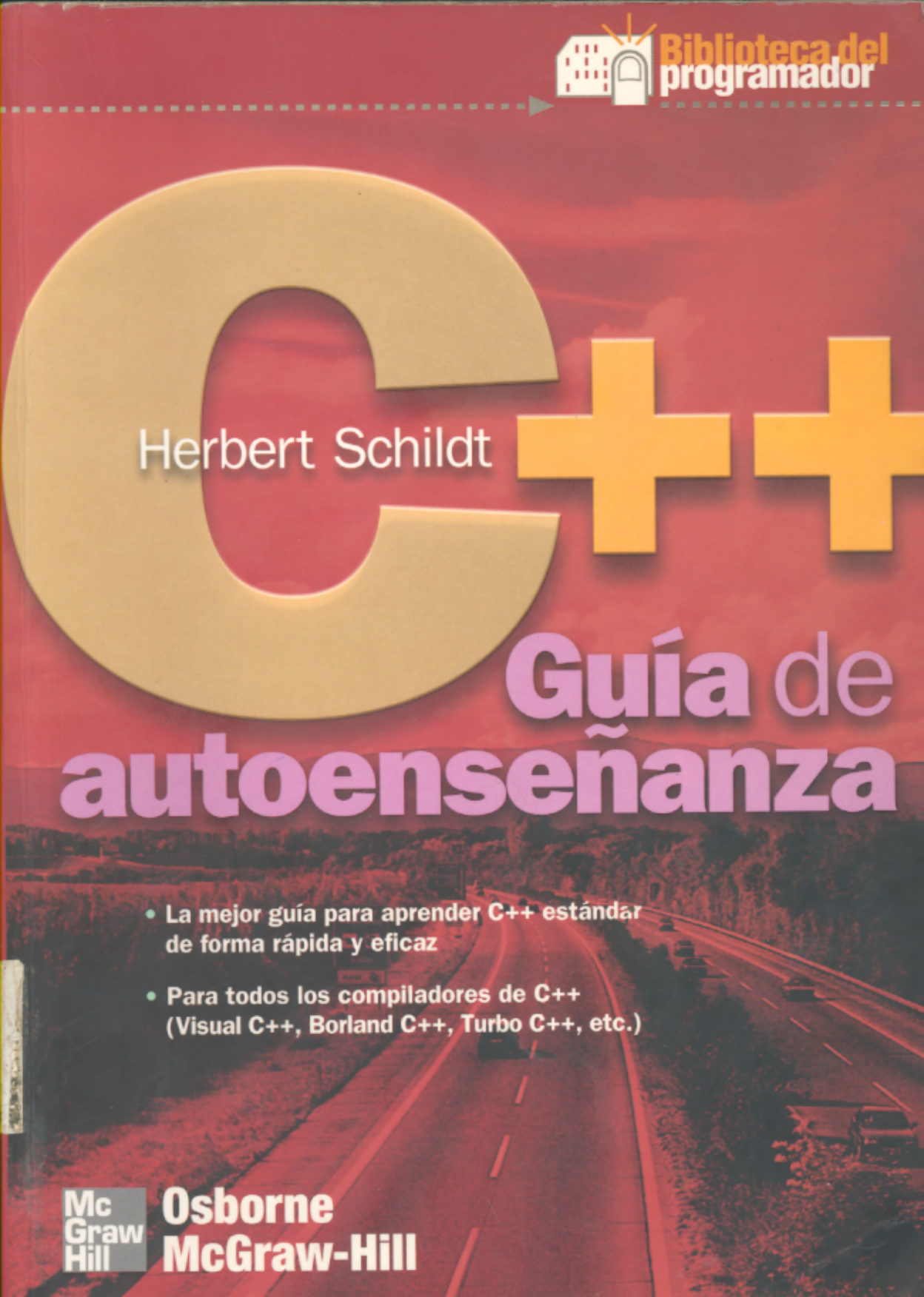

Biblioteca del programador
Herbert Schildt
C++
Guía de autoenseñanza
• La mejor guía para aprender C++ estándar de forma rápida y eficaz
• Para todos los compiladores de C++ (Visual C++, Borland C++, Turbo C++, etc.)
Mc Graw Hill
Osborne McGraw-Hill

C++

Guía de autoenseñanza

Herbert Schildt

Traducción:
CARLOS CERVIGON RÜCKAUER
Licenciado en Informática

Revisión técnica:
ANTONIO VAQUERO SANCHEZ
Catedrático de Informática
Facultad de Ciencias Físicas
Universidad Complutense de Madrid

LUIS HERNANDEZ YAÑEZ
Profesor Titular de Lenguajes y Sistemas Informáticos
Facultad de Ciencias Físicas
UniversidaXd Complutense de Madrid

Osborne/McGraw-Hill

MADRID • BUENOS AIRES • CARACAS • GUATEMALA • LISBOA • MEXICO
NUEVA YORK • PANAMA • SAN JUAN • SANTAFE DE BOGOTA • SANTIAGO • SAO PAULO
AUCKLAND • HAMBURGO • LONDRES • MILAN • MONTREAL • NUEVA DELHI • PARIS
SAN FRANCISCO • SIDNEY • SINGAPUR • ST. LOUIS • TOKIO • TORONTO

ISBN: 0-07-882025-1

ISBN: 84-481-3203-3
Depósito legal: M. 14.092-2001

Compuesto e impreso en Fernández Ciudad, S. L.

PRINTED IN SPAIN - IMPRESO EN ESPAÑA

Contenido

Introducción

C++ es la respuesta al programador de C que trabaja con Programación Orientada a Objetos (POO). Basado en los sólidos fundamentos de C, C++ añade el soporte para POO (y otras nuevas características) sin perder la capacidad, estilo y flexibilidad de C. De hecho, muchos programadores ven C++ como un «C mejorado», independientemente de que permita programación orientada a objetos. Si usted está desarrollando programas orientados a objetos o simplemente desea construir de una forma sencilla programas estructurados, C++ simplifica la labor de programación.

Puesto que C++ está basado en C, casi todos los conocimientos en C son aplicables en C++. Esto se debe, entre otras razones, a que C++ se ha convertido en el lenguaje líder de programación orientada a objetos de la década de los noventa. No existe la necesidad de que el programador experimentado en C aprenda completamente un nuevo lenguaje. En lugar de ello, sólo es necesario aprender algunas pocas características más añadidas por C++. Le sorprenderá la rapidez de sus progresos.

C++ fue inventado en 1980 por Bjarne Stroustrup en los Laboratorios Bell en Murray Hill, New Jersey. Inicialmente recibió la denominación de «C con clases». En 1983 se le dió el nombre de C++. Desde 1980 C++ ha sufrido dos importantes revisiones, una en 1985 y otra en 1990. En la actualidad se está trabajando para elaborar un estándar ANSI (American National Standards Institute) para C++. El primer borrador del estándar propuesto se creó el 25 de enero de 1994. El comité ANSI C++ (al que pertenezco) ha mantenido casi en su totalidad las características definidas por Stroustrup y ha añadido alguna nueva.

El proceso de generación de estándares es, por lo general, muy lento y, probablemente, la adopción final de un estándar en C++ llevará varios años. Por tanto, no olvide que C++ todavía es «un trabajo progresivo» y que algunas de sus características aún están siendo desarrolladas y añadidas. Sin embargo, el contenido de este libro no está sujeto a variación alguna. Es aplicable a todos

los compiladores contemporáneos de C++ y es compatible con el estándar ANSI de C++ propuesto en la actualidad. Por consiguiente, se puede usar este libro con completa confianza.

Novedades de la segunda edición

Este libro es la traducción de la segunda edición de *Teach Yourself C++*. Incluye todo el material contenido en la primera edición. Todo el material previo ha sido actualizado y comprobado de nuevo. Este libro también incluye una nueva sección (en el Capítulo 5) dedicada a los constructores de copias y un nuevo capítulo (Capítulo 11) que trata de las plantillas y de la gestión de excepciones. Las plantillas y la gestión de excepciones son carcaterísticas nuevas de C++; no existían cuando se escribió la primera edición. Sin embargo, ahora son admitidas por varios compiladores y forman parte del estándar ANSI de C++ propuesto.

Uso desde Windows

Si su computadora utiliza Windows y su objetivo es escribir programas basados en Windows, la elección de aprender este lenguaje es perfecta. C++ encaja completamente con la programación en Windows. Sin embargo, en este libro los programas no son programas para Windows. Son programas escritos para consolas. La razón de esto es fácil de entender: los programas Windows son, por su naturaleza, extensos y complicados. Para crear, incluso, un pequeño programa esquemático en Windows se requieren de 50 a 70 líneas de código. Para escribir programas Windows que presenten las características de C++ se necesitarían cientos de líneas de código por cada una de ellas. Dicho de un modo sencillo, Windows no es un entorno apropiado para aprender a programar. Sin embargo, es posible utilizar un compilador basado en Windows para compilar los programas de este libro. Es lo que se necesitaría para emplear la interfaz del indicador de órdenes (consola).

Una vez que se tienen grandes conocimientos sobre C++, éstos pueden aplicarse a la programación en Windows. De hecho, la programación de Windows que emplea C++ permite el uso de bibliotecas de clases que pueden simplificar enormemente el desarrollo de un programa en Winndows (en comparación con la versión equivalente en C). Muchos de los programas Windows creados actualmente se codifican en C++ en vez de en C.

Organización de este libro

Este libro es original porque enseña el lenguaje C++ aplicando un método de aprendizaje supervisado. Presenta cada vez una idea, seguida de numerosos ejemplos y ejercicios que ayudan a dominar cada tema. Este enfoque asegura la comprensión total de cada tema antes de pasar al siguiente.

El material se presenta secuencialmente. Por tanto, asegúrese de entender completamente cada capítulo. En cada uno de ellos se presupone que se ha asimilado el contenido de los anteriores. Al comienzo de cada capítulo (excepto el Capítulo 1) hay una sección de *Comprobación de aptitud* que verifica los conocimientos obtenidos en el capítulo previo. Al final de cada capítulo hay una sección de *Comprobación de aptitud superior* que verifica si se ha aprendido el contenido del mismo. Por último, cada capítulo concluye con una sección de *Comprobación de aptitud integrada* que comprueba la integración del contenido de ese capítulo con el de los anteriores. Las respuestas de los múltiples ejercicios del libro se encuentran en el Apéndice B.

Este libro asume que usted es ya un experimentado programador en C. Dicho de otro modo, no es posible aprender a programar en C++ sin haberlo hecho antes en C. Si no le ha sido posible programar en C, tómese algún tiempo para aprenderlo antes de comenzar a usar este libro.

1 Una perspectiva de C++

OBJETIVOS DEL CAPITULO

C++ es una versión ampliada del lenguaje C. C++ incluye todo lo que forma parte de C y añade soporte para la programación orientada a objetos (POO para abreviar). Además, C++ también contiene muchas mejoras y características que sencillamente lo convierten en «un C mejor», independientemente de la programación orientada a objetos. Con muy pocas excepciones, C++ es un superconjunto de C. Mientras que todo lo que conozca sobre el lenguaje C se puede aplicar a C++, comprender sus características más avanzadas requerirá una importante inversión de tiempo y esfuerzo por su parte. Sin embargo, las recompensas de programar en C++ justificarán de sobra el esfuerzo realizado.

El propósito de este capítulo es presentarle muchas de las características más importantes de C++. Como ya sabe, los elementos de un lenguaje de computadora no existen por sí solos, separados unos de otros. En vez de eso, trabajan juntos para formar el lenguaje completo. Esta interrelación es especialmente pronunciada en C++. De hecho es difícil tratar de forma aislada un aspecto de C++ porque las características de C++ están altamente integradas. Para ayudar a superar este problema, este capítulo da una visión general de varias características de C++. Esta visión hará que pueda entender los ejemplos que se ven posteriormente en el libro. Recuerde que muchos de los temas tratados aquí, se tratarán en profundidad en capítulos posteriores.

Además de introducir varias características importantes de C++, este capítulo también trata algunas de las diferencias entre los estilos de programación de C y C++. Hay muchas características de C++ que permiten mayor flexibilidad en la forma de escribir programas. Aunque algunas de estas características tienen poco o nada que ver con la programación orientada a objetos, se encuentran en muchos programas de C++, por lo que es apropiado tratarlas pronto en este libro.

Puesto que el lenguaje C++ se creó para soportar la programación orientada a objetos, este capítulo comienza con una descripción de la POO. Como ya se verá, muchas características de C++ están relacionadas con la POO de una manera o de otra. En realidad, la teoría de la POO filtra varios aspectos de C++. Sin embargo, es importante entender que C++ se puede usar para escribir programas que están o no están orientados a objetos. La forma de usar C++ depende completamente de usted.

Nota *Este libro asume que usted sabe cómo compilar y ejecutar un programa con el compilador de C++. Si no es así, necesitará consultar su manual de usuario del compilador. (Por las diferencias entre compiladores, es imposible dar instrucciones de compilación para cada uno de ellos en este libro.) Debido a que la mejor forma de aprender a programar es programando, es muy importante que introduzca, compile y ejecute los ejemplos del libro en el orden en el que se presentan.*

Antes de comenzar, debemos hacer unos cuantos comentarios generales sobre la naturaleza y forma de C++. En primer lugar, los programas en C++ casi siempre se parecen físicamente a los programas en C. Como ocurre en C, un programa en C++ comienza la ejecución en **main( )**. Para incluir argumentos de

línea de órdenes, C++ utiliza el mismo convenio de **argc** y **argv** que usa C. C++ tiene las mismas funciones de biblioteca estándar que C. Aunque C++ define unos pocos archivos de cabecera específicos de este lenguaje, también contiene todos los archivos de cabecera que se encuentran en un compilador de C. C++ usa las mismas estructuras de control que C. C++ tiene los mismos tipos de datos incorporados que C.

Conviene recordar que este libro asume que ya se conoce el lenguaje de programación C. Es decir, que hay que estar preparado para programar en C antes de aprender a programar en C++. Si no se conoce C, un buen punto de comienzo es mi libro *Teach yourself C, 2nd Edition* (Berkeley, Osborne/McGraw-Hill, 1994). Utiliza el mismo enfoque sistemático que se utiliza en este libro y cubre en profundidad todo el lenguaje C.

1.1. ¿QUE ES LA PROGRAMACION ORIENTADA A OBJETOS?

La programación orientada a objetos es una nueva manera de enfocar la programación. Desde sus comienzos, la programación ha estado gobernada por varias metodologías. En cada punto crítico en la evolución de la programación se creaba un nuevo enfoque para ayudar al programador a manejar programas cada vez más complejos. Los primeros programas se crearon mediante un proceso de cambio de los conmutadores del panel frontal de la computadora. Obviamente, este enfoque sólo es adecuado para programas pequeños. A continuación se inventó el lenguaje ensamblador que permitió escribir programas más largos. El siguiente avance ocurrió en los años 50 cuando se inventó el primer lenguaje de alto nivel (FORTRAN).

Mediante un lenguaje de alto nivel, un programador estaba capacitado para escribir programas que tuvieran una longitud de varios miles de líneas. Sin embargo, el método de programación usado en el comienzo era un enfoque ad hoc que no solucionaba mucho. Mientras que esto está bien para programas relativamente cortos, se convierte en «código espagueti» ilegible y difícil de tratar cuando se aplica a programas más largos. La eliminación del código espagueti se consiguió con la creación de los *lenguajes de programación estructurados* en los años sesenta. Estos lenguajes incluyen Algol y Pascal. En definitiva, C es un lenguaje estructurado, y casi todos los tipos de programas que se han estado haciendo se podrían llamar programas estructurados. Los programas estructurados se basan en estructuras de control bien definidas, bloques de código, la ausencia (o mínimo uso) del GOTO, y subrutinas independientes que soportan recursividad y variables locales. La esencia de la programación estructurada es la reducción de un programa a sus elementos constitutivos. Mediante la programación estructurada un programador medio puede crear y mantener programas de una longitud superior a 50.000 líneas.

Aunque la programación estructurada nos ha llevado a excelentes resultados cuando se ha aplicado a programas moderadamente complejos, llega a fallar en

algún punto cuando el programa alcanza un cierto tamaño. Para poder escribir programas de mayor complejidad se necesitaba un nuevo enfoque en la tarea de programación. A partir de este punto se inventa la programación orientada a objetos. La POO toma las mejores ideas incorporadas en la programación estructurada y las combina con nuevos y potentes conceptos que permiten organizar los programas de forma más efectiva. La programación orientada a objetos permite descomponer un problema en subgrupos relacionados. Cada subgrupo pasa a ser un objeto autocontenido que contiene sus propias instrucciones y datos que le relacionan con ese objeto. De esta manera, la complejidad se reduce y el programador puede tratar programas más largos.

Todos los lenguajes de POO, incluyendo C++, comparten tres características: encapsulación, polimorfismo y herencia. Analicemos estos conceptos.

Encapsulación

La *encapsulación* es el mecanismo que agrupa el código y los datos que maneja y los mantiene protegidos frente a cualquier interferencia y mal uso. En un lenguaje orientado a objetos, el código y los datos pueden empaquetarse de la misma forma en que se crea una «caja negra» autocontenida. Dentro de la caja son necesarios tanto el código como los datos. Cuando el código y los datos están enlazados de esta manera, se ha creado un *objeto*. En otras palabras, un objeto es el dispositivo que soporta encapsulación.

En un objeto, los datos y el código, o ambos, pueden ser *privados* para ese objeto o *públicos*. Los datos o el código privado sólo los conoce y son accesibles por otra parte del objeto. Es decir, una parte del programa que está fuera del objeto no puede acceder al código o a los datos privados. Cuando el código o los datos son públicos, otras partes del programa pueden acceder a ellos, incluso aunque esté definido dentro de un objeto. Normalmente, las partes públicas de un objeto se utilizan para proporcionar una interfaz controlada a las partes privadas del objeto.

Para todos los propósitos, un objeto es una variable de un tipo definido por el usuario. Puede parecer extraño que un objeto que enlaza código y datos se pueda contemplar como una variable. Sin embargo, en programación orientada a objetos, este es precisamente el caso. Cada vez que se define un nuevo objeto, se está creando un nuevo tipo de dato. Cada instancia específica de este tipo de dato es una variable compuesta.

Polimorfismo

Polimorfismo (del griego, cuyo significado es «muchas formas») es la cualidad que permite que un nombre se utilice para dos o más propósitos relacionados, pero técnicamente diferentes. El propósito del polimorfismo aplicado a la POO es permitir poder usar un nombre para especificar una clase general de acciones. Dentro de una clase general de acciones, la acción específica a aplicar está determinada por el tipo de dato. Por ejemplo, en C, que no se basa significati-

vamente en el polimorfismo, la acción de valor absoluto requiere tres funciones distintas: **abs( )**, **labs( )** y **fabs( )**. Estas funciones calculan y devuelven el valor absoluto de un entero, un entero largo y un valor real, respectivamente. Sin embargo, en C++, que incorpora polimorfismo, a cada función se puede llamar **abs( )**. (Más adelante en este capítulo se muestra una forma de cómo hacer esto.) El tipo de datos utilizado para llamar a la función determina qué versión específica de la función se está usando. Como se verá, en C++ es posible usar un nombre de función para propósitos muy diferentes. Esto se llama *sobrecarga de funciones.*

De forma general, el concepto de polimorfismo es la idea de «una interfaz, múltiples métodos». Esto significa que es posible diseñar una interfaz genérica para un grupo de actividades relacionadas. Sin embargo, la acción específica ejecutada depende de los datos. La ventaja del polimorfismo es que ayuda a reducir la complejidad permitiendo que la misma interfaz se utilice para especificar una *clase general de acción.* Es trabajo del compilador seleccionar la *acción específica* que se aplica a cada situación. El programador no necesita hacer esta selección manualmente. Sólo necesita recordar y utilizar la interfaz general. Como ilustra el ejemplo del párrafo anterior, tener tres nombres para la función de valor absoluto en vez de uno, hace que la actividad general de obtener el valor absoluto de un número sea más compleja de lo que realmente es.

El polimorfismo se puede aplicar tanto a funciones como a operadores. Prácticamente todos los lenguajes de programación contienen una aplicación limitada de polimorfismo cuando se relaciona con los operadores aritméticos. Por ejemplo, en C, el signo + se utiliza para añadir enteros, enteros largos, caracteres y valores reales. En estos casos, el compilador automáticamente sabe qué tipo de aritmética debe aplicar. En C++, se puede ampliar este concepto a otros tipos de datos que se definan. Este tipo de polimorfismo se llama *sobrecarga de operadores.*

El punto clave a recordar sobre el polimorfismo es que permite manejar complejidades más grandes a través de la creación de interfaces estándar para actividades relacionadas.

Herencia

La *Herencia* es el proceso mediante el cual un objeto puede adquirir las propiedades de otro. Más en concreto, un objeto puede heredar un conjunto general de propiedades a las que puede añadir aquellas características que son específicamente suyas. La herencia es importante porque permite que un objeto soporte el concepto de *clasificación jerárquica.* Mucha información se hace manejable gracias a la clasificación jerárquica. Por ejemplo, pensemos en la descripción de una casa. Una casa es parte de una clase general llamada **edificio**. A su vez, **edificio** es una parte de la clase más general **estructura,** que es parte de la clase aún más general de objetos que se puede llamar **obra-hombre.** En cualquier caso, la clase hija hereda todas las cualidades asociadas con la clase padre y le añade sus propias características definitorias. Sin el uso de clasificaciones ordenadas,

cada objeto tendría que definir todas las características que se relacionan con él explícitamente. Sin embargo, mediante el uso de la herencia, es posible describir un objeto estableciendo la clase general (o clases) a las que pertenece, junto con aquellas características específicas que le hacen único. Como ya se verá, la herencia juega un papel muy importante en la POO.

EJEMPLOS

1. La encapsulación no es completamente nueva para la POO. En cierto grado, la encapsulación se puede conseguir usando el lenguaje C. Por ejemplo, cuando se utiliza una función de biblioteca, en realidad se está usando una rutina de caja negra, cuya parte interna no se puede alterar o modificar (exceptuando el hecho de hacerlo mediante acciones malintencionadas).

   Consideremos la función **fopen( )**. Cuando se usa para abrir un archivo, se crean e inicializan diversas variables internas. En lo que se refiere al programa, estas variables están ocultas y no son accesibles. Sin embargo, C++ proporciona un enfoque mucho más seguro a la encapsulación.
2. En el mundo real, los ejemplos de polimorfismo son muy comunes. Por ejemplo, consideremos el volante de un coche. Funciona igual tanto si el coche utiliza dirección asistida, de cremallera o dirección normal. La cuestión es que la interfaz (el volante) es el mismo sin importar qué tipo de mecanismo de dirección (método) se esté utilizando.
3. Como ya se dijo, las propiedades de la herencia y el concepto más general de clasificación son fundamentales para la forma en que está organizado el conocimiento. Por ejemplo, el apio es un miembro de la clase **vegetales,** que es parte de la clase **plantas.** A su vez, las plantas son organismos vivos y así podríamos continuar. Sin la clasificación jerárquica, los sistemas de conocimiento no serían posibles.

EJERCICIO

1. Piense cómo la clasificación y el polimorfismo juegan un importante papel en nuestra vida diaria.

1.2. E/S POR CONSOLA DE C++

Dado que C++ es un superconjunto de C, todos los elementos del lenguaje C están contenidos en el lenguaje C++. Esto implica que todos los programas en C también son por omisión programas en C++. (En realidad hay muy pocas excepciones a esta regla, que se trata más adelante en este libro.) Por lo tanto, es posible escribir programas en C++ que parezcan programas en C. Aunque no hay nada malo en esto, significa que no se va a beneficiar de las características de C++.

La mayoría de los programadores de C++ escriben programas que usan un estilo y características que son exclusivas de C++. Una razón para ello es que ayuda a comenzar a pensar en términos de C++ en vez de hacerlo en términos de C. Además, usando características de C++, cualquiera que lea el programa sabrá inmediatamente que es un programa en C++ y no en C.

Quizás la característica específica más común utilizada por los programadores de C++ es su enfoque a la E/S por consola. Aunque se pueden utilizar funciones como **prinf( )** y **scanf( )**, C++ proporciona un método nuevo y mejor de realizar estas operaciones de E/S. En C++, la E/S se realiza usando *operadores de E/S* en vez de funciones de E/S. El operador de salida es << y el operador de entrada >>. Como sabemos, en C estos son los operadores de desplazamiento a la izquierda y derecha, respectivamente. En C++, todavía guardan su significado original (desplazamiento a izquierda y derecha) pero también tienen como tarea añadida el realizar la entrada y salida. Analicemos esta sentencia de C++:

```
cout << "Muestra esta cadena por pantalla.\n";
```

Esta sentencia hace que se muestre la cadena en la pantalla de la computadora. **cout** es un flujo predefinido que se enlaza automáticamente con la consola cuando un programa C++ comienza su ejecución. Es similar al **stdout** de C. Como en C, la consola de E/S de C++ se puede redirigir, pero para el resto de este estudio se asume que se está utilizando la consola.

Utilizando el operador de salida << es posible sacar cualquier tipo básico de C++. Por ejemplo, esta sentencia muestra el valor 100.99:

```
cout << 100.99;
```

En general, para mostrar algo por consola, utilice esta forma general del operador <<:

cout<<expression;

Aquí, *expression* puede ser cualquier expresión válida de C++ —incluyendo otra expresión de salida.

Para introducir un valor desde el teclado, utilice el operador de entrada >>. Por ejemplo, este fragmento introduce un valor entero en num:

```
int num;
cin >> num;
```

Observe que **num** *no* viene precedido por un **&**. Como ya sabe, cuando se introducen valores utilizando la función de C **scanf( )**, las variables deben pasar sus direcciones a la función para que puedan recibir los valores introducidos por el usuario. Este no es el caso cuando utilizamos operadores de entrada de C++. (Esto se verá más claramente a medida que se aprenda más sobre C++.)

En general, para introducir valores desde el teclado utilice esta forma de >>:

cin>> variable

Nota *Los papeles añadidos de << y >> son ejemplos de sobrecarga de operadores.*

Para usar los operadores de E/S de C++, se debe incluir en el programa el archivo de cabecera **iostream.h**. Este es uno de los archivos de cabecera estándar de C++ y se proporciona con el compilador de C++.

EJEMPLOS

1. Este programa muestra una cadena, dos valores enteros y un valor real doble:

```
#include <iostream.h>

main()
{
  int i, j;
  double d;

  i = 10;
  j = 20;
  d = 99.101;

  cout << "Estos son algunos valores: ";
  cout << i;
  cout << ' ';
  cout << j;
  cout << ' ';
  cout << d;

  return 0;
}
```

Nota *Como muestra este ejemplo,* **main( )** *devuelve el valor 0. Aunque no es técnicamente necesario, es una buena idea que cuando termine un programa devuelva un valor conocido al proceso de llamada (normalmente el sistema operativo). Sin la sentencia de* **return,** *podría devolver un valor indefinido.*

2. Es posible mostrar más de un valor en una sola expresión de E/S. Por ejemplo, esta versión del programa descrito en el Ejemplo 1 muestra una forma más eficiente de escribir la sentencia de E/S:

```
#include <iostream.h>

main( )
{
  int i, j;
  double d;
```

```
  i = 10;
  j = 20;
  d = 99.101;

  cout << "Estos son algunos valores: ";
  cout << i << ' '<< j << ' ' << d;

  return 0;
}
```

Aquí, la línea

```
cout << i << ' '<< j << ' ' << d;
```

muestra varios elementos en una expresión. En general, se puede usar una sola sentencia para mostrar tantos elementos como se quiera. Si esto parece confuso, simplemente recuerde que el operador de salida << se comporta como cualquier otro operador de C++ y puede ser parte de una expresión arbitrariamente larga.

Observe que hay que incluir explícitamente espacios entre los elementos cuando sea necesario. Si no se ponen los espacios, los datos aparecerán juntos en la pantalla.

3. Este programa pide al usuario un valor entero:

```
#include <iostream.h>

main( )
{
  int i;

  cout << "Introduzca un valor: ";
  cin >> i;
  cout << "Este es su número: " << i << "\n";

  return 0;
}
```

4. Este programa pide al usuario un valor entero, un valor real y una cadena. El usuario después usa una sentencia de entrada para leer los otros tres:

```
#include <iostream.h>

main( )
{
  int i;
  float f;
  char s[80];

  cout << "Introduzca un entero, un real y una cadena: ";
  cin >> i >> f >> s;
  cout << "Estos son sus datos: ";
  cout << i << ' ' << f << ' ' << s;

  return 0;
}
```

Como ilustra este ejemplo, se pueden introducir tantos elementos como se quiera en una sentencia de entrada. Como en C, los elementos de datos individuales deben separarse por espacios en blanco (espacios, tabuladores y caracteres de nueva línea).

Cuando se lee una cadena, la entrada se detendrá cuando se lea el primer carácter de espacio en blanco. Por ejemplo, si se introduce esta entrada en el programa anterior:

```
10 100.12 Esto es una prueba
```

el programa mostrará esto:

```
10 100.12 Esto
```

La cadena está incompleta porque la lectura de la cadena se paró con el espacio que hay detrás de **Esto**. El resto de la cadena se deja en el búfer de entrada, esperando una operación de entrada posterior. (Esto es similar a introducir una cadena usando **scanf( )** con el formato **%s**.)

5. Por omisión, cuando se usa >>, toda la entrada es con búfer de línea. Esto significa que no se pasa ninguna información al programa C++ hasta que se pulsa INTRO. (La mayoría de los compiladores de C también utilizan entrada con búfer de línea cuando trabajan con scanf( ), por lo que la entrada con búfer de línea no debe resultarle nueva.) Para ver el efecto de la entrada con búfer de línea, pruebe este programa:

```
#include <iostream.h>

main()
{
  char ch;

  cout << "Introduzca teclas, x para parar.\n";

  do {
    cout << ": ";
    cin >> ch;
} while (ch != 'x');

  return 0;
}
```

Cuando pruebe el programa, tendrá que pulsar INTRO después de introducir cada tecla para enviar el carácter correspondiente al programa.

EJERCICIOS

1. Escriba un programa que introduzca el número de horas que trabaja un empleado y el salario del empleado. Después muestre la paga mayor. (Asegúrese de pedir la entrada.)

2. Escriba un programa que convierta pies a pulgadas. Pida al usuario los pies y muestre el número equivalente en pulgadas. Repita este proceso hasta que el usuario introduzca 0 como número de pies.
3. Aquí se muestra un programa en C. Vuelva a escribirlo para que use sentencias de E/S al estilo de C++.

```
/* Convierta este programa en C al estilo de C++.
   Este programa calcula el mínimo común
   denominador.
*/
#include <stdio.h>
main( )
{
  int a, b, d, min;

  printf("Introduzca dos números: ");
  scanf("%d%d", &a, &b);
  min = a > b ? b : a;
  for(d = 2; d<min; d++)
    if((a%d)==0) && ((b%d)==0)) break;
  if(d==min) {
    printf("No hay denominador común\n");
    return 0;
  }
  printf("El mínimo común denominador es %d\n", d);
  return 0;
}
```

1.3. COMENTARIOS EN C++

En C++ se pueden incluir comentarios en el programa de dos formas diferentes. Primero, se puede usar el estándar, es decir el mecanismo de comentarios al estilo de C. Es decir, iniciar un comentario con /* y terminarlo con /*. Como ocurre con C, este tipo de comentarios no se pueden anidar en C++.

La segunda forma de añadir un comentario al programa de C++ es usar el *comentario de línea única*. Un comentario de línea única comienza con el símbolo // y termina al final de la línea. Aparte del final físico de la línea (es decir, una combinación de retorno carro/avance de línea) un comentario de línea única no utiliza ningún símbolo de final de comentario.

Normalmente, los programadores de C++ usan los comentarios de C para crear comentarios de varias líneas y reservan los comentarios de línea única de C++ para los comentarios breves.

EJEMPLOS

1. Aquí se muestra un programa que contiene comentarios de estilo tanto de C como de C++:

```
/*
   Este es un comentario al estilo de C.
   Este programa determina si un entero
   es impar o par.
*/

#include <iostream.h>

main()
{
  int num; // es un comentario de una sola línea de C++

  // leer el número
  cout << "Introduzca el número a probar: ";
  cin >> num;

  // ver si es par o impar
  if((num%2)==0) cout << "El número es par\n";
  else cout << "El número es impar\n";

  return 0;
}
```

2. Mientras que los comentarios al estilo de C no se pueden anidar, es posible anidar un comentario de una sola línea de C++ dentro de un comentario de varias líneas de C. Por ejemplo, esto es perfectamente válido:

```
/* Este es un comentario de varias líneas
   en el cual // se ha anidado un comentario de una sola línea.
   Aquí termina el comentario de varias líneas.
*/
```

El hecho de que comentarios de línea única se puedan anidar dentro de comentarios multilínea hace más fácil quitar el comentario de varias líneas de código con propósitos depurativos.

EJERCICIOS

1. Como experimento, determine si es válido este comentario (que anida un comentario al estilo de C dentro de un comentario de línea única al estilo de C++):

```
// Esta es una extraña /* forma de hacer un comentario */
```

2. Añada comentarios a las respuestas de los ejercicios de la Sección 1.1.

1.4. CLASES: UN PRIMER CONTACTO

Quizás la característica más importante de C++ es la clase. La clase es el mecanismo que se usa para crear objetos. Por tanto, la clase es el centro de muchas características de C++. Aunque el tema de clases se trata con gran detalle a lo largo de este libro, son tan importantes para la programación en C++ que ahora es necesario hacer un breve estudio sobre ellas.

La sintaxis de una declaración de clase es similar a la de una estructura. Su forma general se muestra aquí:

class nombre-clase {
funciones y variables privadas de la clase
public;
funciones y variables públicas de la clase
} lista de objetos;

En una declaración de clase la *lista de objetos* es opcional. Como con una estructura, más tarde se pueden declarar objetos de clase, según se necesiten. Aunque el *nombre-clase* también es opcional técnicamente, desde un punto de vista práctico, es necesario ponerlo siempre. La razón para ello es que *nombre-clase* se convierte en un nuevo nombre de tipo que se usa para declarar objetos de la clase.

Las funciones y variables declaradas dentro de una declaración de clase se dice que son *miembros* de esa clase. Por omisión, todas las funciones y variables declaradas en una clase son privadas para esa clase. Esto significa que sólo son accesibles por otros miembros de esa clase. Para declarar miembros de clase públicos se utiliza la palabra clave **public**, seguida de dos puntos. Todas las funciones y variables declaradas tras el especificador **public** son accesibles tanto por otros miembros de la clase como por cualquier otra parte del programa que contiene la clase.

A continuación podemos ver una sencilla declaración de clase:

```
class myclass {
  // privado para myclass
  int a;
public:
  void set_a(int num);
  int get_a();
};
```

Esta clase tiene una variable privada, llamada **a**, y dos funciones públicas, **set_a( )** y **get_a( )**. Podemos ver que las funciones están declaradas dentro de una clase utilizando sus formas de prototipo. Las funciones declaradas como parte de una clase se llaman *funciones miembro*.

Dado que **a** es privada, no es accesible por ningún otro código que esté fuera de **myclass**. Sin embargo, dado que **set_a( )** y **get_a( )** son miembros de **myclass**, pueden acceder a **a**. Además, **get_a( )** y **set_a( )** se declaran como miembros públicos de **myclass**, y cualquier otra parte del programa que contenga **myclass** los pueden llamar.

Aunque las funciones **get_a( )** y **set_a( )** están declaradas por **myclass**, todavía no están definidas. Para definir una función miembro, se debe enlazar el nombre tipo de la clase de la que es parte la función miembro con el nombre de la función. Esto se hace precediendo el nombre de la función con el nombre de clase seguido de dos símbolos de dos puntos. Los dos símbolos de dos puntos

se llaman *operador de resolución del ámbito.* Por ejemplo, a continuación se muestra cómo se definen las funciones miembro **set_a( )** y **get_a( )**:

```
void myclass::set_a(int num)
{
  a = num;
}
int myclass::get_a()
{
  return a;
}
```

Observe que tanto **set_a( )** como **get_a( )** tienen acceso a **a**, que es privado a **myclass**. Puesto que **set_a( )** y **get_a( )** son miembros de **myclass**, pueden acceder directamente a sus datos privados.

Cuando defina una función miembro, utilice esta forma general:

```
tipo nombre-clase::nombre-func(lista-parámetros)
{
 ... // cuerpo de la función
}
```

La declaración de **myclass** no definió ningún objeto de tipo **myclass** —sólo define el tipo de objeto que se creará cuando realmente se declare uno. Para crear un objeto, utilice el nombre de clase como un especificador de tipo. Por ejemplo, esta línea declara dos objetos de tipo **myclass**:

```
myclass ob1, ob2; // estos son objetos de tipo myclass
```

Recuerde *Una declaración de clase es una abstracción lógica que define un nuevo tipo que determina cómo será un objeto de ese tipo. Una declaración de objeto crea una entidad física de ese tipo. (Es decir, un objeto ocupa espacio de memoria, pero una definición de tipo no.)*

Una vez creado un objeto de clase, el programa puede referenciar sus miembros públicos usando el operador punto de la misma forma en que se accede a los miembros de una estructura. Suponiendo la declaración de objeto anterior, esta sentencia llama a **set_a( )** para los objetos **ob1** y **ob2**:

```
ob1.set_a(10); // versión ob1 de a hasta 10
ob2.set_a(99); // versión ob2 de a hasta 99
```

Como indica el comentario, estas sentencias establecen una copia de **ob1** de **a** a 10 y una copia de **ob2** a 99. Cada objeto contiene su propia copia de todos los datos declarados en la clase. Esto significa que el **a** de **ob1** es distinto y diferente del **a** vinculado a **ob2**.

