. Una asignación sencilla, *state=eofbit*, habría borrado todos los demás bits.

Estos indicadores del estado del flujo son observables desde fuera de la implementación del flujo. Por ejemplo, podríamos ver que los estados de dos flujos difieren de la siguiente manera:

```
int diff = cin.rdstate()^cout.rdstate();      // rdstate() devuelve el estado
```

No es muy habitual computar diferencias de estados de flujo. Para otros tipos similares, computar las diferencias es esencial. Consideremos, por ejemplo, la comparación de un vector de bits que representa el conjunto de interrupciones que se está manejando con otro que representa el conjunto de interrupciones a la espera de ser manejadas.

Observe el lector que este trapicheo de bits se toma de la implementación de flujos E/S y no de la interfaz de usuario. Una manipulación cómoda de los bits puede ser muy importante, pero, por fiabilidad, mantenibilidad, portabilidad, etcétera, debe mantenerse en niveles bajos del sistema. Para unas nociones más generales sobre conjuntos, vea el lector *set* (§17.4.3), *bitset* (§17.5.3) y *vector<bool>* (§16.3.11) de la biblioteca estándar.

El uso de campos (§C.8.1) es una manera abreviada muy cómoda de desplazar y enmascarar para extraer campos de bits de una palabra. Esto se puede hacer también, por supuesto, usando los operadores lógicos a nivel de bits. Por ejemplo, se podrían extraer así los 16 bits centrales de un *long* de 32 bits:

```
unsigned short middle(long a) { return (a>>8)&0xffff; }
```

No hay que confundir los operadores lógicos a nivel de bits con los operadores lógicos &&, || y !. Estos últimos devuelven *true* o *false* y son útiles fundamentalmente para escribir la prueba en una sentencia *if*, *while* o *for* (§6.3.2, §6.3.3). Por ejemplo, *!0* (no cero) es el valor *true*, mientras que *~0* (complemento de cero) es la configuración de bits todo unos, que en representación de complemento a dos es el valor *-1*.

6.2.5 Incremento y decremento

El operador ++ se usa para expresar incremento directamente, en lugar de expresarlo indirectamente usando una combinación de una adición y una asignación. Por definición, *++lvalue* significa *lvalue+=1*, lo que nuevamente significa *lvalue=lvalue+1* siempre que *lvalue* no tenga efectos colaterales. La expresión que denota el objeto a incrementar es evaluada una vez (sólo). El decremento lo expresa de forma semejante el operador --. Los operadores ++ y -- se pueden usar como operadores tanto prefijos como sufijos. El valor de *++x* es el valor nuevo (es decir, incrementado) de *x*. Por ejemplo, *y=++x* es equivalente a *y=(x+=1)*. Sin embargo, el valor de *x++* es el valor antiguo de *x*. Por ejemplo, *y=x++* es equivalente a *y=(t=x, x+=1, t)*, donde *t* es una variable del mismo tipo que *x*.

Como la suma y resta de punteros, ++ y -- operan sobre punteros en términos de elementos del array a los que apunta el puntero; *p++* hace que *p* apunte al elemento siguiente (§5.3.1).

Los operadores de incremento son especialmente útiles para incrementar y decrementar variables en los bucles. Por ejemplo, se puede copiar una cadena terminada por cero de la manera siguiente:

```
void cpy(char* p, const char* q)
{
    while (*p++ = *q++) ;
}
```

Al igual que C, C++ es a la vez amado y odiado por permitir una codificación tan concisa y orientada a la expresión. Porque

```
while (*p++ = *q++) ;
```

resulta algo más que un poco oscuro para los programadores no acostumbrados a C y porque ese estilo de codificación no es infrecuente en C y C++, merece la pena examinarlo más atentamente.

Consideremos primero una forma más tradicional de copiar un array de caracteres:

```
int length = strlen(q);
for (int i = 0; i<=length; i++) p[i] = q[i];
```

Esto es un despilfarro. La longitud de una cadena terminada por cero se encuentra leyendo la cadena a la búsqueda del cero que la termina. Así pues, leemos la cadena dos veces: una para averiguar su longitud y otra para copiarla. Por tanto, podríamos intentar lo siguiente:

```
int i;
for (i = 0; q[i] !=0, i++) p[i] = q[i];
p[i] = 0;    // cero terminal
```

La variable *i* usada para indexar puede ser eliminada porque *p* y *q* son punteros:

```
while (*q != 0) {
    *p = *q;
    p++;    // apuntar al siguiente carácter
    q++;    // apuntar al siguiente carácter
}
*p = 0;    // cero terminal
```

Como la operación de postincremento nos permite usar primero el valor y luego incrementarlo, podemos reescribir el bucle así:

```
while (*q != 0) {
    *p++ = *q++;
}
*p = 0;    // cero terminal
```

El valor de **p++ = *q++* es **q*. Podemos, por tanto, reescribir el ejemplo así:

```
while ( (*p++ = *q++) !=0) { }
```

En este caso, no nos damos cuenta de que **q* es cero hasta que lo copiamos a **p* e incrementamos *p*. En consecuencia, podemos eliminar la asignación final del cero terminal. Por

último, podemos reducir el ejemplo aún más observando que no necesitamos el bloque vacío y que el «!=0» es redundante porque el resultado de un puntero o condición entera se compara siempre con cero en cualquier caso. Así pues, obtenemos la versión que nos proponíamos descubrir:

```
while (*p++ = *q++) ;
```

¿Es esta versión menos legible que las anteriores? No para un programador experimentado de C o C++. ¿Es esta versión más eficiente en cuanto a tiempo o espacio que las anteriores? Salvo la primera versión, que llamaba a **strlen**(), en realidad no. Qué versión sea más eficiente dependerá de la arquitectura de la máquina y del compilador.

La manera más eficiente de copiar una cadena de caracteres terminada por cero para la máquina concreta del lector debe ser la función de copia de cadena estándar:

```
char* strcpy(char*, const char*);    // de <string.h>
```

Para copias más generales se puede usar el algoritmo **copy** estándar (§2.7.2, §18.6.1). Siempre que sea posible debe el lector usar los recursos de la biblioteca estándar con preferencia sobre el trapicheo de punteros y bytes. Las funciones de la biblioteca estándar pueden ser insertadas (§7.1.1) o incluso implementadas usando instrucciones máquina especializadas. En consecuencia, el lector debe pensárselo cuidadosamente antes de creer que un trozo de código artesanal supera a las funciones de la biblioteca.

6.2.6 Memoria libre

Un objeto con nombre tiene una duración que está determinada por su ámbito (§4.9.4). Sin embargo, a menudo es útil crear un objeto que exista con independencia del ámbito en el que ha sido creado. En particular, es habitual crear objetos que puedan ser usados después de devolver desde la función en la que han sido creados. El operador **new** crea tales objetos y el operador **delete** se puede usar para destruirlos. De los objetos asignados por **new** se dice que están «en la memoria libre» (también que son «objetos de montículo» o que están «asignados a la memoria dinámica»).

Consideremos cómo escribiríamos un compilador siguiendo el estilo usado para la calculadora de sobremesa (§6.1). Las funciones de análisis sintáctico crearían un árbol de expresiones a usar por el generador de código:

```
struct Enode {
    Token_value oper;
    Enode* left;
    Enode* right;
    // ...
};
Enode* expr(bool get)
{
    Enode* left = term(get);
    for (;;)
        switch (curr_tok) {
        case PLUS:
        case MINUS:
        {   Enode* n = new Enode;      // crear un Enode en la memoria libre
            n->oper = curr_tok;
            n->left = left;
            n->right = term(true);
            left = n;
            break;
        }
        default:
            return left;               // devolver nodo
        }
}
```

Un generador de código usaría entonces los nodos resultantes y los borraría:

```
void generate(Enode* n)
{
    switch (n->oper) {
    case PLUS:
        // ...
        delete n;        // borrar un Enode de la memoria libre
    }
}
```

Un objeto creado por **new** existe hasta que es destruido explícitamente por **delete**. El espacio que ocupaba puede ser reutilizado entonces por **new**. Una implementación C++ no garantiza la presencia de un «recolector de basura» que busque objetos no referenciados y los ponga a disposición de **new** para ser reutilizados. En consecuencia, partiré de la base de que los objetos creados por **new** son liberados de modo manual usando **delete**. Si está presente un recolector de basura se puede omitir **delete** en la mayoría de los casos (§9.1).

El operador **delete** sólo se puede aplicar a un puntero devuelto por **new** o a cero. La aplicación de **delete** a cero no tiene ningún efecto.

También se pueden definir versiones más especializadas del operador **new** (§15.6).

6.2.6.1 Arrays

También se pueden crear arrays de objetos usando **new**. Por ejemplo:

```
char* save_string(const char* p)
{
    char* s = new char[strlen(p)+1];
    strcpy(s,p);    // copiar de p a s
    return s;
}
int main(int argc, char* argv[])
{
```

```
    if (argc < 2) exit(1);
    char* p = save_string(argv[1]);
    // ...
    delete[] p;
}
```

El operador ***delete*** sin más se usa para borrar objetos individuales; ***delete*** [ ] se usa para borrar arrays.

Para liberar espacio asignado por ***new***, ***delete*** y ***delete*** [ ] deben ser capaces de determinar el tamaño del objeto asignado. Esto implica que un objeto asignado usando la implementación estándar de ***new*** ocupará ligeramente más espacio que un objeto estático, ya que habitualmente se usa una palabra para contener el tamaño del objeto.