Recuerde *Cada objeto de una clase tiene su propia copia de cada variable declarada dentro de la clase.*

EJEMPLOS

1. Como un primer ejemplo sencillo, este programa usa **myclass,** descrito en el texto, para fijar el valor de **a** para **ob1** y **ob2** y para mostrar el valor de **a** para cada objeto:

```
#include <iostream.h>

class myclass {
  // privado a myclass
  int a;
public:
  void set_a(int num);
  int get_a();
};

void myclass::set_a(int num)
{
  a = num;
}

int myclass::get_a()
{
  return a;
}

main()
{
  myclass ob1, ob2;

 ob1.set_a(10);
  ob2.set_a(99);

  cout << ob1.get_a() << "\n";
  cout << ob2.get_a() << "\n";
  return 0;
}
```

Como es de esperar, este programa muestra en pantalla los valores 10 y 99.

2. En **myclass** del ejemplo anterior, **a** es privado. Esto significa que sólo las funciones miembro de **myclass** pueden acceder directamente a ella. (Esta es una de las razones por las que se requiere la función pública **get_a( )**.) Si se intenta acceder a un miembro privado de una clase desde alguna parte del programa que no es miembro de esa clase, se producirá un error de tiempo de compilación. Por ejemplo, suponiendo que **myclass** está definida como se muestra en el ejemplo precedente, la siguiente función **main( )** producirá un error:

```
//Este fragmento contiene un error
#include <iostream.h>

main()
{
  myclass ob1, ob2;
```

```
  ob1.a = 10; // ¡ERROR! no se puede acceder a un miembro privado
  ob2.a = 99; // mediante funciones no miembro.

 cout << ob1.get_a() << "\n";

 cout << ob2.get_a() << "\n";

  return 0;
}
```

3. Igual que puede haber funciones miembro públicas, también puede haber variables miembro públicas. Por ejemplo, si **a** se hubiera declarado en la sección pública de **myclass,** entonces cualquier parte del programa podría referenciar a **a**, como se muestra a continuación:

```
#include <iostream.h>

class myclass {
public:
  // ahora a es público
  int a;
  // y no hace falta set_a() o get_a()
};
main()
{
  myclass ob1, ob2;

  // aquí, se accede directamente a a
  ob1.a = 10;
  ob2.a = 99;

  cout << ob1.a << "\n";
  cout << ob2.a << "\n";

 return 0;
}
```

En este ejemplo, ya que **a** se declara como un miembro público de **myclass,** es accesible directamente desde **main( )**. Observe cómo el operador punto se usa para acceder a **a**. En general, tanto si se está llamando a una función miembro como accediendo a una variable miembro, se requiere el nombre del objeto seguido del operador punto seguido por el nombre del miembro, para especificar completamente a qué objeto miembro se está refiriendo.

4. Para comprobar el poder de los objetos, veamos un ejemplo más práctico. Este programa crea una clase llamada **stack** que implementa un pila que puede servir para almacenar caracteres:

```
#include <iostream.h>
#define SIZE 10

// Declara una clase pila de caracteres
class stack {
  char stck[SIZE]; // guarda la pila
  int tos; // índice de la cabeza de la pila
```

```
public:
  void init(); // inicializa la pila
  void push(char ch); // mete carácter en la pila
  char pop(); // saca carácter de la pila
};

// Inicializa la pila
void stack::init()
{
  tos = 0;
}

// Mete un carácter.
void stack::push(char ch)
{
  if(tos==SIZE) {
  cout << "La pila está llena";
  return;
  }
  stck[tos] = ch;
  tos++;
}

// Saca un carácter.
char stack::pop()
{
  if(tos==0) {
  cout << "La pila está vacía";
  return 0; // devuelve nulo cuando la pila está vacía
  }
  tos--;
  return stck[tos];
}
main()
{
  stack s1, s2; // Crear dos pilas
  int i;
  // inicializa las pilas
  s1.init();
  s2.init();

  s1.push('a');
  s2.push('x');
  s1.push('b');
  s2.push('y');
  s1.push('c');
  s2.push('z');

  for(i=0; i<3; i++) cout << "Saca de s1: " << s1.pop() << "\n";
  for(i=0; i<3; i++) cout << "Saca de s2: " << s2.pop() << "\n";

  return 0;
}
```

Este programa muestra la siguiente salida:

```
Saca de s1: c
Saca de s1: b
Saca de s1: a
Saca de s2: z
Saca de s2: y
Saca de s2: x
```

Observe más detenidamente este programa. La clase **stack** contiene dos variables privadas: **stck** y **tos.** El array **stck** realmente guarda los caracteres enviados a la pila, y **tos** contiene el índice de la cabeza de la pila. Las funciones públicas de la pila son **init( )**, **push( )** y **pop( )**, que inicializa la pila, mete un valor y lo saca, respectivamente.

Dentro de **main( )** se crean dos pilas, **s1** y **s2**, y se meten tres caracteres en cada pila. Es importante entender que cada objeto pila está separado de los otros. Por ello, los caracteres que se meten en **s1** *de ninguna manera* afectan a los caracteres que se meten en **s2.** Cada objeto contiene su propia copia de **stck** y **tos.** Este concepto es fundamental para entender los objetos. Aunque todos los objetos de una clase comparten sus funciones miembro, cada objeto crea y guarda *sus propios datos.*

EJERCICIOS

1. Si no lo ha hecho todavía, introduzca y ejecute los programas mostrados en los ejemplos de esta sección.
2. Cree una clase llamada **card** que guarde una entrada de catálogo de fichas de biblioteca. Haga que la clase guarde el título del libro, autor y número de copias a mano. Almacene el título y autor como cadenas y el número de ellos disponibles como un entero. Use una función miembro pública llamada **store( )** para almacenar la información de un libro y una función miembro pública llamada **show( )** para mostrar la información. Incluya un **main( )** breve para probar la clase.
3. Cree una clase cola que guarde una cola circular de enteros. Haga que el tamaño de la cola sea de 100 enteros. Incluya una breve función **main( )** que pruebe su funcionamiento.

1.5. ALGUNAS DIFERENCIAS ENTRE C Y C++

Aunque C++ es un superconjunto de C, existen algunas diferencias entre los dos que casi seguro le afectarán cuando empiece a escribir programas en C++. Aunque estas diferencias son pequeñas, suelen estar en los programas de C++. Por lo tanto, antes de seguir conviene tomarse algún tiempo para repasar estas diferencias, que se tratan aquí.

En primer lugar, en C, cuando una función no toma parámetros, su prototipo tiene la palabra **void** dentro de su lista de parámetros de función. Por ejemplo, en C, si una función llamada **f1( )** no toma parámetros (y devuelve a **char**), entonces su prototipo se parecerá a éste:

```
char f1(void);
```

Sin embargo, en C++, void es opcional. Por lo tanto, en C++ el prototipo para **f1( )** se escribe habitualmente así:

```
char f1();
```

C++ se diferencia de C en la forma en que se especifica una lista de parámetros vacía. Si el prototipo precedente se diera en un programa en C, simplemente significaría que *no* se dice nada sobre los parámetros de la función. En C++ significa que la función no tiene parámetros. Esta es la razón por la que varios de los ejemplos anteriores no usaban explícitamente **void** para declarar una lista de parámetros vacía. (El uso de **void** para declarar una lista de parámetros vacía no es ilegal; es simplemente redundante. Puesto que la mayoría de los programadores persiguen la eficiencia como celo profesional, casi nunca se verá **void** utilizado de este modo.) Conviene recordar que en C++ estas dos declaraciones son equivalentes:

```
int f1();
int f1(void);
```

Otra diferencia entre C y C++ es que en un programa de C++ todas las funciones deben estar en forma de prototipo. Recuerde que en C, los prototipos se recomiendan, pero técnicamente son opcionales. En C++ son obligatorios. Como muestran los ejemplos de la sección anterior, un prototipo de una función miembro contenido en una clase también sirve como su prototipo general, y no se requiere ningún otro prototipo aparte.

Una tercera diferencia entre C y C++ es que en C++, si una función se declara para que devuelva un valor, entonces debe devolver un valor. Es decir, si una función tiene un tipo de retorno distinto de **void,** entonces cualquier sentencia **return** dentro de esa función debe contener un valor. En C, una función que no sea **void** no es necesario que devuelva un valor. Si no lo hace, «devuelve» un valor irrelevante .

Otra diferencia entre C y C++ que se encontrará habitualmente en programas de C++ tiene que ver con el lugar dónde se declaran las variables locales. En C, las variables locales se deben declarar sólo al principio de un bloque, con prioridad a cualquier sentencia de «acción». En C++, las variables locales se pueden declarar en cualquier lugar. Una ventaja de este enfoque es que las variables locales se pueden declarar cerca de donde se utilizan por primera vez, lo que ayuda a prevenir efectos laterales no deseados.

EJEMPLOS

1. En un programa en C es un práctica común declarar **main( )** como se muestra aquí si no toma argumentos de línea de órdenes:

   ```
   main(void)
   ```

   Sin embargo, en C++., el uso de **void** es redundante e innecesario.

2. Este breve programa en C++ no compilará porque la función **sum( )** no está como prototipo:

```
// Este programa no se compilará.
#include <iostream.h>

main()
{
  int a, b, c;
  cout << "Introduzca dos números: ";
  cin >> a >> b;
  c = sum(a, b);
  cout << "La suma es: " << c;

  return 0;
}

// Esta función necesita un prototipo.
sum(int a, int b)
{
  return a+b;
}
```

3. Aquí se muestra un breve programa que ilustra cómo se pueden declarar las variables locales en cualquier lugar dentro de un bloque:

```
#include <iostream.h>

main()
{
  int i; // variables locales declaradas al principio del bloque

  cout << "Introduzca un número: ";
  cin >> i;

  // calcula el factorial
  int j, fact=1; // variables declaradas después
  //de las sentencias de acción

  for(j=i; j>=1; j--) fact = fact * j;

  cout << "El factorial es " << fact;

  return 0;
}
```

La declaración de **j** y **fact** cerca del lugar donde se usa por primera vez es de poco valor en este corto ejemplo; sin embargo, en funciones largas, la posibilidad de declarar variables cerca del lugar donde se usa por primera vez puede ayudar a clarificar el código y prevenir efectos laterales no deseados. Esta característica de C++ se usa mucho en los programas de C++ cuando están implicadas funciones largas.

EJERCICIOS

1. El siguiente programa no compilará como un programa C++. ¿Por qué no?

```
#include <iostream.h>

main()
{
  char s[80];

  cout << "Introduzca una cadena: ";
  cin >> s;

  cout << "La longitud es: ";
  cout << strlen(s);

return 0;
}
```

2. Por su cuenta, intente declarar variables locales en varios puntos de un programa de C++. Intente hacer lo mismo en un programa de C y preste atención a las declaraciones que generan errores.

1.6. INTRODUCCION A LA SOBRECARGA DE FUNCIONES

Después de las clases, quizás la siguiente característica más importante y llamativa de C++ es *la sobrecarga de funciones.* La sobrecarga de funciones no sólo proporciona un mecanismo mediante el cual C++ adquiere un tipo de polimorfismo, también constituye la base mediante la cual el entorno de programación de C++ se puede ampliar dinámicamente. Por la importancia de la sobrecarga, comenzamos con una breve introducción.

En C++ dos o más funciones pueden compartir el mismo nombre en tanto en cuanto difiera el tipo de sus argumentos o el número de sus argumentos —o ambos. Cuando dos o más funciones comparten el mismo nombre, se dice que están *sobrecargadas.* Las funciones sobrecargadas pueden ayudar a reducir la complejidad de un programa permitiendo que operaciones relacionadas se referencien mediante el mismo nombre.

Es muy fácil sobrecargar una función: simplemente hay que declarar y definir todas las versiones requeridas. El compilador seleccionará automáticamente la versión correcta para llamar, en base al número y/o tipo de argumentos usados para llamar a la función.

Nota *En C++ también es posible sobrecargar operadores. Sin embargo, antes de poder entender por completo la sobrecarga de operadores es necesario saber más sobre C++.*

EJEMPLOS

1. Uno de los principales usos de la sobrecarga de funciones es conseguir polimorfismo en tiempo de compilación, que se ajusta a la filosofía de un interfaz, muchos métodos. Como ya se sabe, en programación en C es común tener un número de funciones relacionadas que difieren sólo en el tipo de dato sobre el que operan. El ejemplo clásico de esta situación se encuentra en la biblioteca estándar de C. Como se mencionó anteriormente en este capítulo, la biblioteca contiene las funciones **abs( ), labs( )** y **fabs( )**, que devuelven el valor absoluto de un entero, un entero largo y un valor real, respectivamente.

   Sin embargo, debido a que se necesitan tres nombres diferentes junto a los tres tipos diferentes de datos, la situación es más complicada de lo que debería ser. En los tres casos, se devuelve el valor absoluto; sólo son distintos los tipos de datos. En C++, se puede corregir esta situación sobrecargando un nombre para los tres tipos de datos, como lo ilustra este ejemplo:

```
#include <iostream.h>

// Sobrecarga abs() de tres formas
int abs(int n);
long abs(long n);
double abs(double n);

main()
{
  cout << "Valor absoluto de -10: " << abs(-10) << "\n";
  cout << "Valor absoluto de -10L: " << abs(-10L) << "\n";
  cout << "Valor absoluto de -10.01: " << abs(-10.01) << "\n";

  return 0;
}

// abs() para int
int abs(int n)
{
  cout << "Con un entero abs()\n";
  return n<0 ? -n : n;
}

// abs() para largos
long abs(long n)
{
  cout << "Con largos abs()\n";
  return n<0 ? -n : n;
}

// abs() para dobles
double abs(double n)
{
  cout << "Con doble abs()\n";
  return n<0 ? -n : n;
}
```

Como puede ver, este programa define tres funciones llamadas **abs( )** —una para cada tipo de dato. Dentro de **main( )**, a **abs( )** se llama usando tres tipos diferentes de argumentos. El compilador automáticamente llama a la versión correcta de **abs( )** basándose en el tipo de dato usado como argumento.

Aunque este ejemplo es bastante simple, ilustra el valor de la sobrecarga de funciones. Puesto que un solo nombre se puede usar para describir una clase general de acción, se elimina la complejidad artificial causada por los tres nombres ligeramente diferentes —en este caso, **abs( ), fabs( )** y **labs( )**. Ahora sólo tiene que recordar un nombre —aquel que describe la acción *general*. Se deja que el compilador elija la versión *específica* apropiada de la función a llamar (es decir, el método). Esto tiene el efecto de reducir la complejidad. Por tanto, a través del polimorfismo se han reducido tres nombres a uno.

Aunque el uso del polimorfismo en este ejemplo es bastante trivial, debe ser capaz de ver cómo puede ser bastante efectivo en un programa muy largo el enfoque «una interfaz, múltiples métodos».

2. Aquí tenemos otro ejemplo de sobrecarga de funciones. En este caso, la función **date( )** está sobrecargada para aceptar la fecha tanto como una cadena o como tres enteros. En ambos casos la función muestra la fecha que se le ha pasado:

```
#include <iostream.h>

void date(char *date); // fecha como una cadena
void date(int month, int day, int year); // fecha en números

main()
{
  date("8/23/95");
  date(8, 23, 95);

  return 0;
}
// Fecha como una cadena.
void date(char *date)
{
  cout << "Fecha: " << date << "\n";
}

// Fecha como enteros.
void date(int month, int day, int year)
{
  cout << "Fecha: " << month << "/";
  cout << day << "/" << year << "\n";
}
```

Este ejemplo ilustra cómo la sobrecarga de funciones puede proporcionar la interfaz más natural a una función. Puesto que es muy común que la fecha se represente como una cadena o como tres enteros conteniendo el mes, día y año, es libre de seleccionar la forma más conveniente relacionada con la situación del momento.

3. Además, se pueden ver funciones sobrecargadas que difieren en el tipo de datos de sus argumentos. Sin embargo, las funciones sobrecargadas pueden diferir también en el número de argumentos, como se explica en este ejemplo:

```
#include <iostream.h>

void f1(int a);
void f1(int a, int b);

main()
{
  f1(10);
  f1(10, 20);

  return 0;
}

void f1(int a)
{
  cout << "En f1(int a)\n";
}

void f1(int a, int b)
{
  cout << "En f1(int a, int b)\n";
}
```

4. Es importante entender que el tipo devuelto no representa una diferencia suficiente para permitir la sobrecarga de funciones. Si dos funciones difieren sólo en el tipo de datos que devuelven, el compilador no estará siempre preparado para elegir el apropiado al que llamar. Por ejemplo, este fragmento es incorrecto porque es ambiguo:

```
// Esto es incorrecto y no se compilará
int f1(int a);
double f1(int a);
.
.
.
f1(10); // ¿A qué función llama el compilador?
```

Como indica el comentario, el compilador no tiene manera de saber a qué versión de **f1( )** llamar.

EJERCICIOS

1. Cree una función llamada **sroot( )** que devuelva la raíz cuadrada de su argumento. Sobrecargue **sroot( )** de tres formas: haga que devuelva la raíz cuadrada de un entero, un entero largo y un **double.** (Para calcular la raíz cuadrada, puede usar la función de la biblioteca estándar **sqrt( ).**)
2. La biblioteca estándar de C++ contiene estas tres funciones:

```
double atof(const char *s);
int atoi(const char *s);
long atol(const char *s);
```

que devuelven el valor numérico contenido en la cadena a la que apunta **s.** Concretamente, **atof( )** devuelve un **doble, atoi( )** devuelve un entero y **atol( )** devuelve un **long.** ¿Por qué no es posible sobrecargar estas funciones?

3. Cree una función llamada **min( )** que devuelva el más pequeño de los dos argumentos numéricos usados al llamar a la función. Sobrecargue **min( )** para que acepte caracteres, enteros y **doubles** como argumentos.
4. Cree una función llamada **sleep( )** que detenga la computadora el número de segundos especificado mediante su argumento. Sobrecargue **sleep( )** para que se pueda llamar tanto con un entero o una cadena que represente un entero. Por ejemplo, cualquiera de estas dos llamadas a **sleep( )** hará que la computadora se detenga durante 10 segundos:

```
sleep(10);
sleep("10");
```

Demuestre que las funciones funcionan bien incluyéndolas en un breve programa. (Puede usar un bucle de retardo para detener la computadora.)

1.7. PALABRAS CLAVE DE C++

Además de las 32 palabras clave que forman el lenguaje C, el estándar ANSI propuesto para C++ añade 29 palabras más. Estas palabras claves se muestran en la Tabla 1.1. Sin embargo, en el momento de escribir esto, las palabras clave **bool, const_cast, dynamic_cast, false, mutable, namespace, reinterpret_cast, static_cast, true, typeid, using** y **wchar_t,** están en proceso de ser definidas por el comité del estándar del ANSI C++, y no las implementa ningún compilador disponible normalmente. Estas palabras clave no son parte de la especificación original de C++ creada por Bjarne Stroustrup. Básicamente se están añadiendo para permitir que C++ se adecúe a algunas situaciones de casos especiales, y están sujetas a cambio o eliminación. (Estas palabras clave se tratan brevemente en el Apéndice A.) Además, la palabra clave **overload,** está obsoleta, pero se incluye por compatibilidad con los antiguos programas de C++. Revise el manual de usuario del compilador para determinar exactamente qué palabras clave de C++ soporta su compilador.

Tabla 1.1. *Palabras clave de C++*

asm	friend	protected	try
bool	inline	public	typeid
catch	mutable	reinterpret_cast	using
class	namespace	static_cast	virtual
const_cast	new	template	wchar_t
delete	operator	this	
dynamic_cast	overload	throw	
false	private	true	

COMPROBACION DE APTITUD SUPERIOR

1. Dé una breve descripción del polimorfismo, encapsulación y herencia.
2. ¿Cómo se pueden incluir los comentarios en un programa en C++?
3. Escriba un programa que use E/S al estilo de C++ para introducir dos enteros desde el teclado y después mostrar el resultado de elevar el primero a la potencia del segundo. (Por ejemplo, si el usuario introduce 2 y 4, entonces el resultado es 2^4 ó 16.)
4. Cree una función llamada **rev_str( )** que invierta una cadena. Sobrecargue **rev_str( )** para que se pueda llamar con uno o dos arrays de caracteres. Cuando se llame con una cadena, haga que una cadena contenga la invertida. Cuando se llame con dos cadenas, devuelva la cadena invertida en el segundo argumento. Por ejemplo:

```
char s1[80], s2[80];
strcpy(s1, "hola");
rev_str(s1, s2); // cadena invertida va en s2, s1 sin tocar
rev_str(s1); // devuelve la cadena invertida en s1
```

2 *Introducción a las clases*

OBJETIVOS DEL CAPITULO

Este capítulo presenta las clases y los objetos. Es necesaria una lectura detenida porque se tratan muchos temas importantes que están relacionados con prácticamente todos los aspectos de la programación de C++.

COMPROBACION DE APTITUD

Antes de continuar, debería ser capaz de responder correctamente las siguientes preguntas y hacer los ejercicios.

1. Escriba un programa que use E/S al estilo de C++ para pedir al usuario una cadena y después mostrar su longitud.
2. Cree una clase que contenga información de nombres y direcciones. Almacene toda la información en cadenas de caracteres en la parte privada de la clase. Incluya una función pública que almacene el nombre y dirección. Incluya también una función pública que muestre el nombre y la dirección. (Llame a estas funciones **store( )** y **display( )**.)
3. Cree una función sobrecargada llamada **rotate( )** que rote a la izquierda los bits de su argumento y devuelva el resultado. Sobrecargue dicha función para que acepte enteros y **longs**. (Una rotación es similar a un desplazamiento, excepto en que el bit desplazado de un extremo se desplaza al otro extremo.)
4. ¿Qué es incorrecto en el siguiente fragmento?

```
#include <iostream.h>

class myclass {
  int i;
public:
.
.
.
};

main()
{
  myclass ob;
  ob.i = 10;
.
.
.
}
```

2.1. FUNCIONES CONSTRUCTORAS Y DESTRUCTORAS

Si ha estado escribiendo programas durante mucho tiempo, sabrá que es muy común que partes del programa requieran inicialización. La necesidad de inicialización es incluso más común cuando se está trabajando con objetos. De hecho, cuando se aplican a problemas reales, prácticamente cada objeto que se crea va a necesitar algún tipo de inicialización. Para tratar esta situación, C++ permite

incluir una *función constructora* en una declaración de clase. A un constructor de clase se le llama cada vez que se crea un objeto de esa clase. De esta manera, cualquier inicialización que sea necesaria en un objeto la puede realizar automáticamente la función constructora.

Una función constructora tiene el mismo nombre que la clase de la que es parte y no tiene tipo devuelto. Por ejemplo, aquí se muestra una pequeña clase que contiene una función constructora:

```
#include <iostream.h>

class myclass {
  int a;
public:
  myclass(); // constructora
  void show();
};

myclass::myclass()
{
  cout << "En constructor\n";
  a = 10;
}

void myclass::show()
{
  cout << a;
}

main()
{
  myclass ob;

  ob.show();

  return 0;
}
```

En este sencillo ejemplo, el valor de **a** lo inicializa el constructor **myclass( )**. Al constructor se le llama cuando se crea el objeto **ob**. Un objeto se crea cuando se ejecuta la sentencia de declaración del objeto. Es importante entender que en C++, una sentencia de declaración de variable es una «sentencia de acción.» Cuando se programa en C++, es fácil pensar que las sentencias de declaración es como establecer variables. Sin embargo, en C++, dado que un objeto puede tener una función constructora, una sentencia de declaración de variable puede hacer que se produzcan acciones considerables.

Observe cómo se define **myclass( )**. Como se ha visto, no hay tipo devuelto. Según las reglas sintácticas formales de C++, es ilegal que un constructor tenga un tipo devuelto.

Para objetos globales, a un constructor de objetos se le llama una vez, cuando el programa comienza la ejecución. Para objetos locales, al constructor se le llama cada vez que se ejecuta la sentencia de declaración.

El complemento de un constructor es el *destructor*. A esta función se le llama cuando se destruye un objeto. Cuando se trabaja con objetos es muy común tener que realizar algunas acciones cuando se destruye el objeto. Por ejemplo, un objeto que asigna memoria cuando se crea querrá liberar la memoria cuando se destruya. El nombre de un destructor es el nombre de la clase a la que pertenece precedido por el carácter ~. Por ejemplo, esta clase contiene una función destructora.

```
#include <iostream.h>

class myclass {
  int a;
public:
  myclass(); // constructora
  ~myclass(); // destructora
  void show();
};

myclass::myclass()
{
  cout << "En constructor\n";
  a = 10;
}
myclass::~myclass()
{
  cout << "Destruyendo...\n";
}

void myclass::show()
{
  cout << a << "\n";
}

main()
{
  myclass ob;

  ob.show();

  return 0;
}
```

Al destructor de la clase se le llama cuando se destruye un objeto. Los objetos locales se destruyen cuando se salen del ámbito. Los objetos globales se destruyen cuando finaliza el programa.

No es posible obtener la dirección de un constructor ni de un destructor.

Nota *Técnicamente, un constructor o un destructor pueden realizar cualquier tipo de operación. El código que se genera dentro de estas funciones no tiene porqué inicializar o reinicializar nada relacionado con la clase para la que están definidos. Por ejemplo, un constructor para los ejemplos anteriores podría haber calculado el área de 100 círculos. Sin embargo, tener un constructor o un destructor que realiza acciones no directamente relacionadas con la inicialización o destrucción ordenada de un objeto hace que el estilo de programación sea pobre y hay que evitarlo.*

EJEMPLOS

1. Debería recordar que la clase **stack** creada en el Capítulo 1 requería una función de inicialización para establecer la variable del índice de la pila. Este es precisamente el tipo de operación que se diseñó para que llevara a cabo una función constructora. Aquí tenemos una versión mejorada de la clase **stack** que usa un constructor para inicializar automáticamente un objeto pila cuando se crea:

```
#include <iostream.h>

define SIZE 10

// Declara una clase pila para caracteres
class stack {
  char stck[SIZE]; // guarda la pila
  int tos; // índice de la cabeza de la pila
public:
  stack(); // constructor.
  void push(char ch); // mete un carácter en la pila
  char pop(); // saca un carácter de la pila
};

// Inicializar la pila
stack::stack()
{
  cout << "Construyendo una pila\n";
  tos = 0;
}

// Meter un carácter.
void stack::push(char ch)
{
  if(tos==SIZE) {
  cout << "La pila está llena\n";
  return;
  }
  stck[tos] = ch;
  tos++;
}

// Saca un carácter.
char stack::pop()
{
  if(tos==0) {
  cout << "La pila está vacía";
  return 0; // devuelve nulo cuando la pila está vacía
  }
  tos--;
  return stck[tos];
}

main()
```

```
{
  // crea dos pilas que se inicializan automáticamente
  stack s1, s2;
  int i;

  s1.push('a');
  s2.push('x');
  s1.push('b');
  s2.push('y');
  s1.push('c');
  s2.push('z');

  for(i=0; i<3; i++) cout << "Saca de s1: " << s1.pop() << "\n";
  for(i=0; i<3; i++) cout << "Saca de s2: " << s2.pop() << "\n";

  return 0;
}
```

Como se puede ver, ahora la tarea de inicialización la realiza automáticamente la función constructora en vez de una función separada que debe llamar el programa explícitamente. Este es un punto importante. Cuando se realiza una inicialización automáticamente al crearse el objeto, elimina cualquier posibilidad de que, por error, no se realice la inicialización. Esta es otra forma en la que los objetos ayudan a reducir la complejidad del programa. El programador no se tiene que preocupar de inicializar —se realiza automáticamente cuando el objeto comienza a existir.

2. Aquí tenemos un ejemplo que muestra la necesidad de una función destructora y constructora. Crea una sencilla clase de cadena, llamada **strtype,** que contiene una cadena y su longitud. Cuando se crea un objeto **strtype,** se asigna memoria para que guarde la cadena y su longitud inicial se establece en cero. Cuando se destruye un objeto **strtype,** se libera esa memoria.

```
#include <iostream.h>
#include <malloc.h>
#include <string.h>
#include <stdlib.h>

define SIZE 255

class strtype {
  char *p;
  int len;
public:
  strtype(); // constructor
  ~strtype(); //destructor
  void set(char *ptr);
  void show();
};

// Inicializar un objeto cadena.
strtype::strtype()
```

```
{
  p = (char *) malloc(SIZE);
  if(!p) {
  cout << "Error de asignación\n";
  exit(1);
  }
  *p = '\0';
  len = 0;
}

// Libera memoria cuando se destruye un objeto cadena.
strtype::~strtype()
{
  cout << "Liberando p\n";
  free(p);
}

void strtype::set(char *ptr)
{
  if(strlen(p) > SIZE) {
  cout << "Cadena demasiado grande\n";
  return;
  }
  strcpy(p, ptr);
  len = strlen(p);
}

void strtype::show()
{
  cout << p << " - longitud: " << len;
  cout << "\n";
}

main()
{
  strtype s1, s2;

  s1.set("Esto es una prueba");
  s2.set("Me gusta C++");

  s1.show();
  s2.show();

  return 0;
}
```

Nota *Este programa usa* **malloc( )** *y* **free( )** *para asignar y liberar memoria. Aunque esto es perfectamente válido, C++ proporciona otro método para gestionar la memoria dinámicamente, como veremos más adelante en este libro.*

3. Aquí tenemos una forma interesante de usar un constructor y destructor de objetos. Este programa utiliza un objeto de la clase **timer** para medir el intervalo entre el

momento en que se crea un objeto de tipo **timer** y el momento en que se destruye. Cuando se llama a la función destructora del objeto se muestra el tiempo transcurrido. Se podría usar un objeto como éste para medir la duración de un programa o la cantidad de tiempo que dura una función dentro de un bloque. Sólo debe asegurarse de que el objeto se sale de ámbito en el punto en el que se quiere finalizar el intervalo de tiempo.

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

class timer {
  clock_t start;
public:
  timer(); // constructor
  ~timer(); // destructor
};

timer::timer()
{
  start = clock();
}

timer::~timer()
{
  clock_t end;

  end = clock();
  cout << "Tiempo transcurrido: " << (end-start) / CLK_TCK << "\n";
}

main()
{
  timer ob;
  char c;

  // retardo ...
  cout << "Pulse una tecla seguida de INTRO: ";
  cin >> c;

  return 0;
}
```

Este programa usa la función de biblioteca estándar **clock( )**, que devuelve el número de ciclos de reloj que se han producido desde que el programa comenzó a ejecutarse. Al dividir este valor por **CLK_TCK** se convierte este valor a segundos.

EJERCICIOS

1. Vuelva a trabajar sobre la clase **queue** que desarrolló como ejercicio en el Capítulo 1 para reemplazar su función de inicialización por un constructor.

2. Cree una clase llamada **stopwatch** que simule un cronógrafo que sigue el rastro del tiempo transcurrido. Utilice un constructor para que al principio inicialice a cero el tiempo transcurrido. Proporcione dos funciones miembro llamadas **start( )** y **stop( )** que activen y desactiven el temporizador, respectivamente. Incluya una función miembro llamada **show( )** que muestre el tiempo transcurrido. Además, haga que la función destructora muestre automáticamente el tiempo transcurrido cuando se destruye un objeto **stopwatch.** (Para simplificar, muestre el tiempo en segundos.)
3. ¿Qué está mal en el constructor que se muestra en el siguiente fragmento?

```
class sample {
  double a, b, c;
public:
  double sample(); // error, ¿por qué?
};
```

2.2. CONSTRUCTORES CON PARAMETROS

Es posible pasar argumentos a una función constructora. Para permitir esto, simplemente añada los parámetros adecuados a la declaración y definición de la función constructora. Después, cuando declare un objeto, especifique los parámetros como argumentos. Para ver cómo se realiza esto, comencemos con el breve ejemplo mostrado aquí:

```
#include <iostream.h>

class myclass {
  int a;
public:
  myclass(int x); // constructor
  void show();
};

myclass::myclass(int x)
{
  cout << "En el constructor\n";
  a = x;
}
void myclass::show()
{
  cout << a << "\n";
}

main()
{
  myclass ob(4);

ob.show();

return 0;
}
```

Aquí, el constructor de **myclass** toma un parámetro. El valor que se le pasa a **myclass( )** se usa para inicializar **a**. Preste especial atención a cómo se declara **ob** en **main( )**. El valor 4, especificado en los paréntesis que siguen a **ob** es el argumento que se le pasa al parámetro de **x** de **myclass( )**, que se usa para inicializar **a**.

En realidad, la sintaxis para pasar un argumento a un constructor con parámetros es una abreviatura de esta forma más larga:

```
myclass ob = myclass(4);
```

Sin embargo, la mayoría de los programadores de C++ utilizan la forma corta, igual que debe hacer usted.

Nota *A diferencia de las funciones constructoras, las funciones destructoras no pueden tener parámetros. La razón es bastante sencilla de entender: no hay ningún mecanismo mediante el que se puedan pasar argumentos a un objeto que se ha destruido.*

EJEMPLOS

1. Es posible —y de hecho bastante común— pasar un constructor en vez de un argumento. En este ejemplo a **myclass( )** se le pasan dos argumentos:

```
#include <iostream.h>

class myclass {
  int a, b;
public:
  myclass(int x, int y); // constructor
  void show();
};

myclass::myclass(int x, int y)
{
  cout << "En el constructor\n";
  a = x;
  b = y;
}

void myclass::show()
{
  cout << a << ' ' << b << "\n";
}

main()
{
  myclass ob(4, 7);

  ob.show();

  return 0;
}
```

Aquí, a **x** se le pasa el valor 4 y a **y** el valor 7. Este mismo enfoque general se utiliza para pasar cualquier número de argumentos que se desee (hasta el límite que tenga el compilador, por supuesto).

2. Aquí se muestra otra versión de la clase **stack,** que usa un constructor con parámetros para pasar un «nombre» a una pila. Este nombre de un solo carácter se utiliza para identificar a qué pila nos referimos cuando se produce un error.