Observe el lector que un ***vector*** (§3.7.1, §16.3) es un objeto normal y puede, por tanto, ser asignado y desasignado usando ***new*** y ***delete*** sin más. Por ejemplo:

```
void f(int n)
{
    vector<int>* p = new vector<int>(n);    // objeto individual
    int* q = new int[n];                    // array
    // ...
    delete p;
    delete[] q;
}
```

6.2.6.2 Agotamiento de la memoria

Los operadores de almacenamiento libre ***new***, ***delete***, ***new*** [ ] y ***delete*** [ ] se implementan usando las funciones:

```
void* operator new(size_t);         // espacio para objeto individual
void operator delete(void*);
void* operator new[](size_t);       // espacio para array
void operator delete[](void*);
```

Cuando el operador ***new*** necesita asignar espacio para un objeto, llama a ***operator new***() para asignar un número apropiado de bytes. De forma semejante, cuando el operador ***new*** necesita asignar espacio para un array, llama a ***operator new***[ ]( ).

Las implementaciones estándar de ***operator new***( ) y ***operator new***[ ]( ) no inicializan la memoria devuelta.

¿Qué ocurre cuando ***new*** no puede encontrar memoria que asignar? Por defecto, el asignador lanza una excepción ***bad_alloc***. Por ejemplo:

```
void f()
{
    try {
        for(;;) new char[10000];
    }
    catch(bad_alloc) {
        cerr << "Memoria agotada!\n";
    }
}
```

Por mucha memoria que tengamos disponible, esto invocará eventualmente el manejador ***bad_alloc***.

Podemos especificar lo que debe hacer ***new*** al producirse el agotamiento de memoria. Cuando falla ***new***, llama en primer lugar a una función especificada por una llamada a *set_new_handler*( ) declarada en <***new***>, en caso de que exista. Por ejemplo:

```
void sin_memoria()
{
    cerr << "el operador new ha fallado: no queda memoria\n";
    throw bad_alloc();
}
int main()
{
    set_new_handler(sin_memoria);    // hacer de sin_memoria el new_handler
    for(;;) new char[10000];
    cout << "hecho\n";
}
```

Esto nunca llegará a escribir ***hecho***. Por el contrario, escribirá

el operador new ha fallado: no queda memoria

Véase en §14.4.5 una implementación plausible de un ***operator new***( ) que comprueba si hay un nuevo manejador para llamar y lanza ***bad_alloc*** si no lo hay. Un ***new_handler*** haría algo más inteligente que finalizar simplemente el programa. Si el lector sabe cómo funcionan ***new*** y ***delete*** —por ejemplo, porque haya escrito sus propios ***operator_new***( ) y ***operator_delete***( )— el manejador trataría de encontrar memoria para que ***new*** volviera. Dicho de otro modo, el usuario proporcionaría un recolector de basura, con lo que haría que el uso de ***delete*** fuera opcional. Sin embargo, no se trata en absoluto de una tarea para principiantes. Para prácticamente cualquiera que necesite un recolector automático de basura lo adecuado es que adquiera uno que ya haya sido escrito y probado (§C9.1).

Al proporcionar un ***new_handler*** nos ocupamos de la comprobación de agotamiento de memoria para todos los usos ordinarios de ***new*** en el programa. Existen dos formas alternativas de controlar la asignación de memoria. Podemos proporcionar funciones de asignación y liberación no estándar (§15.6) para los usos estándar de ***new*** o confiar en la información adicional sobre asignación proporcionada por el usuario (§10.4.11).

6.2.7 Conversión explícita de tipos

A veces tenemos que tratar con «memoria en bruto»; es decir, con memoria que contiene o va a contener objetos de un tipo no conocido para el compilador. Por ejemplo, un asignador de memoria puede devolver un ***void**** que apunte a la memoria recién asignada o podemos querer establecer que un valor entero determinado sea tratado como dirección de un dispositivo E/S:

```
void* malloc(size_t);
void f()
{
    int* p = static_cast<int*>(malloc(100));          // asignación nueva
                                                      // usada como ints
    IO_device* dl = reinterpret_cast<IO_device*>(0Xff00); // dispositivo
                                                      // en 0Xff00
    //...
}
```

Un compilador no conoce el tipo del objeto al que apunta el ***void****. Tampoco puede saber si el entero ***0Xff00*** es una dirección válida. En consecuencia, la corrección de las conversiones está por completo en manos del programador. La conversión explícita de tipos, denominada a menudo aplicación de *moldes (casting),* es en ocasiones esencial. Sin embargo, tradicionalmente se usa de forma abusiva y es una importante fuente de errores.

El operador ***static_cast*** convierte entre tipos relacionados, como un tipo de puntero a otro, una enumeración a un tipo entero o un tipo en coma flotante a un tipo entero. El operador ***reinterpret_cast*** maneja las conversiones entre tipos no relacionados tales como un entero a puntero. Esta distinción permite al compilador aplicar una comprobación de tipos mínima para ***static_cast*** y hace más fácil que el programador encuentre las conversiones más peligrosas representadas como ***reinterpret_cast***. Algunos ***static_cast*** son portables, pero pocos ***reinterpret_cast*** lo son. Apenas hay garantías para ***reinterpret_cast***, pero en general produce un valor de un tipo nuevo que tiene la misma configuración de bits que su argumento. Si el destino tiene como mínimo tantos bits como el valor original, podemos aplicar ***reinterpret_cast*** al resultado para devolverlo a su tipo original y usarlo. El resultado de un ***reinterpret_cast*** está garantizado para que sólo sea utilizable si su tipo de resultado es el tipo exacto usado para definir el valor de que se trate. Observe el lector que ***reinterpret_cast*** es la modalidad de conversión que se debe usar para punteros a funciones (§7.7).

Si al lector le tienta usar una conversión explícita de tipos, tómese tiempo para considerar si es *realmente* necesaria. En C++ la conversión explícita de tipos es innecesaria en la mayor parte de los casos en los que es necesaria en C (§1.6) y también en muchos casos en los que era necesaria en versiones anteriores de C++ (§1.6.2, §B.2.3). En muchos programas la conversión explícita de tipos puede ser evitada por completo; en otros, su uso debe estar localizado en unas pocas rutinas. En este libro sólo se usa la conversión explícita de tipos en situaciones realistas en §6.2.7, §7.7, §13.5, §15.4 y §25.4.1.

Se proporciona igualmente una forma de conversión comprobada en tiempo de ejecución, ***dynamic_cast*** (§15.4.1) y un molde para eliminar cualificadores ***const***, ***const_cast*** (§15.4.2.1).

C++ ha heredado de C la notación ***(T)e***, que realiza cualquier conversión que pueda ser expresada como una combinación de ***static_cast***, ***reinterpret_cast*** y ***const_cast*** para hacer un valor de tipo ***T*** a partir de la expresión ***e*** (§B.2.3). Este molde de estilo C es mucho más peligroso que los operadores de conversión con nombre porque es más difícil de descubrir la notación en un programa grande y la modalidad de conversión que pretende el programador no está explicitada. Esto es, ***(T)e*** podría estar haciendo una conversión portable entre tipos relacionados, una conversión no portable entre tipos no relacionados o eliminando el modificador ***const*** de un tipo de puntero. Si no se conocen los tipos exactos de ***T*** y ***e*** es imposible determinarlo.

6.2.8 Constructores

La construcción de un valor de tipo ***T*** a partir de un valor ***e*** puede expresarse con la notación funcional ***T(e)***. Por ejemplo:

```
void f(double d)
{
    int i = int(d);                 // truncar d
    complex z = complex(d);         // hacer un complejo a partir de d
    //...
}
```

La construcción ***T(e)*** se denomina a veces *molde de estilo función.* Para un tipo ***T*** predefinido, ***T(e)*** es equivalente a ***static_cast<T>(e)***. Desafortunadamente, esto implica que el uso de ***T(e)*** no es siempre seguro. Para tipos aritméticos, los valores pueden ser truncados e incluso la conversión explícita de un tipo entero más largo a otro más corto (como de ***long*** a ***char***) puede dar lugar a comportamiento no definido. Procuro usar esta notación sólo cuando la construcción de un valor está bien definida; es decir, para conversiones aritméticas con estrechamiento (en inglés, *narrowing*) (§C.6), para conversiones de enteros a enumeraciones (§4.8) y la construcción de objetos de tipos definidos por el usuario (§2.5.2, §10.2.3).

Las conversiones de puntero no se pueden expresar directamente usando la notación ***T(e)***. Por ejemplo, ***char*(2)*** es un error de sintaxis. Por desgracia, la protección que proporciona la notación de constructor contra estas conversiones peligrosas puede sortearse usando nombres ***typedef*** (§4.9.7) para tipos de puntero.

La notación de constructor ***T()*** se usa para expresar el valor por defecto del tipo ***T***. Por ejemplo:

```
void f(double d)
{
    int j = int();                  // valor int por defecto
    complex z = complex x();        // valor complejo por defecto
    //...
}
```

El valor de un uso explícito del constructor para un tipo predefinido es ***0*** convertido a ese tipo (§4.9.5). Así pues, ***int()*** es otra forma de escribir ***0***. Para un tipo ***T*** definido por el usuario, ***T()*** es definido por el constructor por defecto (§10.4.2), en caso de que exista.

El uso de la notación de constructor para los tipos predefinidos es especialmente importante cuando se escriben plantillas. El programador no sabe entonces si un parámetro de la plantilla se va a referir a un tipo predefinido o a un tipo definido por el usuario (§16.3.4, §17.4.1.2).