```
#include <iostream.h>

define SIZE 10

// Declara una clase pila de caracteres
class stack {
  char stck[SIZE]; // guarda la pila
  int tos; // índice de la cabeza de la pila
  char who; // identifica la pila
public:
  stack(char c); // constructor
  void push(char ch); // mete carácter en la pila
  char pop(); // saca carácter de la pila
};

// Inicializa la pila
stack::stack(char c)
{
  tos = 0;
  who = c;
  cout << "Construyendo la pila " << who << "\n";
}

// Mete un carácter.
void stack::push(char ch)
{
  if(tos==SIZE) {
    cout << "La pila " << who << " está llena\n";
    return;
  }
  stck[tos] = ch;
  tos++;
}

// Saca un carácter.
char stack::pop()
{
  if(tos==0) {
    cout << "La pila " << who << " está vacía\n";
    return 0; // devuelve nulo cuando la pila está vacía
  }
  tos--;
  return stck[tos];
}

main()
```

```
{
  // crea dos pilas que se inicializan automáticamente
  stack s1('A'), s2('B');
  int i;

  s1.push('a');
  s2.push('x');
  s1.push('b');
  s2.push('y');
  s1.push('c');
  s2.push('z');

  // Esto generará algunos mensajes de error

  for(i=0; i<5; i++) cout << "Saca de s1: " << s1.pop() << "\n";
  for(i=0; i<5; i++) cout << "Saca de s2: " << s2.pop() << "\n";

  return 0;
}
```

El hecho de dar un «nombre» a los objetos, como se muestra en este ejemplo, es especialmente útil durante la depuración, cuando es importante saber qué objeto genera un error.

3. A continuación se muestra una forma diferente de implementar la clase **strtype** (desarrollada anteriormente) que utiliza una función constructora con parámetros:

```
#include <iostream.h>
#include <malloc.h>
#include <string.h>
#include <stdlib.h>

class strtype {
  char *p;
  int len;
public:
  strtype(char *ptr);
  ~strtype();
  void show();
};

strtype::strtype(char *ptr)
{
  len = strlen(ptr);
  p = (char *) malloc(len+1);
  if(!p) {
    cout << "Error de asignación\n";
    exit(1);
  }
  strcpy(p, ptr);
}

strtype::~strtype()
{
  cout << "Liberando p\n";
  free(p);
}

void strtype::show()
```

```
{
  cout << p << " - longitud: " << len;
  cout << "\n";
}

main()
{
  strtype s1("Esto es una prueba"), s2("Me gusta C++");

  s1.show();
  s2.show();
  return 0;
}
```

En esta versión de **strtype,** a la cadena se le asigna un valor inicial usando la función constructora.

4. Aunque los ejemplos previos han usado constantes, a un constructor de objetos se le puede pasar cualquier expresión válida, incluyendo variables. Por ejemplo, este programa utiliza la entrada del usuario para construir un objeto:

```
#include <iostream.h>

class myclass {
  int i, j;
public:
  myclass(int a, int b);
  void show();
};

myclass::myclass(int a, int b)
{
  i = a;
  j = b;
}

void myclass::show()
{
  cout << i << ' ' << j << "\n";
}

main()
{
  int x, y;

  cout << "Introduzca dos enteros: ";
  cin >> x >> y;

  // usa las variables para construir ob
  myclass ob(x, y);

  ob.show();

  return 0;
}
```

Este programa ilustra un punto importante sobre objetos. Se pueden construir cuando se necesiten para ajustarse a la situación exacta en el momento de su creación. A medida que aprenda más sobre C++, verá lo útil que es construir objetos «sobre la marcha».

EJERCICIOS

1. Modifique la clase **stack** para que asigne memoria dinámicamente para la pila. Haga que el tamaño de la pila lo especifique un parámetro de la función constructora. (No olvide liberar esta memoria con una función destructora.)
2. Cree una clase llamada **t_and_d** que cuando se cree se le pase la hora y fecha actual del sistema como un parámetro a su constructor. Haga que la clase incluya una función miembro que muestre esta hora y fecha en la pantalla. (Pista: Utilice las funciones estándar de hora y fecha de la biblioteca estándar para encontrar y mostrar la fecha.)
3. Cree una clase llamada **box,** cuya función constructora recibe tres valores **double,** que representen la longitud de los lados del cuadro. Haga que la clase **box** calcule el volumen del cubo y guarde el resultado en una variable **double.** Incluya una función miembro llamada **vol( )** que muestre el volumen de cada objeto **box.**

2.3. INTRODUCCION A LA HERENCIA

Aunque la herencia se trata más en detalle en el Capítulo 7, es necesario presentarla en este momento. Por lo que respecta y se aplica a C++, la herencia es un mecanismo por el cual una clase puede heredar las propiedades de otra. La herencia permite construir una jerarquía de clases, partiendo de la más general a la más específica.

Para empezar, es necesario definir dos términos normalmente usados al tratar la herencia. Cuando una clase hereda otra, la clase que se hereda se llama la *clase base.* La clase que hereda se llama *clase derivada.* En general, el proceso de herencia comienza con la definición de una clase base. La clase base define todas las cualidades que serán comunes a cualquier clase derivada. En esencia, la clase base representa la descripción más general de un conjunto de rasgos. Una clase derivada hereda esos rasgos generales y añade aquellas propiedades que son específicas a esa clase.

Para entender cómo una clase puede heredar otra, comencemos primero con un ejemplo que, aunque simple, muestra muchas características clave de la herencia.

Para empezar, aquí tenemos la declaración para la clase base:

```
// Define la clase base.
class B {
  int i;
public:
  void set_i(int n);
  int get_i();
};
```

Utilizando esta clase base, aquí mostramos una clase derivada que la hereda:

```
// Define la clase derivada.
class D : public B {
  int j;
public:
  void set_j(int n);
  int mul();
};
```

Observe atentamente esta declaración. Fíjese que tras el nombre de clase **D,** hay dos puntos seguidos por la palabra clave **public** y el nombre de clase **B.** Esto le dice al compilador que la clase **D** heredará todos los componentes de la clase **B.** La palabra clave **public** le indica al compilador que **B** se heredará, así como que todos los elementos públicos de la clase base serán elementos públicos de la clase derivada. Sin embargo, todos los elementos privados de la clase base permanecen privados para ella y no son accesibles directamente por la clase derivada.

Aquí se muestra un programa completo que usa las clases **B** y **D:**

```
// Un sencillo ejemplo de herencia.
#include <iostream.h>

// Define la clase base.
class base {
  int i;
public:
  void set_i(int n);
  int get_i();
};

// Define la clase derivada.
class derived : public base {
  int j;
public:
  void set_j(int n);
  int mul();
};

// Establece el valor de i en la base.
void base::set_i(int n)
{
  i = n;
}

// Devuelve el valor de i en la base.
int base::get_i()
{
  return i;
}
```

```
// Establece el valor j en la derivada.
void derived::set_j(int n)
{
  j = n;
}

// Devuelve el valor de i de la base por el j de la derivada.
int derived::mul()
{
  // la clase derivada puede llamar a funciones miembro públicas
  de la clase base
  return j * get_i();
}

main()
{
  derived ob;

  ob.set_i(10); // carga i en la base
  ob.set_j(4); // carga j en la derivada

  cout << ob.mul(); // muestra 40

  return 0;
}
```

Observe la definición de **mul( ).** Fíjese que se llama a **get_i( ),** que es un miembro de la clase base **B,** no de **D,** sin enlazarla a ningún objeto específico. Esto es posible porque los miembros públicos de **B** se vuelven miembros públicos de **D.** Sin embargo, la razón por la que **mul( )** debe llamar a **get_i( )** en vez de acceder a **i** directamente es que el miembro privado de una clase base (en este caso, **i**) permanece privado a ella, y no es accesible por ninguna clase derivada. La razón por la que los miembros públicos de una clase no tienen acceso a clases derivadas es para mantener la encapsulación. Si los miembros privados de una clase pudieran hacerse públicos simplemente heredando la clase, la encapsulación se podría evitar fácilmente.

Aquí se muestra la forma general usada para heredar una clase base:

```
class nombre-clase-derivada : especificador-acceso nombre-clase-base {
  .
  .
  .
};
```

Aquí, el *especificador-acceso* es una de la tres palabras clave siguientes: **public, private** o **protected.** Por ahora sólo usaremos **public** cuando heredemos una clase. Posteriormente en este libro se dará una descripción completa de los especificadores de acceso.

EJEMPLO

1. Aquí se muestra un programa que define una clase base genérica llamada **fruit** que describe ciertas características de la fruta. Esta clase la heredan dos clases derivadas llamadas **Apple** y **Orange.** Estas clases aportan información específica a **fruit** que está relacionada con estos tipos de fruta.

```
// Un ejemplo de herencia de clase.
#include <iostream.h>
#include <string.h>

enum yn {no, yes};
enum color {red, yellow, green, orange};

void out(enum yn x);

char *c[] = {
  "red", "yellow", "green", "orange"};

// Clase fruit genérica.
class fruit {
// en esta clase base todos los elementos son públicos
public:
  enum yn annual;
  enum yn perennial;
  enum yn tree;
  enum yn tropical;
  enum color clr;
  char name[40];
};

// Clase Apple derivada.
class Apple : public fruit {
  enum yn cooking;
  enum yn crunchy;
  enum yn eating;
public:
  void seta(char *n, enum color c, enum yn ck, enum yn crchy,
            enum yn e);
  void show();
};

// Clase Orange derivada.
class Orange : public fruit {
  enum yn juice;
  enum yn sour;
  enum yn eating;
public:
  void seto(char *n, enum color c, enum yn j, enum yn sr,
            enum yn e);
  void show();
};
```

```
void Apple::seta(char *n, enum color c, enum yn ck,
                 enum yn crchy, enum yn e)
{
  strcpy(name, n);
  annual = no;
  perennial = yes;
  tree = yes;
  tropical = no;
  clr = c;
  cooking = ck;
  crunchy = crchy;
  eating = e;
}

void Orange::seto(char *n, enum color c, enum yn j,
                  enum yn sr, enum yn e)
{
  strcpy(name, n);
  annual = no;
  perennial = yes;
  tree = yes;
  tropical = yes;
  clr = c;
  juice = j;
  sour = sr;
  eating = e;
}

void Apple::show()
{
  cout << name << " la manzana es: " << "\n";
  cout << "Anual: "; out(annual);
  cout << "Perenne: "; out(perennial);
  cout << "Arbol: "; out(tree);
  cout << "Tropical: "; out(tropical);
  cout << "Color: " << c[clr] << "\n";
  cout << "Buena para cocinar: "; out(cooking);
  cout << "Crujiente: "; out(crunchy);
  cout << "Buena para comer: "; out(eating);
  cout << "\n";
}

void Orange::show()
{
  cout << name << " la naranja es: " << "\n";
  cout << "Anual: "; out(annual);
  cout << "Perenne: "; out(perennial);
  cout << "Arbol: "; out(tree);
  cout << "Tropical: "; out(tropical);
  cout << "Color: " << c[clr] << "\n";
  cout << "Buena para zumo: "; out(juice);
  cout << "Agria: "; out(sour);
  cout << "Buena para comer: "; out(eating);
  cout << "\n";
}

void out(enum yn x)
```

```
{
  if(x==no) cout << "no\n";
  else cout << "si\n";
}

main()
{
  Apple a1, a2;
  Orange o1, o2;

  a1.seta("Delicia Roja", red, no, yes, yes);
  a2.seta("Jonathan", red, yes, no, yes);

  o1.seto("Navel", orange, no, no, yes);
  o2.seto("Valencia", orange, yes, yes, no);

  a1.show();
  a2.show();

  o1.show();
  o2.show();

  return 0;
}
```

Como puede ver, la clase base **fruit** define varias características que son comunes a todos los tipos de fruta. (Por supuesto que para que este ejemplo sea lo suficientemente corto como para ajustarse convenientemente en un libro, la clase **fruit** se ha simplificado.) Por ejemplo, todas las frutas crecen en árboles de hoja caduca o perenne. Todas las frutas crecen en árboles u otro tipo de plantas, viñas o arbustos. Todas las frutas tienen un color y un nombre. Esta clase base la heredan las clases **Apple** y **Orange.** Cada una de estas clases aporta información específica a su tipo de fruta.

Este ejemplo ilustra la razón básica de la herencia. Aquí, se crea una clase base que define los rasgos generales asociados con *todas* las frutas. Se deja que las clases derivadas aporten los rasgos que sean específicos a cada caso *individual.*

Este programa ilustra otro hecho importante sobre la herencia: una clase base no la «posee» exclusivamente una sola clase derivada. Una clase base la puede heredar cualquier número de clases.

EJERCICIO

1. Dada la siguiente clase base:

```
class area_cl {
public:
  double height;
  double width;
};
```

Cree dos clases derivadas llamadas **box** e **isósceles** que hereden **area_cl.** Haga que cada clase incluya una función llamada **area( )** que devuelva el área de un cuadro o un triángulo isósceles, respectivamente. Use constructores con parámetros para inicializar **height** y **width.**

2.4. PUNTEROS A OBJETO

Hasta ahora se ha estado accediendo a miembros de un objeto usando el operador punto. Este es el método correcto cuando se está trabajando con un objeto. Sin embargo, también es posible acceder a un miembro de un objeto a través de un puntero a ese objeto. Cuando sea este el caso, se emplea el operador de flecha (→) en vez del operador punto. (Es exactamente el mismo modo como se usa el operador flecha cuando se usa un puntero a una estructura.)

Un puntero a objeto se declara exactamente igual que se declara un puntero a cualquier otro tipo de variable. Se especifica su nombre de clase y luego se precede el nombre de variable con un asterisco. Para obtener la dirección de un objeto, se precede al objeto con el operador **&**, igual que cuando se toma la dirección de cualquier otro tipo de variable.

De igual forma que los punteros a otros tipos, cuando se incrementa un puntero a objeto, apuntará al siguiente objeto de su tipo.

EJEMPLO

1. Aquí tenemos un sencillo ejemplo que usa un puntero a objeto:

```
#include <iostream.h>

class myclass {
  int a;
public:
  myclass(int x); // constructor
  int get();
};

myclass::myclass(int x)
{
  a = x;
}

int myclass::get()
{
  return a;
}

main()
{
  myclass ob(120); // crea el objeto
  myclass *p; // crea un puntero al objeto
```

```
  p = &ob; // pone la dirección de ob en p

  cout << "Valor utilizando el objeto: " << ob.get();
  cout << "\n";

  cout << "Valor utilizando el puntero: " << p->get();

  return 0;
}
```

Observe cómo la declaración:

```
myclass *p;
```

crea un puntero a un objeto de **myclass( )**. Es importante entender que la creación de un puntero a objeto *no* crea un objeto —sólo crea un puntero a un objeto. La dirección de **ob** se pone en **p** utilizando esta sentencia:

```
p = &ob;
```

Finalmente el programa muestra cómo se puede referenciar un objeto mediante el uso de un puntero a él.

Volveremos al tema de punteros a objeto en el Capítulo 4, una vez que sepamos más sobre C++. Hay varias características especiales relacionadas con ello.

2.5. LAS CLASES, ESTRUCTURAS Y UNIONES ESTAN RELACIONADAS

Como ya se ha visto, la clase es sintácticamente similar a la estructura. Sin embargo, puede sorprenderle el hecho de que la clase y la estructura tienen prácticamente idénticas capacidades. En C++, la definición de una estructura se ha ampliado para que pueda también incluir funciones miembro, incluyendo funciones constructoras y destructoras, de la misma manera que puede hacerlo una clase. De hecho, la única diferencia entre una estructura y una clase es que, por omisión, los miembros de una clase son privados y los miembros de una estructura son públicos. Aquí se muestra la sintaxis ampliada de una estructura:

```
struct nombre-etiqueta {
   // función y miembros públicos
private:
   // función y miembros privados
} lista-objetos;
```

De hecho, según la sintaxis normal de C++, tanto **struct** como **class** crean nuevos *tipos* de clase. Observe que se presenta una nueva palabra clave. Es **private** y le indica al compilador que los miembros que le siguen son privados para esa clase.

Por encima, parece una redundancia el hecho de que las estructuras y clases

tengan las mismas competencias. Muchos recién llegados a C++ se preguntan por qué existe esta aparente duplicación. De hecho, no es raro escuchar el comentario de que la palabra clave **class** es innecesaria.

La respuesta a esta línea de razonamiento tiene una forma «fuerte» y una «débil». La razón «fuerte» se debe a guardar una compatibilidad ascendente desde C. Tal y como está definido actualmente C++, una estructura estándar de C también es perfectamente aceptable en un programa de C++. Puesto que en C todos los miembros de estructuras son públicos por omisión, este convenio también se guarda en C++. Además, puesto que **class** es una entidad sintácticamente separada de **struct,** la definición de una clase es libre de evolucionar de una forma que puede no ser compatible con una definición de estructura al estilo de C. Puesto que las dos son distintas, la futura dirección de C++ no está restringida por términos de compatibilidad.

La razón «débil» de tener dos construcciones similares es que no hay desventaja en ampliar la definición de una estructura en C++ para incluir funciones miembro.

Aunque las estructuras tienen las mismas capacidades que las clases, muchos programadores restringen su uso de las estructuras para ceñirse a su forma de estilo de C y no las usan para incluir funciones miembro. Muchos programadores usan la palabra clave **class** cuando definen objetos que tienen tanto datos como código. Sin embargo, esto es un asunto de estilo y está sujeto a una cuestión de preferencias. (A partir de esta sección, este libro reserva el uso de **struct** para objetos que no tienen funciones miembro.)

Si encuentra interesante la conexión entre clases y estructuras, también lo será la siguiente revelación sobre C++: ¡las uniones y las clases también están relacionadas! En C++, una unión también define un tipo de clase que puede contener como miembros tanto funciones como datos. Una unión es como una estructura en que , por omisión, todos los miembros son públicos hasta que se usa el especificador **private.** Lo único que hay en una unión es que todos los miembros comparten la misma posición de memoria (igual que en C). Las uniones pueden contener funciones constructoras y destructoras. Afortunadamente, las uniones de C son cada vez más compatibles con las uniones de C++.

Aunque estructuras y clases parecen a primera vista ser redundantes, este no es el caso de las uniones. En un lenguaje orientado a objetos, es importante preservar la encapsulación. Así, la capacidad de las uniones para enlazar código y datos permite crear tipos de clases en los que todos los datos usan una posición compartida. Esto es algo que no se puede hacer usando una clase.

En lo que respecta a C++ hay varias restricciones que aplicar a las uniones. Primero, no pueden heredar ninguna otra clase y no se pueden usar como clases base por ningún otro tipo. Las uniones no deben tener ningún miembro **static.** Además no pueden contener ningún objeto que tenga un constructor o un destructor. (Aunque la unión, en sí, puede tener un constructor y destructor.)

Nota *Algunos compiladores de C++ no permiten que las uniones tengan miembros privados, por lo que para una mejor compatibilidad, sería mejor evitar el uso de private dentro de una unión.*

EJEMPLOS

1. Aquí se muestra un breve programa que usa **struct** para crear una clase:

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


// usa struct para definir un tipo de clase
struct st_type {
  st_type(double b, char *n);
  void show();
private:
  double balance;
  char name[40];
} ;


st_type::st_type(double b, char *n)
{
  balance = b;
  strcpy(name, n);
}


void st_type::show()
{
  cout << "Nombre: " << name;
  cout << ": $" << balance;
  if(balance<0.0) cout << "***";
  cout << "\n";
}


main()
{
  st_type acc1(100.12, "Johnson");
  st_type acc2(-12.34, "Hedricks");


  acc1.show();
  acc2.show();
  return 0;
}
```

Observe que, como se ha expuesto, los miembros de una estructura son públicos por omisión. La palabra clave **private** se debe usar para declarar miembros privados.

Además, observe una diferencia entre la estructuras al estilo de C y estructuras al estilo de C++. En C++, el nombre-etiqueta de la estructura se convierte en un nombre de tipo completo que se puede usar para declarar objetos. En C, el nombre-etiqueta está precedido por la palabra clave **struct** para que se convierta en un tipo completo.

Aquí se muestra el mismo programa usando una clase:

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

class cl_type {
  double balance;
  char name[40];
public:
  cl_type(double b, char *n);
  void show();
} ;

cl_type::cl_type(double b, char *n)
{
  balance = b;
  strcpy(name, n);
}

void cl_type::show()
{
  cout << "Nombre: " << name;
  cout << ": $" << balance;
  if(balance<0.0) cout << "***";
  cout << "\n";
}

main()
{
  cl_type acc1(100.12, "Johnson");
  cl_type acc2(-12.34, "Hedricks");

acc1.show();
  acc2.show();

  return 0;
}
```

2. A continuación se muestra un ejemplo que utiliza una unión para mostrar el patrón de bits binario, byte a byte, contenido dentro de un valor **double.**

```
#include <iostream.h>

union bits {
  bits(double n);
  void show_bits();
  double d;
  unsigned char c[sizeof(double)];
};

bits::bits(double n)
{
  d = n;
}
```

```
void bits::show_bits()
{
  int i, j;

  for(j = sizeof(double)-1; j>=0; j--) {
    cout << "Patrón de bits en el byte " << j << ": ";
    for(i = 128; i; i >>= 1)
      if(i & c[j]) cout << "1";
      else cout << "0";
    cout << "\n";
  }
}

main()
{
  bits ob(1991.829);

  ob.show_bits();

  return 0;
}
```

La salida de este programa es

```
Patrón de bits en el byte 7: 01000000
Patrón de bits en el byte 6: 10011111
Patrón de bits en el byte 5: 00011111
Patrón de bits en el byte 4: 01010000
Patrón de bits en el byte 3: 11100101
Patrón de bits en el byte 2: 01100000
Patrón de bits en el byte 1: 01000001
Patrón de bits en el byte 0: 10001001
```

3. Tanto las estructuras como las uniones pueden tener constructores y destructores. El siguiente ejemplo muestra la clase **strtype** repetida pero como una estructura. Contiene una función constructora y otra destructora.

```
#include <iostream.h>
#include <malloc.h>
#include <string.h>
#include <stdlib.h>

struct strtype {
  strtype(char *ptr);
  ~strtype();
  void show();
private:
  char *p;
  int len;
};

strtype::strtype(char *ptr)
```

```
{
  len = strlen(ptr);
  p = (char *) malloc(len+1);
  if(!p) {
    cout << "Error de asignación\n";
    exit(1);
  }
  strcpy(p, ptr);
}

strtype::~strtype()
{
  cout << "Liberando p\n";
  free(p);
}

void strtype::show()
{
  cout << p << " - longitud: " << len;
  cout << "\n";
}

main()
{
  strtype s1("Esto es una prueba"), s2("Me gusta C++");

  s1.show();
  s2.show();

  return 0;
}
```

EJERCICIOS

1. Vuelva a escribir la clase **stack** presentada en la Sección 2.1 para que utilice una estructura en vez de una clase.
2. Utilice una clase **union** para intercambiar los bytes de mayor y menor orden de un entero (suponga enteros de 16-bits; si su computadora usa enteros de 32 bits, intercambie los bits de un **short int**).

2.6. FUNCIONES INSERTADAS

Antes de continuar este análisis de las clases, es necesaria una disgresión breve pero relacionada con el tema. En C++, es posible definir funciones a las que no se llama realmente, pero se insertan en el código en el momento de cada llamada. Es casi igual que la forma en que trabaja una macro con parámetros al estilo de C. La ventaja de las funciones *insertadas* es que no tienen nada asociado con el mecanismo de llamada y vuelta de la función. Esto significa que

las funciones insertadas se pueden ejecutar más rápidamente que las funciones normales. (Recuerde que las instrucciones de máquina que generan la llamada y vuelta de la función tardan tiempo cada vez que se llama a una función. Si hay parámetros, lleva incluso más tiempo.)

La desventaja de las funciones insertadas es que si son demasiado largas y se las llama demasiado a menudo, el programa aumentará su longitud. Por esta razón, sólo unas pocas funciones se declaran como funciones insertadas.

Para declarar una función insertada, simplemente hay que preceder la definición de la función con el especificador **inline.** Por ejemplo, este breve programa muestra cómo declarar una función insertada:

```
// Ejemplo de una función insertada
#include <iostream.h>

inline int even(int x)
{
  return !(x%2);
}

main()
{
  if(even(10)) cout << "10 es par\n";
  if(even(11)) cout << "11 es par\n";

  return 0;
}
```

En este ejemplo, la función **even(),** que devuelve verdadero si el argumento es par, se declara como insertada. Esto significa que la línea:

```
if(even(10)) cout << "10 es par\n";
```

es funcionalmente equivalente a:

```
if(!(10%2)) cout << "10 es par\n";
```

Este ejemplo también señala otra importante característica del uso de **inline:** una función insertada se tiene que definir *antes de* llamarla. Si no es así, el compilador no tiene forma de saber que tiene que insertarla. Esto es por lo que **even( )** se define antes que **main( ).**

La ventaja de usar **inline** en vez de macros con parámetros es doble. Primero, proporciona una forma más estructurada de ampliar pequeñas funciones insertadas. Por ejemplo, cuando se crea un macro con parámetros es fácil olvidar que se necesita un paréntesis extra para asegurar una correcta expansión insertada en cada caso. El uso de funciones insertadas previene de muchos problemas.

En segundo lugar, una función insertada debe poder optimizarse más profundamente por el compilador que una expansión de macro. En cualquier caso, los programadores de C++ prácticamente nunca usan macros con parámetros,

ya que en vez de eso se basan en el **inline** para evitar la complicación de una llamada a función asociada con una función corta.

Es importante entender que el especificador **inline** es una *solicitud*, no una orden para el compilador. Si, por varias razones, el compilador no es capaz de cumplir la petición, la función se compila como una función normal y la solicitud de **inline** se ignora.

Dependiendo del compilador, se pueden aplicar varias restricciones a una función insertada. Por ejemplo, algunos compiladores no insertarán una función si ésta contiene una variable **static,** una sentencia de bucle, un **switch** o un **goto,** o si la función es recursiva. Compruebe esto en el manual de usuario del compilador para ver las restricciones específicas a las funciones insertadas que puedan afectarle.

Recuerde *Si se viola cualquier restricción de inline, el compilador es libre de generar una función normal.*

EJEMPLOS

1. Cualquier tipo de función se puede insertar, incluyendo funciones que son miembros de clases. Por ejemplo, aquí se ha insertado la función miembro **divisible( )** para una ejecución más rápida. (La función devuelve verdadero si su primer argumento es divisible por su segundo argumento.)

```
// Demuestra una función miembro insertada.
#include <iostream.h>

class samp {
  int i, j;
public:
  samp(int a, int b);
  int divisible(); // insertada en su definición
};

samp::samp(int a, int b)
{
  i = a;
  j = b;
}

/* Devuelve 1 si i es divisible entre j.
  Esta función miembro está insertada.
*/
inline int samp::divisible()
{
  return !(i%j);
}

main()
```

```
{
  samp ob1(10, 2), ob2(10, 3);

  // esto es verdadero
  if(ob1.divisible()) cout << "10 es divisible por 2\n";

  // esto es falso
  if(ob2.divisible()) cout << "10 es divisible por 3\n";

  return 0;
}
```

2. Es perfectamente permisible insertar una función sobrecargada. Por ejemplo, este programa sobrecarga **min( )** de tres formas. Cada forma se declara también como **inline.**

```
#include <iostream.h>

// Sobrecarga min() de tres formas.

// enteros
inline int min(int a, int b)
{
  return a<b ? a : b;
}

// enteros largos
inline long min(long a, long b)
{
  return a<b ? a : b;
}

// doubles
inline double min(double a, double b)
{
  return a<b ? a : b;
}

main()
{
  cout << min(-10, 10) << "\n";
  cout << min(-10.01, 100.002) << "\n";
  cout << min(-10L, 12L) << "\n";

  return 0;
}
```

EJERCICIOS

1. En el Capítulo 1 se sobrecargó la función **abs( )**, para que pudiera encontrar el valor absoluto de enteros, enteros largos y **doubles.** Modifique ese programa, para que esas funciones se inserten.

2. ¿Por qué el compilador puede no insertar la siguiente función?

```
void f1()
{
  int i;

for(i=0; i<10; i++) cout << i;
}
```

2.7. INSERCION AUTOMATICA

Si la definición de una función miembro es suficientemente corta, su definición se puede incluir dentro de la declaración de clase. Hacer esto provoca que la función se convierta automáticamente en una función insertada, si es posible. Cuando una función se define dentro de una declaración de clase, la palabra clave **inline** no es necesaria. (Sin embargo, no es un error usarla en esta situación.) Por ejemplo, la función **divisible( )** de la sección anterior se puede insertar automáticamente como se muestra aquí.

```
#include <iostream.h>

class samp {
  int i, j;
public:
  samp(int a, int b);

  /* divisible se define aquí y se inserta
     automáticamente. */
  int divisible() { return !(i%j); }
};

samp::samp(int a, int b)
{
  i = a;
  j = b;
}

main()
{
  samp ob1(10, 2), ob2(10, 3);

  // esto es verdadero
  if(ob1.divisible()) cout << "10 es divisible por 2\n";

  // esto es falso
  if(ob2.divisible()) cout << "10 es divisible por 3\n";

  return 0;
}
```

Como se puede ver, el código asociado con **divisible( )** se da dentro de la declaración de la clase **samp.** Observe además que no se necesita o no se permite ninguna otra definición de **divisible( ).** Definir **divisible( )** dentro de **samp** provoca que se convierta en una función insertada automáticamente.

Cuando una función definida dentro de una declaración de clase no se puede convertir en una función insertada (porque se ha violado alguna restricción) se convierte automáticamente en una función normal.

Observe cómo **divisible( )** está definida dentro de **samp,** particularmente el cuerpo. Todo ocurre en una línea. Este formato es muy común en programas de C++ cuando una función se declara dentro de una declaración de clase. Esto permite que la declaración sea más compacta. Sin embargo, la clase **samp** se podría haber escrito de la siguiente manera:

```
class samp {
  int i, j;
public:
  samp(int a, int b);

  /* divisible se define aquí y se inserta
     automáticamente. */
  int divisible()
  {
  return !(i%j);
  }
};
```

En esta versión, la estructura de **divisible( )** usa más o menos el estilo de sangrado estándar. Desde el punto de vista del compilador, no hay diferencia entre el estilo compacto y el estilo estándar. Sin embargo, el estilo compacto se encuentra normalmente en programas de C++ cuando las funciones cortas se definen dentro de una definición de clase.

Las mismas restricciones que se aplican a las funciones insertadas «normales» se aplican a las funciones insertadas automáticamente dentro de una declaración de clase.

EJEMPLOS

1. Quizás el uso más común de funciones insertadas definidas dentro de una clase es definir funciones constructoras o destructoras. Por ejemplo, la clase **samp** se puede definir de forma más eficiente así:

```
#include <iostream.h>

class samp {
  int i, j;
public:
  // constructor insertado
  samp(int a, int b) { i = a; j = b; }
  int divisible() { return !(i%j); }
};
```

La definición de **samp( )** dentro de la clase **samp** es suficiente, y no se necesita o se permite ninguna otra definición de **samp( )**.

2. Algunas veces se puede incluir una función corta en una declaración de clase incluso aunque la característica de inserción automática sea de poco o ningún valor. Considere esta declaración de clase:

```
class myclass {
  int i;
public:
  myclass(int n) { i = n; }
  void show() { cout << i; }
};
```

Aquí la función **show( )** se convierte en una función insertada automáticamente. Sin embargo, como debe saber, las operaciones de E/S son (generalmente) muy lentas si las comparamos con las operaciones de memoria/UCP en las que se pierde cualquier sobrecarga por eliminar la llamada a la función. Incluso en programas de C++ es todavía común ver pequeñas funciones de este tipo declaradas dentro de una clase, simplemente por cuestión de conveniencia y porque no hace ningún daño.

EJERCICIOS

1. Convierta la clase **stack** de la Sección 2.1, Ejemplo 1, para que use funciones insertadas automáticamente cuando sea apropiado.
2. Convierta la clase **strtype** de la Sección 2.2, Ejemplo 3, para que use funciones insertadas automáticamente.

COMPROBACION DE APTITUD SUPERIOR

En este punto debería ser capaz de realizar los siguientes ejercicios y de responder las preguntas.

1. ¿Qué es un constructor? ¿Qué es un destructor? ¿Cuándo se ejecutan?
2. Cree una clase llamada **line** que dibuje una línea en la pantalla. Guarde la longitud de la línea en una variable entera privada llamada **len.** Haga que el constructor de **line** tome un parámetro: la longitud de la línea. Haga que el constructor almacene la longitud y dibuje la línea. Si el sistema no soporta gráficos, muestre la línea usando el carácter *. Opcional: Proporcione a **line** un destructor que borre la línea.
3. ¿Qué muestra el siguiente programa?

```
#include <iostream.h>

main()
{
  int i = 10;
  long l = 1000000;
  double d = -0.0009;

  cout << i << ' ' << l << ' ' << d;
  cout << "\n";

  return 0;
}
```

4. Añada otra clase derivada que herede el **area_cl** de la Sección 3, Ejercicio 1. Llame a esta clase **cylinder** y haga que calcule el área de su superficie. Pista: El área de la superficie de un cilindro es 2 * *pi* * R^2 + *pi* * *D* * *altura.*
5. ¿Qué es una función insertada? ¿Cuáles son sus ventajas y desventajas?
6. Modifique el siguiente programa para que todas las funciones miembro se inserten automáticamente:

```
#include <iostream.h>

class myclass {
  int i, j;
public:
  myclass(int x, int y);
  void show();
};

myclass::myclass(int x, int y)
{
  i = x;
  j = y;
}
void myclass::show()
{
  cout << i << " " << j << "\n";
}

main()
{
  myclass count(2, 3);

  count.show();

  return 0;
}
```

7. ¿Cuál es la diferencia entre una clase y una estructura?

COMPROBACION DE APTITUD INTEGRADA

Esta sección comprueba cómo ha asimilado el contenido de este capítulo con el del capítulo anterior.

1. Cree una clase llamada **prompt.** Pase a su función constructora una cadena de petición de su elección. Haga que el constructor muestre la cadena y después introduzca un entero. Almacene este valor en una variable privada llamada **count.** Cuando se destruya un objeto de tipo **prompt,** haga que suene la alarma de la computadora tantas veces como el usuario indicó.
2. En el Capítulo 1 se creó un programa que convertía pies en pulgadas. Ahora, cree una clase que haga lo mismo. Haga que la clase almacene el número de pies y su equivalente en número de pulgadas. Pasar al constructor de clase el número de pies y haga que el constructor muestre el número de pulgadas.
3. Cree una clase llamada **dice** que contenga una variable entera privada. Cree una función llamada **roll( )** que use el generador de números aleatorios estándar llamado **rand( )** para generar un número entre el 1 y el 6. Después haga que **roll( )** muestre ese valor.

3 *Profundización en las clases*

OBJETIVOS DEL CAPITULO

En este capítulo se continuará el estudio de la clase. Aprenderá cómo asignar objetos, pasar objetos a funciones y devolver objetos desde las funciones. Además aprenderá un nuevo e importante tipo de función: la función amiga.

COMPROBACION DE APTITUD

Antes de continuar, debe ser capaz de responder correctamente a las siguientes preguntas y realizar los ejercicios.

1. Dada la siguiente clase, ¿cuáles son los nombres de sus funciones constructora y destructora?

```
class widgit {
  int x, y;
public:
  // ... se rellena con funciones constructoras y destructoras
};
```

2. ¿Cuándo se llama a una función constructora? ¿Cuándo se llama a una función destructora?
3. Dada la siguiente clase base, muestre cómo la puede heredar una clase derivada llamada **Mars.**

```
class planet {
  int moons;
  double dist_from_sun;
  double diameter;
  double mass;
public:
  // ...
};
```

4. Hay dos maneras de hacer que una función se inserte en el código. ¿Cuáles son?
5. Indique dos posibles restricciones a las funciones insertadas.
6. Dada la siguiente clase, muestre cómo se declara un objeto llamado **ob** que pasa el valor 100 a **a** y *X* a **c**.

```
class sample {
  int a;
  char c;
public:
  sample(int x, char ch) { a = x; c = ch; }
  // ...
};
```

3.1. ASIGNACION DE OBJETOS

Un objeto se puede asignar a otro a condición de que ambos objetos sean del mismo tipo. Por omisión, cuando un objeto se asigna a otro, se hace una copia a nivel de bits de todos los miembros. Por ejemplo, cuando se asigna un objeto llamado **A** a otro objeto llamado **B,** los contenidos de todos los datos de **A** se copian en los miembros equivalentes de **B**. Considere este corto ejemplo:

```
// Un ejemplo de asignación de objetos.
#include <iostream.h>

class myclass {
  int a, b;
public:
  void set(int i, int j) { a = i; b = j; }
  void show() { cout << a << ' ' << b << "\n"; }
};

main()
{
  myclass o1, o2;

  o1.set(10, 4);

  // asigna o1 a o2
  o2 = o1;

  o1.show();
  o2.show();

  return 0;
}
```

Aquí, el objeto **o1** tiene sus variables miembro **a** y **b** fijadas con los valores 10 y 4 respectivamente. A continuación, **o1** se asigna a **o2.** Esto hace que el valor actual de **o1.a** se asigne a **o2.a** y **o1.b** se asigne a **o2.b.** De esta manera, cuando se ejecuta este programa muestra

```
10 4
10 4
```

Recuerde que una asignación entre dos objetos simplemente hace que los datos de esos objetos sean idénticos. Los dos objetos están completamente separados. Por ejemplo, después de la asignación, la llamada a **o1.set( )** para establecer el valor de **o1.a** no tiene efecto en **o2** o en su valor **a.**

EJEMPLOS

1. Sólo se pueden usar objetos del mismo tipo en una sentencia de asignación. Si los objetos no son del mismo tipo, se informa de un error en tiempo de compilación. Además, no es suficiente que los tipos sean físicamente similares —sus nombres de tipo deben ser iguales. Por ejemplo, este no es un programa válido:

```
// Este programa tiene un error.
#include <iostream.h>

class myclass {
  int a, b;
public:
  void set(int i, int j) { a = i; b = j; }
  void show() { cout << a << ' ' << b << "\n"; }
};

/* Esta clase es similar a myclass, pero usa un nombre
   de clase diferente y por tanto parece un tipo distinto
   para el compilador.
*/
class yourclass {
  int a, b;
public:
  void set(int i, int j) { a = i; b = j; }
  void show() { cout << a << ' ' << b << "\n"; }
};

main()
{
  myclass o1;
  yourclass o2;

  o1.set(10, 4);

  o2 = o1; // ERROR, objetos de diferente tipo

  o1.show();
  o2.show();

  return 0;
}
```

   En este caso, incluso aunque **myclass** y **yourclass** son físicamente lo mismo, debido a que tienen diferentes nombres tipo, el compilador los trata como tipos diferentes.

2. Es importante entender que todos los miembros de un objeto se asignan a otro cuando se realiza la asignación. Esto incluye datos complejos como arrays. Por ejemplo, en la siguiente versión del ejemplo **stack,** sólo **s1** tiene caracteres introducidos. Sin embargo, debido a la asignación, el array **stck** de **s2** también contendrá los caracteres **a, b** y **c.**

```
#include <iostream.h>

define SIZE 10

// Declara una clase pila de caracteres
class stack {
  char stck[SIZE]; // guarda la pila
  int tos; // índice de la cabeza de la pila
public:
  stack(); // constructor
  void push(char ch); // mete carácter en la pila
  char pop(); // saca carácter de la pila
};

// Inicializa la pila
stack::stack()
{
  cout << "Construyendo una pila\n";
  tos = 0;
}

// Mete un carácter.
void stack::push(char ch)
{
  if(tos==SIZE) {
    cout << "La pila está llena";
    return;
  }
  stck[tos] = ch;
  tos++;
}

// Saca un carácter.
char stack::pop()
{
  if(tos==0) {
    cout << "La pila está vacía";
    return 0; // devuelve nulo cuando la pila está vacía
  }
  tos--;
  return stck[tos];
}

main()
{
  // crea dos pilas que se inicializan automáticamente
  stack s1, s2;
  int i;

  s1.push('a');
  s1.push('b');
  s1.push('c');

  // copia de s1
  s2 = s1; // ahora s1 y s2 son idénticos


  for(i=0; i<3; i++) cout << "Saca de s1: " << s1.pop() << "\n";
  for(i=0; i<3; i++) cout << "Saca de s2: " << s2.pop() << "\n";
  return 0;
}
```

3. Debe tener cuidado al asignar un objeto a otro. Por ejemplo, aquí tenemos la clase **strtype** desarrollada en el Capítulo 2, junto con un corto **main( )**. Trate de encontrar un error en este programa.