6.3 Resumen de sentencias

Veamos un resumen y algunos ejemplos de sentencias C++:

Sintaxis de sentencias

sentencia:
declaración
{ *lista-de-sentencias*$_{opt}$ }
try { *lista-de-sentencias*$_{opt}$ } *lista-de-manejadores*
expresión$_{opt}$;

if (*condición*) *sentencia*
if (*condición*) *sentencia* ***else*** *sentencia*
switch (*condición*) *sentencia*

switch (*condición*) *sentencia*
do *sentencia* ***while*** (*expresión*) ;
for (*sentencia-inic-for condición*$_{opt}$; *expresión*$_{opt}$) *sentencia*

case *expresión-constante* : *sentencia*
default : *sentencia*
break ;
continue ;

return *expresión*$_{opt}$;

goto *identificador* ;
identificador : *sentencia*

lista-de-sentencias:
sentencia lista-de-sentencias$_{opt}$

condición:
expresión
especificador-de-tipo declarador = expresión

lista-de-manejadores:
catch (*declaración-de-excepciones*) { *lista-de-sentencias*$_{opt}$ }
lista-de-manejadores lista-de-manejadores$_{opt}$

Observe el lector que una declaración es una sentencia y que no hay sentencia de asignación ni sentencia de llamada de procedimiento; las asignaciones y las llamadas de función son expresiones. Las sentencias para manejo de excepciones, los *bloques-try*, se describen en §8.3.1.

6.3.1 Declaraciones y sentencias

Una declaración es una sentencia. A menos que una variable sea declarada ***static***, su inicializador se ejecuta siempre que el flujo de control pasa por la declaración (véase también §10.4.8). La razón para permitir declaraciones en todos los lugares en los que se puede usar una sentencia (y en algunos otros lugares; §6.3.2.1, §6.3.3.1) es posibilitar que el programador minimice los errores causados por las variables no inicializadas y permitir una mejor localidad del código. Rara vez hay razón para introducir una variable antes de que haya un valor para que ella lo contenga. Por ejemplo:

```
void f(vector<string>& v, int i, const char* p)
{
  if (p==0) return;
  if (i<0 || i<v.size() ) error( "indice erroneo" );
  string s = v[i];
  if (s == p) {
    // ...
  }
  // ...
}
```

La capacidad para colocar declaraciones detrás de código ejecutable es esencial para muchas constantes y para estilos de programación con asignación sencilla en los que un valor de un objeto no cambia después de la inicialización. Para los tipos definidos por el usuario, posponer la definición de una variable hasta que hay disponible un inicializador adecuado puede dar lugar también a un mejor rendimiento. Por ejemplo:

```
String s; /* ... */ s = "Lo mejor es enemigo de lo bueno.";
```

puede fácilmente ser mucho más lento que

```
String s = "Voltaire";
```

La razón más frecuente para declarar una variable sin inicializador es que requiera una sentencia para inicializarla. Son ejemplos de ello las variables y arrays de entrada.

6.3.2 Sentencias de selección

Un valor puede ser probado por una sentencia ***if*** o por una sentencia ***switch***:

if (*condición*) *sentencia*
if (*condición*) *sentencia* ***else*** *sentencia*
switch (*condición*) *sentencia*

Los operadores de comparación

```
==   !=   <   <=   >   >=
```

devuelven el booleano ***true*** si la comparación es verdadera y ***false*** en caso contrario.

En una sentencia ***if*** la primera (o única) sentencia es ejecutada si la expresión es distinta de cero y la segunda sentencia (si está especificada) es ejecutada en caso contrario. Esto implica que cualquier expresión entera o de puntero puede usarse como condición. Por ejemplo, si ***x*** es un entero, entonces

```
if (x) //..
```

significa

```
if (x != 0) //...
```

Para un puntero *p*,

```
if (p) //...
```

es una sentencia directa de la prueba «*p* apunta a un objeto válido», mientras que

```
if (p !=0) //...
```

formula la misma pregunta indirectamente comparando con un valor que no apunta a ningún objeto. Observe el lector que la representación del puntero **0** no es todo ceros en todas las máquinas (§5.1.1). Todos los compiladores que he comprobado generan el mismo código para las dos formas de la prueba.

Los operadores lógicos

```
&&  ||  !
```

se usan habitualmente en las condiciones. Los operadores && y || no evaluarán su segundo argumento a menos que sea necesario hacerlo. Por ejemplo,

```
if (p && 1<p->count) //...
```

prueba primero que *p* es distinto de cero. Prueba *1<p->count* sólo si *p* es distinto de cero.

Algunas *sentencias-if* pueden ser sustituidas convenientemente por *expresiones-condicionales*. Por ejemplo,

```
if (a <= b)
  max = b;
else
  max = a;
```

está mejor expresado del siguiente modo:

```
max = (a<=b) ? b : a;
```

No es necesario encerrar entre paréntesis la condición, pero encuentro que el código es más fácil de leer cuando se hace así.

Alternativamente, una *sentencia-switch* puede escribirse como un conjunto de *sentencias-if*. Por ejemplo:

```
switch (val) {
case 1:
  f();
  break;
case 2:
  g();
  break;
default:
  h();
  break;
}
```

se podría expresar alternativamente como

```
if (val == 1)
  f();
else if (val == 2)
  g();
else
  h();
```

El significado es el mismo, pero la primera versión (*switch*) es preferible porque la naturaleza de la operación (probar un valor con un conjunto de constantes) es explícita. Esto hace la sentencia *switch* más fácil de leer en ejemplos no triviales. Puede dar lugar asimismo a la generación de código mejor.

El lector debe tener cuidado de que los casos de las sentencias *switch* terminen de alguna manera si no quiere que sigan ejecutando el caso siguiente. Consideremos:

```
switch (val) {          // cuidado
case 1:
  cout << "caso 1\n";
case 2:
  cout << "caso 2\n";
default:
  cout << "por defecto: caso no encontrado\n";
}
```

Invocado con *val==1*, lo anterior imprime

```
caso 1
caso 2
por defecto: caso no encontrado
```

para gran sorpresa de los no iniciados. Es buena idea comentar los (raros) casos en los que un fracaso es intencionado, de modo que un fracaso no comentado pueda ser tomado por un error. Un *break* es la forma más frecuente de terminar un caso, pero un *return* es a menudo útil (§6.1.1).

6.3.2.1 Declaraciones en las condiciones

Para evitar el uso indebido accidental de una variable, habitualmente es buena idea introducir la variable en el ámbito más pequeño posible. En particular, suele ser mejor retrasar la definición de una variable local hasta que se le da un valor inicial. De esa forma no pueden surgir problemas por usar la variable antes de que sea asignado su valor inicial.

Una de las aplicaciones más elegantes de estos dos principios es declarar una variable en una condición. Consideremos:

```
if (double d = prim(true)) {
  left /= d;
  break;
}
```

Aquí *d* es declarada e inicializada y el valor de *d* tras la inicialización es comprobado como valor de la condición. El ámbito de *d* se extiende desde el punto en el que se declara hasta

el final de la sentencia controlada por la condición. Por ejemplo, si hubiera habido una rama ***else*** para la sentencia ***if***, ***d*** habría estado en el ámbito de las dos ramas.

La alternativa obvia y tradicional es declarar ***d*** antes de la condición. Sin embargo, esto abre el ámbito (literalmente) para el uso de ***d*** antes de su inicialización o después de su vida útil prevista:

```
double d;
// ...
d2 = d;        // ¡vaya!
// ...
if (d = prim(true)) {
    left /= d;
    break;
}
// ...
d = 2.0;       // dos usos de d sin relación entre sí
```

Además de los beneficios lógicos de declarar las variables en las condiciones, al hacerlo se consigue también código fuente más compacto.

Una declaración en una condición debe declarar e inicializar una sola variable o ***const***.

6.3.3 Sentencias de iteración

Un bucle puede expresarse como una sentencia ***for***, ***while*** o ***do***:

```
while ( condición ) sentencia
do sentencia while ( expresión ) ;
for ( sentencia-inicial-for condición_opt ; expresión_opt ) sentencia
```

Cada una de estas sentencias ejecuta una sentencia (denominada secuencia *controlada* o *cuerpo del bucle)* de manera repetida hasta que la condición se hace falsa o el programador rompe el bucle de alguna otra forma.

La *sentencia-for* está prevista para expresar bucles bastante regulares. La variable de bucle, la condición de terminación y la expresión que actualiza la variable de bucle pueden ser presentadas «por adelantado» en una sola línea. Esto puede aumentar considerablemente la legibilidad y, con ello, reducir la frecuencia de errores. Si no es necesaria inicialización, se puede omitir la sentencia de inicialización. Si se omite la *condición* la *sentencia-for* iterará indefinidamente a menos que el uso salga explícitamente de ella por un ***break***, ***return***, ***goto***, ***throw*** o alguna forma menos obvia, como una llamada a *exit* ( ) (§9.4.1.1). Si se omite la *expresión* debemos actualizar algura forma de variable de bucle en el cuerpo del bucle. Si el bucle no es de la variedad sencilla «introducir una variable de bucle, probar la condición, actualizar la variable de bucle», a menudo se expresa mejor como *sentencia-while*. Una *sentencia-for* es también útil para expresar un bucle sin una condición explícita de terminación:

```
for ( ; ; ) {    // "para siempre"
    // ...
}
```

Una *sentencia-while* ejecuta sencillamente su sentencia controlada hasta que su condición se hace ***false***. Tiendo a preferir las *sentencias-while* a las *sentencias-for* cuando no hay una variable de bucle obvia o cuando la actualización de una variable de bucle llega naturalmente en medio del cuerpo del bucle. Un bucle de entrada es un ejemplo de bucle en el que no hay variable de bucle obvia:

```
while (cin>>ch) // ...
```

Según mi experiencia, la *sentencia-do* es una fuente de errores y confusión. La razón es que su cuerpo se ejecuta siempre una vez antes de que sea evaluada la condición. Sin embargo, para que el cuerpo funcione correctamente, debe haber algo muy parecido a la condición desde la primera vez. Más a menudo de lo que habría imaginado que esa condición no retenía como estaba previsto ni cuando se escribía y probaba por primera vez ni más adelante, después de que se hubiera modificado el código que la precedía. También prefiero la condición «por adelantado, donde pueda verla». En consecuencia, tiendo a evitar las *sentencias-do*.