```
// Este programa contiene un error.

#include <iostream.h>
#include <malloc.h>
#include <string.h>
#include <stdlib.h>

class strtype {
  char *p;
  int len;
public:
  strtype(char *ptr);
  ~strtype();
  void show();
};

strtype::strtype(char *ptr)
{
  len = strlen(ptr);
  p = (char *) malloc(len+1);
  if(!p) {
    cout << "Error de asignación\n";
    exit(1);
  }
  strcpy(p, ptr);
}

strtype::~strtype()
{
  cout << "Liberando p\n";
  free(p);
}

void strtype::show()
{
  cout << p << " - longitud: " << len;
  cout << "\n";
}

main()
{
  strtype s1("Esto es una prueba"), s2("Me gusta C++");

  s1.show();
  s2.show();

  // asigna s1 a s2 - - esto genera un error
  s2 = s1;

  s1.show();
  s2.show();

  return 0;
}
```

El problema con este programa es bastante insidioso. Cuando se crean **s1** y **s2,** ambos asignan memoria para guardar sus respectivas cadenas. En **p** se almacena un puntero a la memoria asignada a cada objeto. Cuando se destruye un objeto **strtype,** esta memoria se borra. Sin embargo, cuando **s1** se asigna a **s2,** el **p** de **s2** ahora apunta a la misma memoria que el **p** de **s1.** De esta manera, cuando estos objetos se destruyen, la memoria apuntada por el **p** de **s1** se libera *dos veces* y la memoria originalmente apuntada por el **p** de **s2** no se libera *en absoluto.*

Mientras sea favorable en este contexto, este tipo de problema si se produce en un programa real hará que falle el sistema de asignación dinámica y posiblemente incluso provoque un fallo en el programa. Como se puede ver en los ejemplos precedentes, al asignar un objeto a otro debemos asegurarnos de no destruir información que pueda necesitarse posteriormente.

EJERCICIOS

1. ¿Qué está mal en el siguiente fragmento?

```
// Este programa tiene un error.
#include <iostream.h>

class cl1 {
  int i, j;
public:
  cl1(int a, int b) { i = a; j = b; }
  // ...
};

class cl2 {
  int i, j;
public:
  cl2(int a, int b) { i = a; j = b; }
  // ...
};

main()
{
  cl1 x(10, 20);
  cl2 y(0, 0);
  x = y;

  // ...
}
```

2. Usando la clase **queue** que se creó en el Capítulo 2, Sección 1, Ejercicio 1, muestre cómo se puede asignar una cola a otra.
3. Si la clase **queue** de la pregunta anterior asigna dinámicamente memoria para guardar la cola, ¿por qué en esta situación puede que una cola no se pueda asignar a otra?

3.2. PASO DE OBJETOS A FUNCIONES

Los objetos se pueden pasar a funciones como argumentos de la misma manera que se pasan otros tipos de datos. Simplemente hay que declarar el parámetro como un tipo de clase y después usar un objeto de esa clase como un argumento cuando se llama a la función. Igual que con otro tipo de datos, por omisión todos los objetos se pasan por valor a una función.

EJEMPLOS

1. Aquí tenemos un corto ejemplo en el que se pasa un objeto a una función:

```
#include <iostream.h>

class samp {
  int i;
public:
  samp(int n) { i = n; }
  int get_i() { return i; }
};

// Devuelve el cuadrado de o.i.
int sqr_it(samp o)
{
  return o.get_i() * o.get_i();
}

main()
{
  samp a(10), b(2);

  cout << sqr_it(a) << "\n";
  cout << sqr_it(b) << "\n";

  return 0;
}
```

Este programa crea una clase llamada **samp** que contiene una variable entera llamada **i.** La función **sqr_it( )** toma un argumento de tipo **samp** y devuelve el cuadrado del valor **i** del objeto. La salida de este programa es 100 seguido de 4.

2. Como se ha visto, el método predeterminado de pasar parámetros en C++, incluyendo objetos, es por valor. Esto significa que se hace una copia idéntica del argumento y esta es la copia usada por la función. Por lo tanto los cambios del objeto dentro de la función no afectan al objeto que llama. Esto se prueba en el siguiente ejemplo:

```
/*
   Recuerde, los objetos, como otros parámetros, se pasan
   por valor. Así los cambios en el parámetro dentro de
   la función no tienen efecto en el objeto usado en
   la llamada.
*/
#include <iostream.h>

class samp {
  int i;
public:
  samp(int n) { i = n; }
  void set_i(int n) { i = n; }
  int get_i() { return i; }
};

/* Establece o.i a su cuadrado. Sin embargo, esto
   no tiene efecto en el objeto usado para llamar a sqr_it().
*/
void sqr_it(samp o)
{
   o.set_i(o.get_i() * o.get_i());

  cout << "La copia de a tiene un valor i de " << o.get_i();
  cout << "\n";
}

main()
{
  samp a(10);

  sqr_it(a); // a pasado por valor

  cout << "Pero, a.i no se cambia en main: ";
  cout << a.get_i(); // muestra 10

  return 0;
}
```

La salida de este programa es

```
La copia de a tiene un valor i de 100
Pero, a.i no se cambia en main: 10
```

3. Como en otro tipo de variables, la dirección de un objeto se puede pasar a una función, por lo que el argumento usado en la llamada lo puede modificar la función. Por ejemplo, la siguiente versión del programa del ejemplo precedente modifica el valor del objeto cuya dirección se usa en la llamada a **sqr_it( ).**

```
/*
   Ahora que se pasa la dirección de un objeto
   a sqr_it(), la función puede modificar el valor
   del argumento cuya dirección se utiliza en la llamada
*/
```

```
#include <iostream.h>

class samp {
  int i;
public:
  samp(int n) { i = n; }
  void set_i(int n) { i = n; }
  int get_i() { return i; }
};

/* Establece o.i a su cuadrado. Esto afecta al argumento
   de llamada.
*/
void sqr_it(samp *o)
{
  o->set_i(o->get_i() * o->get_i());

  cout << "La copia de a tiene un valor i de " << o->get_i();
  cout << "\n";
}

main()
{
  samp a(10);

  sqr_it(&a); // pasa la dirección de a a sqr_it()

  cout << "Ahora, el a de main() se ha modificado: ";
  cout << a.get_i(); // muestra 100

  return 0;
}
```

Este programa muestra ahora la siguiente salida:

```
La copia de a tiene un valor i de 100
Ahora, el a de main() se ha modificado: 100
```

4. Cuando se hace una copia de un objeto cuando se pasa a una función, significa que un nuevo objeto comienza a existir. También, cuando termina la función a la que se le pasó el objeto, la copia del argumento se destruye. Esto sugiere dos preguntas. Primero, ¿se llama al constructor del objeto cuando se hace la copia? Segundo, ¿se llama al destructor del objeto cuando se destruye la copia? La respuesta puede parecer sorprendente en principio.

   Cuando se hace una copia de un objeto para usarla en una llamada a función, *no* se llama la función constructora. La razón es sencilla de entender si se piensa en ello. Puesto que una función constructora se usa generalmente para inicializar algunos aspectos de un objeto, no se tiene que llamar cuando se hace una copia de un objeto ya existente pasado a una función. Hacerlo podría alterar los contenidos del objeto. Cuando se pasa un objeto a una función, se quiere el estado actual del objeto, no su estado inicial.

   Sin embargo, cuando la función finaliza y la copia se destruye, *se llama* a la función destructora. Esto es porque el objeto puede realizar alguna operación que

no se debe hacer cuando sale fuera de ámbito. Por ejemplo, la copia puede asignar memoria que se tiene que liberar.

Para resumir, cuando se crea una copia de un objeto porque se usa como argumento para una función, no se llama a la función constructora.

Sin embargo, cuando la copia se destruye (normalmente al salir del ámbito cuando se vuelve de la función), se llama a la función destructora.

El siguiente programa demuestra el estudio precedente:

```
#include <iostream.h>

class samp {
  int i;
public:
  samp(int n) {
  i = n;
  cout << "Construyendo\n";
  }
  ~samp() { cout << "Destruyendo\n"; }
  int get_i() { return i; }
};

// Devuelve el cuadrado de o.i.
int sqr_it(samp o)
{
  return o.get_i() * o.get_i();
}

main()
{
  samp a(10);

  cout << sqr_it(a) << "\n";

  return 0;
}
```

Esta función muestra lo siguiente:

```
Construyendo
Destruyendo
100
Destruyendo
```

Como puede ver, sólo se hace una llamada al constructor. Esto sucede cuando se crea **a.** Sin embargo, se hacen dos llamadas al destructor. Una es para la copia creada cuando se pasa **a** a **sqr_it( ).** La otra es para **a.**

El hecho de que el destructor del objeto que es la copia del argumento usado para llamar a una función se ejecute cuando finaliza la función puede ser una fuente de problemas. Por ejemplo, si el objeto usado como argumento asigna memoria dinámica y libera esa memoria cuando se destruye, entonces su copia liberará la misma memoria cuando se llama a su destructor. Esto dejará el objeto original dañado y sin uso efectivo. (Véase como ejemplo el Ejercicio 2, un poco antes en

esta sección.) Es importante protejerse de este tipo de error y estar seguro de que la función destructora de la copia de un objeto usada como argumento no provoca efectos laterales que modifiquen el argumento original.

Como puede suponer, una forma de evitar el problema de una función destructora del parámetro que destruye los datos requeridos por el argumento de la llamada es pasar la dirección del objeto y no el objeto en sí mismo. Cuando se pasa una dirección, no se crea un nuevo objeto, y por lo tanto, no se llama a ningún destructor cuando se vuelve de la función. (Como se verá en el siguiente capítulo, C++ proporciona una variación de este tema que ofrece una alternativa muy elegante.) Sin embargo, existe una solución incluso mejor, que se puede usar cuando se haya aprendido un tipo especial de constructor llamado *constructor de copia*. Un constructor de copia permite definir de forma precisa cómo se hacen las copias de objetos. (Los constructores de copia se tratan en el Capítulo 5.)

EJERCICIOS

1. Usando el ejemplo **stack** de la Sección 3.1, Ejemplo 2, añada una función llamada **showstack( )** a la que se le pasa un objeto de tipo **stack.** Haga que la función muestre los contenidos de una pila.
2. Como ya sabe, cuando un objeto se pasa a una función, se hace una copia de ese objeto. Además, cuando se vuelve de esa función, se llama a la función destructora de la copia. Teniendo esto en cuenta ¿qué está mal en el siguiente programa?

```
// Este programa tiene un error.
#include <iostream.h>
#include <stdlib.h>

class dyna {
  int *p;
public:
  dyna(int i);
  ~dyna() { free(p); cout << "liberando \n"; }
  int get() { return *p; }
};

dyna::dyna(int i)
{
  p = (int *) malloc(sizeof(int));
  if(!p) {
    cout << "Error de asignación\n";
    exit(1);
  }

  *p = i;
}

// Devuelve el valor negativo de *ob.p
int neg(dyna ob)
{
  return -ob.get();
}
```

```
main()
{
  dyna o(-10);

  cout << o.get() << "\n";
  cout << neg(o) << "\n";

  dyna o2(20);
  cout << o2.get() << "\n";
  cout << neg(o2) << "\n";

  cout << o.get() << "\n";
  cout << neg(o) << "\n";

  return 0;
}
```

3.3. OBJETOS DEVUELTOS POR FUNCIONES

Igual que se pueden pasar objetos a funciones, las funciones pueden devolver objetos. Para hacerlo, primero hay que declarar la función para que devuelva un tipo de clase. Segundo, hay que devolver un objeto de ese tipo usando la sentencia normal **return.**

Sin embargo, hay un punto importante que entender sobre los objetos devueltos por la funciones: Cuando un objeto es devuelto por una función, se crea automáticamente un objeto temporal que guarda el valor devuelto. Este es el objeto que realmente devuelve la función. Después de devolver el valor, este objeto se destruye. La destrucción de este objeto temporal puede causar efectos laterales inesperados en algunas situaciones, como se muestra posteriormente en el Ejemplo 2.

EJEMPLOS

1. Aquí se muestra un ejemplo de una función que devuelve un objeto:

```
// Devuelve un objeto
#include <iostream.h>
#include <string.h>

class samp {
  char s[80];
public:
  void show() { cout << s << "\n"; }
  void set(char *str) { strcpy(s, str); }
};

// Devuelve un objeto de tipo samp
samp input()
```

```
{
  char s[80];
  samp str;

  cout << "Introduzca una cadena: ";
  cin >> s;

  str.set(s);

  return str;
}

main()
{
  samp ob;

  // asigna el objeto devuelto a ob
  ob = input();
  ob.show();

  return 0;
}
```

En este ejemplo, **input( )** crea un objeto local llamado **str** y después lee una cadena desde el teclado. Esta cadena se copia en **str.s** y después la función devuelve **str.** Después este objeto se asigna a **ob** dentro de **main( )** cuando se devuelve en la llamada a **input( )**.

2. Debe tener cuidado al devolver objetos desde funciones si esos objetos contienen funciones destructoras, porque el objeto devuelto sale fuera de ámbito tan pronto como el valor se devuelve a la rutina de llamada. Por ejemplo, si el objeto devuelto por la función tiene un destructor que libera dinámicamente memoria asignada, esa memoria puede quedar liberada incluso aunque el objeto al que se asigna el valor devuelto lo siga utilizando. Por ejemplo, observe esta versión incorrecta del programa anterior:

```
// Error generado al devolver un objeto.
#include <iostream.h>
#include <string.h>
#include <stdlib.h>

class samp {
  char *s;
public:
  samp() { s = '\0'; }
  ~samp() { if(s) free(s); cout << "Liberando s\n"; }
  void show() { cout << s << "\n"; }
  void set(char *str);
};

// Carga una cadena.
void samp::set(char *str)
```

```
{
  s = (char *) malloc(strlen(str));
  if(!s) {
    cout << "Error de asignación\n";
    exit(1);
  }

  strcpy(s, str);
}

// Devuelve un objeto de tipo samp.
samp input()
{
  char s[80];
  samp str;

  cout << "Introduzca una cadena: ";
  cin >> s;

  str.set(s);
  return str;
}

main()
{
  samp ob;

  // asigna el objeto devuelto a ob
  ob = input(); // ¡¡¡¡Esto produce un error!!!!
  ob.show();

  return 0;
}
```

A continuación se muestra la salida de este programa:

```
Introduzca una cadena: Hola
Liberando s
Liberando s
Hola
Liberando s
Asignación de puntero nulo
```

Observe que se llama tres veces a la función destructora **samp.** Primero se llama cuando el objeto local str sale de ámbito al volver de **input( ).** La segunda vez se llama a **~samp( )** cuando se destruye el objeto temporal devuelto por **input( ).**

Conviene recordar que cuando se devuelve un objeto desde una función, se genera *automáticamente un objeto temporal invisible (para nosotros)* que guarda el valor devuelto. En este caso, este objeto es simplemente una copia de **str,** que es valor devuelto por la función. Por lo tanto, después de volver de la función, se ejecuta el destructor del objeto temporal. Finalmente, al destructor del objeto **ob,** dentro de **main( ),** se le llama cuando finaliza el programa.

El problema es que, en esta situación, la primera vez que se ejecuta el destructor, se libera la memoria asignada para guardar la cadena de entrada por **input( )**. Así, no sólo las otras dos llamadas al destructor de **samp** intentan liberar una parte de memoria dinámica ya liberada, sino que también destruyen el sistema de asignación dinámica en el proceso, como evidencia el mensaje de tiempo de ejecución «Asignación de puntero nulo». (Dependiendo del compilador, del modelo de memoria usado para compilar, y de otros aspectos, se puede ver o no ver este mensaje si se prueba este programa.)

El punto clave que hay que entender de este ejemplo es que cuando un objeto es devuelto desde una función, el objeto temporal usado para hacer efectiva la devolución habrá llamado a su función destructora. Por lo tanto, se deben evitar objetos devueltos en los que se dé esta situación. (Como se aprenderá en el Capítulo 5, es posible usar un constructor de copia para manejar esta situación.)

EJERCICIOS

1. Para ilustrar exactamente cuando se construye y destruye un objeto, cuando se devuelve desde una función, cree una clase llamada **who.** Haga que el constructor de **who** tome un argumento de carácter que se usará para identificar un objeto. Haga que el constructor muestre un mensaje similar a éste cuando construya el objeto:

```
Construyendo who #x
```

   donde **x** es el carácter identificativo asociado con cada objeto. Cuando un objeto se destruye, haga que se muestre un mensaje parecido a este:

```
Destruyendo who #x
```

   donde, de nuevo **x** es el carácter identificativo. Por último, cree una función llamada **make_who( )** que devuelva un objeto **who.** Dé a cada objeto un único nombre. Observe la salida del programa.
2. Aparte de la liberación incorrecta de la memoria asignada dinámicamente, piense en una situación en la que podría ser impropio devolver un objeto desde una función.

3.4. INTRODUCCION A LAS FUNCIONES AMIGAS

Habrá momentos en los que se quiera que una función tenga acceso a los miembros privados de una clase sin que esa función sea realmente un miembro de esa clase. De cara a esto, C++ soporta las funciones amigas. Una función amiga no es un miembro de una clase, pero todavía tiene acceso a sus elementos privados.

Dos motivos por los que las funciones amigas son útiles tienen que ver con la sobrecarga de operadores y la creación de ciertos tipos de funciones de E/S. Tendremos que esperar hasta más adelante para ver en acción estos usos de las funciones amigas. Sin embargo, una tercera razón para las funciones amigas es

que habrá momentos en los que una función tenga acceso a los miembros privados de dos o más clases diferentes. Este uso es el que se examina aquí.

Una función amiga se define como una función no miembro normal. Sin embargo, dentro de la declaración de clase para la que será una función amiga, está también incluido su prototipo, precedido por la palabra clave **friend.** Para entender cómo funciona, examine este breve programa:

```
// Un ejemplo de función amiga.
#include <iostream.h>

class myclass {
  int n, d;
public:
  myclass(int i, int j) { n = i; d = j; }
  // declara una función amiga de myclass
  friend int isfactor(myclass ob);
};

/* Aquí está la definición de la función amiga. Devuelve verdadero
   si d es un factor de n. Observe que en la definición de
   isfactor() no se utiliza la palabra clave friend.
*/
int isfactor(myclass ob)
{
  if(!(ob.n % ob.d)) return 1;
  else return 0;
}

main()
{
  myclass ob1(10, 2), ob2(13, 3);

  if(isfactor(ob1)) cout << "2 es un factor de 10\n";
  else cout << "2 no es un factor de 10\n";

  if(isfactor(ob2)) cout << "3 es un factor de 13\n";
  else cout << "3 no es un factor de 13\n";

  return 0;
}
```

En este ejemplo, **myclass** declara su función constructora y la función amiga **isfactor( )** dentro de su declaración de clase. Debido a que **isfactor( )** es una función amiga de **myclass, isfactor( )** tiene acceso a sus áreas privadas. Esto es por lo que dentro de **isfactor( ),** es posible referirse directamente a **ob.n** y **ob.d.**

Es importante entender que una función amiga no es un miembro de la clase de la que es amiga. Por lo tanto, no es posible llamar a una función amiga usando un nombre de objeto y un operador de acceso a miembro de clase (un punto o flecha). Por ejemplo, suponiendo el ejemplo anterior, esta sentencia está mal:

```
ob1.isfactor(); // error, isfactor no es una función miembro
```

En vez de ello, las funciones amigas se llaman igual que las funciones normales.

Aunque una función amiga tiene conocimiento de los elementos privados de la clase de la que es amiga, sólo puede acceder a ellos a través de un objeto de la clase. Es decir, a diferencia de un miembro de **myclass,** que se pueden referir directamente a **n** o **d**, una función amiga puede acceder a estas variables sólo en conjunción con un objeto que esté declarado dentro o pasado a la función amiga.

Nota *El párrafo anterior descubre una importante cuestión secundaria. Cuando una función miembro se refiere a un elemento privado, lo hace directamente porque una función miembro sólo se ejecuta en conjunción con un objeto de esa clase. De esta manera, cuando una función miembro se refiere a un elemento privado, el compilador sabe a qué objeto pertenece ese elemento privado por el objeto que está enlazado a la función cuando se llama a esa función miembro. Sin embargo, una función amiga no está unida a ningún objeto. Simplemente garantiza el acceso a los elementos privados de una clase. Así, dentro de la función amiga no tiene sentido referirse a un miembro privado sin referirse a un objeto específico.*

Debido a que las funciones amigas no son miembros de una clase, normalmente se le pasarán uno o más objetos de la clase para la que están definidas. Este es el caso de **isfactor( )**. Se le pasa un objeto de **myclass,** llamado **ob**. Sin embargo, debido a que **isfactor( )** es una función amiga de **myclass**, puede acceder a los elementos privados de **ob**. Si **isfactor( )** no fuese una función amiga de **myclass,** no sería posible acceder a **ob.d** o **ob.n** ya que **n** y **d** son miembros privados de **myclass.**

Nota *Una función amiga no es miembro y no se puede calificar mediante un nombre de objeto. Se tiene que llamar como una función normal.*

Una función amiga no se hereda. Es decir, cuando una clase base incluye una función amiga, la función amiga no es una función amiga de la clase derivada.

Otro punto importante sobre las funciones amigas es que una función amiga puede ser amiga de más de una clase.

EJEMPLOS

1. Un uso común (y bueno) de una función amiga se da cuando dos tipos diferentes de clases tienen alguna cantidad en común que hay que comparar. Por ejemplo, considere el siguiente programa, que crea una clase llamada **car** y una clase llamada **truck,** cada una conteniendo, como una variable privada, la velocidad del vehículo que representa:

```
#include <iostream.h>

class truck; // una referencia anticipada
```

```
class car {
  int passengers;
  int speed;
public:
  car(int p, int s) { passangers = p; speed = s; }
  friend int sp_greater(car c, truck t);
};

class truck {
  int weight;
  int speed;
public:
  truck(int w, int s) { weight = w, speed = s; }
  friend int sp_greater(car c, truck t);
};
/* Devuelve positivo si la velocidad de car es mayor que la de
truck.
  Devuelve 0 si las velocidades son las mismas.
  Devuelve negativo si la velocidad de truck es mayor
  que la de car.
*/
int sp_greater(car c, truck t)
{
  return c.speed-t.speed;
}

main()
{
  int t;
  car c1(6, 55), c2(2, 120);
  truck t1(10000, 55), t2(20000, 72);

cout << "Comparando c1 y t1:\n";
  t = sp_greater(c1, t1);
  if(t<0) cout << "El camión es más rápido.\n";
  else if(t==0) cout << "La velocidad del coche y del camión
es la misma.\n";
  else cout << "El coche es más rápido.\n";

cout << "\nComparando c2 y t2:\n";
  t = sp_greater(c2, t2);
  if(t<0) cout << "El camión es más rápido.\n";
  else if(t==0) cout << "La velocidad del coche y del camión
es la misma.\n";
  else cout << "El coche es más rápido.\n";

  return 0;
}
```

Este programa contiene la función **sp_greater( ),** que es una función amiga de las clases **car** y **truck.** (Como se ha visto, una función puede ser amiga de dos o más clases.) Esta función devuelve positivo si el objeto **car** es más rápido que el objeto **truck,** cero si sus velocidades son iguales y negativo si **truck** va más deprisa.

Este programa ilustra un importante elemento sintáctico de C++: la *referencia anticipada.* Ya que **sp_greater( )** toma parámetros de ambas clases **car** y **truck,**

lógicamente es imposible declarar ambas antes de incluir **sp_greater( )** en ellas. Por lo tanto, necesita alguna forma de advertirle al compilador sobre un nombre de clase sin declararlo en ese momento. A esto se le llama referencia anticipada. En C++, para decirle al compilador que un identificador es el nombre de una clase, se utiliza una línea como ésta antes de usar por primera vez el nombre de clase:

class nombre-clase;

Por ejemplo, en el programa anterior, la referencia anticipada es

```
class truck;
```

Ahora, **truck** se puede usar en la declaración de función amiga de **sp_greater( )** sin generar un error en tiempo de compilación.

2. Una función puede ser miembro de una clase y amiga de otra. Por ejemplo, aquí tenemos el ejemplo anterior reescrito para que **sp_greater( )** sea miembro de **car** y amiga de **truck:**

```
#include <iostream.h>

class truck; // una referencia anticipada

class car {
  int passengers;
  int speed;
public:
  car(int p, int s) { passengers = p; speed = s; }
  int sp_greater(truck t);
};

class truck {
  int weight;
  int speed;
public:
  truck(int w, int s) { weight = w, speed = s; }

  // observe el nuevo uso del operador de resolución de ámbito
  friend int car::sp_greater(truck t);
};

/* Devuelve positivo si la velocidad del coche es mayor que
   la del camión.
Devuelve 0 si la velocidad es la misma
Devuelve negativo si la velocidad del camión es mayor que
la del coche
*/
int car::sp_greater(truck t)
{
  /* Dado que sp_greater() es miembro de car, sólo
  se le tiene que pasar un objeto truck. */

  return speed-t.speed;
}
```

```
main()
{
  int t;
  car c1(6, 55), c2(2, 120);
  truck t1(10000, 55), t2(20000, 72);

  cout << "Comparando c1 y t1:\n";
  t = c1.sp_greater(t1); // evoca a la función miembro de car
  if(t<0) cout << "El camión es más rápido.\n";
  else if(t==0) cout << "La velocidad del coche y del camión
  es la misma.\n";
  else cout << "El coche es más rápido.\n";

  cout << "\nComparando c2 y t2:\n";
  t = c2.sp_greater(t2); //evoca a la función miembro de car
  if(t<0) cout << "El camión es más rápido.\n";
  else if(t==0) cout << "La velocidad del coche y del camión
  es la misma.\n";
  else cout << "El coche es más rápido.\n";

  return 0;
}
```

Observe el nuevo uso del operador de resolución de ámbito como sucede en la declaración de función amiga dentro de la declaración de clase **truck.** En este caso se usa para indicarle al compilador que la función **sp_greater( )** es un miembro de la clase **car.**

Una forma fácil de recordar cómo usar el operador de resolución de ámbito es que el nombre de clase seguido del operador de resolución de ámbito seguido del nombre del miembro, especifica completamente un miembro de clase.

De hecho, cuando se hace referencia a un miembro de una clase, nunca es incorrecto especificar completamente su nombre. Sin embargo, cuando un objeto se utiliza para llamar a una función miembro o acceder a una variable miembro, el nombre completo es redundante y casi no se usa. Por ejemplo,

```
t = c1.sp_greater(t1)
```

se puede escribir usando (redundante) el operador de resolución de ámbito y el nombre de clase **car:**

```
t = c1.car::sp_greater(t1)
```

De cualquier manera, dado que **c1** es un objeto de tipo **car,** el compilador ya *sabe que* **sp_greater( )** es un miembro de la clase **car,** haciendo innecesaria la especificación completa de la clase.

EJERCICIO

1. Imagine una situación en la que dos clases, llamadas **pr1** y **pr2,** mostradas aquí, comparten una impresora. Además, imagine que otras partes del programa necesitan saber cuando la impresora está siendo utilizada por un objeto cualquiera de estas dos clases. Cree una función llamada **inuse( )** que devuelva verdadero cuando la impresora esté utilizando cualquiera de ellos y falso en cualquier otro caso. Haga que esta función sea una función amiga de **pr1** y **pr2.**

```
class pr1 {
  int printing;
  // ...
public:
  pr1() { printing = 0; }
  void set_print(int status) { imprimiendo = estado; }
  // ...
};

class pr2 {
  int printing;
  // ...
public:
  pr2() { imprimiendo = 0; }
  void set_print(int status) { imprimiendo = estado; }
  // ...
};
```

COMPROBACION DE APTITUD SUPERIOR

Antes de continuar, debe ser capaz de responder las siguientes preguntas y realizar los ejercicios:

1. ¿Qué prerrequisito único se debe encontrar a la hora de asignar un objeto a otro?
2. Dado este fragmento de clase:

```
class samp {
  double *p;
public:
  samp(double d) {
  p = (double *) malloc(sizeof(double));
  if(!p) exit(1); // error de asignación
  *p = d;
  }
  ~samp() { free(p); }
  // ...
};

// ...
samp ob1(123.09), ob2(0.0);
// ...
ob2 = ob1;
```

   ¿qué problema produce la asignación de **ob1** a **ob2**?
3. Dada esta clase:

```
class planet {
  int moons;
  double dist_from_sun; // en millas
  double diameter;
  double mass;
public:
  //...
  double get_miles() { return dist_from_sun; }
};
```

Cree una función llamada **light( )** que tome como argumento un objeto de tipo **planet** y devuelva el número de segundos que tarda la luz del sol en llegar al planeta. (Asuma que la luz viaja a 186.000 millas por segundo y que **dist_from_sun** está especificado en millas.)

4. ¿Puede pasarse la dirección de un objeto a una función como argumento?
5. Usando la clase **stack,** escriba una función llamada **loadstack( )** que devuelva una pila que ya está cargada con las letras del alfabeto (a-z). Asigne esta pila a otro objeto en la rutina de llamada y compruebe que contiene el alfabeto. Asegúrese de cambiar el tamaño de la pila para que sea suficientemente grande para guardar el alfabeto.
6. Explique porqué hay que ser cuidadoso cuando se pasan objetos a una función o se devuelven objetos de una función.
7. ¿Qué es una función amiga?

COMPROBACION DE APTITUD INTEGRADA

Esta sección comprueba cómo ha asimilado el contenido de este capítulo con el de capítulos anteriores.

1. Las funciones se pueden sobrecargar mientras difiera el número o tipo de sus parámetros. Sobrecargue **loadstack( )** del Ejercicio 5 de la sección Comprobación de Aptitud Superior para que tome como parámetro un entero llamado **upper.** En la versión sobrecargada, si **upper** es 1, cargue la pila con el alfabeto en mayúsculas. En cualquier otro caso, cargue la pila con minúsculas.
2. Usando la clase **strtype** mostrada en la Sección 3.1, Ejemplo 3, añada una función amiga que tome como argumento un puntero a un objeto de tipo **strtype** y devuelva un puntero a la cadena apuntada por ese objeto. (Es decir, haga que la función devuelva **p**.) Llame a esta función **get_string( ).**
3. Experimento: Cuando un objeto de una clase derivada se asigna a otro objeto de la misma clase derivada, ¿ se copia también la información asociada con la clase base? Para saberlo, utilice las dos clases siguientes y escriba un programa que demuestre lo que ocurre.

```
class base {
  int a;
public:
  void load_a(int n) { a = n; }
  int get_a() { devuelve a; }
};

class derived : public base {
  int b;
public:
  void load_b(int n) { b = n; }
  int get_b() { return b; }
};
```

4 Arrays, punteros y referencias

OBJETIVOS DEL CAPITULO

Este capítulo examina diversos aspectos importantes sobre arrays de objetos y punteros a objetos. Finaliza con una discusión de una de las principales novedades de C++: la referencia. La referencia es decisiva para muchas de las características de C++, por lo que se aconseja una lectura cuidadosa.

COMPROBACION DE APTITUD

Antes de continuar, debería ser capaz de responder a estas preguntas y de realizar estos ejercicios.

1. Cuando un objeto se asigna a otro, ¿qué es lo que ocurre exactamente?
2. ¿Pueden presentarse problemas o efectos colaterales cuando se asigna un objeto a otro? (Dé un ejemplo.)
3. Cuando se pasa un objeto como argumento a una función, se realiza una copia del mismo. ¿Se llama a la función constructora de la copia? ¿Se llama al destructor?
4. Los objetos se pasan implícitamente a las funciones por valor, lo que significa que lo que le ocurra a la copia dentro de la función no va a afectar al argumento utilizado en la llamada. ¿Puede existir alguna excepción a este principio? Si lo hay, dé un ejemplo.
5. Dada la siguiente clase, cree una función llamada **make_sum( )** que devuelva un objeto del tipo **summation**. Esta función debe pedir un número al usuario y después construir un objeto con ese valor y devolverlo a la rutina invocadora. Demuestre que la función trabaja adecuadamente.

```
class summation {
  int num;
  long sum; // suma de num
public:
  void set_sum(int n);
  void show_sum() {
    cout << num << " la suma es " << sum << "\n";
  }
};

void summation::set_sum(int n)
{
  int i;
  num = n;

  sum = 0;
  for(i=1; i<=n; i++)
    sum += i;
}
```

6. En la pregunta anterior, no se definió la función **set_sum( )** dentro de la declaración de la clase **summation**. Dé una razón de por qué esto podría ser necesario para algunos compiladores.
7. Dada la siguiente clase, muestre cómo añadir una función amiga llamada **isneg( )**, que acepte un parámetro de tipo **myclass** y devuelva verdadero si **num** es negativo y falso en cualquier otro caso.

```
class myclass {
  int num;
public:
  myclass(int x) { num = x; }
};
```

8. ¿Puede una función ser amiga de más de una clase?

4.1. ARRAYS DE OBJETOS

Como ya ha sido comentado en varias ocasiones, los objetos son variables y tienen las mismas capacidades y atributos que cualquier otro tipo de variables. Por consiguiente, es perfectamente posible disponer objetos en un array. La sintaxis para declarar un array de objetos es exactamente la utilizada para declarar un array de cualquier tipo de variable. Aún más, el acceso a los arrays de objetos es igual al de los arrays de otros tipos de variables.

EJEMPLOS

1. A continuación se muestra un ejemplo de un array de objetos:

```
#include <iostream.h>

class samp {
  int a;
public:
  void set_a(int n) { a = n; }
  int get_a() { return a; }
};
main()
{
  samp ob[4];
  int i;

  for(i=0; i<4; i++) ob[i].set_a(i);

  for(i=0; i<4; i++) cout << ob[i].get_a( );

  cout << "\n";

  return 0;
}
```

Este programa crea un array con cuatro objetos del tipo **samp** y después almacena en cada elemento un valor entre 0 y 3. Nótese cómo las funciones miembro son llamados para cada elemento del array. El nombre del array, en este caso **ob**, se indexa; después se aplica el operador del miembro de acceso, seguido del nombre del miembro que se va a llamar.

2. Si un tipo de clase incluye un constructor, puede inicializarse un array de objetos. Aquí, por ejemplo, **ob** es un array inicializado:

```
// Inicialización de un array.
#include <iostream.h>

class samp {
  int a;
public:
  samp(int n) { a = n; }
  int get_a() { return a; }
};

main()
{
  samp ob[4] = { -1, -2, -3, -4 };
  int i;

  for(i=0; i<4; i++) cout << ob[i].get_a() << ' ';

  cout << "\n";

  return 0;
}
```

Este programa muestra en la pantalla «-1 -2 -3 -4». En este ejemplo, los valores que van desde -1 hasta -4 se pasan a la función constructora de **ob**.

En realidad, la sintaxis presentada en la lista de inicialización ha sido sustituida por esta forma extendida (vista por primera vez en el Capítulo 2):

```
samp ob[4] = { samp(-1), samp(- 2),
  samp(-3), samp(- 4) };
```

Sin embargo, cuando se inicializa una sola dimensión, la forma utilizada en el programa es la más normal (aunque, como se verá más adelante, esta forma operará sólo con arrays cuyos constructores tengan un único argumento).

3. También es posible crear arrays de objetos multidimensionales. Por ejemplo, a continuación se muestra un programa que crea e inicializa un array de objetos bidimensional:

```
// Creación de un array bidimensional de objetos.
#include <iostream.h>

class samp {
  int a;
public:
  samp(int n) { a = n; }
  int get_a() { return a; }
};

main()
```

```
{
  samp ob[4][2] = {
    1, 2,
    3, 4,
    5, 6,
    7, 8
  };
  int i;

  for(i=0; i<4; i++) {
    cout << ob[i][0].get_a() << ' ';
    cout << ob[i][1].get_a() << "\n";
  }

  cout << "\n";

  return 0;
}
```

Este programa muestra

```
1 2
3 4
5 6
7 8
```

4. Como ya es sabido, un constructor puede tener más de un argumento. Cuando se inicializa un array de objetos, cuyo constructor tiene más de un argumento, debe utilizarse la forma de inicialización alternativa mencionada anteriormente. Veámoslo con un ejemplo:

```
#include <iostream.h>

class samp {
  int a, b;
public:
  samp(int n, int m) { a = n; b = m; }
  int get_a() { return a; }
  int get_b() { return b; }
};

main()
{
  samp ob[4][2] = {
    samp(1, 2), samp(3, 4),
    samp(5, 6), samp(7, 8),
    samp(9, 10), samp(11, 12),
    samp(13, 14), samp(15, 16)
  };

  int i;

  for(i=0; i<4; i++) {
```

```
    cout << ob[i][0].get_a() << ' ';
    cout << ob[i][0].get_b() << "\n";
    cout << ob[i][1].get_a() << ' ';
    cout << ob[i][1].get_b() << "\n";
  }

  cout << "\n";

  return 0;
}
```

En este ejemplo, el constructor de **samp** tiene dos argumentos. El array **ob** se declara e inicializa en **main( )** utilizando llamadas directas al constructor de **samp**. Esto es necesario porque la sintaxis formal de C++ sólo permite, en una lista separada por comas, un argumento a la vez. No existe modo alguno, por ejemplo, de especificar dos o más argumentos por entrada en la lista.

Por tanto, cuando se inicializan arrays de objetos que tienen constructores de más de un argumento, debe utilizarse la sintaxis de inicialización de «forma extendida» en vez de la de «forma reducida».

Nota *Siempre es posible utilizar la forma extendida de inicialización incluso si el objeto sólo tiene un argumento. Sin embargo, para este caso es más conveniente la forma reducida.*

El programa anterior muestra:

```
1 2
3 4
5 6
7 8
9 10
11 12
13 14
15 16
```

EJERCICIOS

1. Utilizando la siguiente declaración de clase, cree un array de diez elementos e inicialice el elemento **ch** con valores desde la A hasta la J. Demuestre que el array contiene realmente estos valores.

```
#include <iostream.h>

class letters {
  char ch;
public:
  letters(char c) { ch = c; }
  char get_ch() { return ch; }
};
```

2. Utilizando la siguiente declaración de clase, cree un array con diez elementos, inicialice **num** con valores desde el 1 hasta el 10 e inicialice **sqr** con la raíz cuadrada de **num**.

```
#include <iostream.h>

class squares {
  int num, sqr;
public:
  squares(int a, int b) { num = a; sqr = b; }
  void show() {cout << num << ' ' << sqr << "\n"; }
};
```

3. Cambie la inicialización de **ob** del Ejercicio 1 para utilizar la forma extendida. (Esto es, llame explícitamente al constructor de **ob** de la lista de inicialización.)

4.2. USO DE PUNTEROS A OBJETOS

Como se discutió en el Capítulo 2, se puede acceder a los objetos mediante punteros. Como ya se sabe, cuando se utiliza un puntero a un objeto, las funciones miembro del objeto se referencian utilizando el operador flecha (→) en lugar del operador punto (.).

La aritmética de punteros a un objeto es igual a la de cualquier otro tipo de datos: está relacionada con el objeto. Por ejemplo, cuando se incrementa el puntero a un objeto, apunta al próximo objeto. Cuando se decrementa el puntero a un objeto, éste apunta al objeto anterior.

EJEMPLOS

1. A continuación se muestra un ejemplo de la aritmética de los punteros a objetos:

```
// Punteros a objetos.
#include <iostream.h>

class samp {
  int a, b;
public:
  samp(int n, int m) { a = n; b = m; }
  int get_a() { return a; }
  int get_b() { return b; }
};

main()
{
  samp ob[4] = {
    samp(1, 2),
    samp(3, 4),
    samp(5, 6),
```

```
    samp(7, 8)
  };
  int i;

  samp *p;
  p = ob; // obtención de la dirección de comienzo del array

  for(i=0; i<4; i++) {
    cout << p->get_a() << ' ';
    cout << p->get_b() << "\n";
    p++; // avance al próximo objeto
  }
  cout << "\n";

  return 0;
}
```

Este programa muestra:

```
1 2
3 4
5 6
7 8
```

Como puede verse por la salida presentada, cada vez que se incrementa **p**, éste apunta al próximo objeto del array.

EJERCICIOS

1. Plantee de nuevo el Ejemplo 1 de modo que presente el contenido del array **ob** en orden inverso.
2. Vuelva al Ejemplo 3 de la Sección 4.1, de manera que el acceso al array bidimensional se haga a través de un puntero. Truco: En C++, como en C, todos los arrays se almacenan contiguamente, de izquierda a derecha y de arriba a abajo.

4.3. EL PUNTERO this

C++ consta de un puntero especial denominado **this**. **this** es un puntero que se pasa automáticamente a cualquier miembro cuando se invoca. Es un puntero al objeto que genera la llamada. Por ejemplo, dada la siguiente sentencia:

```
ob.f1(); // se asume que ob es un objeto
```

la función **f1( )** recibe automáticamente un puntero a **ob** —que es el objeto que genera la llamada. Este puntero se referencia como **this**.

Es importante entender que sólo se pasa a los miembros punteros **this**. Por ejemplo, una función amiga no tiene un puntero **this**.

EJEMPLO

1. Como ya se ha visto, cuando un miembro referencia a otro miembro de una clase, lo hace sin limitar la referencia con la clase o con la especificación del objeto. Por ejemplo, examine este pequeño programa, que crea una sencilla clase inventario:

```
// Demostración del puntero this.
#include <iostream.h>
#include <string.h>

class inventory {
  char item[20];
  double cost;
  int on_hand;
public:
  inventory(char *i, double c, int o)
  {
    strcpy(item, i);
    cost = c;
    on_hand = o;
  }
  void show();
};

void inventory::show()
{
  cout << item;
  cout << ": $" << cost;
  cout << " Existencias: " << on_hand << "\n";
}

main()
{
  inventory ob("llave de tuerca", 4.95, 4);

  ob.show();

  return 0;
}
```

Como puede observarse, dentro del constructor **inventory( )** y del miembro **show( )**, los atributos **item**, **cost**, y **on-hand** son referenciados directamente. Esto se debe a que sólo puede llamarse a un miembro cuando éste está enlazado a un objeto. Por consiguiente, el compilador sabe cuáles son los datos del objeto que van a ser referenciados.

Sin embargo, hay una explicación más sutil. Cuando se llama a un miembro se pasa automáticamente un puntero **this** al objeto que provocó la llamada. Así pues, el programa anterior podría volver a escribirse como se muestra a continuación:

```
// Demostración del puntero this.
#include <iostream.h>
#include <string.h>
```

```
class inventory {
  char item[20];
  double cost;
  int on_hand;
public:
  inventory(char *i, double c, int o)
  {
    strcpy(this->item, i); // acceso a los atributos
    this->cost = c; // a través del puntero
    this->on_hand = o; // this
  }
  void show();
};

void inventory::show()
{
  cout << this->item; // uso de this para acceder a los atributos
  cout << ": $" << this->cost;
  cout << " Existencias: " << this->on_hand << "\n";
}
main()
{
  inventory ob("llave de tuerca", 4.95, 4);

  ob.show();

  return 0;
}
```

Aquí, el acceso a través del puntero **this** a las funciones miembro de **ob** es explícito. De este modo, en **show( )** estas dos sentencias son equivalentes:

```
cost = 123.23;
this->cost = 123.23;
```

De hecho, la primera forma es, hablando sencillamente, una abreviatura de la segunda.

Mientras que ningún programador en C++ utilizaría el puntero **this** para acceder al miembro de una clase, del modo que se ha mostrado anteriormente, porque la forma reducida es mucho más sencilla, es importante entender lo que conlleva la forma reducida.

El puntero **this** tiene varios usos, incluyendo la asistencia en la sobrecarga de operadores. En el Capítulo 6 se describirá con detalle este uso. Por ahora, el aspecto más importante sobre el que aprender es que todos los miembros se pasan implícitamente como un puntero al objeto que realizó la llamada.

EJERCICIO

1. Dado el siguiente programa, transforme todas las referencias a los miembros de una clase en referencias explícitas con el puntero **this**.

```
#include <iostream.h>

class myclass {
  int a, b;
public:
  myclass(int n, int m) { a = n; b = m; }
  int add() { return a+b; }
  void show();
};

void myclass::show()
{
  int t;

  t = add(); // llamada al miembro
  cout << t << "\n";
}

main()
{
  myclass ob(10, 14);

  ob.show();
  return 0;
}
```

4.4. USO DE new Y delete

Hasta ahora, cuando se necesitaba asignar memoria dinámica, se hacía uso de **malloc( )** y la memoria asignada se liberaba utilizando **free( )**. Además de estas funciones estándar que siguen siendo válidas en C++, C++ proporciona un método más conveniente y seguro para asignar y liberar memoria. En C++, puede asignarse memoria utilizando **new** y liberarse mediante **delete**. Estos operadores adoptan la siguiente forma general:

p-var = new type;
delete p-var;

En este caso, *type* es el especificador del tipo del objeto para el que se asigna memoria y *p-var* es un puntero a este tipo. **new** es un operador que devuelve un puntero a la memoria asignada dinámicamente, que debe ser lo suficientemente grande como para contener al objeto de tipo *type*. **delete** devuelve la memoria cuando ya no se necesita.

Al igual que **malloc( )**, si no hay suficiente memoria disponible para atender una petición de asignación, **new** devuelve un puntero nulo. Del mismo modo, debe llamarse a **delete** sólo con un puntero obtenido previamente mediante **new**. Si se llama a **delete** con un puntero incorrecto, se bloquea el sistema de asignación, interrumpiéndose posiblemente el programa.

Aunque **new** y **delete** realizan funciones similares a **malloc( )** y **free( )**, presentan varias ventajas añadidas. Primero, **new** asigna automáticamente la memoria suficiente para albergar un objeto del tipo especificado. No es necesario utilizar, por ejemplo, **sizeof**, para calcular el número de bytes requeridos. Esto reduce la posibilidad de que se produzcan errores. Segundo, **new** devuelve automáticamente un puntero del tipo especificado. No tiene por qué utilizarse un molde de tipo explícito como sucede cuando se asigna memoria utilizando **malloc( )** (veáse la nota siguiente). Tercero, tanto **new** como **delete** pueden sobrecargarse, permitiendo que el programador realice, a la medida, su propio sistema de asignación de memoria. Cuarto, es posible inicializar un objeto asignado dinámicamente. Por último, no se necesita incluir **malloc.h** (o **stdlib.h**) en los programas.

Nota *En C no se requiere ningún molde de tipo cuando se asigna el valor devuelto por* **malloc( )** *a un puntero, porque el* **void*** *devuelto por* **malloc( )** *se transforma de modo automático en un puntero compatible con el tipo del puntero del lado izquierdo de la asignación. Sin embargo, no sucede lo mismo en C++ que, cuando se utiliza* **malloc( )**, *necesita un molde de tipo explícito. La razón de esta diferencia es la de permitir que C++ realice una comprobación más estricta de los tipos devueltos por las funciones. No obstante, el operador* **new** *realiza el molde de tipo automáticamente.*

Una vez introducidos **new** y **delete**, éstos serán los operadores utilizados en lugar de **malloc( )** y **free( )**.

EJEMPLOS

1. Este primer breve ejemplo muestra un programa que asigna memoria a un entero:

```
// Un ejemplo sencillo de new y de delete.
#include <iostream.h>

main()
{
  int *p;

  p = new int; // asignación de espacio para un entero

  // asegúrese siempre de que la asignación se ha efectuado
  if(!p) {
    cout << "Error de asignación\n";
    return 1;
  }

  *p = 1000;

  cout << "El entero contenido en p es: " << *p << "\n";

  delete p; // se elimina la memoria

  return 0;
}
```

Obsérvese cómo se comprueba el valor devuelto por **new** antes de ser utilizado. No debe presuponerse jamás que el puntero devuelto por **new** es correcto.

2. A continuación se presenta un ejemplo que asigna dinámicamente un objeto:

```
// Asignación dinámica de objetos.
#include <iostream.h>

class samp {
  int i, j;
public:
  void set_ij(int a, int b) { i=a; j=b; }
  int get_product() { return i*j; }
};

main()
{
  samp *p;

  p = new samp; // asignación del objeto
  if(!p) {
    cout << "Error de asignación\n";
    return 1;
  }

  p->set_ij(4, 5);

  cout << "El producto es: " << p->get_product() << "\n";

  return 0;
}
```

EJERCICIOS

1. Escriba un programa que utilice **new** para asignar dinámicamente un **float**, un **long** y un **char**. Dé valores a estas variables dinámicas y muéstrelos. Por último, utilizando **delete**, libere toda la memoria asignada dinámicamente.
2. Cree una clase que contenga el nombre de una persona y su número de teléfono. Haciendo uso de **new**, asigne dinámicamente un objeto de esta clase y coloque dentro de los campos correspondientes del objeto su nombre y su número de teléfono.

4.5. *MAS SOBRE new Y delete*

Esta sección trata de dos características adicionales de **new** y **delete**. La primera es que los objetos asignados dinámicamente pueden recibir valores iniciales. La segunda es que pueden crearse arrays asignados dinámicamente.

Un objeto asignado dinámicamente puede tomar un valor inicial utilizando esta forma de la sentencia **new**:

p-var = new type (initial-value);

Para asignar dinámicamente un array unidimensional, debe utilizarse esta forma de **new**:

p-var = new type [size];

Después de ejecutarse esta sentencia, *p-var* apuntará a la primera posición de un array de *size* elementos del tipo especificado. Por diversas razones técnicas, no es posible inicializar un array que ha sido asignado dinámicamente.

Para eliminar un array asignado dinámicamente, debe usarse la siguiente forma de **delete**:

delete [ ] p-var;

Esta sintaxis da lugar a que el compilador llame a la función destructor para cada elemento del array. *p-var no* se libera en múltiples ocasiones. *p-var* sólo se libera una vez.

Nota *En compiladores antiguos, puede requerirse especificar el tamaño del array que se va a eliminar entre los corchetes de la sentencia* **delete**. *Esto es lo que ocurría en la definición original de C++. Sin embargo, ya no se necesita la especificación del tamaño.*

EJEMPLOS

1. Este programa asigna memoria a un entero e inicializa dicha memoria:

```
// Un ejemplo de inicialización de una variable dinámica.
#include <iostream.h>

main()
{
  int *p;

  p = new int (9); // se almacena el valor inicial 9

  if(!p) {
    cout << "Error de asignación\n";
    return 1;
  }

  cout << "El entero contenido en p es: " << *p << "\n";

  delete p; // se elimina la memoria

  return 0;
  }
```

Como puede verse, este programa muestra el valor 9, que es el valor dado inicialmente a la memoria apuntada por **p**.

2. El siguiente programa pasa valores iniciales a un objeto asignado dinámicamente:

```
// Asignación dinámica de objetos.
#include <iostream.h>

class samp {
  int i, j;
public:
  samp(int a, int b) { i=a; j=b; }
  int get_product() { return i*j; }
};

main()
{
  samp *p;

  p = new samp(6, 5); // asignación de objetos inicializados
  if(!p) {
    cout << "Error de asignación\n";
    return 1;
  }

  cout << "El producto es: " << p->get_product() << "\n";

  delete p;

  return 0;
}
```

Cuando se asigna el objeto **samp**, se llama a su constructor automáticamente y se le pasan los valores 6 y 5.

3. El siguiente programa asigna un array de enteros:

```
// Un ejemplo sencillo de new y delete.
#include <iostream.h>

main()
{
  int *p;

  p = new int [5]; // asignación de memoria para 5 enteros

  // asegúrese siempre de que se ha producido la asignación
  if(!p) {
    cout << "Error de asignación\n";
    return 1;
  }

  int i;

  for(i=0; i<5; i++) p[i] = i;
  for(i=0; i<5; i++) {
```

```
    cout << "Este es el entero contenido en p[" << i << "]: ";
    cout << p[i] << "\n";
  }

  delete [] p; // se elimina la memoria

  return 0;
}
```

Este programa muestra lo siguiente:

```
Este es el entero contenido en p[0]: 0
Este es el entero contenido en p[1]: 1
Este es el entero contenido en p[2]: 2
Este es el entero contenido en p[3]: 3
Este es el entero contenido en p[4]: 4
```

4. Este programa crea un array dinámico de objetos:

```
// Asignación dinámica de objetos.
#include <iostream.h>

class samp {
  int i, j;
public:
  void set_ij(int a, int b) { i=a; j=b; }
  int get_product() { return i*j; }
};

main()
{
  samp *p;
  int i;

  p = new samp [10]; // asignación de un objeto array
  if(!p) {
    cout << "Error de asignación\n";
    return 1;
  }

  for(i=0; i<10; i++)
    p[i].set_ij(i, i);

  for(i=0; i<10; i++) {
    cout << "Producto [" << i << "] es: ";
    cout << p[i].get_product() << "\n";
  }
  delete [] p;
  return 0;
}
```

Este programa muestra lo siguiente:

```
Producto [0] es: 0
Producto [1] es: 1
Producto [2] es: 4
```

```
Producto [3] es: 9
Producto [4] es: 16
Producto [5] es: 25
Producto [6] es: 36
Producto [7] es: 49
Producto [8] es: 64
Producto [9] es: 81
```

5. La siguiente versión del programa anterior da un destructor a **samp**, de modo que cuando se libera **p** se llama al destructor de cada elemento:

```
// Asignación dinámica de objetos.
#include <iostream.h>

class samp {
  int i, j;
public:
  void set_ij(int a, int b) { i=a; j=b; }
  ~samp() { cout << "Destrucción...\n"; }
  int get_product() { return i*j; }
};

main()
{
  samp *p;
  int i;

  p = new samp [10]; // asignación de un objeto array
  if(!p) {
    cout << "Error de asignación\n";
    return 1;
  }

  for(i=0; i<10; i++)
    p[i].set_ij(i, i);

  for(i=0; i<10; i++) {
    cout << "Producto [" << i << "] es: ";
    cout << p[i].get_product() << "\n";
  }

  delete [] p;
  return 0;
}
```

Este programa muestra lo siguiente:

```
Producto [0] es: 0
Producto [1] es: 1
Producto [2] es: 4
Producto [3] es: 9
Producto [4] es: 16
Producto [5] es: 25
Producto [6] es: 36
```

```
Producto [7] es: 49
Producto [8] es: 64
Producto [9] es: 81
Destrucción...
Destrucción...
Destrucción...
Destrucción...
Destrucción...
Destrucción...
Destrucción...
Destrucción...
Destrucción...
Destrucción...
```

Como puede verse, se llama diez veces al destructor de **samp** —una vez por cada elemento del array.

EJERCICIOS

1. Transforme el siguiente código en uno equivalente que haga uso de **new**.

```
char *p;

p = (char *) malloc(100);
// ...
strcpy(p, "Esto es una prueba");
```

Truco: Una cadena simplemente es un array de caracteres.

2. Utilizando **new**, asigne memoria a un **double** y déle un valor inicial de −123.0987.

4.6. REFERENCIAS

C++ consta de una particularidad relacionada con los punteros, denominada *referencia*. Una referencia es un puntero implícito que, a todos los efectos, se comporta como cualquier otro nombre para una variable. Existen tres modos de utilizar una referencia. Primero, una referencia puede pasarse a una función. Segundo, una referencia puede ser devuelta por una función. Por último, puede crearse una referencia independiente. En esta sección se analizan cada una de estas aplicaciones de la referencia, comenzando por los parámetros de referencia.

Sin lugar a dudas, el uso más importante de una referencia es como parámetro de una función. Para facilitar la comprensión de un parámetro por referencia y su forma de operar, comenzaremos con un programa que utiliza un puntero (no una referencia) como parámetro:

```
#include <iostream.h>

void f(int *n); // uso de un parámetro puntero

main()
```

```
{
  int i = 0;

  f(&i);

  cout << "Este es el nuevo valor de i: " << i << '\n';

  return 0;
}

void f(int *n)
{
  *n = 100; // almacena 100 en el argumento apuntado por n
}
```

En este programa, **f( )** almacena el valor 100 en el entero apuntado por **n**. Además, cuando **f( )** finaliza, **i** contiene el valor 100.

Este programa demuestra cómo se utiliza un puntero como parámetro para crear, de modo manual, un mecanismo de paso de parámetros con llamada por referencia. En un programa C, es la única manera de realizar una llamada por referencia. Sin embargo, en C++ puede automatizarse completamente este proceso utilizando un parámetro por referencia. Para comprobar esto, escribiremos de nuevo el programa anterior. A continuación, se ofrece una versión que utiliza un parámetro por referencia:

```
#include <iostream.h>

void f(int &n); // declaración de un parámetro por referencia

main()
{
  int i = 0;

  f(i);

  cout << "Este es el nuevo valor de i: " << i << '\n';

  return 0;
}
// f() ahora utiliza un parámetro por referencia
void f(int &n)
{
  // observe que no se necesita el * en la siguiente sentencia
  n = 100; // almacena 100 en el argumento usado para llamar a f()
}
```

Examinemos este programa cuidadosamente. Primero, para declarar una variable o parámetro por referencia el nombre de la variable debe ir precedido por **&**. Así es como **n** se declara como un parámetro de **f( )**. Ahora que **n** es una referencia, ya no es necesario —o incluso erróneo— aplicar el operador *. En su lugar, cada vez que se utiliza **n** dentro de **f( )** automáticamente es considerada como un puntero al argumento utilizado para llamar a **f( )**. Esto significa que la sentencia

```
n = 100;
```

en realidad coloca el valor 100 en la variable utilizada para llamar a **f( )** que, en este caso, es **i**. Además, cuando se llama a **f( )**, no es necesario que el argumento vaya precedido de **&**. En lugar de ello, debido a que **f( )** se declara con un parámetro por referencia, la dirección del argumento se pasa *automáticamente* a **f( )**.

Para resumir, cuando se utiliza un parámetro por referencia, el compilador pasa automáticamente como argumento la dirección de la variable utilizada. No es necesario (de hecho, no está permitido) generar a mano la dirección del argumento precedido de **&**. Además, dentro de la función, el compilador usa automáticamente la variable apuntada por el parámetro de referencia. No se requiere (y nuevamente, no está permitido) emplear el *. De este modo, un parámetro por referencia automatiza completamente el mecanismo de paso de parámetros por referencia.

Es fundamental comprender que no puede modificarse aquello a lo que apunta una referencia. Por ejemplo, si la sentencia

```
n++;
```

estuviera incluida en **f( )** (dentro del programa anterior), en **main( )**, **n** apuntaría a **i**. En vez de incrementar **n**, esta sentencia incrementa el valor de la variable referenciada (en este caso **i**).

Los parámetros por referencia ofrecen varias ventajas sobre los punteros (más o menos) equivalentes alternativos. Primero, desde un punto de vista práctico, ya no es necesario pasar la dirección de un argumento. Cuando se utiliza un parámetro por referencia, se pasa la dirección automáticamente. Segundo, y de acuerdo a la opinión de muchos programadores, los parámetros por referencia presentan una interfaz más clara y elegante que la del incómodo mecanismo de punteros explícitos. Tercero, como se comprobará en la próxima sección, cuando se pasa un objeto por referencia a una función no se realiza ninguna copia. Este es un modo de eliminar las dificultades asociadas con la copia de un argumento que, cuando se llama a su función destructora, puede perjudicar cualquier otra zona necesitada en el programa.

EJEMPLOS

1. El ejemplo clásico de paso de argumentos por referencia es la función **swap( )**, que intercambia el valor de los argumentos por los que ha sido llamada. A continuación se presenta una versión de **swap( )** que utiliza referencias para cambiar el valor de sus dos argumentos enteros:

```
#include <iostream.h>

void swap(int &x, int &y);

main()
{
  int i, j;

  i = 10;
  j = 19;
```

```
  cout << "i: " << i << ", ";
  cout << "j: " << j << "\n";

  swap(i, j);
  cout << "Después del intercambio: ";
  cout << "i: " << i << ", ";
  cout << "j: " << j << "\n";

  return 0;
}

void swap(int &x, int &y)
{
  int t;

  t = x;
  x = y;
  y = t;
}
```

Si **swap( )** hubiera sido programada utilizando punteros en vez de referencias, presentaría el siguiente aspecto:

```
void swap(int *x, int *y)
{
  int t;

  t = *x;
  *x = *y;
  *y = t;
}
```

Como puede observarse, el uso de la versión con referencias de **swap( )** elimina la necesidad del operador *.

2. El siguiente programa utiliza la función **round( )** para redondear un valor **double**. El valor que hay que redondear se pasa por referencia.

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

void round(double &num);

main()
{
  double i = 100.4;

  cout << i << " el valor redondeado es ";
  round(i);
  cout << i << "\n";

  i = 10.9;
  cout << i << " el valor redondeado es ";
  round(i);
  cout << i << "\n";

  return 0;
```

```
}
void round(double &num)
{
  double frac;
  double val;

  // descomposición de un número en sus partes entera y decimal
  frac = modf(num, &val);

  if(frac < 0.5) num = val;
  else num = val+1.0;
}
```

round( ) utiliza una función de una biblioteca estándar, relativamente confusa, llamada **modf( )**, para descomponer un número en su parte entera y su parte decimal. Se devuelve la parte decimal; la parte entera se guarda en la variable apuntada por su segundo parámetro.

EJERCICIOS

1. Programe una función llamada **neg( )** que invierta el signo de su parámetro entero. Hágalo de dos maneras —la primera utilizando un parámetro puntero y, a continuación, utilizando un parámetro por referencia. Incluya un breve programa para mostrar su funcionamiento.
2. ¿Qué hay erróneo en el siguiente programa?

```
// Este programa tiene un error.
#include <iostream.h>

void triple(double &num);

main()
{
  double d = 7.0;

  triple(&d);

  cout << d;
  return 0;
}

  // Triple valor de num.
void triple(double &num)
{
  num = 3 * num;
}
```

3. Mencione algunas de las ventajas de los parámetros por referencia.

4.7. PASO DE REFERENCIAS A OBJETOS

Como ya se discutió en el Capítulo 2, cuando se pasa un objeto a una función, mediante el uso del mecanismo implícito del paso de parámetros por valor, se hace una copia del objeto. Aunque no se llame a la función constructora del parámetro, cuando la función finaliza se llama a su función destructora. A medida que se producen nuevas llamadas pueden presentarse serios problemas en algunas instancias —por ejemplo, cuando el destructor libera memoria dinámica.

Una solución a este problema es el paso de un objeto por referencia. (La otra solución involucra el uso de constructores de copia, que se discutirán en el Capítulo 5.) Cuando se pasa el objeto por referencia no se hace copia alguna y, por consiguiente, no se llama a su función destructora cuando la función finaliza. No debe olvidarse, sin embargo, que los cambios efectuados en el objeto dentro de la función afectan al objeto utilizado como argumento.

Nota *Es muy importante entender que una referencia no es un puntero. Por tanto, cuando se pasa un objeto por referencia, el operador de acceso a un atributo utiliza el punto (.), no la flecha (→).*

EJEMPLO

1. El siguiente ejemplo muestra la utilidad de pasar un objeto por referencia. En primer lugar, se presenta una versión de un programa que pasa por valor un objeto de **myclass** a una función llamada **f( )**:

```
#include <iostream.h>

class myclass {
  int who;
public:
  myclass(int n) {
    who = n;
    cout << "Construcción " << who << "\n";
  }
  ~myclass() { cout << "Destrucción " << who << "\n"; }
  int id() { return who; }
};

// o es pasado por valor.
void f(myclass o)
{
  cout << "Recibido " << o.id() << "\n";
}

main()
{
  myclass x(1);

  f(x);

  return 0;
}
```

Esta función muestra lo siguiente:

```
Construcción 1
Recibido 1
Destrucción 1
Destrucción 1
```

Como puede verse, se llama dos veces a la función destructora —primero, cuando se destruye la copia del objeto 1 al terminar **f( )** y, de nuevo, cuando el programa finaliza.

Sin embargo, si se cambia el programa de manera que **f( )** utilice un parámetro por referencia, no se efectúa ninguna copia y, por consiguiente, al finalizar **f( )** no se llama a ningún destructor:

```
#include <iostream.h>

class myclass {
  int who;
public:
  myclass(int n) {
    who = n;
    cout << "Construcción " << who << "\n";
  }
  ~myclass() { cout << "Destrucción " << who << "\n"; }
  int id() { return who; }
};

// Ahora se pasa o por referencia.
void f(myclass &o)
{
  // !!!Observe cómo aún se utiliza el operador . !!!
  cout << "Recibido " << o.id() << "\n";
}

main()
{
  myclass x(1);

  f(x);

  return 0;
}
```

Esta versión presenta la siguiente salida:

```
Construcción 1
Recibido 1
Destrucción 1
```

Recuerde *Cuando acceda a las funciones miembro de un objeto mediante una referencia, utilice el operador punto, no la flecha.*

EJERCICIO

1. ¿Qué hay erróneo en el siguiente programa? Demuestre cómo eliminarlo utilizando un parámetro por referencia.

```
// Este programa tiene un error.
#include <iostream.h>
#include <string.h>
#include <stdlib.h>

class strtype {
  char *p;
public:
  strtype(char *s);
  ~strtype() { delete p; }
  char *get() { return p; }
};

strtype::strtype(char *s)
{
  int l;

  l = strlen(s);

  p = new char [l];
  if(!p) {
    cout << "Error de asignación\n";
    exit(1);
  }

  strcpy(p, s);
}

void show(strtype x)
{
  char *s;

  s = x.get();
  cout << s << "\n";
}

main()
{
  strtype a("Hola"), b("gente");

  show(a);
  show(b);

  return 0;
}
```

4.8. DEVOLUCION DE REFERENCIAS

Una función puede devolver una referencia. Como se describirá en el Capítulo 6, la devolución de una referencia puede ser muy útil cuando se sobrecargan determinados tipos de operadores. Sin embargo, también puede usarse para utilizar una función en el lado izquierdo de una sentencia de asignación. Su efecto es potente y espectacular.

EJEMPLOS

1. Para comenzar, se presenta un programa muy sencillo que contiene una función que devuelve una referencia:

```
// Un sencillo ejemplo de una función que devuelve una referencia.

#include <iostream.h>

int &f();
int x;

main()
{
  f() = 100;

  cout << x << "\n";

  return 0;
}

// Devolución de una referencia entera
int &f()
{
  return x; // devolución de una referencia a x
}
```

En este caso, la función **f( )** se declara como una función que devuelve una referencia a un entero.

Dentro del cuerpo de esta función, la sentencia

```
return x;
```

no devuelve el valor de la variable global **x**, sino que devuelve automáticamente la dirección de **x** (en forma de referencia). De este modo, dentro de **main( )**, la sentencia

```
f() = 100;
```

pone en **x** el valor 100, porque **f( )** ha devuelto una referencia a dicha variable.

Sintetizando, la función **f( )** devuelve una referencia. Por tanto, cuando **f( )** se utiliza en el lado izquierdo de una sentencia de asignación, la asignación se producirá sobre la referencia devuelta por **f( )**. Puesto que **f( )** devuelve una referencia a **x** (en este ejemplo), es **x** la que recibe el valor 100.

2. Debe tenerse cuidado cuando se devuelve una referencia de que el objeto referenciado no esté fuera del *ámbito de aplicación*. Por ejemplo, considere esta leve modificación sobre la función **f( )**:

```
// Devuelve una referencia entera
int &f()
{
  int x; // x ahora es una variable local
  return x; // devuelve una referencia a x
}
```

En este caso, **x** es local a **f( )** y estará fuera del ámbito de aplicación cuando **f( )** finalice. Esto significa que la referencia devuelta por **f( )** no puede utilizarse.

Nota *Algunos compiladores de C++ no permiten que se devuelva una referencia en una variable local. Sin embargo, este problema puede manifestarse de otras formas, como en el caso de la asignación dinámica de objetos.*

3. La devolución de una referencia está bien empleada cuando se crea un tipo de array acotado. Como ya es sabido, en C y C++ no se comprueban los límites del array. Por consiguiente, es posible sobrepasar o no alcanzar los límites del mismo. Sin embargo, en C++, puede crearse una **clase** array que realice automáticamente la comprobación de los límites.

La **clase** array contiene dos funciones esenciales —una que almacena la información dentro del array y otra que la recupera. Estas funciones pueden comprobar, en tiempo de ejecución, que no se sobrepasan los límites del array.

El siguiente programa construye un array de comprobación de límites para caracteres:

```
// Un ejemplo de array acotado.
#include <iostream.h>
#include <stdlib.h>

class array {
  int size;
  char *p;
public:
  array(int num);
  char &put(int i);
  char get(int i);
};

array::array(int num)
{
  p = new char [num];
  if(!p) {
```

```
    cout << "Error de asignación\n";
    exit(1);
  }
  size = num;
  }
// Almacenamiento de información dentro del array.
char &array::put(int i)
{
  if(i<0 || i>=size) {
    cout << "¡¡¡Error en los límites!!!\n";
    exit(1);
  }
  return p[i]; // devolución de una referencia a p[i]
}

// Extrae información del array.
char array::get(int i)
{
  if(i<0 || i>=size) {
    cout << "¡¡¡Error en los límites!!!\n";
    exit(1);
  }
  return p[i]; // devolución de un carácter
}

main()
{
  array a(10);

  a.put(3) = 'X';
  a.put(2) = 'R';

  cout << a.get(3) << a.get(2);
  cout << "\n";

  // ahora se genera un error de límites en tiempo de ejecución
  a.put(11) = '!';

  return 0;
}
```

Este es un ejemplo práctico de funciones que devuelven referencias y es conveniente examinarlo detalladamente. Nótese que la función **put( )** devuelve una referencia al elemento del array especificado por el parámetro **i**. Esta referencia puede utilizarse después en el lado izquierdo de una sentencia de asignación para almacenar información en el array —si el índice especificado por **i** no está fuera de los límites. La función inversa es **get( )**, que devuelve el valor almacenado en el índice especificado, si el índice está dentro del rango. En ocasiones a esta manera de establecer un array se le denomina *array seguro.*

Otro aspecto destacable del programa anterior es que el array se asigna dinámicamente utilizando **new.** Esto permite la declaración de arrays de diferentes longitudes.

Como ya se ha mencionado, el modo de llevar a cabo la comprobación de límites en este programa es una aplicación práctica de C++. Si es necesario tener verificados los límites de un array en tiempo de ejecución, ésta es una forma de hacerlo. Sin embargo, no debe olvidarse que la comprobación de los límites ralentiza el acceso al array. Por tanto, sólo debe incluirse la comprobación de límites cuando existe la posibilidad real de sobrepasarlos.

EJERCICIOS

1. Escriba un programa que cree un array bidimensional seguro, de dos por tres, de enteros. Demuestre que funciona.
2. ¿Es válido el siguiente fragmento de código? Si no lo es, ¿por qué?

```
int &f();
.
.
.
int *x;

x = f();
```

4.9. *REFERENCIAS INDEPENDIENTES Y RESTRICCIONES*

Aunque no se utiliza frecuentemente, es posible crear una *referencia independiente*. Una referencia independiente es una variable de referencia que, a todos los efectos, es simplemente otro nombre para otra variable. Debido a que no pueden asignarse nuevos valores a las referencias, una referencia independiente debe inicializarse cuando se declara.

Nota *Puesto que a veces se utilizan referencias independientes, es importante conocerlas. Sin embargo, la mayoría de los programadores piensan que no existe ninguna necesidad de utilizarlas y que lo único que aportan a un programa es confusión. Además, en C++ existen las referencias independientes debido, fundamentalmente, a que no hay razones precisas para desestimarlas. En la medida de lo posible, debería evitarse su uso.*

Existen un conjunto de restricciones que se aplican a todos los tipos de referencias. Una referencia no puede referenciarse. No pueden crearse arrays de referencias y no se puede referenciar un campo bits. Las referencias deben inicializarse excepto si son atributos de una clase, valores devueltos o parámetros de funciones.

Recuerde *Las referencias son similares a los punteros, pero no son punteros.*

EJEMPLOS

1. A continuación se presenta un programa que contiene una referencia independiente:

```
#include <iostream.h>

main()
{
  int x;
  int &ref = x; // creación de una referencia independiente

  x = 10; // estas dos sentencias
  ref = 10; // son funcionalmente equivalentes

  ref = 100;
  // esto imprime dos veces el número 100
  cout << x << ' ' << ref << "\n";

  return 0;
}
```

   En este programa, la referencia independiente **ref** sirve como otro nombre diferente para **x**. Desde un punto de vista práctico, **x** y **ref** son equivalentes.
2. Una referencia independiente puede referenciar a una constante. Por ejemplo, la siguiente sentencia es válida:

```
const int &ref = 10;
```

   Nuevamente, hay poco beneficio con el uso de este tipo de referencia, pero es posible encontrarla alguna vez en otros programas.