6.3.3.1 Declaraciones en sentencias *for*

Una variable puede ser declarada en la parte del inicializador de una *sentencia-for*. Si ese inicializador es una declaración, la variable (o variables) que introduce está en ámbito hasta el final de la *sentencia-for*. Por ejemplo:

```
void f(int v[], int max)
{
    for (int i = 0; i<max; i++) v[i] = i*i;
}
```

Si es necesario conocer el valor final de un índice después de salir de un bucle ***for***, la variable índice debe ser declarada fuera del bucle ***for*** (§6.3.4).

6.3.4 *goto*

C++ tiene el denigrado ***goto***:

```
goto identificador ;
identificador : sentencia
```

goto tiene pocos usos en programación general de alto nivel, pero puede ser muy útil cuando es generado código C++ por un programa en lugar de ser escrito directamente por una persona; por ejemplo, se pueden usar ***goto*** en un analizador sintáctico generado a partir de una gramática por un programa generador de analizadores sintácticos. También puede ser importante ***goto*** en los raros casos en los que es esencial una eficiencia óptima, por ejemplo, en el bucle interno de alguna aplicación en tiempo real.

Uno de los pocos usos sensatos de goto en código ordinario es para escaparse de un bucle anidado o *sentencia-switch* (con un ***break*** sólo se escapa del bucle inclusivo más interno o *sentencia-switch*). Por ejemplo:

```
void f()
```

```
    int i;
    int j;
    for (i = 0; i<n; i++)
        for (j = 0; j<m; j++) if (nm[i][j] == a) goto encontrado;
    // no encontrado
    // ...
encontrado:
    // nm[i][j] == a
}
```

Hay también una sentencia ***continue*** que va efectivamente hasta el final de una sentencia de bucle, como se explica en §6.1.5.

6.4 Comentarios y sangrado

Un uso sensato de los comentarios y una utilización coherente de las sangrías puede hacer mucho más agradable la tarea de leer y entender un programa. Se usan varios estilos coherentes de sangrado. No veo razones fundamentales para preferir uno a otro (aunque, como la mayoría de los programadores, tengo mis preferencias, que se reflejan en este libro). Lo mismo se puede decir de los estilos de comentarios.

Los comentarios se pueden usar mal de formas que afectan gravemente a la legibilidad de un programa. El compilador no entiende el contenido de los comentarios, por lo que no hay forma de asegurarse de que un comentario

[1] tiene sentido,
[2] describe el programa y
[3] está actualizado.

La mayoría de los programas contienen comentarios que son incomprensibles, ambiguos y directamente erróneos. Un comentario malo es peor que un comentario inexistente. Esta observación se dirige a comentarios como los siguientes:

```
// la variable "v" debe ser inicializada
// la variable "v" debe ser usada sólo por la función "f()"
// llamar a la función 'init()" antes de llamar a cualquier otra función de este archivo
// llame a la función 'cleanup()" al final de su programa
// no use la función "weird()"
// la función "f()" toma dos argumentos
```

Comentarios así se pueden hacer innecesarios con un uso adecuado de C++. Por ejemplo, se podrían utilizar las reglas de enlazado (§9.2) y las de visibilidad, inicialización y limpieza de las clases (§10.4.1) para hacer que los ejemplos anteriores fueran redundantes.

Una vez que se ha establecido algo claramente en el lenguaje, no debe mencionarse por segunda vez en un comentario. Por ejemplo:

```
a = b+c;        // a se convierte en b+c
count++;        // incrementar el contador
```

Estos comentarios son peor que sencillamente redundantes. Aumentan la cantidad de texto que tiene que examinar el lector, suelen oscurecer la estructura del programa y pueden ser erróneos. Observe el lector, sin embargo, que se usan ampliamente comentarios semejantes con fines didácticos en manuales de lenguaje de programación como el presente. Éste es uno de los aspectos en que un programa de un libro de texto difiere de un programa real.

Mis preferencias son:

[1] Un comentario para cada archivo fuente que indique lo que las declaraciones que contiene tienen en común, referencias a manuales, consejos generales de mantenimiento, etcétera.
[2] Un comentario para cada clase, plantilla y espacio de nombres.
[3] Un comentario para cada función no trivial que indique su finalidad, el algoritmo utilizadado (a menos que sea evidente) y tal vez algo sobre las hipótesis que hace en cuanto a su entorno.
[4] Un comentario para cada variable y constante global y de espacio de nombres.
[5] Unos pocos comentarios donde el código no sea obvio y/o portable.
[6] Muy poco más.

Por ejemplo:

```
// tbl.c: Implementación de la tabla de símbolos.
/*
    Eliminación gaussiana con pivotamiento parcial.
    Ver Ralston: "A first course..." pág. 411.
*/
// swap() supone la disposición de pila de un SGI R6000.
/*********************************************

Copyright(c) 1997 AT&T, Inc.
Reservados todos los derechos

*********************************************/
```

Un conjunto de comentarios bien elegidos y bien redactados es parte esencial de todo buen programa. Escribir buenos comentarios puede ser tan difícil como escribir el propio programa. Es un arte que merece la pena cultivar.

Observe también el lector que si se usan exclusivamente comentarios // en una función, cualquier parte de esa función se puede comentar fuera utilizando comentarios de estilo /* */, y viceversa.

6.5 Consejos

[1] Prefiera la biblioteca estándar a otras bibliotecas y al «código artesanal»; §6.1.8.
[2] Evite las expresiones complicadas; §6.2.3.
[3] En caso de duda sobre la precedencia de los operadores, use paréntesis; §6.2.3.
[4] Evite la conversión explícita de tipos (moldes); §6.2.7.
[6] Use la notación ***T*(*e*)** exclusivamente para construcciones bien definidas; §6.2.8.
[7] Evite las expresiones con un orden de evaluación no definido; § 6.2.2.
[8] Evite ***goto***; §6.3.4.

[9] Evite las *sentencias-do*; §6.3.3.
[10] No declare una variable hasta que tenga un valor con el que inicializarla; § 6.3.1, §6.3.2.1, §6.3.3.1.
[11] Haga comentarios concisos; §6.4.
[12] Mantenga un estilo coherente de sangrado; §6.4.
[13] Prefiera definir un miembro ***operator new*** ( ) (§15.6) a sustituir el ***operator new*** ( ) global; §6.2.6.2.
[14] Cuando lea entrada, tenga siempre en cuenta la posibilidad de que esté mal formada; §6.1.3.

6.6 Ejercicios

1. (*1) Reescriba la sentencia ***for*** siguiente como una sentencia ***while*** equivalente

```
for (i=0; i<long_max; i++) if (linea_de_entrada[i] == '?') recuento_de_busquedas++;
```

   Reescríbala para que use un puntero como variable controlada, es decir, para que la prueba sea de la forma `*p=='?'`.
2. (*1) Ponga todos los paréntesis en las siguientes expresiones:

```
a = b + c * d << 2 & 8
a & 077 != 3
a == b || a == c && c < 5
c = x != 0
0 <= i < 7
f(1,2)+3
a = - 1 ++ b -- -5
a = b == c ++
a = b = c = 0
a[4][2] *= *b ? c : * d * s
a-b, c=d
```

3. (*2) Lea una secuencia de pares (nombre,valor) separados por espacios en blanco en los que cada nombre sea una sola palabra separada por espacio en blanco y cada valor sea un entero o un valor en coma flotante. Calcule e imprima la suma y la media para cada nombre y la suma y la media para todos los nombres. Pista: §6.1.8.
4. (*1) Escriba una tabla de valores para operaciones lógicas a nivel de bits (§6.2.4) para todas las combinaciones posibles de operandos ***0*** y ***1***.
5. (*1,5) Encuentre 5 construcciones de C++ para las cuales el significado sea no definido (§C.2). (*1,5) Encuentre 5 construcciones de C++ para las cuales el significado esté definido por la implementación (§C.2).
6. (*1) Encuentre 10 ejemplos de código C++ no portable.
7. (*2) Escriba 5 expresiones para las cuales el orden de evaluación sea no definido. Ejecútelas para ver qué hace con ellas una implementación o —preferiblemente— varias.
8. (*1,5) ¿Qué ocurre si divide por cero en su sistema? ¿Qué ocurre si se produce desbordamiento y subdesbordamiento?
9. (*1) Ponga todos los paréntesis en las expresiones siguientes:

```
*p++
*--p
++a++
(int*)p->m
*p.m
*a[i]
```

10. (*2) Escriba las siguientes funciones: ***strlen*** ( ), que devuelve la longitud de una cadena de estilo C; ***strcpy*** ( ), que copia una cadena en otra; y ***strcmp*** ( ), que compara dos cadenas. Considere cuáles deben ser los tipos de argumento y los tipos devueltos. Compare luego sus funciones con las versiones de la biblioteca estándar declaradas en <***ccstring***> (<***string.h***>) y especificadas en §20.4.1.
11. (*1) Vea cómo reacciona su compilador a los errores siguientes:

```
void f(int a, int b)
{
  if (a = 3) //...
  if (a&077 == 0)   //...
  a := b+i;
}
```

   Invente más errores sencillos y vea cómo reacciona el compilador.
12. (*2) Modifique el programa de §6.6[3] para que calcule también la mediana.
13. (*2) Escriba una función ***cat*** ( ) que tome dos argumentos de cadena de estilo C y devuelva una cadena que sea la concatenación de los argumentos. Use ***new*** para encontrar memoria para el resultado.
14. (*2) Escriba una función ***rev*** ( ) que tome como argumento una cadena e invierta sus caracteres. Es decir, después de ***rev*** (***p***) el último carácter de ***p*** será el primero, etcétera.
15. (*1,5) ¿Qué hace el ejemplo siguiente?

```
void send(int* to, int* from, int count)
// Comentario útil borrado deliberadamente.
{
  int n = (count+7) / 8;
  switch (count%8) {
  case 0: do { *to++ = *from++;
  case 7:      *to++ = *from++;
  case 6:      *to++ = *from++;
  case 5:      *to++ = *from++;
  case 4:      *to++ = *from++;
  case 3:      *to++ = *from++;
  case 2:      *to++ = *from++;
  case 1:      *to++ = *from++;
    } while (--n>0);
  }
}
```

   ¿Por qué escribiría alguien algo así?

16. (*2) Escriba una función ***atoi*** (***const char****) que tome una cadena de dígitos y devuelva el ***int*** correspondiente. Por ejemplo, ***atoi*** ("***123***") es ***123***. Modifique ***atoi*** () para que maneje la notación octal y hexadecimal de C++ además de números decimales sin más. Modifique ***atoi*** () para que maneje la notación constante de caracteres de C++.
17. (*2) Escriba una función ***itoa*** (***int i***, ***char b***[]) que cree una representación de cadena de ***i*** en ***b*** y devuelva ***b***.
18. (*2) Teclee el ejemplo de la calculadora y póngalo a funcionar. No «ahorre tiempo» usando un texto ya introducido. Aprenderá más si encuentra y corrige «pequeños errores tontos».
19. (*2) Modifique la calculadora para que informe del número de línea de los errores.
20. (*3) Permita a un usuario definir funciones en la calculadora. Pista: Defina una función como una secuencia de operaciones exactamente como las habría tecleado un usuario. Una secuencia así se puede almacenar como cadena de caracteres o como lista de componentes léxicos. A continuación lea y ejecute esas operaciones cuando sea llamada la función. Si desea una función definida por el usuario que tome argumentos tendrá que inventar una notación para ella.
21. (*1,5) Convierta la calculadora de sobremesa de modo que use una estructura ***symbol*** en lugar de usar las variables estáticas ***number_value*** y ***string_value***.
22. (*2,5) Escriba un programa que quite los comentarios de un programa C++. Es decir, que lea de ***cin***, elimine tanto los comentarios // como los comentarios /* */ y escriba el resultado en ***cout***. No se preocupe de hacer que la salida tenga un aspecto agradable (eso sería objeto de otro ejercicio, mucho más difícil). No se preocupe por la incorrección de los programas. Tenga en cuenta //, /* y */ en comentarios, cadenas y constantes de caracteres.
23. (*2) Examine algunos programas para hacerse una idea de la variedad de sangrías, nombres y estilos de comentarios que se usan.

7

Funciones

La iteración es humana;
la recursión, divina
L. Peter Deutsch

Declaraciones y definiciones de función — paso de argumentos — sobrecarga de funciones — resolución de la ambigüedad — argumentos por defecto — ***stdargs*** — punteros a funciones — macros — consejos — ejercicios.

7.1 Declaraciones de función

La forma típica de conseguir que se haga algo en un programa C++ es llamar a una función que lo haga. Definir una función es la forma de especificar cómo se debe hacer una operación. No se puede llamar a una función a menos que haya sido declarada previamente.

Una declaración de función da el nombre de la función, el tipo del valor devuelto (si lo hay) por ella, y el número y tipo de los argumentos que deben ser proporcionados en una llamada a la función. Por ejemplo:

```
Elem* next_elem();
char* strcpy(char* to, const char* from);
void exit(int);
```

La semántica del paso de argumentos es idéntica a la semántica de inicialización. Los tipos de los argumentos son comprobacos y cuando es necesario se produce la conversión implícita de tipos de los argumentos. Por ejemplo:

```
double sqrt(double);

double sr2 = sqrt(2);       // llamar a sqrt() con el argumento double(2)
double sq3 = sqrt("tres");  // error: sqrt() requiere un argumento de tipo double
```

No hay que subestimar el valor de esta comprobación y conversión de tipo

Una declaración de función puede contener nombres de argumentos. Esto puede ayudar a un lector de un programa, pero el compilador sencillamente no tiene en cuenta esos

nombres. Como se mencionó en §4.7, ***void*** como tipo devuelto significa que la función no devuelve ningún valor.

7.1.1 Definiciones de función

Toda función que es llamada en un programa debe estar definida en algún lugar (sólo una vez). Una definición de función es una declaración de función en la cual se presenta el cuerpo de la función. Por ejemplo:

```
extern void swap(int*, int*);    // declaración
void swap (int* p, int* q)       // definición
{
  int t = *p;
  *p = *q;
  *q = t;
}
```

El tipo de la definición y todas las declaraciones para una función deben especificar el mismo tipo. Los nombres de los argumentos, sin embargo, no forman parte del tipo y no es necesario que sean idénticos.

No es infrecuente tener definiciones de función con argumentos no usados:

```
void search(table* t, const char* key, const char*)
{
  // no hay uso del tercer argumento
}
```

Como se muestra, se puede indicar que un argumento no se usa no dándole nombre. Habitualmente surgen argumentos sin nombre a partir de la simplificación de código y de la planificación previa de extensiones. En ambos casos, dejar el argumento en su sitio, aunque sin usar, garantiza que los llamadores no sean afectados por el cambio.

Una función puede ser definida para que sea ***inline***. Por ejemplo:

```
inline int fac(int n) { return (n<2) ? 1 : n*fac(n-1); }
```

El especificador ***inline*** es una pista que indica al compilador que debe intentar generar código para una llamada de ***fac*** ( ) insertada, en lugar de establecer primero el código para la función y llamar luego por medio del mecanismo habitual de llamada. Un compilador más inteligente puede generar la constante ***720*** para una llamada ***fac*** (***6***). La posibilidad de que haya funciones insertadas mutuamente recursivas, funciones insertadas que hacen recursión o no dependiendo de la entrada, etcétera, hace imposible garantizar que todas las llamadas a una función ***inline*** sean realmente insertadas. El grado de inteligencia de un compilador no puede legislarse, por lo que un compilador podría generar ***720***, otro ***6*fac*** (***5***) y otro una llamada no insertada ***fac*** (***6***).

Para hacer la inserción posible en ausencia de una compilación y unas capacidades de enlace extraordinariamente inteligentes, la definición —y no sólo la declaración— de una función insertada debe estar dentro del ámbito (§9.2). Un especificador ***inline*** no afecta al significado de una llamada a la función.

7.1.2 Variables estáticas

Una variable local se inicializa cuando el flujo de ejecución alcanza su definición. Por defecto, esto ocurre en todas las llamadas a la función y cada invocación de la función tiene su propia copia de la variable. Si una variable local es declarada ***static***, se usará un único objeto asignado estáticamente para representar esa variable en todas las llamadas a la función. Será inicializada sólo la primera vez que el flujo de ejecución llegue a su definición. Por ejemplo:

```
void f(int n)
{
  while (n--) {
    static int n = 0;          // inicializada una vez
    int x = 0;                 // inicializada n veces
    cout << "n == " << n++ << ", x== " << x++ << '\n';
  }
}
void main()
{
  f(3);
}
```

Esto imprime:

```
n == 0, x == 0
n == 1, x == 0
n == 2, x == 0
```

Una variable estática proporciona una función con «memoria» sin necesidad de introducir una variable global, a la que podrían tener acceso y podrían corromper otras funciones (véase también §10.2.4).

7.2 Paso de argumentos

Cuando se llama a una función se aparta memoria para sus argumentos formales y cada uno de ellos es inicializado por su argumento real correspondiente. La semántica del paso de argumentos es idéntica a la semántica de inicialización. En especial, se comprueba el tipo de un argumento real con el tipo del argumento formal correspondiente y se realizan todas las conversiones de tipo estándar y definidas por el usuario. Hay reglas especiales para el paso de arrays (§7.2.1), un recurso para pasar argumentos no comprobados (§7.6) y un recurso para especificar argumentos por defecto (§7.5). Consideremos:

```
void f(int val, int& ref)
{
  val++;
  ref++;
}
```

Cuando se llama a *f*( ), *val*++ incrementa una copia local del primer argumento real, mientras que *ref*++ incrementa el segundo argumento real. Por ejemplo,

```
void g()
{
  int i = 1;
  int j = 1;
  f(i,j);
}
```

incrementará *j* pero no *i*. El primer argumento, *i*, es pasado *por valor;* el segundo argumento, *j*, es pasado *por referencia*. Como se mencionó en §5.5, las funciones que modifican argumentos llamados por referencia pueden hacer que los programas sean difíciles de leer y deben ser evitadas casi siempre (pero véase §21.2.1). Sin embargo, puede ser notablemente más eficiente pasar un objeto grande por referencia que pasarlo por valor. En ese caso, el argumento sería declarado *const* para indicar que se usa la referencia sólo por razones de eficiencia y no para posibilitar que la función llamada cambie el valor del objeto:

```
void f(const Large& arg)
{
  // el valor de "arg" no se puede modificar sin uso explícito de la conversión de tipo
}
```

La ausencia de *const* en la declaración de un argumento de referencia se toma como una manifestación de intento de modificar la variable:

```
void g(Large& arg);   // dar por supuesto que g() modifica arg
```

De forma parecida, declarar *const* un argumento de puntero dice a los lectores del código que el valor de un objeto apuntado por ese argumento no es modificado por la función. Por ejemplo:

```
int strlen(const char*);          // número de caracteres de una cadena de estilo C
char* strcpy(char* to, const char* from);   // copiar una cadena de estilo C
int strcmp(const char*, const char*);       // comparar cadenas de estilo C
```

La importancia de usar argumentos *const* aumenta con el tamaño del programa.