EJERCICIO

1. Intente pensar en un buen uso de una referencia independiente.

COMPROBACION DE APTITUD SUPERIOR

Al llegar a este punto, debería ser capaz de responder a estas preguntas y de realizar los ejercicios.

1. Dada la siguiente clase, cree un array bidimensional de dos por cinco, dando a cada objeto del array el valor inicial que desee.

```
class a_type {
  double a, b;
public:
  a_type(double x, double y) {
    a = x;
    b = y;
  }
  void show() { cout << a << ' ' << b << "\n"; }
};
```

2. Modifique la solución del problema anterior de modo que el acceso al array se lleve a cabo mediante un puntero.
3. ¿Qué es el puntero **this**?
4. Muestre las formas generales de **new** y **delete**. ¿Cuáles son sus ventajas frente a **malloc( )** y **free( )**?
5. ¿Qué es una referencia? ¿Qué ventaja tiene utilizar un parámetro por referencia?
6. Elabore una función llamada **recip( )** que tome un parámetro por referencia **double**. Permita que la función transforme el valor de dicho parámetro en su inverso. Escriba un programa para verificar que funciona.

COMPROBACION DE APTITUD INTEGRADA

Esta sección comprueba cómo ha asimilado el contenido de este capítulo con el de los anteriores.

1. Dado un puntero a un objeto, ¿qué operador se utiliza para acceder a un atributo del objeto?
2. En el Capítulo 2, se creó una clase **strtype** que asignaba dinámicamente espacio a una cadena. Confeccione de nuevo la clase **strtype** (mostrada aquí para su comodidad) para que utilice **new** y **delete**.

```
#include <iostream.h>
#include <malloc.h>
#include <string.h>
#include <stdlib.h>

class strtype {
  char *p;
  int len;
public:
  strtype(char *ptr);
  ~strtype();
  void show();
};

strtype::strtype(char *ptr)
{
  len = strlen(ptr);
  p = (char *) malloc(len+1);
  if(!p) {
  cout << "Error de asignación\n";
  exit(1);
  }
  strcpy(p, ptr);
}

strtype::~strtype()
{
  cout << "Liberando p\n";
  free(p);
}
```

```
void strtype::show()
{
  cout << p << " - longitud: " << len;
  cout << "\n";
}

main()
{
  strtype s1("Esto es una prueba"), s2("Me gusta C++");

  s1.show();
  s2.show();

  return 0;
}
```

3. Elabore de nuevo cualquier programa de este capítulo para que haga uso de una referencia.

5 *Sobrecarga de funciones*

OBJETIVOS DEL CAPITULO

En este capítulo se aprenderán más cosas sobre las funciones de sobrecarga. Aunque ya se introdujo anteriormente este tema en el libro, existen otros aspectos del mismo que es necesario cumplimentar. Entre ellos se encuentran incluidos la sobrecarga de funciones constructoras, la creación de constructores de copias y el modo de evitar la ambigüedad cuando se utiliza la sobrecarga.

COMPROBACION DE APTITUD

Antes de continuar, debería ser capaz de responder a estas preguntas y de realizar estos ejercicios.

1. ¿Qué es una referencia? Mencione dos usos importantes de la misma.
2. Muestre cómo se asigna un **float** y un **int** mediante **new**. Muestre también cómo liberarlos utilizando **delete**.
3. ¿Cuál es la forma general utilizada de **new** para inicializar una variable dinámica? Dé un ejemplo concreto.
4. Dada la siguiente clase, muestre cómo inicializar un array de diez elementos de modo que **x** tome los valores del 1 al 10.

```
class samp {
  int x;
public:
  samp(int n) { x = n; }
  int getx() { return x; }
};
```

5. Mencione alguna ventaja de los parámetros por referencia. Mencione una desventaja.
6. ¿Pueden inicializarse los arrays asignados dinámicamente?
7. Utilizando el siguiente prototipo, cree una función llamada **mag( )** que eleve **num** al orden de magnitud especificado por **order**:

```
void mag(long &num, long order);
```

Por ejemplo, si **num** es 4 y order es 2, después de que **mag( )** finalice **num** tendrá el valor 400. Construya un programa que demuestre que la función lleva a cabo lo esperado.

5.1. SOBRECARGA DE FUNCIONES CONSTRUCTORAS

Es posible —realmente, es lo normal— sobrecargar la función constructora de una clase. (Sin embargo, no es posible sobrecargar un destructor.) Hay tres razones fundamentales por las que se puede querer sobrecargar una función constructora: para ganar flexibilidad, para permitir arrays y para construir cons-

tructores de copias. En esta sección se discutirán las dos primeras. Los constructores de copias se discutirán en la próxima sección.

Una cosa a tener en cuenta, cuando se estudien los ejemplos, es que la función constructora de una clase debe proporcionar un modelo para cada uno de los modos en los que se declare un objeto de esa clase. Si no se encuentra el modelo, se produce un error de compilación. Esta es la razón por la que son tan comunes en los programas de C++ las funciones constructoras sobrecargadas.

EJEMPLOS

1. Quizá el uso más frecuente de las funciones constructoras sobrecargadas es el de ofrecer la opción de inicializar o no un objeto. Por ejemplo, en el siguiente programa, **o1** recibe un valor inicial y **o2** no. Si se elimina el constructor que tiene la lista de argumentos vacía, el programa no compilará porque no hay ningún constructor que coincida con un objeto sin inicializar de tipo **samp**. Lo contrario también es cierto: Si se elimina el constructor parametrizado, el programa no compilará porque no hay un modelo para el objeto inicializado. Se necesitan ambos para que el programa compile correctamente.

```
#include <iostream.h>

class myclass {
  int x;
public:
  // dos formas de sobrecarga de un constructor
  myclass() { x = 0; } // sin inicializador
  myclass(int n) { x = n; } // inicializador
  int getx() { return x; }
};

  main()
{
  myclass o1(10); // declaración con valor inicial
  myclass o2; // declaración sin inicializador

  cout << "o1: " << o1.getx() << '\n';
  cout << "o2: " << o2.getx() << '\n';

  return 0;
}
```

2. Otra razón típica por la que se sobrecarga un constructor es para permitir que tanto los objetos como los arrays de objetos aparezcan dentro de un programa. Como ya sabrá, por su propia experiencia en programación, es bastante normal inicializar una variable, pero no lo es tanto inicializar un array. (Bastante a menudo los valores de los arrays se asignan utilizando información que sólo se conoce cuando se está ejecutando el programa.) De este modo, para admitir arrays de objetos sin inicializar, junto con objetos inicializados, debe incluirse un constructor que permita la inicialización y otro que no.

Por ejemplo, partiendo de la clase **myclass** del Ejemplo 1, estas dos declaraciones son válidas:

```
myclass ob(10);
myclass ob[5];
```

El hecho de proporcionar constructores de inicialización y de no inicialización da lugar a que las variables pueden inicializarse o no, según se necesite. Por ejemplo, este programa declara dos arrays del tipo **myclass**; uno está inicializado y el otro no:

```
#include <iostream.h>

class myclass {
  int x;
public:
  // dos modos de sobrecarga de un constructor
  myclass() { x = 0; } // sin inicializador
  myclass(int n) { x = n; } // inicializador
  int getx() { return x; }
};

main()
{
  myclass o1[10]; // declaración del array sin inicializadores

  // declaración con inicializadores
  myclass o2[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  int i;

  for(i=0; i<10; i++) {
    cout << "o1[" << i << "]: " << o1[i].getx();
    cout << '\n';
    cout << "o2[" << i << "]: " << o2[i].getx();
    cout << '\n';
  }

  return 0;
}
```

En este ejemplo, la función constructora pone a cero todos los elementos de **o1**. Los elementos de **o2** se inicializan según se muestra en el programa.

3. Otra razón para sobrecargar las funciones constructoras es la de permitir que el programador seleccione el método más conveniente de inicializar un objeto. Para ver cómo se lleva a cabo, primero, examine el próximo ejemplo, que crea una clase que contiene las fechas de un calendario. El constructor **date( )** se sobrecarga de dos formas. En una de ellas, la fecha se acepta como una cadena de caracteres. En la otra, la fecha se pasa como tres enteros.

```
#include <iostream.h>
#include <stdio.h> // incluida para sscanf()
```

```
class date {
  int day, month, year;
public:
  date(char *str);
  date (int m, int d, int y) {
    day = d;
    month = m;
    year = y;
  }
  void show() {
    cout << month << '/' << day << '/';
    cout << year << '\n';
  }
};

date::date(char *str)
{
  sscanf(str, "%d%*c%d%*c%d", &month, &day, &year);
}

main()
{
  // construcción del objeto fecha utilizando una cadena
  date sdate("11/1/95");

  // construcción del objeto fecha utilizando enteros
  date idate(11, 1, 95);

  sdate.show();
  idate.show();
  return 0;
}
```

La ventaja de sobrecargar el constructor **date( )**, de acuerdo a lo mostrado en este programa, está en la libertad para utilizar cualquiera de las versiones, en función de la situación en la que esté siendo usada. Por ejemplo, si se crea un objeto **date** a partir de la entrada de un usuario, la versión de la cadena es la más sencilla de emplear. Sin embargo, si el objeto **date** se construye a partir de algún tipo de cálculo interno, probablemente tenga más sentido la versión de los tres parámetros enteros.

Aunque es posible sobrecargar un constructor tantas veces como se desee, realizarlo en exceso tiene un efecto destructivo sobre la clase. Desde el punto de vista del estilo, lo más adecuado es sobrecargar un constructor para mejorar solamente aquellas situaciones con alta probabilidad de ocurrir frecuentemente. Por ejemplo, sobrecargar **date( )** una tercera vez, de manera que se pueda introducir la fecha como tres enteros en octal, no tiene mucho sentido. Sin embargo, sobrecargarla para aceptar un objeto de tipo **time_t** (un tipo que almacena la fecha y la hora del sistema) podría ser muy útil. (Véanse los ejercicios de Comprobación de aptitud superior del final del capítulo, que contienen un ejemplo que hace precisamente esto.)

4. Existe otra situación en la que será necesario sobrecargar la función constructora de una clase: cuando se asigna un array dinámico de esa clase. Como se recordará del capítulo anterior, no puede inicializarse un array dinámico. Por tanto, si la clase

contiene un constructor que tiene un inicializador, debe incluirse una versión sobrecargada que no tenga inicializador. Por ejemplo, a continuación se presenta un programa que asigna un array de objetos dinámicamente:

```
#include <iostream.h>

class myclass {
  int x;
public:
  // dos modos de sobrecargar un constructor
  myclass() { x = 0; } // sin inicializador
  myclass(int n) { x = n; } // inicializador
  int getx() { return x; }
  void setx(int n) { x = n; }
};

main()
{
  myclass *p;
  myclass ob(10);  // inicialización de una única variable

  p = new myclass[10]; // aquí no pueden utilizarse
  inicializadores
  if(!p) {
    cout << "Error de asignación\n";
    return 1;
  }

  int i;

  // inicialización de todos los elementos de ob
  for(i=0; i<10; i++) p[i] = ob;

  for(i=0; i<10; i++) {
    cout << "p[" << i << "]: " << p[i].getx();
    cout << '\n';
  }

  return 0;
}
```

Sin la versión sobrecargada de **myclass( )** que no tiene inicializador, la sentencia **new** habría generado un error en tiempo de compilación y el programa no habría compilado correctamente.

EJERCICIOS

1. Dada esta clase parcialmente definida

```
class strtype {
  char *p;
```

```
  int len;
public:
  char *getstring() { return p; }
  int getlength() { return len; }
};
```

añada a la misma dos funciones constructoras. La primera no debe tener parámetros. Asígnela 255 bytes de memoria (utilizando **new**), inicialice esa memoria con una cadena vacía y déle a **len** un valor de 255. El segundo constructor debe tener dos parámetros. El primero es la cadena que se utiliza para inicializar y el otro es el número de bytes a asignar. Esta versión debe asignar la cantidad de memoria especificada y copiar la cadena en dicha memoria. Realice todas las verificaciones de límites que sean necesarias y demuestre con un pequeño programa que los constructores funcionan.

2. En el Ejercicio 2 del Capítulo 2, Sección 1, usted creó una emulación de un cronógrafo. Amplíe aquella solución de manera que la clase **stopwatch** ofrezca un constructor sin parámetros (como ya lo hacía) y una versión sobrecargada que acepte la hora del sistema en la forma devuelta por la función estándar **clock( )**. Demuestre que esta mejora funciona.
3. Piense en diferentes formas en las que una función constructora sobrecargada puede beneficiarle en sus propias tareas de programación.

5.2. CREACION Y USO DE UN CONSTRUCTOR DE COPIAS

Una de las formas más importantes de un constructor sobrecargado es el *constructor de copias*. Como se ha podido ver en los numerosos ejemplos de los capítulos anteriores, cuando se pasa un objeto a una función, o ésta lo devuelve, pueden presentarse dificultades. Como se verá en esta sección, un modo de evitar estos problemas es definir un constructor de copias, que es un tipo especial de función constructora sobrecargada.

Para comenzar, planteemos de nuevo el problema para el que se ha diseñado el constructor de copias que lo va a resolver. Cuando se pasa un objeto a una función, se realiza una copia bit a bit (i.e. exacta) de ese objeto y se guarda en el parámetro de la función que recibe el objeto. Sin embargo, hay casos en los que no es deseable una copia idéntica. Por ejemplo, si el objeto contiene un puntero a la memoria asignada, la copia apuntará a la *misma* memoria que el objeto original. Por consiguiente, si la copia efectúa un cambio sobre el contenido de esta memoria, ¡la memoria también cambiará para el objeto original! Cuando la función finaliza se destruirá la copia mediante una llamada a su destructor. Esto puede ocasionar efectos colaterales indeseables que, además, afectarán al objeto original.

Se produce una situación parecida cuando una función devuelve un objeto. Normalmente, el compilador generará un objeto temporal que mantiene una copia del valor devuelto por la función. (Esto se hace automáticamente y cae fuera de su control.) Este objeto temporal desaparece, una vez que se devuelve

su valor a la rutina que provocó la llamada, mediante una llamada al destructor temporal. Sin embargo, si el destructor elimina alguna información necesitada por esa rutina (por ejemplo, si libera memoria asignada dinámicamente), continuarán los problemas.

En el fondo de todos estos problemas está el hecho de la copia bit a bit del objeto. Para prevenirlos, el programador necesita definir con exactitud lo que sucede cuando se hace una copia de un objeto, de modo que puedan evitarse efectos colaterales indeseables. La forma de llevar esto a cabo es mediante la creación de un constructor de copias. La definición de un constructor de copias permite especificar completamente lo que ocurre exactamente cuando se hace la copia de un objeto.

Es importante entender que C++ define dos tipos distintos de situaciones en las que se da el valor de un objeto a otro. La primera es la asignación. La segunda situación es la inicialización, que puede tener lugar de tres maneras:

- Cuando se utiliza un objeto para inicializar otro en una sentencia de declaración.
- Cuando se pasa un objeto como parámetro a una función.
- Cuando se crea un objeto temporal para ser usado como el valor devuelto por una función.

El constructor de copias sólo se aplica a la inicialización. No se aplica a la asignación.

Por omisión, cuando se realiza una inicialización, el compilador proporciona automáticamente una copia bit a bit. (Esto es, C++ ofrece automáticamente un constructor de copias implícito, que simplemente duplica el objeto.) Sin embargo, es posible especificar con precisión cómo un objeto inicializará a otro, definiendo un constructor de copias. Una vez definido, el constructor de copias será llamado siempre que se utilice un objeto para inicializar otro.

Recuerde *Los constructores de copias no influyen en las operaciones de asignación.*

Todos los constructores de copias presentan esta forma general:

```
classname (const classname &obj) {
 // cuerpo del constructor
}
```

Aquí, *obj* es una referencia a un objeto que se utiliza para inicializar otro objeto. Por ejemplo, supuesta una clase denominada **myclass**, y que **y** es un objeto de tipo **myclass**, las siguientes sentencias llamarían al constructor de copias de **myclass**:

```
myclass x = y; // y inicializando explícitamente x
func1(y); // y pasado como un parámetro
y = func2(); // y recibiendo un objeto devuelto
```

En los dos primeros casos, se pasaría al constructor de copias una referencia a **y**. En el tercero, se pasa al constructor de copias una referencia al objeto devuelto por **func2( )**.

EJEMPLOS

1. A continuación se muestra un ejemplo que ilustra la necesidad de un constructor de copias. Este programa crea un tipo restringido de array de enteros «seguro», que previene que se sobrepasen los límites del array. Se asigna el espacio de cada array utilizando **new**, y dentro de cada objeto array se mantiene un puntero a la memoria.

```
/* Este programa crea una clase de array "seguro". Puesto que
   se asigna dinámicamente el espacio para el array,
   se suministra un constructor de copias para asignar memoria
   cuando se utiliza un objeto array para inicializar otro.
*/
#include "iostream.h"
#include "stdlib.h"

class array {
  int *p;
  int size;
public:
  array(int sz) { // constructor
    p = new int[sz];
    if(!p) exit(1);
    size = sz;
    cout << "Uso del constructor 'normal'\n";
  }
  ~array() {delete [] p;}

  // constructor de copias
  array(const array &a);

  void put(int i, int j) {
    if(i>=0 && i<size) p[i] = j;
  }
  int get(int i) {
    return p[i];
  }
};

/* Constructor de copias.

En el caso siguiente, se asigna específicamente memoria
para la copia, y la dirección de esta memoria se asigna a p.
Por tanto, p no está apuntando a la misma memoria asignada
dinámicamente al objeto original:

*/
  array::array(const array &a) {
```

```
  int i;

  p = new int[a.size]; // asignación de memoria para la copia
  if(!p) exit(1);
  for(i=0; i<a.size; i++) p[i] = a.p[i]; // contenido de la
  //copia
  cout << "Uso del constructor de copias\n";
}

main()
{
  array num(10); // esta sentencia llama al constructor
  //"normal"
  int i;

  // colocación de algunos valores en el array
  for(i=0; i<10; i++) num.put(i, i);

  // presentación de num
  for(i=9; i>=0; i--) cout << num.get(i);
  cout << "\n";

  // creación de otro array e inicialización con num
  array x = num; // esta sentencia invoca al constructor de
  copias

  // presentación de x
  for(i=0; i<10; i++) cout << x.get(i);

  return 0;
}
```

Cuando **num** se utiliza para inicializar **x**, se llama al constructor de copias, se asigna memoria para el nuevo array y se almacena en **x.p** y el contenido de **num** se copia en el array de **x**. De este modo, **x** y **num** tienen arrays que contienen los mismos valores, pero cada array es independiente y distinto. (Esto es, **num.p** y **x.p** no apuntan a la misma zona de memoria.) Si el constructor de copias no hubiera sido creado, entonces la inicialización bit a bit **array x = num** ¡habría dado lugar a que los arrays de **x** y de **num** compartieran la misma memoria! (Esto es, **num.p** y **x.p** habrían apuntado realmente a la misma posición.)

El constructor de copias solamente se llama para las inicializaciones. Por ejemplo, la siguiente secuencia no llama al constructor de copias definido en el programa anterior:

```
array a(10);
array b(10);

b = a; // no llama al constructor de copias
```

En este caso, **b = a** realiza la operación de asignación.

2. Para ver cómo el constructor de copias ayuda a prevenir algunos de los problemas asociados con el paso de determinados tipos de objetos a funciones, observe el siguiente (incorrecto) programa:

```
// Este programa tiene un error.
#include <iostream.h>
#include <string.h>
#include <stdlib.h>

class strtype {
  char *p;
public:
  strtype(char *s);
  ~strtype() { delete [] p; }
  char *get() { return p; }
};

strtype::strtype(char *s)
{
  int l;

  l = strlen(s);

  p = new char [l];
  if(!p) {
    cout << "Error de asignación\n";
    exit(1);
  }

  strcpy(p, s);
}

void show(strtype x)
{
  char *s;

  s = x.get();
  cout << s << "\n";
}

main()
{
  strtype a("Hola"), b("gente");

  show(a);
  show(b);

  return 0;
}
```

En este programa, cuando un objeto **strtype** se pasa a **show( )**, se hace una copia bit a bit (puesto que no se ha definido un constructor de copias) y se guarda en el parámetro **x**. De este modo, cuando finaliza la función, **x** pierde su valor y se elimina. Esto, por supuesto, da lugar a una llamada al destructor de **x**, que libera **x.p**. Sin embargo, la memoria que se ha liberado es la misma memoria que todavía está siendo utilizada por el objeto empleado para llamar a la función. Esto conduce a un error.

La solución del problema anterior es la definición de un constructor de copias para la clase **strtype**, que asigne memoria a la copia cuando se cree. Este es el enfoque usado en el siguiente programa corregido:

```
/* Este programa utiliza un constructor de copias para
permitir que los objetos strtype sean pasados a funciones */
#include <iostream.h>
#include <string.h>
#include <stdlib.h>

class strtype {
  char *p;
public:
  strtype(char *s); // constructor
  strtype(const strtype &o); // constructor de copias
  ~strtype() { delete [] p; } // destructor
  char *get() { return p; }
};

// Constructor "normal"
strtype::strtype(char *s)
{
  int l;

  l = strlen(s);

  p = new char [l];
  if(!p) {
    cout << "Error de asignación\n";
    exit(1);
  }

  strcpy(p, s);
}

// Constructor de copias
strtype::strtype(const strtype &o)
{
  int l;

  l = strlen(o.p);

  p = new char [l]; // asignación de memoria para nueva copia
  if(!p) {
    cout << "Error de asignación\n";
```

```
    exit(1);
  }

  strcpy(p, o.p); // copia de la cadena en la copia
}

void show(strtype x)
{
  char *s;

  s = x.get();
  cout << s << "\n";
}

main()
{
  strtype a("Hola"), b("gente");

  show(a);
  show(b);
  return 0;
}
```

Ahora, cuando finaliza **show( )** y **x** pierde su valor, la memoria apuntada por **x.p** (que se liberará) no es la misma que la utilizada por el objeto pasado a la función.

EJERCICIOS

1. También se invoca al constructor de copias cuando una función genera el objeto temporal utilizado como el valor devuelto por una función (para aquellas funciones que devuelven objetos). Teniendo esto en cuenta, consideremos la siguiente salida:

```
Construcción normal
Construcción normal
Construcción de copias
```

Esta salida fue creada por el siguiente programa. Explique por qué y describa con exactitud lo que sucede.

```
#include <iostream.h>

class myclass {
public:
  myclass();
  myclass(const myclass &o);
  myclass f();
};

// Constructor normal
```

```
myclass::myclass()
{
   cout << "Constructor normal\n";
}

// Constructor de copias
myclass::myclass(const myclass &o)
{
  cout << "Construcción de la copia\n";
}

// Devolución de un objeto.
myclass myclass::f()
{
  myclass temp;

  return temp;
}

main()
{
  myclass obj;

  obj = obj.f();

  return 0;
}
```

2. Explique qué hay erróneo en el siguiente programa y después corríjalo.

```
// Este programa contiene un error.
#include <iostream.h>
#include <stdlib.h>

class myclass {
  int *p;
public:
  myclass(int i);
  ~myclass() { delete p; }
  friend int getval(myclass o);
};

myclass::myclass(int i)
{
  p = new int;

  if(!p) {
    cout << "Error de asignación\n";
    exit(1);
  }
  *p = i;
}

int getval(myclass o)
```

```
{
  return *o.p; // obtención del valor
}

main()
{
  myclass a(1), b(2);

  cout << getval(a) << " " << getval(b);
  cout << "\n";
  cout << getval(a) << " " << getval(b);

  return 0;
}
```

3. Explique, empleando sus conocimientos, el propósito de un constructor de copias y sus diferencias con un constructor normal.

5.3. *EL ANACRONISMO* overload

Cuando se ideó C++, se requería la palabra clave **overload** para crear una función sobrecargada. Aunque ya no es necesario utilizarla, con el fin de mantener la compatibilidad con los programas antiguos de C++, todavía se conserva entre las palabras clave y todos los compiladores de C++ aún aceptan la sintaxis de sobrecarga del viejo estilo. A pesar de que debería evitarse su uso, es posible ver **overload** en programas ya existentes, de modo que es una buena idea entender cómo se aplicaba.

La forma general de **overload** se muestra en esta línea:

overload nombre-func;

donde *func-name* es el nombre de la función que va a ser sobrecargada. Esta sentencia debe preceder a las declaraciones de la función sobrecargada. Por ejemplo, la siguiente sentencia le comunica al compilador que se va a sobrecargar una función llamada **timer( )**:

```
overload timer;
```

Recuerde *Puesto que en el actual C++* ***overload*** *es un anacronismo, debería evitarse su uso.*

5.4. *UTILIZACION DE ARGUMENTOS IMPLICITOS*

C++ tiene una característica que está relacionada con la sobrecarga de funciones. Esta característica se denomina *argumento implícito* y permite dar un valor implícito a un parámetro cuando no se especifica el argumento correspondiente

en la llamada a la función. Como se observará, el uso de argumentos implícitos, básicamente, es una forma abreviada de sobrecarga de funciones.

Para dar un argumento implícito a un parámetro, simplemente hay que poner a continuación del parámetro un signo de igualdad y el valor que se desee dar por defecto, si el correspondiente argumento no está presente cuando se llama a la función. Por ejemplo, esta función da a sus dos parámetros el valor implícito 0:

```
void f(int a=0, int b=0);
```

La sintaxis es parecida a la de la inicialización de una variable.

Esta función puede llamarse de tres modos diferentes. Primero, puede llamarse con ambos argumentos especificados. Segundo, puede llamarse sólo con el primer argumento especificado. En este caso, **b** tendrá por omisión el valor cero. Por último, puede llamarse a **f( )** sin ningún argumento, tomando **a** y **b** el valor implícito cero. Por tanto, las siguientes llamadas de **f( )** son todas válidas:

```
f(); // a y b valen por omisión 0
f(10); // a vale 10, b por omisión 0
f(10, 99) // a vale 10, b vale 99
```

En este ejemplo queda claro que no hay modo de darle a **a** un valor por omisión y especificar **b**.

Cuando se crea una función que tiene uno o más argumentos implícitos éstos sólo deben especificarse una vez: bien en la definición de la función o en su prototipo, pero no en ambos. Esta regla se aplica aunque sólo se dupliquen los mismos valores implícitos. (Esta restricción es un escollo en la sintaxis formal de C++ .)

Como ya habrá adivinado, todos los parámetros implícitos deben situarse a la derecha de cualquier parámetro que no tenga valor implícito. Además, una vez que se comienzan a definir parámetros implícitos no pueden especificarse parámetros que no tengan valores implícitos.

Otra cuestión sobre los argumentos implícitos: deben ser constantes o variables globales. No pueden ser variables locales u otros parámetros.

EJEMPLOS

1. A continuación se muestra un programa que ilustra el ejemplo descrito en la discusión anterior:

```
// Primer ejemplo sencillo de argumentos implícitos.
include <iostream.h>

void f(int a=0, int b=0)
{
  cout << "a: " << a << ", b: " << b;
  cout << '\n';
}
main()
{
  f();
```

```
  f(10);
  f(10, 99);

  return 0;
}
```

Como cabe esperar, este programa presenta la siguiente salida:

```
a: 0, b: 0
a: 10, b: 0
a: 10, b: 99
```

No debe olvidarse de que una vez especificado el primer argumento implícito, todos los parámetros restantes también deben tener valores implícitos. Por ejemplo, esta versión, ligeramente diferente, de **f( )** origina un error en tiempo de compilación:

```
void f(int a=0, int b) // ¡error! b también debe tener valor
implícito
{
  cout << "a: " << a << ", b: " << b;
  cout << '\n';
}
```

2. Para entender cómo están relacionados los argumentos implícitos con la sobrecarga de funciones, consideremos, en primer lugar, el siguiente programa que sobrecarga la función llamada **box_area( )**. Esta función devuelve el área de un rectángulo.

```
// Cálculo del área de un rectángulo utilizando
funciones sobrecargadas
#include <iostream.h>

// Devolución del área de un rectángulo no cuadrangular.
double box_area(double length, double width)
{
  return length * width;
}

// Devuelve el área de un rectángulo cuadrangular.
   double box_area(double length)
{
  return length * length;
}

main()
{
  cout << "el área del cuadro de 10 x 5.8 es: ";
  cout << box_area(10.0, 5.8) << '\n';

  cout << "el área del cuadro de 10 x 10 es: ";
  cout << box_area(10.0) << '\n';
  return 0;
}
```

En este programa se ha sobrecargado **box_area( )** de dos formas. En la primera, se pasan a la función ambas dimensiones del rectángulo. Esta versión se usa cuando el rectángulo no es cuadrado. Sin embargo, cuando el rectángulo es un cuadrado, sólo necesita especificarse un argumento y se produce una llamada a la segunda versión de **box_area( )**.

Si se medita sobre ello, queda claro que en esta situación no existe realmente necesidad de tener dos funciones diferentes. En su lugar, el segundo parámetro puede fijarse de forma implícita a algún valor que actúe como un indicador para **box_area( )**. Cuando la función ve ese valor, utiliza el parámetro **length** dos veces. El siguiente es un ejemplo de este enfoque:

```
// Cálculo del área de un rectángulo utilizando
argumentos implícitos.
#include <iostream.h>

// Devuelve el área de un cuadrado.
double box_area(double length, double width = 0)
{
  if(!width) width = length;
  return length * width;
}

main()
{
  cout << "el área de un rectángulo de 10 x 5.8 es: ";
  cout << box_area(10.0, 5.8) << '\n';

  cout << "el área de un rectángulo de 10 x 10 es: ";
  cout << box_area(10.0) << '\n';

  return 0;
}
```

En este caso, el valor implícito de **width** es cero. Se eligió este valor porque ningún cuadro puede tener una anchura nula. (En realidad, un rectángulo con anchura nula es una línea.) De este modo, si **box_area( )** ve este valor implícito, sustituye automáticamente el valor de **width** por el valor de **length**.

Como se muestra en el ejemplo, los argumentos implícitos proporcionan a menudo una alternativa sencilla para la sobrecarga de funciones. (Por supuesto, existen muchas situaciones en las que todavía se necesita la sobrecarga de funciones.)

3. No sólo es lícito dar a las funciones constructoras argumentos implícitos, sino que también es normal. Como pudo verse anteriormente en este capítulo, muchas veces se sobrecarga una función constructora simplemente para permitir que se creen objetos inicializados y no inicializados. En muchos casos, puede evitarse la sobrecarga de un constructor dándole uno o más argumentos implícitos. Examine, por ejemplo, el siguiente programa:

```
#include <iostream.h>

class myclass {
  int x;
```

```
public:
  /*Uso de argumentos implícitos en vez de sobrecargar
    el constructor de myclass. */
  myclass(int n = 0) { x = n; }
  int getx() { return x; }
};

  main()
{
  myclass o1(10); // declaración con valor inicial
  myclass o2; // declaración sin inicialización

  cout << "o1: " << o1.getx() << '\n';
  cout << "o2: " << o2.getx() << '\n';

  return 0;
}
```

Como muestra este ejemplo, dándole a **n** el valor implícito cero es posible crear objetos que tengan valores iniciales explícitos y otros para los que el valor implícito es suficiente.

4. Otra buena aplicación para un argumento implícito tiene lugar cuando se utiliza un parámetro para seleccionar una opción. Es posible dar a ese parámetro un valor implícito que, utilizado como un indicador, comunica a la función que continúe en la opción previamente seleccionada. Por ejemplo, en el siguiente programa la función **print( )** muestra una cadena en la pantalla. Si su parámetro **how** se pone a **ignore**, el texto se muestra como está. Si **how** es **upper**, el texto se muestra en mayúsculas. Si **how** es **lower**, el texto se muestra en minúsculas. Cuando no se especifica **how**, su valor implícito es -1, que indica a la función que utilice el último valor de **how**.

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

const int ignore = 0;
const int upper = 1;
const int lower = 2;

void print(char *s, int how = -1);

main()
{
  print("Hola gente\n", ignore);
  print("Hola gente\n", upper);
  print("Hola gente\n"); // continuación en mayúsculas
  print("Hola gente\n", lower);
  print("Esto es todo\n"); // continuación en minúsculas

  return 0;
}

/* Imprime una cadena con la opción especificada. Si no se
```

```
indica nada, usa la última opción.
*/
void print(char *s, int how)
{
  static int oldcase = ignore;

  // si no especifica nada usa la última opción
  if(how<0) how = oldcase;
  while(*s) {
    switch(how) {
     case upper: cout << (char) toupper(*s);
       break;
     case lower: cout << (char) tolower(*s);
       break;
     default: cout << *s;
    }
    s++;
  }
  oldcase = how;
}
```

Esta función presenta la siguiente salida:

```
Hola Gente
HOLA GENTE
HOLA GENTE
hola gente
Esto es todo
```

5. Con anterioridad se presentó en este capítulo la forma general de un constructor de copias. Esta forma general sólo se mostró con un parámetro. Sin embargo, es posible crear constructores de copias que tengan argumentos adicionales, mientras los argumentos adicionales tomen valores implícitos. Por ejemplo, lo siguiente es también una forma aceptable de un constructor de copias:

```
myclass(const myclass &obj, int x=0) {
  // cuerpo del constructor
}
```

Mientras que el primer argumento sea una referencia al objeto que se copia y los otros argumentos sean implícitos, la función es calificada como un constructor de copias. Esta flexibilidad permite crear constructores de copias que tengan otros usos.

6 Aunque los argumentos implícitos son potentes y adecuados, pueden no utilizarse. No hay razón por la que, cuando son usados correctamente, los argumentos implícitos no permitan a una función realizar su tarea de un modo eficiente y sencillo. Sin embargo, este es el único caso en el que el valor implícito dado a un parámetro tiene sentido. Por ejemplo, si el argumento es el valor requerido nueve veces de diez, es obviamente una buena idea dotar a la función de un argumento implícito. Sin embargo, en situaciones en las que no exista un valor que vaya a ser utilizado con más probabilidad que otro, no tiene mucho sentido emplear un valor implícito. En realidad, hacer uso de un argumento implícito, cuando no va ser utilizado, deses-

tructura el programa y contribuye a confundir a cualquiera que tenga que hacer uso de la función.

Al igual que en la sobrecarga de funciones, para llegar a ser un buen programador de C++ es necesario saber cuándo utilizar un argumento implícito.

EJERCICIOS

1. En la biblioteca estándar de C++ está la función **strtol( )**, con el siguiente prototipo:

   ```
   long strtol(const char *start, const **end, int base);
   ```

   La función transforma la cadena numérica apuntada por *start* en un entero largo. El número de referencia de la cadena numérica se especifica mediante *base*. Cuando la función finaliza, *end* apunta al carácter de la cadena inmediatamente posterior a ese número. Devuelve el entero largo equivalente a la cadena numérica. *base* debe estar comprendido entre 2 y 38. Sin embargo, en la mayoría de los casos, base toma el valor 10.

   Cree una función llamada **mystrtol( )** que haga lo mismo que **strtol( )**, excepto que *base* tome implícitamente el valor de 10. (Utilice, si lo desea, la función **strtol( )** para realizar la conversión. Esta función necesita el fichero cabecera **stdlib.h**.) Demuestre que la versión realizada funciona correctamente.
2. ¿Cuál es el error del siguiente prototipo de función?

   ```
   char *f(char *p, int x = 0, char *q);
   ```

3. La mayoría de los compiladores de C++ están provistos de funciones estándar que permiten posicionar el cursor y otras opciones similares. Si su compilador tiene estas funciones, escriba una función llamada **myclreol( )** que borre una línea desde la posición actual del cursor hasta el final de la misma. Construya la función con un parámetro que especifique el número de posiciones que hay que borrar. Si no se especifica el valor del parámetro, automáticamente se borrará la línea completa. En cualquier otro caso, se borrarán el número de posiciones especificadas por el mismo.

4. ¿Dónde está el error en el siguiente prototipo, que hace uso de un argumento implícito?

   ```
   int f(int count, int max = count);
   ```

5.5. SOBRECARGA Y AMBIGÜEDAD

Cuando se utiliza la sobrecarga de funciones es posible que en los programas se introduzca ambigüedad. La ambigüedad provocada por la sobrecarga puede generarse por la conversión de tipos, los parámetros por referencia y los argumentos implícitos. Además, ciertos tipos de ambigüedad se deben a las propias funciones de sobrecarga. Otros, son consecuencia del modo en el que son llamadas las funciones sobrecargadas. Para que un programa compile sin errores debe eliminarse la ambigüedad.

EJEMPLOS

1. Uno de los tipos más normales de ambigüedad está originado por las reglas de conversión automática entre tipos. Como es sabido, cuando se llama a una función con un argumento que es de un tipo compatible (pero no el mismo) con el parámetro que se va a pasar, el tipo del argumento se convierte automáticamente al tipo final. A menudo este hecho se califica como *promoción de tipo* y es perfectamente válido. De hecho, es esta clase de conversión de tipos la que permite que una función como **putchar( )** sea llamada con un carácter, a pesar de que su argumento se especifica como un **int**. Sin embargo, en algunos casos, esta conversión automática de tipos, si la función está sobrecargada, provocará una situación ambigua. Para entenderlo, examine este programa:

```
// Este programa contiene un error de ambigüedad.
#include <iostream.h>

float f(float i)
{
  return i / 2.0;
}

double f(double i)
{
  return i / 3.0;
}

main()
{
  float x = 10.09;
  double y = 10.09;

  cout << f(x); // no ambiguo - utilizar f(float)
  cout << f(y); // no ambiguo - utilizar f(double)

  cout << f(10); // ambiguo, ¿conversión de 10 a doble o a
  //real

  return 0;
}
```

Como indican los comentarios en **main( )**, el compilador es capaz de seleccionar la versión correcta de **f( )** cuando es llamada con una variable **float** o **double**. Sin embargo, ¿qué sucede cuando es llamada con un entero? ¿El compilador llama a **f(float)** o a **f(double)**? (¡Ambas conversiones son correctas!) En cualquiera de los casos, es igualmente correcto transformar un entero a un **float** o a un **double**. Es así como se crea una situación ambigua.

Este ejemplo señala también que se puede introducir ambigüedad en la llamada a una función sobrecargada. El hecho es que no existe ambigüedad inherente a las versiones sobrecargadas de **f( )**, siempre que la llamada se efectúe con argumentos que no sean ambiguos.

2. A continuación se muestra otro ejemplo de sobrecarga de funciones que no es ambigua en sí misma. Sin embargo, cuando se realiza la llamada con el tipo de argumento erróneo, las reglas de conversión automática de C++ dan lugar a una situación ambigua.

```
// Este programa es ambiguo.
#include <iostream.h>

void f(unsigned char c)
{
  cout << c;
  }

void f(char c)
{
  cout << c;
}

main()
{
  f('c');
  f(86); // ¿¿¿a qué versión de f() se llama???

  return 0;
}
```

En este caso, cuando se llama a **f( )** con la constante numérica 86, el compilador no sabe si llamar a **f(unsigned char)** o a **f(char)**. Cualquier conversión sería correcta produciéndose, por tanto, una situación ambigua.

3. Cuando se intenta sobrecargar funciones en las que la única diferencia está en que una utiliza un parámetro por referencia y la otra un parámetro implícito por valor, se produce otro tipo de ambigüedad. Debido a la sintaxis formal de C++ , no es posible que el compilador sepa a qué función llamar. Recuerde que no hay diferencia sintáctica entre llamar a una función que toma un parámetro por valor y llamar a una función que toma un parámetro por referencia. Por ejemplo:

```
// Un programa ambiguo.

int f(int a, int b)
{
  return a+b;
}

// esto en el fondo es ambiguo
int f(int a, int &b)
{
  return a-b;
}

main()
```

```
{
  int x=1, y=2;

  cout << f(x, y); // ¿¿¿a qué versión de f() se llama???

  return 0;
}
```

En este caso, **f(x,y)** es ambigua porque podría llamarse a cualquier versión de la función. De hecho, el compilador dará un error, incluso si se especifica esta sentencia, porque la sobrecarga de las dos funciones es ambigua y no podría resolverse ninguna referencia a las mismas.

4. Se produce un nuevo tipo de ambigüedad cuando se sobrecarga una función en la que una o más funciones sobrecargadas utilizan un argumento implícito. Considere este programa:

```
// Ambigüedad basada en argumentos implícitos y sobrecarga.
#include <iostream.h>

int f(int a)
{
  return a*a;
}

int f(int a, int b = 0)
{
  return a*b;
}

main()
{
  cout << f(10,2); // llamada a f(int, int)
  cout << f(10); // ambiguo - ¿¿¿llamada a f(int) o a
  //f(int, int)???

  return 0;
}
```

De nuevo puede verse que la llamada a la función *no* es en el fondo ambigua. La llamada a **f(10,2)** es perfectamente aceptable y carente de ambigüedad. Sin embargo, el compilador no tiene forma de saber si la llamada **f(10)** está invocando a la primera versión de **f( )** o a la segunda con el valor de **b** implícito.

EJERCICIOS

1. Intente compilar cada uno de los anteriores programas ambiguos. Recuerde los tipos de mensajes de error que se producen. Esto le ayudará a reconocer los errores de ambigüedad cuando aparezcan en sus propios programas.

5.6. BUSQUEDA DE LA DIRECCION DE UNA FUNCION SOBRECARGADA

Para finalizar con este capítulo, le enseñaremos a encontrar la dirección de una función sobrecargada. Como ocurre en C, es posible asignar la dirección de una función (esto es, su punto de entrada) a un puntero y acceder a la misma a través del puntero. La dirección de una función se obtiene poniendo su nombre en el lado derecho de una sentencia de asignación, sin paréntesis ni argumentos. Por ejemplo, si **zap( )** es una función, de acuerdo a lo descrito, éste es un método correcto de asignar **p** a la dirección de **zap( )**:

```
p = zap;
```

En C, puede utilizarse cualquier tipo de puntero para apuntar a una función porque sólo existe una función a la que puede apuntarse. Sin embargo, en C++, la situación es un poco más complicada porque una función puede estar sobrecargada. Por ello, deberá existir un mecanismo que determine cuál es la dirección de la función obtenida.

La solución es eficaz y elegante. Cuando se trata de obtener la dirección de una función sobrecargada, es el modo de declarar el puntero el que determina la dirección de la función sobrecargada. En el fondo, la declaración del puntero se compara con las de las funciones sobrecargadas. La función cuya declaración coincide determina la dirección que va a ser utilizada.

EJEMPLOS

1. A continuación se presenta un programa que contiene dos versiones de una función llamada **space( )**. La primera versión obtiene el número **count** de espacios en la pantalla. La segunda versión obtiene el número **count** de cualquier tipo de carácter que se pase en **ch**. En **main( )**, se declaran dos punteros a función. El primero se especifica como un puntero a una función que tiene sólo un parámetro entero. El segundo se declara como un puntero a una función que tiene dos parámetros.

```
/* Presentación de la asignación de punteros a función
   para funciones sobrecargadas. */
#include <iostream.h>
// Obtención del número count de espacios.
void space(int count)
{
  for( ; count; count--) cout << ' ';
}

// Obtención del número count de caracteres.
void space(int count, char ch)
{
  for( ; count; count--) cout << ch;
```

```
}

main()
{
  /* Creación de un puntero a una función void con
     un parámetro int. */
  void (*fp1)(int);

  /* Creación de un puntero a una función void con
     un parámetro int y un parámetro carácter. */
  void (*fp2)(int, char);

  fp1 = space; // obtiene la dirección de space(int)

  fp2 = space; // obtiene la dirección de space(int,
   char)

  fp1(22); // cuenta 22 espacios
  cout << "|\n";

  fp2(30, 'x'); // cuenta 30 x
  cout << "|\n";

  return 0;
}
```

Como se indica en los comentarios, el compilador es capaz de determinar la función sobrecargada de la que obtener la dirección de base a partir de la cual se declaran **fp1** y **fp2**.

A modo de resumen: Cuando se asigna la dirección de una función sobrecargada al puntero de una función, es la declaración del puntero la que determina la dirección de la función que va a ser asignada. Además, la declaración del puntero a función debe coincidir exactamente con una y sólo una de las funciones sobrecargadas. Si así no fuera, se produce ambigüedad, dando lugar a errores en tiempo de compilación.

EJERCICIOS

1. Las dos siguientes funciones están sobrecargadas. Muestre cómo obtener la dirección de cada una de ellas.

```
int dif(int a, int b)
{
  return a-b;
}

float dif(float a, float b)
{
  return a-b;
}
```

COMPROBACION DE APTITUD SUPERIOR

Al llegar a este punto, debería ser capaz de responder a estas preguntas y de realizar estos ejercicios.

1. Sobrecargue el constructor **date( )** de la Sección 5.1, Ejemplo 3, de manera que acepte un parámetro de tipo **time_t**. (Recuerde que **time_t** es un tipo definido mediante funciones estándar de tiempo y fechas, que se encuentran en la librería de su compilador C++ .)
2. ¿Cuál es el error presente en el siguiente fragmento de código?

```
class samp {
  int a;
public:
  samp(int i) { a = i; }
   // ...
};

// ...

main()
{
  samp x, y(10);

  // ...
}
```

3. Dé dos razones por las que pueda querer (o necesitar) sobrecargar el constructor de una clase.
4. ¿Cuál es la forma general de un constructor de copias?
5. ¿Qué tipo de operaciones se efectúan cuando se invoca un constructor de copias?
6. Explique brevemente qué hace la palabra clave **overload** y por qué ya no se necesita.
7. Explique en pocas palabras lo que es un argumento implícito.
8. Construya una función denominada **reverse( )** que tenga dos parámetros. El primer parámetro, llamado **str**, es un puntero a una cadena que se invierte cuando finaliza la función. El segundo parámetro, llamado **count**, especifica el número de caracteres de **str** a invertir. Déle a **count** un valor implícito, de modo que, cuando aparezca, indique a **reverse( )** que invierta la cadena completa.
9. ¿Cuál es el error del siguiente prototipo?

```
char *wordwrap(char *str, int size=0, char ch);
```

10. Explique algunas de las formas en las que puede introducirse ambigüedad cuando se sobrecargan funciones.
11. ¿Cuál es el error en el siguiente fragmento de código?

```
void compute(double *num, int divisor=1);
void compute(double *num);
// ...
compute(&x);
```

12. Cuando se asigna la dirección de una función sobrecargada a un puntero, ¿qué es lo que determina la versión de la función que va a ser utilizada?

COMPROBACION DE APTITUD INTEGRADA

Esta sección comprueba cómo ha asimilado el contenido de este capítulo con el de los anteriores.

1. Construya una función denominada **order( )** que tenga dos parámetros enteros por referencia. Si el primer argumento es mayor que el segundo, intercámbielos. En cualquier otra situación, la función no debe hacer nada. Esto es, ordene los dos argumentos usados para llamar a **order( )** de modo que, cuando la función finalice, el primer argumento sea menor que el segundo. Por ejemplo, dado

```
int x=1, y=0;
order(x, y);
```

x sea 0 e y sea 1.

2. ¿Por qué las dos siguientes funciones sobrecargadas son esencialmente ambiguas?

```
int f(int a);
int f(int &a);
```

3. Explique la relación entre el uso de un argumento implícito y la sobrecarga de funciones.
4. Dada la siguiente clase parcial, añada las funciones constructoras necesarias de modo que las dos declaraciones en **main( )** sean correctas. (Sugerencia: Es necesario sobrecargar **samp( )** dos veces.)

```
class samp {
  int a;
public:
  // inclusión de las funciones constructoras
  int get_a() { return a; }
};

main()
{
  samp ob(88); // inicialización de ob a 88
  samp obarray[10]; // el elemento 10 del array sin inicializar

  // ...
}
```

5. Explique brevemente por qué se necesitan los constructores de copias.

6 Introducción a la sobrecarga de operadores

OBJETIVOS DEL CAPITULO

Este capítulo introduce otra importante característica de C++: el operador de sobrecarga. Esta característica permite definir el significado de los operadores de C++ con relación a las clases creadas. A través de los operadores de sobrecarga de una clase pueden añadirse nuevos tipos de datos a un programa sin demasiado esfuerzo.

COMPROBACION DE APTITUD

Antes de continuar, debe ser capaz de responder a estas preguntas y de realizar estos ejercicios.

1. Demuestre cómo sobrecargar el constructor de la siguiente clase de modo que también puedan crearse objetos sin inicializar. (Cuando construya objetos sin inicializar, déle a **x** y **a** y el valor 0.)

```
class myclass {
  int x, y;
public:
  myclass(int i, int j) { x=i; y=j; }
  // ...
};
```

2. Haciendo uso de la clase de la primera pregunta, demuestre cómo se puede evitar la sobrecarga de **myclass( )** mediante argumentos implícitos.
3. ¿Cuál es el error de la siguiente declaración?

```
int f(int a=0, double balance);
```

4. ¿Dónde está el error de estas dos funciones sobrecargadas?

```
void f(int a);
void f(int &a);
```

5. ¿Cuándo es adecuado utilizar argumentos implícitos? ¿Cuándo es probablemente una mala solución?
6. Dada la siguiente definición de clase, ¿es posible asignar dinámicamente un array de estos objetos?

```
class test {
  char *p;
  int *q;
  int count;
public:
  test(char *x, int *y, int c) {
    p = x;
    q = y;
  count = c;
  }
  // ...
};
```

7. ¿Qué es un constructor de copias y en qué circunstancias se utiliza?

6.1. FUNDAMENTOS DE LA SOBRECARGA DE OPERADORES

La sobrecarga de operadores es similar a la sobrecarga de funciones. De hecho, la sobrecarga de un operador es realmente un tipo de sobrecarga de función. Sin embargo, se aplican algunas reglas adicionales. Por ejemplo, un operador siempre se sobrecarga con relación a una clase. Las diferencias restantes se discutirán a medida que surjan.

Cuando se sobrecarga un operador, el operador no pierde su contenido original. En contrapartida, gana un contenido adicional relacionado con la clase para la que se definió.

Para sobrecargar un operador se crea una *función operadora*. Lo más normal es que una función operadora sea un miembro o una amiga de la clase para la que se define. Sin embargo, hay una ligera diferencia entre una función operadora miembro y una función operadora amiga. La primera parte de este capítulo discute la creación de funciones operadoras atributo. A continuación se discuten las funciones operadoras amigas.

La forma general de una función operadora miembro es la siguiente:

```
return-type class-name::operator#(arg-list)
{
   // operación que se va a realizar
}
```

A menudo, el tipo devuelto por una función operadora es la clase para la que se define. (Sin embargo, una función operadora es libre de devolver cualquier tipo.) El operador que va a ser sobrecargado se sustituye por #. Por ejemplo, si va a sobrecargarse el operador +, el nombre de la función debería ser **operator**+. El contenido de *arg-list* varía dependiendo del modo de implementación de la función operadora y del tipo del operador que se va a sobrecargar.

Hay que recordar dos importantes restricciones cuando se sobrecarga un operador. La primera es que no puede cambiarse la precedencia del operador. La segunda es que el número de operandos que tiene un operador no puede modificarse. Por ejemplo, el operador / no puede sobrecargarse para que tenga sólo un operando.

La mayoría de los operadores de C++ pueden estar sobrecargados. Los únicos operadores que no se pueden sobrecargar son:

```
. :: .* ?
```

Tampoco pueden sobrecargarse los operadores del preprocesador. (El operador .* es muy especializado y está fuera del alcance de este libro.)

Recuerde que C++ define muchos operadores, incluyendo los operadores subscritos [ ] y los operadores de llamada a función. Sin embargo, este capítulo se concentra en la sobrecarga de los operadores utilizados más comúnmente.

Excepto para el =, las funciones operadoras se heredan de alguna clase derivada. No obstante, una clase derivada puede sobrecargar cualquier operador que elija (incluyendo los sobrecargados por la clase base) relacionado con ella.

Usted ya ha utilizado dos operadores sobrecargados: << y >>. Estos operadores han sido sobrecargados para llevar a cabo las funciones de la consola de E/S. Como ya se mencionó, la sobrecarga de estos operadores para realizar E/S no impide que puedan realizar sus funciones tradicionales de desplazamiento a la izquierda y a la derecha.

Mientras que es posible tener una función operadora que realice *cualquier* actividad —esté o no relacionada con el uso normal del operador— es mejor que las acciones de un operador sobrecargado se ajusten al uso normal de ese operador. Cuando se crean operadores sobrecargados que se desvían de este principio, se corre el riesgo de desestructurar sustancialmente el programa realizado. Por ejemplo, la sobrecarga del /, de manera que se escriban 300 veces en un archivo del disco la frase «Me gusta C++», conduce básicamente a una utilización confusa y errónea de la sobrecarga de operadores.

Lo descrito en el párrafo anterior puede no ser siempre cierto, habrá ocasiones en las que se necesite utilizar un operador de forma distinta a su uso normal. Los dos mejores ejemplos son los operadores << y >>, que se sobrecargan para realizar la consola de E/S. No obstante, aún en estos casos, las flechas izquierda y derecha proporcionan una «pista» visual de su significado. Por consiguiente, si se necesita sobrecargar un operador de un modo diferente al estándar, se recomienda hacer el mayor esfuerzo posible para utilizar un operador adecuado.

Un apunte final: las funciones operadoras pueden no tener argumentos implícitos.

6.2. SOBRECARGA DE OPERADORES BINARIOS

Cuando una función operadora atributo sobrecarga un operador binario, la función tendrá sólo un parámetro. Este parámetro contendrá al objeto que esté en el lado derecho del operador. El objeto del lado izquierdo es el que genera la llamada a la función operadora y se pasa implícitamente a través de **this**.

Es importante entender que las funciones operadoras pueden escribirse de modos muy distintos. Los ejemplos siguientes y los del resto del capítulo no muestran todas las posibilidades, pero presentan varias de las técnicas más comunes.

EJEMPLOS

1. El siguiente programa sobrecarga el operador + + con respecto a la clase **coord**. Esta clase se utiliza para guardar las coordenadas X,Y.

```
// Sobrecarga de + relacionada con la clase coord.
#include <iostream.h>
class coord {
  int x, y; // valores coordenados
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  coord operator+(coord ob2);
};

// Sobrecarga de + relacionada con la clase coord.
coord coord::operator+(coord ob2)
{
  coord temp;
  temp.x = x + ob2.x;
  temp.y = y + ob2.y;

  return temp;
}

main()
{
  coord o1(10, 10), o2(5, 3), o3;
  int x, y;

  o3 = o1 + o2; // suma de dos objetos - llamada a operator+()

  o3.get_xy(x, y);
  cout << "(o1+o2) X: " << x << ", Y: " << y << "\n";

  return 0;
}
```

Este programa presenta la siguiente salida:

```
(o1+o2) X: 15, Y: 13
```

Analicemos detenidamente este programa. La función **operator+( )** devuelve un objeto de tipo **coord**, que guarda la suma de las coordenadas X de cada operando en **x** y la suma de las coordenadas Y de cada operando en **y**. Obsérvese que dentro de **operator+( )** se utiliza un objeto temporal, llamado **temp**, para almacenar el resultado y este es el objeto que se devuelve. También hay que mencionar que no se *modifica ningún* operando. La razón de introducir **temp** es fácil de entender. En esta situación (como en la mayoría), el operador + se ha sobrecargado de una manera consistente con su uso aritmético normal. Por tanto, era fundamental que no se modificase ningún operando. Por ejemplo, al sumar 10 + 4, el resultado es 14, pero ni el 10 ni el 4 se modifican. Por ello se necesita un objeto temporal que mantenga el resultado.

La razón de que la función **operator+( )** devuelva un objeto de tipo **coord** es porque permite que el resultado de la suma de objetos **coord** sea utilizado en expresiones posteriores. Por ejemplo, la sentencia

```
o3 = o1 + o2;
```

es correcta sólo porque el resultado de **o1 + o2** es un objeto **coord**, que puede asignarse a **o3**. Si hubiera devuelto un tipo diferente, esta sentencia habría sido errónea. Además, con la devolución de un objeto **coord**, el operador de suma permite una cadena de sumas. Por ejemplo, la siguiente sentencia es válida:

```
o3 = o1 + o2 + o1 + o3;
```

Aunque existirán situaciones en las que se desee una función operadora que lo que devuelve sea diferente del objeto para el que se definió, la mayoría de las veces las funciones operadoras creadas devolverán un objeto de la clase para la que fueron definidas. (La excepción a esta regla aparece cuando se sobrecargan los operadores lógicos y los relacionales. Esta situación se examina en la sección «Sobrecarga de operadores lógicos y relacionales» de este capítulo.)

Una última nota sobre este ejemplo. Debido a que se devuelve un objeto **coord**, la siguiente sentencia también es correcta:

```
(o1+o2).get_xy(x, y);
```

En este caso, se utiliza directamente el objeto temporal devuelto por **operator+( )**. Por supuesto que, una vez ejecutada esta sentencia, el objeto se destruye.

2. La siguiente versión del programa anterior sobrecarga el operador − y el = con relación a la clase **coord**.

```
// Sobrecarga de +, - e = relativa a la clase coord.
#include <iostream.h>

class coord {
  int x, y; // valores coordenados
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  coord operator+(coord ob2);
  coord operator-(coord ob2);
  coord operator=(coord ob2);
};

// Sobrecarga de + relativa a la clase coord.
coord coord::operator+(coord ob2)
{
  coord temp;

  temp.x = x + ob2.x;
  temp.y = y + ob2.y;

  return temp;
}

// Sobrecarga de - relativa a la clase coord.
coord coord::operator-(coord ob2)
{
  coord temp;
```

```
  temp.x = x - ob2.x;
  temp.y = y - ob2.y;

  return temp;
}

// Sobrecarga de = relativa a la clase coord.
coord coord::operator=(coord ob2)
{
  x = ob2.x;
  y = ob2.y;

  return *this; // devolución del objeto que se asigna
}

main()
{
  coord o1(10, 10), o2(5, 3), o3;
  int x, y;

  o3 = o1 + o2; // suma de dos objetos - esto llama a operator+()
  o3.get_xy(x, y);
  cout << "(o1+o2) X: " << x << ", Y: " << y << "\n";

  o3 = o1 - o2; // substracción de dos objetos
  o3.get_xy(x, y);
  cout << "(o1=o2) X: " << x << ", Y: " << y << "\n";

  o3 = o1; // asignación de un objeto
  o3.get_xy(x, y);
  cout << "(o3=o1) X: " << x << ", Y: " << y << "\n";

  return 0;
}
```

La función **operator − ( )** se realiza de forma similar a **operator + ( )**. Sin embargo, muestra un punto crucial cuando se sobrecarga un operador para el que es importante el orden de los operandos. Cuando se construyó la función **operator + ( )**, no importaba el orden de los operandos. (Esto es, A + B es igual a B + A.) La operación de resta, sin embargo, depende del orden. Por tanto, para sobrecargar correctamente el operador de substracción, es necesario restar del operando de la derecha el operando de la izquierda. Puesto que es el operando de la izquierda el que genera la llamada a **operator − ( )**, la resta debe hacerse en el siguiente orden:

```
x - ob2.x;
```

Recuerde *Cuando se sobrecarga un operador binario, el operando izquierdo se pasa implícitamente a la función y el operando derecho se pasa como un argumento.*

Examinemos ahora la función operadora de asignación. Lo primero a observar es que el operando izquierdo (esto es, el objeto al que se asigna el valor) se modifica

mediante la operación. Esto mantiene el significado normal de la asignación. El segundo detalle a destacar es que la función devuelve ***this**. Esto es, la función **operator=( )** devuelve el objeto al que ha sido asignado. La razón de esto es la de permitir que se realicen una serie de asignaciones. Como ya debería saber, en C++, la siguiente sentencia es sintácticamente correcta (y, realmente, muy utilizada):

```
a = b = c = d = 0;
```

Mediante la devolución de ***this**, el operador de asignación sobrecargado permite que los objetos de tipo **coord** puedan utilizarse de un modo similar. Por ejemplo, esto es perfectamente aceptable:

```
o3 = o2 = o1;
```

No debe olvidar que no hay ninguna regla que obligue a una función de asignación sobrecargada a devolver el objeto que recibe la asignación. Sin embargo, si se desea que la función sobrecargada = se comporte, con relación a su clase, del mismo modo que lo haría con los tipos incorporados, debe devolver ***this**.

3. Es posible sobrecargar un operador con relación a una clase, de modo que el operando del lado derecho sea un objeto de un tipo incorporado, como un entero, en vez de la clase de la que la función operadora es un miembro. Por ejemplo, aquí el operador + se sobrecarga para sumar un valor entero y un objeto **coord**:

```
// Sobrecarga de + para ob + int así como ob + ob.
#include <iostream.h>

class coord {
  int x, y; // valores coordenados
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  coord operator+(coord ob2); // ob + ob
  coord operator+(int i); // ob + int
};

// Sobrecarga de + relativa a la clase coord.
coord coord::operator+(coord ob2)
{
  coord temp;

  temp.x = x + ob2.x;
  temp.y = y + ob2.y;

  return temp;
}

// Sobrecarga de + para ob + int
coord coord::operator+(int i)
{
  coord temp;
```

```
  temp.x = x + i;
  temp.y = y + i;
  return temp;
}

main()
{
  coord o1(10, 10), o2(5, 3), o3;
  int x, y;

  o3 = o1 + o2; // suma de dos objetos - llamada a operator+(coord)
  o3.get_xy(x, y);
  cout << "(o1+o2) X: " << x << ", Y: " << y << "\n";

  o3 = o1 + 100; // suma objeto + entero - llama a operator+(int)
  o3.get_xy(x, y);
  cout << "(o1+100) X: " << x << ", Y: " << y << "\n";

  return 0;
}
```

Es importante recordar que cuando se sobrecarga una función operadora miembro, de manera que pueda utilizarse un objeto en una operación que haga uso de un tipo incorporado, el tipo incorporado debe estar en el lado derecho del operador. La razón de ello es fácil de comprender: Es el objeto del lado izquierdo el que genera la llamada a la función operadora, sin embargo, ¿qué ocurre cuando el compilador recibe la siguiente sentencia?

```
o3 = 19 + o1; // int + ob
```

No existe una operación incorporada definida para gestionar la suma de un entero a un objeto. La función sobrecargada **operator+(int i)** funciona sólamente cuando el objeto está en el lado izquierdo. Por tanto, esta sentencia genera un error en tiempo de compilación. (Pronto veremos un modo de resolver esta restricción.)

4. Puede utilizarse un parámetro por referencia en una función operadora. Por ejemplo, éste es un modo perfectamente válido de sobrecargar el operador + con relación a la clase **coord**:

```
// Sobrecarga de + relativa a la clase coord utilizando referencias.

coord coord::operator+(coord &ob2)
{
  coord temp;

  temp.x = x + ob2.x;
  temp.y = y + ob2.y;

  return temp;
}
```

Una razón para usar un parámetro por referencia en una función operadora es la eficiencia. El paso de objetos como parámetros a funciones conlleva, a menudo, un elevado coste y consume una cantidad importante de ciclos de CPU. Sin embargo, siempre es más rápido y eficiente pasar la dirección de un objeto. Si el operador se va a utilizar frecuentemente, el uso de un parámetro por referencia mejorará significativamente el rendimiento.

Otro motivo para usar un parámetro por referencia es el de evitar el problema originado en la destrucción de la copia de un operando. Como ya se ha indicado en capítulos previos, cuando se pasa un argumento por valor se hace una copia del mismo. Si el objeto tiene una función destructora, cuando la función finaliza, se llama al destructor de la copia. En algunos casos, es posible que el destructor destruya algo necesitado por el objeto invocador. Si así ocurriera, el hecho de utilizar un parámetro por referencia en lugar de un parámetro por valor es un miembro sencillo (y eficiente) de contrarrestar el problema. No hay que olvidar, no obstante, que también podría definirse un constructor de copias que, en el caso general, evitaría este problema.

EJERCICIOS

1. Sobrecargue los operadores * y / con relación a la clase **coord**. Demuestre que ambos funcionan.
2. ¿Por qué razón el siguiente fragmento de código hace un uso inapropiado de un operador sobrecargado?

```
coord coord::operator%(coord ob)
{
  double i;
.
  cout << "Introduzca un número: ";
  cin >> i;
  cout << "la raíz de " << i << " es ";
  cout << sqr(i);
}
```

3. Estudie qué es lo que sucede si modifica el tipo devuelto por las funciones operadoras para que devuelvan tipos diferentes a **coord**. Observe los errores producidos.

6.3. SOBRECARGA DE LOS OPERADORES LOGICOS Y RELACIONALES

Es posible sobrecargar los operadores lógicos y relacionales. Cuando se sobrecargan dichos operadores, para que se comporten normalmente, no se deseará que las funciones operadoras devuelvan un objeto de la clase para la que fueron definidas. En lugar de ello, devolverán un entero que indique verdadero o falso. Esto no sólo permite a estas funciones operadoras devolver un valor verdadero/falso, sino que también posibilita que los operadores se integren en expresiones lógicas y relacionales más extensas que admitan otros tipos de datos.

EJEMPLO

1. En el próximo programa se han sobrecargado los operadores == y &&:

```
// Sobrecarga de == y && relativa a la clase coord.
#include <iostream.h>

class coord {
  int x, y; // valores coordenados
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  int operator==(coord ob2);
  int operator&&(coord ob2);
};

// Sobrecarga de == para coord.
int coord::operator==(coord ob2)
{
  if(x==ob2.x && y==ob2.y) return 1;
  else return 0;
}

// Sobrecarga de && para coord.
int coord::operator&&(coord ob2)
{
  return ((x && ob2.x) && (y && ob2.y));
}

main()
{
  coord o1(10, 10), o2(5, 3), o3(10, 10), o4(0, 0);

  if(o1==o2) cout << "o1 igual a o2\n";
  else cout << "o1 y o2 son diferentes \n";

  if(o1==o3) cout << "o1 igual a o3\n";
  else cout << "o1 y o3 son diferentes\n";

  if(o1&&o2) cout << "o1 && o2 es verdadero\n";
  else cout << "o1 && o2 es falso\n";

  if(o1&&o4) cout << "o1 && o4 es verdadero\n";
  else cout << "o1 && o4 es falso\n";

  return 0;
}
```

EJERCICIO

1. Sobrecargue los operadores < y > con relación a la clase **coord**.

6.4. SOBRECARGA DE UN OPERADOR UNARIO

La sobrecarga de un operador unario es similar a la sobrecarga de un operador binario, excepto que sólo hay un operando del que ocuparse. Cuando se sobrecarga un operador unario utilizando un miembro, el miembro no tiene parámetros. Puesto que sólo hay un operando, es el operando el que genera la llamada a la función operadora. No es necesario otro parámetro.

EJEMPLOS

1. El siguiente programa sobrecarga el operador de incremento (++) con relación a la clase **coord**:

```
// Sobrecarga de ++ relativa a la clase coord.
#include <iostream.h>

class coord {
  int x, y; // valores coordenados

public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  coord operator++();
};

// Sobrecarga de ++ para la clase coord.
coord coord::operator++()
{
  x++;
  y++;

  return *this;
}

main()
{
  coord o1(10, 10);
  int x, y;

  ++o1; // incremento de un objeto
  o1.get_xy(x, y);
  cout << "(++o1) X: " << x << ", Y: " << y << "\n";

  return 0;
}
```

Debido a que el operador de incremento ha sido diseñado para aumentar su operando en una unidad, el operador ++ sobrecargado modifica el objeto sobre el que trabaja. La función también devuelve el objeto que incrementa. Esto permite que el operador de incremento sea utilizado en una sentencia más larga, como ésta:

```
o2 = o1++;
```

Al igual que con los operadores binarios, no hay ninguna regla que obligue a sobrecargar un operador unario para que refleje su significado normal. Sin embargo, en la mayoría de las ocasiones, será esto precisamente lo que se desee hacer.

2. En versiones previas de C++ cuando se sobrecargaba un operador de incremento o de decremento, no había modo de determinar si el ++ o el -- sobrecargados precedían o seguían a su operando. De acuerdo con el programa anterior, estas dos sentencias serían idénticas:

```
o1++;
++o1;
```

Sin embargo, la especificación actual de C++ (incluyendo el estándar ANSI C++ propuesto) ha definido un miembro mediante el que el compilador puede distinguir entre estas dos sentencias. Para realizar esto, se crean dos versiones de la función **operator++()**. La primera se define como se mostró en el ejemplo anterior. La segunda se declara de este modo:

```
coord coord::operator++(int notused);
```

Si el ++ precede al operando, se llama a la función **operator++()**. Por el contrario, si el ++ sigue al operando, se utiliza la función **operator++(int notused)**. Si la diferencia entre el incremento o decremento prefijo y postfijo es importante para los objetos de una clase, será necesario realizar ambas funciones operadoras.

Nota *La característica que permite que C++ distinga entre la aplicación prefija o postfija de los operadores de incremento puede no estar admitida por los compiladores más antiguos.*

3. Como es sabido, en C++ el signo menos es un operador unario y binario. Es asombroso el modo de sobrecargar dicho operador para que mantenga ambos usos con relación a la clase creada. La solución es, en realidad, bastante sencilla: simplemente se sobrecarga dos veces, una vez como operador binario y otra vez como operador unario. Este programa describe la manera de hacerlo:

```
// Sobrecarga de - relativa a la clase coord.
#include <iostream.h>

class coord {
  int x, y; // valores coordenados
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
```

```
  coord operator-(coord ob2); // menos binario
  coord operator-(); // menos unario
};

// Sobrecarga de - relativa a la clase coord.
coord coord::operator-(coord ob2)
{
  coord temp;

  temp.x = x - ob2.x;
  temp.y = y - ob2.y;

  return temp;
}

// Sobrecarga unaria de - para la clase coord.
coord coord::operator-()
{
  x = -x;
  y = -y;
  return *this;
}

main()
{
  coord o1(10, 10), o2(5, 7);
  int x, y;

  o1 = o1 - o2; // resta
  o1.get_xy(x, y);
  cout << "(o1-o2) X: " << x << ", Y: " << y << "\n";

  o1 = -o1; // negación
  o1.get_xy(x, y);
  cout << "(-o1) X: " << x << ", Y: " << y << "\n";

  return 0;
}
```

Como puede observarse, cuando se sobrecarga el menos como un operador binario, sólo tiene un parámetro. Cuando se sobrecarga como un operador unario tiene dos parámetros. Esta diferencia en el número de parámetros es lo que permite que el menos se sobrecargue para ambas operaciones. Según indica el programa, cuando el signo menos se utiliza como un operador binario se llama a la función **operator – (coord ob2)**. Cuando se utiliza el menos unario se llama a la función **operator – ( )**.

EJERCICIOS

1. Sobrecargue el operador `--` para la clase **coord**. Construya la forma prefija y postfija.

2. Sobrecargue el operador + para la clase **coord** de manera que pueda utilizarse como operador binario (mostrado anteriormente) y como operador unario. Cuando sea usado como operador unario permita que el + transforme cualquier coordenada negativa en su valor positivo.

6.5. USO DE FUNCIONES OPERADORAS AMIGAS

Como se mencionó al comienzo del capítulo, es posible sobrecargar un operador con relación a una clase utilizando una función en lugar de un miembro. Como es sabido, una función amiga no tiene un puntero **this**. En el caso de un operador binario, esto quiere decir que a una función operadora amiga se le pasan explícitamente ambos operandos. Para operadores unarios, sólo se pasa un operando. El resto de los detalles siguen siendo igual, no hay ningún motivo para utilizar una función amiga en vez de una función operadora miembro, con la única excepción que se describe en los ejemplos.

Recuerde *No puede utilizarse una función amiga para sobrecargar el operador de asignación. El operador de asignación sólo puede sobrecargarse mediante una función operadora miembro.*

EJEMPLOS

1. En este ejemplo se sobrecarga **operator+( )** para la clase **coord** utilizando una función amiga:

```
// Sobrecarga de + relativa a la clase coord utilizando una amiga.