Observe el lector que la semántica del paso de argumentos es diferente de la semántica de asignación. Esto es importante para argumentos *const*, argumentos de referencia y argumentos de algunos tipos definidos por el usuario (§10.4.4.1).

Un literal, una constante y un argumento que requiera conversión pueden pasarse como argumento *const*&, pero no como argumento no *const*. Permitir las conversiones para un argumento *const T*& asegura que a un argumento así se le pueda dar exactamente el mismo conjunto de valores que a un argumento *T* pasando el valor en un temporal, si es necesario. Por ejemplo:

```
float fsqrt(const float&);      // raiz cuadrada de estilo Fortran que toma
                                // un argumento de referencia

void g(double d)
{
  float r = fsqrt(2.0f);        // pasar ref a temp que contiene 2.0f
  r = fsqrt(r);                 // pasar referencia a r
  r = fsqrt(d);                 // pasar ref a temp que contiene float(d)
}
```

Al no permitir las conversiones para argumentos de referencia no *const* (§5.5) se evita la posibilidad de que se produzcan errores tontos derivados de la introducción de temporales. Por ejemplo:

```
void update(float& i);

void g(double d, float r)
{
  update(2.0f);     // error: argumento const
  update(r);        // pasar referencia a r
  update(d);        // error: se requiere conversión de tipo
}
```

Si se hubieran permitido estas llamadas, *update*( ) habría actualizado en silencio los temporales que inmediatamente se habrían borrado. Habitualmente esto habría sido una sorpresa desagradable para el programador.

7.2.1 Argumentos de array

Si se usa un array como argumento de una función se pasa un puntero a su elemento inicial. Por ejemplo:

```
int strlen(const char*);

void f()
{
  char v[] = "un array";
  int i = strlen(v);
  int j = strlen("Nicholas");
};
```

Es decir, un argumento de tipo *T*[ ] se convertirá en *T** cuando sea pasado como argumento. Esto implica que una asignación a un elemento de un argumento de array modifica el valor de un elemento del array de argumento. En otras palabras, los arrays difieren de los demás tipos en que un array no es (ni puede ser) pasado por valor.

El tamaño de un array no está disponible para la función llamada. Esto puede ser una molestia, pero hay varias maneras de sortear el problema. Las cadenas de estilo C terminan por cero, por lo que es fácil calcular su tamaño. Para los demás arrays se puede pasar un segundo argumento que especifique el tamaño. Por ejemplo:

```
void compute1(int* vec_ptr, int vec_size);      // una manera

struct Vec {
  int* ptr;
  int size;
};
void compute2(Vec v);       // otra manera
```

Alternativamente, se puede usar un tipo como ***vector*** (§3.7.1, §16.3) en lugar de un array.

Los arrays multidimensionales son más espinosos (§C.7), pero a menudo se pueden usar arrays de punteros en su lugar y no es necesario ningún tratamiento especial. Por ejemplo:

```
char* day[] = {
    "lun", "mar", "mie", "jue", "vie", "sab", "dom"
};
```

También en este caso ***vector*** y tipos similares son alternativas a los arrays y punteros de bajo nivel predefinidos.

7.3 Devolución de valores

Desde una función no declarada ***void*** debe ser devuelto un valor (sin embargo, ***main***() es especial; §3.2). A la inversa, no puede ser devuelto un valor desde una función ***void***. Por ejemplo:

```
int f1() { }        // error: no hay valor devuelto
void f2() { }       // bien
int f3() { }        // bien
void f4() { }       // error: valor devuelto en función void
int f5() { }        // error: falta valor devuelto
void f6() { }       // bien
```

Un valor devuelto es especificado por una sentencia de devolución. Por ejemplo:

```
int fac(int n) { return (n>1) ? n*fac(n-1) : 1; }
```

De una función que se llama a sí misma se dice que es recursiva.

Puede haber más de una sentencia devuelta en una función:

```
int fac2(int n)
{
    if (n>1) return n*fac(n-1);
    return 1;
}
```

Como la semántica de paso de argumentos, la semántica de devolución de valor de funciones es idéntica a la semántica de inicialización. Se considera que una sentencia de devolución inicializa una variable sin nombre del tipo devuelto. El tipo de una expresión de devolución se comprueba con el tipo del tipo devuelto y se realizan todas las conversiones de tipos estándar y definidas por el usuario. Por ejemplo:

```
double f() { return 1; }    // 1 se convierte implícitamente a double(1)
```

Cada vez que se llama a una función se crea una nueva copia de sus argumentos y variables locales (automáticas). La memoria es reutilizada después de que la función vuelve por lo que nunca debe ser devuelto un puntero a una variable local. El contenido de la localización a la que se apunta cambiará de forma impredecible:

```
int* fp() { int local = 1; /* ... */ return &local; }    // mal
```

Este error es menos frecuente que el error equivalente producido usando referencias:

```
int& fr() { int local = 1; /* .. */ return local; }    // mal
```

Afortunadamente, los compiladores pueden advertir fácilmente sobre la devolución de referencias a variables locales.

Una función ***void*** no puede devolver ningún valor. Sin embargo, una llamada a una función ***void*** no arroja ningún valor, por lo que una función ***void*** puede usar una llamada a una función ***void*** como expresión de una sentencia ***return***. Por ejemplo:

```
void g(int* p);
void h(int* p) { /* ... */ return g(p);    // bien: devolución de "ningún valor"
```

Esta forma de devolución es importante al escribir funciones de plantilla cuyo tipo devuelto es un parámetro de plantilla (véase §18.4.4.2).

7.4 Nombres de función sobrecargados

Casi siempre es buena idea dar nombres diferentes a las funciones diferentes, pero cuando algunas funciones realizan conceptualmente la misma tarea sobre objetos de tipos diferentes puede ser más cómodo darles el mismo nombre. Usar el mismo nombre para operaciones con tipos diferentes se denomina *sobrecarga*. Es una técnica que se usa ya para las operaciones básicas en C++. Es decir, sólo hay un nombre para la suma, +, pero se puede usar para sumar valores de tipo entero, en coma flotante y puntero. Es fácil ampliar esta idea a funciones definidas por el programador. Por ejemplo:

```
void print(int);            // imprimir un int
void print(const char*);    // imprimir una cadena de caracteres de estilo C
```

Por lo que se refiere al compilador, lo único que tienen en común las funciones con el mismo nombre es precisamente el nombre. Es de suponer que son funciones parecidas en algún sentido, pero el lenguaje no limita ni ayuda al programador. Así pues, los nombres de función sobrecargados son sobre todo una comodidad notacional. Esta comodidad es significativa para funciones con nombres convencionales como ***sqrt***, ***print*** y ***open***. Cuando un nombre es significativo desde el punto de vista semántico, esta comodidad se convierte en esencial. Es lo que ocurre, por ejemplo, con operadores como +, * y << en el caso de los constructores (§11.7) y en programación genérica (§2.7.2, capítulo 18). Cuando se llama a una función *f*, el compilador debe calcular a cuál de las funciones de nombre *f* se debe invocar. Esto se hace comparando los tipos de los argumentos reales con los tipos de los argumentos formales de todas las funciones llamadas *f*. La idea es invocar la función que mejor concuerde con los argumentos y dar un error de tiempo de compilación si ninguna función concuerda de manera óptima. Por ejemplo:

```
void print(double);
void print(long);

void f()
{
    print(1L);      // print(long)
    print(1.0);     // print(double)
    print(1);       // error, ambiguo: ¿print(long(1)) o print(double(1))?
}
```

Para encontrar la versión adecuada a la que llamar de un conjunto de funciones sobrecargadas se busca una concordancia óptima del tipo de la expresión del argumento y los parámetros (argumentos formales) de las funciones. Para acercarse a nuestras nociones de qué es razonable, se aplican por orden una serie de criterios:

[1] Concordancia exacta; es decir, concordancia no usando —o no usando sólo— conversiones triviales (por ejemplo, nombre de array a puntero, nombre de función a puntero a función y ***T*** a ***const T***).

[2] Concordancia usando promociones; es decir, promociones integrales (***bool*** a ***int***, ***char*** a ***int***, ***short*** a ***int*** y sus homólogos ***unsigned***; §6.1), ***float*** a ***double*** y ***double*** a ***long double***.

[3] Concordancia usando conversiones estándar (por ejemplo, ***int*** a ***double***, ***double*** a ***int***, ***Derived**** a ***Base**** (§12.2), ***T**** a ***void*** (§5.6), ***int*** a ***unsigned int***; §C.6).

[4] Concordancia usando conversiones definidas por el usuario (§11.4)

[5] Concordancia usando la elipsis . . . en una declaración de función (§7.6).

Si se encuentran dos concordancias al más alto nivel donde se encuentra una concordancia, la llamada es rechazada por ambigua. Las reglas de resolución son así de complicadas sobre todo para tener en cuenta las complicadas reglas de C y C++ para los tipos predefinidos (§C.6). Por ejemplo:

```
void print(int);
void print(const char*);
void print(double);
void print(long);
void print(char);

void h(char c, int i, short s, float f)
{
    print(c);          // concordancia exacta: invocar print(char)
    print(i);          // concordancia exacta: invocar print(int)
    print(s);          // promoción integral: invocar print(int)
    print(f);          // promoción float a double: print(double)

    print('a');        // concordancia exacta: invocar print(char)
    print(49);         // concordancia exacta: invocar print(int)
    print(0);          // concordancia exacta: invocar print(int)
    print("a");        // concordancia exacta: invocar print(const char*)
}
```

La llamada a ***print***(***0***) invoca a ***print***(***int***) porque ***0*** es un ***int***. La llamada a ***print***('***a***') invoca a ***print***(***char***) porque '***a***' es un ***char*** (§4.3.1). La razón por la que distinguimos entre conversiones y promociones es que queremos marcar nuestra preferencia por las promociones —seguras— como la de ***char*** a ***int***, sobre las conversiones —inseguras— como la de ***int*** a ***char***.

La resolución de la sobrecarga es independiente del orden de declaración de las funciones consideradas.

La sobrecarga descansa sobre un conjunto relativamente complicado de reglas y en ocasiones al programador le sorprenderá qué función es llamada. Así pues, ¿por qué preocuparse? Consideremos la alternativa a la sobrecarga. A menudo necesitamos operaciones similares realizadas sobre objetos de tipos diversos. Sin sobrecarga, hemos de definir varias funciones con nombres diferentes:

```
void print_int(int);
void print_char(char);
void print_string(const char*);    // cadena de estilo C

void g(int i, char c, const char* p, double d)
{
    print_int(i);          // bien
    print_char(c);         // bien
    print_string(p);       // bien
    print_int(c);          // ¿bien? llama a print_int(int(c))
    print_char(i);         // ¿bien? llama a print_char(char(i))
    print_string(i);       // error
    print_int(d);          // ¿bien? llama a print_int(int(d))
}
```

En comparación con el ***print***() sobrecargado, hemos de recordar diversos nombres y hemos de recordar usarlos correctamente. Esto puede ser tedioso, acaba con los intentos de hacer programación genérica (§2.7.2) y generalmente anima al programador a centrarse en cuestiones de tipificación de nivel relativamente bajo. Como no hay sobrecarga, todas las conversiones estándar son aplicables a los argumentos para estas funciones, lo que puede conducir también a errores. En el ejemplo anterior, implica que sólo una de las cuatro llamadas con argumento «erróneo» es capturada por el compilador. Por tanto, la sobrecarga puede aumentar las posibilidades de que un argumento inapropiado sea rechazado por el compilador.

7.4.1 Sobrecarga y tipo devuelto

En la resolución de la sobrecarga no se consideran los tipos devueltos. La razón es mantener la resolución para un operador individual (§11.2.1, §11.2.4) o llamada de función independiente del contexto. Consideremos:

```
float sqrt(float);
double sqrt(double);

void f(double da, float fla)
{
    float fl = sqrt(da);     // llamar a sqrt(double)
    double d = sqrt(da);     // llamar a sqrt(double)
    fl = sqrt(fla);          // llamar a sqrt(float)
    d = sqrt(fla);           // llamar a sqrt(float)
}
```

Si se tuviera en cuenta el tipo devuelto, no sería posible examinar aisladamente una llamada de ***sqrt***() y determinar a qué función se ha llamado.

7.4.2 Sobrecarga y ámbitos

Las funciones que se declaran en ámbitos diferentes no se sobrecargan. Por ejemplo:

```
void f(int);
void g()
{
  void f(double);
  f(1);        // llamar a f(double)
}
```

Claramente `f(int)` habría sido la mejor concordancia para `f(1)`, pero sólo `f(double)` está en ámbito. En tales casos se pueden sumar o restar declaraciones locales para obtener el comportamiento deseado. Como siempre, la ocultación intencionada puede ser una técnica útil, pero la ocultación inintencionada es una fuente de sorpresas. Cuando se desea sobrecargar traspasando ámbitos de clase (§15.2.2) o ámbitos de espacio de nombres (§8.2.9.2) se pueden usar *declaraciones-using* o *directivas-using* (§8.2.2).

7.4.3 Resolución manual de la ambigüedad

Declarar demasiado pocas (o demasiadas) versiones sobrecargadas de una función puede conducir a ambigüedades. Por ejemplo:

```
void f1(char);
void f1(long);
void f2(char*);
void f2(int*);
void k(int i)
{
  f1(i);       // ambiguo: f1(char) o f1(long)
  f2(0);       // ambiguo: f2(char*) o f2(int*)
}
```

Siempre que sea posible, lo que hay que hacer en esos casos es considerar el conjunto de versiones sobrecargadas de una función como un todo y ver si tiene sentido de acuerdo con la semántica de la función. A menudo se puede resolver el problema añadiendo una versión que resuelva la ambigüedad. Por ejemplo, añadir

```
inline void f1(int n) { f1(long(n)); }
```

resolvería todas las ambigüedades parecidas a `f1(i)` en favor del tipo `long int` más grande.

También se puede añadir una conversión explícita de tipo para resolver una llamada concreta. Por ejemplo:

```
f2(static_cast<int*>(0));
```

Sin embargo, esto no es casi siempre más que un feo parche. Enseguida podría hacerse otra llamada similar y habría que resolverla.

A algunos no iniciados en C++ les irritan los errores de ambigüedad sobre los que informa el compilador. Los programadores más experimentados saben apreciar esos mensajes de error como indicadores útiles de errores de diseño.

7.4.4 Resolución para argumentos múltiples

Dadas las reglas de resolución de la sobrecarga, es posible asegurarse de que se va a usar el algoritmo (función) más sencillo cuando la eficiencia o precisión del cálculo ofrezca diferencias significativas para los tipos que intervienen. Por ejemplo:

```
int pow(int, int);
double pow(double, double);
complex pow(double, complex);
complex pow(complex, int);
complex pow(complex, double);
complex pow(complex, complex);
void k(complex z)
{
  int i = pow(2,2);            // invocar a pow(int,int)
  double d = pow(2.0,2.0);     // invocar a pow(double,double)
  complex z2 = pow(2,z);       // invocar a pow(double,complex)
  complex z3 = pow(z,2);       // invocar a pow(complex,int)
  complex z3 = pow(z,z);       // invocar a pow(complex,complex)
}
```

En el proceso de elegir entre dos funciones sobrecargadas con dos o más argumentos, se encuentra una concordancia óptima para cada argumento usando las reglas de §7.4. Se llama a una función que es la concordancia óptima para un argumento y una concordancia mejor o igual para todos los demás argumentos. Si no existe una función así se rechaza la llamada por ambigua. Por ejemplo:

```
void g()
{
  double d = pow(2.0,2);       // error: ¿pow(int(2.0),2) o pow(2.0,double(2))?
}
```

La llamada es ambigua porque `2.0` es la concordancia óptima para el primer argumento de `pow(double,double)` y `2` es la concordancia óptima para el segundo argumento de `pow(int,int)`.

7.5 Argumentos por defecto

A menudo una función general necesita más argumentos de los que son necesarios para manejar casos sencillos. En particular, las funciones que construyen objetos (§10.2.3) proporcionan a menudo diversas opciones en bien de la flexibilidad. Consideremos una función para imprimir un entero. Parece razonable que dé al usuario una opción sobre la base a imprimir, pero en la mayor parte de los programas los enteros se imprimirán como valores decimales enteros. Por ejemplo:

```
void print(int value, int base = 10);   // la base por defecto es 10
void f()
{
```

```
    print(31);
    print(31,10);
    print(31,16);
    print(31,2);
}
```

produciría la salida siguiente:

```
31 31 1f 11111
```

El efecto de un argumento por defecto se puede lograr alternativamente por sobrecarga:

```
void print(int value, int base);
inline void print(int value) { print(value,10); }
```

Sin embargo, la sobrecarga hace menos evidente para el lector que el propósito es tener una sola función de impresión más una abreviatura.

Un argumento por defecto sufre comprobación de tipo en el momento de la declaración de la función y es evaluado en el momento de la llamada. Sólo se pueden proporcionar argumentos por defecto para los últimos argumentos. Por ejemplo:

```
int f(int, int=0, char* =0);   // bien
int g(int =0, int =0, char*);  // error
int h(int =0, int, char* =0);  // error
```

Observe el lector que el espacio entre * y = es significativo (*= es un operador de asignación; §6.2):

```
int nasty(char*=0);            // error de sintaxis
```

Un argumento por defecto puede ser repetido en una declaración posterior en el mismo ámbito, pero no cambiado. Por ejemplo:

```
void f(int x = 7);
void f(int = 7);        // bien
void f(int = 8);        // error: argumentos por defecto diferentes
void g()
{
  void f(int x = 9);    // bien: esta declaración oculta la externa
  // ...
}
```

Declarar un nombre en un ámbito anidado de modo que el nombre oculte una declaración del mismo nombre en un ámbito externo es propenso a errores.

7.6 Número de argumentos no especificado

Para algunas funciones no es posible especificar el número y tipo de todos los argumentos previstos en una llamada. Una función así se declara finalizando la lista de las declaraciones de argumentos con la elipsis (. . .) que significa «y tal vez algún argumento más». Por ejemplo:

```
int printf(const char* ...);
```

Esto especifica que una llamada a la función *printf*() (§21.8) de la biblioteca estándar C debe tener al menos un argumento, un *char**, pero puede tener otros o no. Por ejemplo:

```
printf("Hola, mundo!\n");
printf("Mi nombre es %s %s\n", nombre, apellido);
printf("%d + %d = %d\n",2,3,5);
```

Una función semejante debe basarse en información no disponible para el compilador cuando interpreta su lista de argumentos. En el caso de *printf*(), el primer argumento es una cadena de formato con secuencias de caracteres especiales que permiten a *printf*() manejar correctamente otros argumentos; *%s* significa «esperar un argumento *char**» y *%d* significa «esperar un argumento int». Sin embargo, en general el compilador no puede saber eso, por lo que no puede garantizar que estén los argumentos esperados o que un argumento sea del tipo adecuado. Por ejemplo:

```
#include<stdio.h>
int main()
{
  printf("Mi nombre es %s %s\n",2);
}
```

compilará y (en el mejor de los casos) producirá una salida de aspecto extraño (pruébelo el lector).

Evidentemente, si un argumento no ha sido declarado, el compilador no tiene la información necesaria para realizar la comprobación y conversión de tipos estándar. En ese caso, un *char* o un *short* se pasa como *int* y un *float* se pasa como *double*. Esto no es necesariamente lo que espera el programador.

Un programa bien diseñado puede necesitar como máximo unas pocas funciones para las cuales los tipos de argumentos no estén totalmente especificados. Se pueden usar funciones sobrecargadas y funciones que utilizan argumentos por defecto para ocuparse de la comprobación de tipos en la mayoría de los casos cuando se está considerando la posibilidad de dejar tipos de argumentos no especificados. Sólo cuando tanto el número *como* el tipo de argumentos varía es necesaria la elipsis. El uso más frecuente de la elipsis es para especificar una interfaz para funciones de biblioteca C definidas antes de que C++ proporcionara alternativas:

```
int fprintf(FILE*, const char* ...);   // de<cstdio>
int execl(const char* ...);            // de cabecera UNIX
```

En *<cstdarg>* se puede encontrar un conjunto estándar de macros para acceder a los argumentos no especificados en tales funciones. Consideremos la escritura de una función de error que toma un argumento entero que indica la gravedad del error seguido de un número arbitrario de cadenas. La idea es componer el mensaje de error pasando cada palabra como un argumento de cadena independiente. La lista de argumentos de cadena debería terminar por un puntero nulo a *char*:

```
extern void error(int ...);
extern char* itoa(int, char[]);     // ver §6.6[17]
const char* Null_cp = 0;
```

```
int main (int argc, char* argv[])
{
  switch (argc) {
  case 1:
    error (0, argv[0], Null_cp);
    break;
  case 2:
    error (0, argv[0], argv[1], Null_cp);
    break;
  default:
    char buffer[8];
    error (1, argv[0], "con", itoa (argc-1, buffer), 'argumentos", Null_cp);
  }
  //...
}
```

La función *itoa* ( ) devuelve la cadena de caracteres que representa su argumento entero.

Observe el lector que usar el entero **0** como terminador no habría sido portable: en algunas implementaciones el entero cero y el puntero nulo no tienen la misma representación. Esto ilustra las sutilezas y trabajo extra a que se enfrenta el programador una vez que ha sido suprimida la comprobación de tipos usando la elipsis.

La función de error se podría definir así:

```
void error (int severity ...)      // "gravedad" seguido por una lista de char*
                                   // terminada por cero
{
  va_list ap;
  va_start (ap, severity);         // comienzo de arg
  for (;;) {
    char* p = va_arg (ap, char*);
    if (p == 0) break;
    cerr << p << ' ';
  }
  va_end (ap);                     // limpieza de arg
  cerr << '\n';
  if (severity) exit (severity);
}
```

En primer lugar, se define e inicializa una *va_list* con una llamada a *va_start* ( ). La macro *va_start* toma el nombre de la *va_list* y el nombre del último argumento formal como argumentos. En cada llamada el programador debe proporcionar un tipo: *va_arg* ( ) da por supuesto que se ha pasado un argumento real de ese tipo, pero en general no tiene forma de asegurarlo. Antes de volver desde una función en la que se ha usado *va_start* ( ) hay que llamar a *va_end* ( ). La razón es que *va_start* ( ) puede modificar la pila de forma tal que no se pueda volver con éxito; *va_end* ( ) deshace esas posibles modificaciones.

7.7 Puntero a función

Hay sólo dos cosas que se puedan hacer a una función: llamarla y tomar su dirección. El puntero obtenido al tomar la dirección de una función se puede usar entonces para llamar a la función. Por ejemplo:

```
void error(string s) { /* ... */ }
void (*efct) (string);     // puntero a función
void f()
{
  efct = &error;           // efct apunta a error
  efct("error");           // llamar a error a través de efct
}
```

El compilador descubrirá que *efct* es un puntero y llamará a la función apuntada. Es decir, la desreferenciación de un puntero a función usando * es opcional. De forma similar, es opcional usar & para obtener la dirección de una función:

```
void (*f1) (string) = &error;     // bien
void (*f2) (string) = error;      // también bien; mismo significado que &error
void g()
{
  f1("Vasa");                     // bien
  (*f1)("Mary Rose");             // también bien
}
```

Los punteros a funciones tienen tipos de argumentos declarados exactamente igual que las propias funciones. En las asignaciones de punteros debe concordar exactamente el tipo completo de la función. Por ejemplo:

```
void (*pf) (string);      // puntero a void(string)
void f1(string);          // void(string)
int f2(string);           // int(string)
void f3(int*);            // void(int*)
void f()
{
  pf = &f1;               // bien
  pf = &f2;               // error: tipo devuelto incorrecto
  pf = &f3;               // error: tipo de argumento incorrecto
  pf("Hera");             // bien
  pf(1);                  // error: tipo de argumento incorrecto
  int i = pf("Zeus");     // error: void asignado a int
}
```

Las reglas para el paso de argumentos son las mismas para llamadas directas a una función y para llamadas a una función a través de un puntero.

A menudo es cómodo definir un nombre para un tipo de puntero-a-función con el fin de evitar usar todo el tiempo la sintaxis de declaración, poco obvia. Veamos un ejemplo de una cabecera de sistema UNIX:

```
(*SIG_TYP) (int) ;        // de <signal.h>
(*SIG_ARG TYP) (int) ;
ignal(int, SIG_ARG_TYP) ;
```

un array de punteros a funciones. Por ejemplo, el sistema de menú para
s basado en ratón está implementado usando arrays de punteros a fun-
ntan operaciones. No es posible describir aquí en detalle el sistema, pero
la siguiente:

```
(*PF) ();

[] = {          // operaciones de edición
aste, &copy, &search

] = {           // gestión de archivos
append, &close, &write
```

inicializar entonces punteros que controlan acciones seleccionadas de
con los botones del ratón:

```
= edit_ops;
= file_ops;
```

ación completa se necesita más información para definir cada elemento
mplo, debe estar almacenada en alguna parte una cadena que especifi-
va a visualizar. Cuando se usa el sistema, el significado de los botones
recuentemente con el contexto. Esos cambios son realizados (en parte)
r de los punteros del ratón. Cuando un usuario selecciona un elemento
l elemento 3 para el botón 2, se ejecuta la operación asociada:

```
() ;    // llamar a la tercera función del botón 2
```

ar a apreciar la fuerza expresiva de los punteros a funciones es intentar
mo el anterior sin ellos —y sin usar sus parientes cercanos y de mejor
as funciones virtuales (§12.2.6)—. Se puede modificar un menú en tiem-
sertando funciones nuevas en la tabla de operadores. También es fácil
nenús en tiempo de ejecución.
funciones se pueden usar para proporcionar una forma sencilla de rutinas
ecir, rutinas que se pueden aplicar a objetos de muchos tipos diferentes:

```
*CFT) (const void*, const void*) ;
oid* base, size_t n, size_t sz, CFT cmp)

 los "n" elementos del vector "base" en orden ascendente
 función de comparación a la que apunta "cmp".
ntos son de tamaño "sz".
ión de shells (Knuth, vol. 3, pág. 84)


ap=n/2; 0<gap; gap/=2)
    for (int i=gap; i<n; i++)
      for (int j=i-gap; 0<0j; j-=gap) {
        char* b = static_cast<char*>(base) ;  // molde necesario
        char* pj = b+j*sz;                    // &base[j]
        char* pjg = b+j*sz;                   // &base[j+gap]
        if (cmp(pj,pjg) <0) {                 // permutar base[j] y base[j+gap]
          for (int k=0; k<sz; k++) {
            char temp = pj[k] ;
            pj[k] = pjg[k];
            pjg[k] = temp;
          }
        }
      }
}
```

La rutina *ssort* ( ) no conoce el tipo de los objetos que clasifica, sólo el número de elementos (el tamaño del array), el tamaño de cada elemento y la función a llamar para realizar una comparación. El tipo de *ssort* ( ) se ha elegido para que sea el mismo que el tipo de la rutina de clasificación de la biblioteca C estándar, *qsort* ( ) . Los programas reales usan *qsort* ( ) , el algoritmo *sort* (§18.7.1) de la biblioteca estándar C++ o una rutina de clasificación especializada. Aunque es un estilo de código habitual en C, no es la manera más elegante de expresar este algoritmo en C++ (véase §13.3, §13.5.2).

Se podría usar una función de clasificación así para clasificar una tabla como la siguiente:

```
struct User {
  char* name;
  char* id;
  int dept;
};
User heads[] = {
  "Ritchie D.M.",     "dmr",   11271,
  "Sethi R.",         "ravi",  11272,
  "Szymanski T.G.",   "tgs",   11273,
  "Schryer N.L.",     "nls",   11274,
  "Schryer N.L.",     "nls",   11275,
  "Kernighan B.W.",   "bwk",   11276
};
void print_id(User* v, int n)
{
  for (int i=0; ; i<n; i++)
    cout << v[i].name << '\t' << v[i].id << '\t' << v[i].dept << '\n';
}
```

Para ser capaces de clasificar, tenemos que definir primero funciones de comparación adecuadas. Una función de comparación debe devolver un valor negativo si el primer argumento es menor que el segundo, cero si los argumentos son iguales y un número positivo en otro caso.