#include <iostream.h>

class coord {
  int x, y; // valores coordenados
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  friend coord operator+(coord ob1, coord ob2);
};

// Sobrecarga de + utilizando una amiga.
coord operator+(coord ob1, coord ob2)
{
  coord temp;

  temp.x = ob1.x + ob2.x;
  temp.y = ob1.y + ob2.y;

  return temp;
}
```

```
main()
{
  coord o1(10, 10), o2(5, 3), o3;
  int x, y;

  o3 = o1 + o2; // suma de dos objetos - esto llama a operator+()
  o3.get_xy(x, y);
  cout << "(o1+o2) X: " << x << ", Y: " << y << "\n";

  return 0;
}
```

Observe que se pasa el operando izquierdo como primer parámetro y que el operando derecho se pasa como segundo parámetro.

2. La sobrecarga de un operador utilizando una amiga ofrece una característica muy importante que no posee un miembro. Mediante el uso de una función operadora amiga, puede permitirse que los objetos sean utilizados en operaciones que contengan tipos incorporados, donde el tipo incorporado está en el lado izquierdo del operador. Como pudo verse previamente en este capítulo, puede sobrecargarse una función operadora miembro binaria, de modo que el operando izquierdo sea un objeto y que el operando derecho sea un tipo incorporado. Pero no es posible utilizar un miembro para conseguir que el tipo incorporado está en el lado izquierdo del operador. Por ejemplo, partiendo de una función operadora miembro sobrecargada, la primera de las siguientes sentencias es correcta, mientras que la segunda no lo es:

```
ob1 = ob2 + 10; // correcta
ob1 = 10 + ob2; // incorrecta
```

Para que todas las sentencias puedan estructurarse como la primera, siempre hay que asegurar que el objeto sea el operando izquierdo y que el tipo incorporado se sitúe en el lado derecho, lo que puede resultar una restricción complicada de llevar a cabo. La solución a este problema es la de la sobrecarga de funciones operadoras amigas y la definición de las dos posibles situaciones.

Como ya es sabido, a una función operadora amiga se le pasan explícitamente ambos operandos. De esta manera es posible definir una función amiga sobrecargada en la que el operando izquierdo sea un objeto y el operando derecho sea cualquier otro tipo de dato. A continuación, se sobrecarga de nuevo el operador, siendo el operando izquierdo de un tipo incorporado y el operando derecho un objeto. El siguiente programa describe este miembro:

```
// Uso de funciones operadoras amigas para incorporar flexibilidad.
#include <iostream.h>

class coord {
  int x, y; // valores coordenados

public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  friend coord operator+(coord ob1, int i);
  friend coord operator+(int i, coord ob1);
};
```

```
// Sobrecarga para ob + int.
coord operator+(coord ob1, int i)
{
  coord temp;

  temp.x = ob1.x + i;
  temp.y = ob1.y + i;

  return temp;
}

// Sobrecarga para entero + objeto.
coord operator+(int i, coord ob1)
{
  coord temp;

  temp.x = ob1.x + i;
  temp.y = ob1.y + i;

  return temp;
}

main()
{
  coord o1(10, 10);
  int x, y;

  o1 = o1 + 10; // objeto + entero
  o1.get_xy(x, y);
  cout << "(o1+10) X: " << x << ", Y: " << y << "\n";

  o1 = 99 + o1; // entero + objeto
  o1.get_xy(x, y);
  cout << "(99+o1) X: " << x << ", Y: " << y << "\n";

  return 0;
}
```

La consecuencia de sobrecargar funciones operadoras amigas en ambas situaciones conduce a que ahora las dos siguientes sentencias sean válidas:

```
o1 = o1 + 10;
o1 = 99 + o1;
```

3. Si desea utilizar una función operadora amiga para sobrecargar los operadores unarios ++ y --, debe pasarse el operando a la función como un parámetro por referencia. Esto se debe a que las funciones amigas no tienen puntero **this**. Recuerde que los operadores de incremento y decremento conllevan la modificación del operando. Sin embargo, si se sobrecargan estos operadores usando una amiga, el operando se pasa como un parámetro por valor. De este modo, cualquier modificación sufrida por el parámetro dentro de la función operadora amiga no afectará al objeto que generó la llamada. Y puesto que, cuando se utiliza una función amiga, no se pasa implícitamente ningún puntero al objeto (esto es, no hay puntero **this**), el operador incremento o decremento no afectan al operando.

Sin embargo, cuando se pasa el operando a una función amiga como un parámetro por referencia, los cambios que tengan lugar dentro de la función amiga afectarán al objeto que genera la llamada. Por ejemplo, a continuación se muestra un programa que sobrecarga el operador ++ usando una función amiga:

```
// Sobrecarga de ++ usando una amiga.
#include <iostream.h>

class coord {
  int x, y; // valores coordenados
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  friend coord operator++(coord &ob);
};

// Sobrecarga de ++ usando una amiga.
coord operator++(coord &ob) // uso de parámetro por referencia
{
  ob.x++;
  ob.y++;

  return ob; // devolución del objeto generador de la llamada
}

main()
{
  coord o1(10, 10);
  int x, y;

  ++o1; // o1 se pasa por referencia
  o1.get_xy(x, y);
  cout << "(++o1) X: " << x << ", Y: " << y << "\n";

  return 0;
}
```

Si se está utilizando un compilador moderno, cuando se emplea una función operadora **amiga** también podrá distinguirse entre la forma prefija y postfija de los operadores incremento y decremento, de la misma manera que ocurría con los miembros. Basta simplemente con añadir, cuando se define la versión postfija, un parámetro entero. A continuación se muestran los prototipos de las versiones prefija y postfija del operador incremento para la clase **coord**.

```
coord operator++(coord &ob); // prefija
coord operator++(coord &ob, int notused); // postfija
```

Si el ++ precede a su operando, se llama a la función **operator++(coord&ob)**. Sin embargo, si el ++ sigue al operando, se utiliza la función **operator++(coord&ob, int notused)**. En este caso, **notused** tendrá el valor 0.

EJERCICIOS

1. Sobrecargue, utilizando funciones amigas, los operadores —y / para la clase **coord**.
2. Sobrecargue la clase **coord** de forma que pueda utilizar objetos **coord** en operaciones en las que se multiplique un valor entero por cada coordenada. Permita que las operaciones utilicen cualquier orden: *ob*int* o *int*ob*.
3. Explique por qué la solución del Ejercicio 2 necesita funciones operadoras amigas.
4. Muestre, utilizando una función amiga, cómo sobrecargar el `--` con relación a la clase **coord**. Defina las formas prefija y postfija.

6.6. UNA VISION MAS DETALLADA DEL OPERADOR DE ASIGNACION

Como ya se ha visto, es posible sobrecargar el operador de asignación con relación a una clase. Cuando se aplica el operador de asignación a un objeto, por defecto, se lleva a cabo una copia exacta del objeto del lado derecho en el objeto del lado izquierdo. Si esto es lo que se necesita, no hay ninguna razón para construir una función operadora **operator=( )**. No obstante, hay situaciones en las que no se necesita una copia exacta. En el Capítulo 3 se vieron algunos ejemplos sobre esto, en el caso de la asignación de memoria a un objeto. En estas situaciones, puede desearse contar con una operación de asignación diferente.

EJEMPLO

1. A continuación se muestra una versión de **strtype** distinta de las presentadas en capítulos precedentes. Esta versión sobrecarga el operador = de modo que, con la operación de asignación, no se sobreescribe el puntero **p**:

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

class strtype {
  char *p;
  int len;
public:
  strtype(char *s);
  ~strtype() {
  cout << "Liberando " << (unsigned) p << '\n';
  delete [] p;
  }
  char *get() { return p; }
  strtype &operator=(strtype &ob);
};

strtype::strtype(char *s)
```

```
{
  int l;

  l = strlen(s);

  p = new char [l];
  if(!p) {
  cout << "Error de asignación\n";
  exit(1);
  }

  len = l;
  strcpy(p, s);
}

// Asignación de un objeto.
strtype &strtype::operator=(strtype &ob)
{
  // análisis de si se necesita más memoria
  if(len < ob.len) { // necesidad de asignar más memoria
    delete [] p;
    p = new char [ob.len];
     if(!p) {
      cout << "Error de asignación\n";
      exit(1);
    }
  }
  len = ob.len;
  strcpy(p, ob.p);
  return *this;
}

main()
{
  strtype a("Hola"), b("gente");

  cout << a.get() << '\n';
  cout << b.get() << '\n';

  a = b; // ahora no se sobreescribe p

  cout << a.get() << '\n';
  cout << b.get() << '\n';

  return 0;
}
```

Como puede verse, el operador de asignación sobrecargado impide que **p** se sobreescriba. Primero comprueba si el objeto del lado izquierdo ha asignado la memoria suficiente para almacenar la cadena que se le asigna. Si así no fuera, se libera la memoria y se asigna otro área. A continuación la cadena se copia en dicha memoria y la longitud se copia en **len.**

Hay que destacar otras dos importantes características de la función **operator=( )**. La primera es que tiene un parámetro por referencia. Esto impide que se haga una copia del objeto del lado derecho de la asignación. De acuerdo a lo dicho en capítulos anteriores, cuando al pasar un objeto a una función se realiza una

copia del mismo, esta copia se elimina cuando finaliza la función. En este caso, la destrucción de la copia llamaría a la función destructor, que liberaría a **p**. Sin embargo, este es el mismo **p** que aún necesita el objeto utilizado como argumento. El uso del parámetro por referencia impide este problema.

La segunda característica importante de la función **operator=( )** es que devuelve una referencia y no un objeto. La razón para ello es la misma que la de usar un parámetro por referencia. Cuando una función devuelve un objeto se crea un objeto temporal que se destruye cuando la función finaliza. Esto implica una llamada al destructor del objeto temporal que da lugar a que se libere **p**, pero el objeto al que se asigna un valor todavía necesita **p** (y la memoria a la que apunta). Por consiguiente, la devolución de una referencia impide que se cree un objeto temporal.

Nota *Como se indicó en el Capítulo 5, la creación de un constructor de copias es otro modo de prevenir los problemas descritos en los dos párrafos precedentes. Pero el constructor de copias puede no ser tan buena solución como la de utilizar un parámetro por referencia y la de devolver un tipo por referencia. Esto se debe a que, en cualquier circunstancia, el empleo de una referencia previene del coste adicional asociado a la copia de un objeto. Como puede observarse, en C++, existen varios miembros para lograr el mismo resultado. Aprender a elegir entre ellos debe ser un cometido de cualquier buen programador de C++.*

EJERCICIO

1. Dada la siguiente declaración de clase, introduzca todos los detalles para crear un tipo array «seguro». Sobrecargue también el operador de asignación de forma que la memoria asignada a cada array no se destruya accidentalmente. (Remítase al Capítulo 4 para recordar cómo se crea un array seguro».)

```
class dynarray {
  int *p;
  int size;
public:
  dynarray(int s);
  int &put(int i);
  int get(int i);
  // creación de la función operator=()
};
```

COMPROBACION DE APTITUD SUPERIOR

Al llegar a este punto, debería ser capaz de responder a estas preguntas y de realizar estos ejercicios.

1. Sobrecargue los operadores >> y << con relación a la clase **coord** de modo que sean correctas los siguientes tipos de operaciones:

```
ob << integer
ob >> integer
```

Compruebe que sus operadores desplazan los valores **x** e **y** la cantidad especificada.

2. Dada la clase

```
class three_d {
  int x, y, z;
public:
  three_d(int i, int j, int k)
  {
  x = i; y = j; z = k;
  }
  three_d() { x=0; y=0; z=0; }
  void get(int &i, int &j, int &k)
  {
  i = x; j = y; k = z;
  }
};
```

sobrecargue para esta clase los operadores +, −, ++ y −−. (Para los operadores de incremento y decremento, haga la sobrecarga sólo en la forma prefija.)

3. Resuelva de nuevo el Ejercicio 2 para que las funciones operadoras utilicen parámetros por referencia en lugar de parámetros por valor. (Sugerencia: Necesitará funciones amigas para los operadores incremento y decremento.)
4. ¿En qué se diferencian las funciones amigas de las funciones operadoras miembro?
5. Explique por qué puede necesitar sobrecargar el operador de asignación.
6. ¿Puede la función **operator=( )** ser una función amiga?
7. Sobrecargue el operador + para la clase **three_d** de modo que acepte los siguientes tipos de operaciones:

```
ob + double;
double + ob;
```

8. Sobrecargue los operadores ==, != y || con relación a la clase **three_d**.

COMPROBACION DE APTITUD INTEGRADA

Esta sección comprueba cómo ha asimilado el contenido de este capítulo con el de los anteriores.

1. Construya una clase **strtype** que admita los siguientes tipos de operadores:

   - concatenación de cadenas utilizando el operador +
   - asignación de cadenas utilizando el operador =
   - comparación de cadenas utilizando <, > y =.

   Utilice cadenas de longitud fija. Es un ejercicio desafiante, pero con algún conocimiento (y con la práctica) debería ser capaz de acometerlo.

7 Herencia

OBJETIVOS DEL CAPITULO

El concepto de herencia ha sido introducido previamente en este libro. Este es el momento de describirlo con mayor profundidad. La herencia es uno de los tres principios de la POO y, como tal, es una importante característica de C++. En C++, la herencia no sólo admite el concepto de clasificación jerárquica; en el Capítulo 10 se demostrará cómo la herencia proporciona el soporte para el polimorfismo, otra característica básica de la POO.

Los temas que se desarrollan en este capítulo incluyen el control del acceso a la clase base y el especificador de acceso **protegido**, la herencia de múltiples clases base, el paso de argumentos a los constructores de la clase base y las clases base virtuales.

COMPROBACION DE APTITUD

Antes de continuar, debería ser capaz de responder a estas preguntas y de realizar estos ejercicios.

1. Cuando se sobrecarga un operador, ¿pierde parte de su funcionalidad?
2. ¿Debe sobrecargarse un operador con relación a una clase?
3. ¿Puede cambiarse la precedencia de un operador sobrecargado? ¿Puede modificarse el número de operandos?
4. Dado el siguiente programa, desarrollado en parte, introduzca las funciones operadoras que se necesiten:

```
#include <iostream.h>

class array {
  int nums[10];
public:
  array();
  void set(int n[10]);
  void show();
  array operator+(array ob2);
  array operator-(array ob2);
  int operator==(array ob2);
};

array::array()
{
  int i;
  for(i=0; i<10; i++) nums[i] = 0;
}
  void array::set(int *n)
{
  int i;

  for(i=0; i<10; i++) nums[i] = n[i];
}
```

```
void array::show()
{
  int i;

  for(i=0; i<10; i++)
  cout << nums[i] << ' ';

  cout << "\n";
}

// Introducción de las funciones operadoras.

main()
{
  array o1, o2, o3;

  int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

  o1.set(i);
  o2.set(i);

  o3 = o1 + o2;
  o3.show();

  o3 = o1 - o3;
  o3.show();

  if(o1==o2) cout << "o1 igual a o2\n";
  else cout << "o1 no es igual a o2\n";

  if(o1==o3) cout << "o1 igual a o3\n";
  else cout << "o1 no es igual a o3\n";

  return 0;
}
```

Consiga que el operador sobrecargado + sume cada elemento de cada operando. Consiga que el operador sobrecargado − reste cada elemento del operando derecho del operando izquierdo. Permita que el operador sobrecargado == devuelva verdadero si los operandos son iguales y falso en cualquier otro caso.

5. Modifique la solución del Ejercicio 4 para sobrecargar los operadores mediante funciones amigas.
6. Utilizando la clase y las funciones de soporte del Ejercicio 4, sobrecargue el operador ++ mediante una función miembro y el operador −− mediante una función amiga. (Sobrecargue sólo las formas prefijas de ++ y −−.)
7. ¿Puede sobrecargarse el operador de asignación utilizando una función amiga?

7.1. CONTROL DEL ACCESO A LA CLASE BASE

Cuando una clase hereda otra, se usa esta forma general:

```
class derived-class-name : access base-class-name {
  // ...
}
```

En ella, *access* es una de estas tres palabras clave: **public, private** o **protected**. Discutiremos el especificador de acceso **protected** en la próxima sección de este capítulo. Los otros dos se describen aquí.

El especificador de acceso determina cuántos elementos de la clase base son heredados por la clase derivada. Cuando el especificador de acceso para la clase base heredada es **public**, todos los atributos públicos de la base se convierten en atributos públicos de la clase derivada. Si el especificador de acceso es **private**, todos los atributos públicos de la clase base pasan a ser atributos privados de la clase derivada. Sea cual sea el caso, cualquier atributo privado de la base sigue siendo privado para ella y es inaccesible para la clase derivada.

Es importante entender que si el especificador de acceso es **private**, los atributos públicos de la base son atributos privados de la clase derivada, pero estos atributos son accesibles para los miembros de la clase derivada.

EJEMPLOS

1. A continuación se muestra una pequeña clase base y una clase derivada que la hereda (como pública):

```
#include <iostream.h>

class base {
  int x;
public:
  void setx(int n) { x = n; }
  void showx() { cout << x << '\n'; }
};

// Herencia pública.
class derived : public base {
  int y;
public:
  void sety(int n) { y = n; }
  void showy() { cout << y << '\n'; }
};

main()
{
  derived ob;
```

```
  ob.setx(10); // miembro de acceso de la clase base
  ob.sety(20); // miembro de acceso de la clase derivada

  ob.showx(); // miembro de acceso de la clase base
  ob.showy(); // miembro de acceso de la clase derivada

  return 0;
}
```

Como se muestra en este programa, puesto que **base** se hereda como pública, los miembros públicos de **base**, ––**setx( )** y **showx( )**––, se convierten en miembros públicos de **derived** y son, por tanto, accesibles desde cualquier parte del programa. En concreto, es lícito llamarlos en **main( )**.

2. Es importante entender que porque una clase derivada herede una clase base como pública no significa que la clase privada tenga acceso a los atributos privados de la clase base. Por ejemplo, la siguiente incorporación que se hace a **derived**, del ejemplo previo, es errónea:

```
class base {
  int x;
public:
  void setx(int n) { x = n; }
  void showx() { cout << x << '\n'; }
};

// Herencia pública - !Existe un error!
class derived : public base {
  int y;
public:
  void sety(int n) { y = n; }

  /* No es posible acceder a los atributos privados de la
  clase base. x es un atributo privado de base y no puede
  utilizarse dentro de derived. */
  void show_sum() { cout << x+y << '\n'; } // ¡Error!

  void showy() { cout << y << '\n'; }
};
```

En este ejemplo, la clase **derived** intenta acceder a **x**, que es un atributo privado de **base**. Sin embargo, éste es un error porque las zonas privadas de la clase base continuan siendo privadas para ella, *independientemente del modo en que fueron heredadas.*

3. En este ejemplo se muestra el programa del Ejemplo 1, pero **base** se hereda de forma privada. Este cambio da lugar a un error en el programa, como se indica en los comentarios:

```
// Este programa contiene un error.
#include <iostream.h>

class base {
  int x;
```

```
public:
  void setx(int n) { x = n; }
  void showx() { cout << x << '\n'; }
};

// Herencia de base como privada.
class derived : private base {
  int y;
public:
  void sety(int n) { y = n; }
  void showy() { cout << y << '\n'; }
};

main()
{
  derived ob;

  ob.setx(10); // ERROR - ahora es privada para la clase derivada
  ob.sety(20); // miembro de acceso de la clase derivada - CORRECTO

  ob.showx(); // ERROR - ahora es privada para la clase derivada
  ob.showy(); // miembro de acceso de la clase derivada - CORRECTO

  return 0;
}
```

De acuerdo a los comentarios de este (incorrecto) programa, tanto **showx( )** como **setx( )** son privados para **derived** y no son accesibles desde fuera de ella.

Es importante entender que **showx( )** y **setx( )** son todavía públicos para **base**, sin importar cómo los herede cualquier clase derivada. Esto significa que un objeto de tipo **base** podría acceder, desde cualquier lugar, a estas funciones.

No obstante, para los objetos de tipo **derived** son funciones privadas. Dado este fragmento de código:

```
base base_ob;

base_ob.setx(1); // es válido porque base_ob es de tipo base
```

la llamada a **setx( )** es válida porque **setx( )** es público para **base**.

4. Como ya se ha enunciado, aunque los atributos públicos de una clase base sean atributos privados de una clase derivada, al heredarse con el especificador **private**, todavía son accesibles *dentro* de la clase derivada. Por ejemplo, a continuación se muestra una versión «fija» del programa anterior:

```
// Este programa es fijo.
#include <iostream.h>

class base {
  int x;
public:
  void setx(int n) { x = n; }
  void showx() { cout << x << '\n'; }
};

// Herencia de base privada.
```

```
class derived : private base {
  int y;
public:
  // setx es accesible dentro de derived
  void setxy(int n, int m) { setx(n); y = m; }
  // showx es accesible dentro de derived
  void showxy() { showx(); cout << y << '\n'; }
};

main()
{
  derived ob;

  ob.setxy(10, 20);

  ob.showxy();

  return 0;
}
```

En esta situación, se accede a las funciones **setx( )** y **showx( )** desde la clase derivada, lo que es perfectamente válido porque son miembros privados de esa clase.

EJERCICIOS

1. Examine este esquema:

```
#include <iostream.h>

class mybase {
  int a, b;
public:
  int c;
  void setab(int i, int j) { a = i; b = j; }
  void getab(int &i, int &j) { i = a; j = b; }
};

class derived1 : public mybase {
// ...
};

class derived2 : private mybase {
// ...
};

main()
{
  derived1 o1;
  derived2 o2;
  int i, j;

// ...
}
```

¿Cuál de las siguientes sentencias, utilizadas en **main( )**, es correcta?

```
A. o1.getab(i, j)
B. o2.getab(i, j);
C. o1.c = 10;
D. o2.c = 10;
```

2. ¿Qué sucede cuando se hereda como público un atributo público? ¿Qué sucede cuando se hereda como privado?
3. Si aún no lo ha hecho, estudie todos los ejemplos de esta sección. Realice algunos cambios en los especificadores de acceso y observe los resultados.

7.2. USO DE ATRIBUTOS PROTEGIDOS

De acuerdo a lo dicho en la sección previa, una clase derivada no tiene acceso a los atributos privados de la clase base. Esto significa que si la clase derivada necesita acceder a algún atributo de la base, dicho atributo debe ser público. No obstante, se darán situaciones en las que se desee que un atributo de una clase base sea privado, pero que, al mismo tiempo, una clase derivada pueda acceder a él. Para lograrlo, C++ incluye el especificador de acceso **protected**.

El especificador de acceso **protected** es equivalente al especificador **private**, con la única excepción de que los atributos protegidos de una clase base son accesibles para los miembros de cualquier clase derivada de esa base. Fuera de la base o de las clases derivadas, no es posible acceder a los atributos protegidos.

El especificador de acceso **protected** puede aparecer en cualquier lugar dentro de la declaración de la clase, aunque se coloca (por omisión) normalmente después de la declaración de los atributos privados y antes de la de los atributos públicos. La forma general completa de la declaración de una clase es:

```
class class-name {
   // atributos privados
protected: // opcional
   // atributos protegidos
public:
   // atributos públicos
};
```

Cuando una clase derivada hereda de una clase base como **público** un atributo protegido, éste se convierte para la clase derivada en un atributo protegido. Si se hereda de la base como **private**, entonces el atributo protegido de la base se convierte en atributo protegido de la clase derivada.

Una clase base también puede ser heredada como **protected** por una clase derivada. Cuando se da este caso, los atributos protegidos y públicos de la clase base pasan a ser atributos protegidos en la clase derivada. (Por supuesto, los atributos privados de la clase base permanecen siendo privados para ella y no son accesibles para la clase derivada.)

El especificador de acceso **protected** también puede utilizarse con estructuras y uniones.

EJEMPLOS

1. Este programa muestra cómo puede accederse a los atributos protegidos, privados y públicos de una clase:

```
#include <iostream.h>

class samp {
  // privado implícitamente
  int a;
protected: // todavía privado con relación a samp
  int b;
public:
  int c;

  samp(int n, int m) { a = n; b = m; }
  int geta() { return a; }
  int getb() { return b; }
};

main()
{
  samp ob(10, 20);

  // ob.b = 99; ¡Error! b está protegida y por tanto privada
  ob.c = 30; // CORRECTO, c es pública

  cout << ob.geta() << ' ';
  cout << ob.getb() << ' ' << ob.c << '\n';

  return 0;
}
```

Como puede verse, la línea comentada no se admite en **main( )** porque **b** está protegida y por tanto todavía es privada para **samp**.

2. El siguiente programa muestra lo que sucede cuando se heredan como públicos atributos protegidos:

```
#include <iostream.h>

class base {
protected: // privado para base
  int a, b; // pero aún accesible para la clase derivada
public:
  void setab(int n, int m) { a = n; b = m; }
};

class derived : public base {
  int c;
  public:
  void setc(int n) { c = n; }
```

```
  // esta función tiene acceso a a y a b desde base
  void showabc() {
  cout << a << ' ' << b << ' ' << c << '\n';
  }
};

main()
{
  derived ob;

  /* a y b no son accesibles aquí porque son
  privadas para las clase base y derivada. */
  ob.setab(1, 2);
  ob.setc(3);

  ob.showabc();

  return 0;
}
```

Puesto que **a** y **b** están protegidas en **base** y son heredadas como públicas por **derived**, pueden ser utilizadas por miembros de **derived**. Más allá de estas dos clases, **a** y **b** son privadas e inaccesibles.

3. Como se ha comentado anteriormente, cuando se hereda una clase base como **protected** los atributos públicos y protegidos de la clase base pasan a ser atributos protegidos de la clase derivada. A continuación se muestra el programa anterior ligeramente modificado, ya que **base** se hereda como **protected** en vez de como **public**:

```
// Este programa no va a compilar.
#include <iostream.h>

class base {
protected: // privado para base
  int a, b; // pero aún accesible para derivada
public:
  void setab(int n, int m) { a = n; b = m; }
};

class derived : protected base { // herencia protegida
  int c;
public:
  void setc(int n) { c = n; }

  // esta función tiene acceso a a y a b desde base
  void showabc() {
  cout << a << ' ' << b << ' ' << c << '\n';
  }
};

main()
{
  derived ob;
```

```
    // ERROR: setab() ahora es un miembro protegido de base.
    ob.setab(1, 2); // setab() no es accesible aquí.

    ob.setc(3);

    ob.showabc();

    return 0;
}
```

Según describen los comentarios, ya que **base** se hereda como **protected** sus elementos protegidos y públicos pasan a ser atributos protegidos de **derived** y por ello son inaccesibles desde **main( )**.

EJERCICIOS

1. ¿Qué ocurre cuando se hereda un atributo protegido como público? ¿Qué ocurre cuando se hereda como privado? ¿Qué ocurre cuando se hereda como protegido?
2. Explique por qué es necesaria la categoría protegida.
3. En el Ejercicio 1 de la Sección 7.1, si **a** y **b** fueran, dentro de **myclass**, atributos protegidos en vez de privados (por omisión), ¿cambiaría alguna de las respuestas a dicho ejercicio?. Si así fuera, ¿cuáles serían ahora?

7.3. CONSTRUCTORES, DESTRUCTORES Y HERENCIA

Es posible que la clase base, la clase derivada o ambas tengan funciones constructoras y/o destructoras. En esta sección se examinan varios aspectos relacionados con esta situación.

Cuando una clase base y una clase derivada tienen funciones constructoras y destructoras, las funciones constructoras se ejecutan en orden descendente. Las funciones destructoras se ejecutan en orden inverso. Esto es, el constructor de la clase base se ejecuta antes que el constructor de la clase derivada. El orden inverso es el seguido por las funciones destructoras: el destructor de la clase derivada se ejecuta antes que el destructor de la clase base.

Si se analiza, tiene sentido que las funciones constructoras se ejecuten en orden descendente. Puesto que la clase base no conoce la existencia de clases derivadas, cualquier inicialización que efectúe es independiente, y posiblemente un prerequisito, de cualquier inicialización realizada por la clase derivada. Por tanto, debe ejecutarse en primer lugar.

Por otra parte, el destructor de una clase derivada debe ejecutarse antes que el destructor de la clase base ya que la clase base sostiene a la clase derivada. Si se ejecutara primero el destructor de la clase base, se destruiría la clase derivada. Es por esto por lo que debe llamarse al destructor de la clase base antes de que el objeto desaparezca.

Hasta este momento en ninguno de los ejemplos precedentes se han pasado argumentos al constructor de una clase base o de una derivada. Sin embargo, es posible hacerlo. Cuando la clase derivada sólo tiene una inicialización, se pasan los argumentos al constructor de la clase derivada en la forma normal. No obstante, si se requiere pasar un argumento al constructor de la clase base es necesario un trabajo mayor. Para llevarlo a cabo se establece un canal de paso de argumentos. En primer lugar, se pasan todos los argumentos necesarios para las clases base y derivada al constructor de la clase derivada. Haciendo uso de una forma expandida de declaración del constructor de la clase derivada se pasan los argumentos a la clase base. La sintaxis de paso de un argumento de la clase derivada a la clase base es ésta:

```
derived-constructor (arg-list) : base(arg-list) {
   // cuerpo del constructor de la clase derivada
}
```

Se admite que tanto la clase base como la clase derivada utilicen el mismo argumento. También es posible que la clase derivada ignore todos los argumentos y sólo los pase a la clase base.

EJEMPLOS

1. A continuación se presenta un pequeño programa que detalla el orden de ejecución de las funciones constructoras y destructoras de la clase base y de la derivada:

```
#include <iostream.h>

class base {
public:
  base() { cout << "Construcción de la clase base\n"; }
  ~base() { cout << "Destrucción de la clase base\n"; }
};

class derived : public base {
public:
  derived() { cout << "Construcción de la clase derivada\n"; }
  ~derived() { cout << "Destrucción de la clase derivada\n"; }
};

main()
{
  derived o;

  return 0;
}
```

Este programa presenta la siguiente salida:

```
Construcción de la clase base
Construcción de la clase derivada
Destrucción de la clase derivada
Destrucción de la clase base
```

Como puede verse, los constructores se ejecutan en orden descendente mientras que los destructores se ejecutan en orden inverso.

2. Este programa muestra el modo de pasar un argumento al constructor de una clase derivada:

```
#include <iostream.h>

class base {
public:
  base() { cout << "Construcción de la clase base\n"; }
  ~base() { cout << "Destrucción de la clase base\n"; }
};

class derived : public base {
  int j;
public:
  derived(int n) {
    cout << "Construcción de la clase derivada\n";
    j = n;
  }
  ~derived() { cout << "Destrucción de la clase derivada\n"; }
  void showj() { cout << j << '\n'; }
};

main()
{
  derived o(10);

  o.showj();

  return 0;
}
```

El argumento se pasa al constructor de la clase derivada de forma normal.

3. En el siguiente ejemplo el constructor de la clase base y de la derivada tienen un argumento. En este caso específico, ambos utilizan el mismo argumento y la clase derivada simplemente pasa el argumento a la clase base.

```
#include <iostream.h>

class base {
  int i;
public:
  base(int n) {
    cout << "Construcción de la clase base\n";
    i = n;
  }
  ~base() { cout << "Destrucción de la clase base\n"; }
  void showi() { cout << i << '\n'; }
};
```

```
class derived : public base {
  int j;
public:
  derived(int n) : base(n) { // paso de arg a la clase base
    cout << "Construcción de la clase derivada\n";
    j = n;
  }

  ~derived() { cout << "Destrucción de la clase derivada\n"; }
  void showj() { cout << j << '\n'; }
};

main()
{
  derived o(10);

  o.showi();
  o.showj();

  return 0;
}
```

Preste especial atención a la declaración del destructor de **derived**. Observe cómo el parámetro **n** (que recibe el argumento inicializado) es utilizado por **derived( )** y pasado a **base( )**.

4. En la mayoría de los casos, las funciones constructoras de las clases base y derivada *no* utilizan el mismo argumento. Cuando se da esta situación y es necesario pasarles uno o más argumentos, deben pasarse al constructor de la clase derivada *todos* los argumentos requeridos tanto por la clase derivada como por la clase base. Después, la clase derivada pasa simplemente a la clase base los argumentos que ella necesite. Por ejemplo, este programa muestra cómo pasar un argumento al constructor de la clase derivada y otro a la clase base:

```
#include <iostream.h>

class base {
  int i;
public:
  base(int n) {
  cout << "Construcción de la clase base\n";
  i = n;
  }
  ~base() { cout << "Destrucción de la clase base\n"; }
  void showi() { cout << i << '\n'; }
};

  class derived : public base {
  int j;
public:
  derived(int n, int m) : base(m) { // paso de arg a la clase base
  cout << "Construcción de la clase derivada\n";
  j = n;
  }
```

```
  ~derived() { cout << "Destrucción de la clase derivada\n"; }
  void showj() { cout << j << '\n'; }
};

main()
{
  derived o(10, 20);

  o.showi();
  o.showj();

  return 0;
}
```

5. Es necesario entender que el constructor de la clase derivada tenga un argumento que sirva para pasar argumentos a la clase base. Si la clase derivada no necesita un argumento, lo ignora y simplemente lo pasa a la clase base. Por ejemplo, en este fragmento de código, **derived( )** no utiliza el parámetro **n**. Sólo se lo pasa a **base( )**:

```
class base {
  int i;
public:
  base(int n) {
    cout << "Construcción de la clase base\n";
    i = n;
  }
  ~base() { cout << "Destrucción de la clase base\n"; }
  void showi() { cout << i << '\n'; }
};

class derived : public base {
  int j;
public:
  derived(int n) : base(n) { // paso de arg a la clase base
  cout << "Construcción de la clase derivada\n";
  j = 0; // n no se utiliza aquí
  }
  ~derived() { cout << "Destrucción de la clase derivada\n"; }
  void showj() { cout << j << '\n'; }
};
```

EJERCICIOS

1. Dado el siguiente esquema, complete la función constructora de **myderived**. Hágalo de manera que pase un puntero a una cadena de inicialización de **mybase**. Incluya, también, en **myderived( )** la inicialización de **len** con la longitud de la cadena.

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

```
class mybase {
  char str[80];
public:
  mybase(char *s) { strcpy(str, s); }
  char *get() { return str; }
};

class myderived : public mybase {
  int len;
public:
  // incluya myderived() aquí
  int getlen() { return len; }
  void show() { cout << get() << '\n'; }
};

main()
{
  myderived ob("hola");

  ob.show();
  cout << ob.getlen() << '\n';

  return 0;
}
```

2. Utilizando el siguiente esquema, construya las funciones constructoras **car( )** y **truck( )** apropiadas. Permita que pasen a **vehicle** los argumentos necesarios. Además, **car( )** debe inicializar **passengers** y **truck( )** a **loadlimit** en la forma especificada por la creación de un objeto.

```
#include <iostream.h>

// Una clase para diferentes tipos de vehículos.
class vehicle {
  int num_wheels;
  int range;
public:
  vehicle(int w, int r)
  {
    num_wheels = w; range = r;
  }
  void showv()
  {
    cout << "Ruedas: " << num_wheels << '\n';
    cout << "Autonomía: " << range << '\n';
  }
};

class car : public vehicle {
  int passengers;
public:
  // incluya aquí el constructor de car()
  void show()
```

```
  {
    showv();
    cout << "Pasajeros: " << passengers << '\n';
  }
};

class truck : public vehicle {
  int loadlimit;
public:
  // incluya aquí el constructor de truck()
  void show()
  {
    showv();
    cout << "límite de carga " << loadlimit << '\n';
  }
};

main()
{
  car c(5, 4, 500);
  truck t(30000, 12, 1200);

  cout << "Coche: \n";
  c.show();
  cout << "\nCamión:\n";
  t.show();

  return 0;
}
```

car( ) y **truck( )** deben declarar los objetos de este modo:

```
car ob(passengers, wheels, range);
truck ob(loadlimit, wheels, range);
```

7.4. HERENCIA MULTIPLE

Existen dos modos en los que una clase derivada puede heredar más de una clase base. El primero, en el que una clase derivada puede ser usada como la clase base de otra clase derivada, creándose una jerarquía de clases multinivel. En este caso, se dice que la clase base original es una clase base *indirecta* de la segunda clase derivada. (No olvide que cualquier clase —sin tener en cuenta cómo fue creada— puede utilizarse como una clase base.) El segundo, en el que una clase derivada puede heredar directamente más de una clase base. En esta situación, se combinan dos o más clases bases para facilitar la creación de la clase derivada. Cuando están involucradas múltiples clases bases pueden surgir diferentes problemas, que se examinan en esta sección.

Cuando se utiliza una clase base para derivar de ella una clase, que será empleada como clase base de otra clase derivada, las funciones constructoras de

las tres clases son llamadas en el orden de derivación. (Esta es una generalización del principio que se estudió previamente en este capítulo.) Las funciones destructoras se llaman en orden inverso. Por tanto, si la clase *D1* hereda la clase *B1* y *D2* hereda *D1*, entonces se llama primero al constructor de *B1*, seguido del de *D1* y por último del de *D2*. Los destructores son llamados en orden inverso.

Cuando una clase derivada hereda directamente múltiples clases bases, utiliza esta declaración expandida:

```
class derived-class-name : access base1,
                           access base2,
                           . . . , access baseN
{
   // ... cuerpo de la clase
}
```

En esta declaración, *base1* hasta *baseN* son los nombres de las clases base y *access* es el especificador del acceso, que puede ser distinto para cada clase base. Cuando se heredan múltiples clases bases, los constructores se ejecutan en el orden, de izquierda a derecha, en el que se especifican las clases bases. Los destructores se ejecutan en orden opuesto.

Cuando una clase hereda múltiples clases bases que tienen constructores con argumentos, la clase derivada les pasa los argumentos que necesitan utilizando la siguiente forma expandida de la función constructora de la clase derivada:

```
derived-constructor(arg-list) : base1(arg-list),
                                base2(arg-list),
                                . . . ,
                                baseN(arg-list)
{
   // cuerpo del constructor de la clase derivada
}
```

En este caso, *base1* hasta *baseN* son los nombres de las clases bases.

Cuando una clase derivada hereda una jerarquía de clases, cada clase derivada de la cadena debe pasar a su clase base predecesora los argumentos que ésta necesite.

EJEMPLOS

1. A continuación se muestra un ejemplo de una clase derivada que hereda una clase derivada de otra clase. Observe cómo se pasan los argumentos desde **D2** hasta **B1** a través de la cadena.

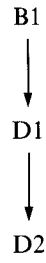
```
// Herencia múltiple
#include <iostream.h>

class B1 {
  int a;
```

```
public:
  B1(int x) { a = x; }
  int geta() { return a; }
};

// Herencia directa de la clase base.
class D1 : public B1 {
  int b;
public:
  D1(int x, int y) : B1(y) // paso de y a B1
  {
   b = x;
  }
  int getb() { return b; }
};

// Herencia de una clase derivada y de una base indirecta.
class D2 : public D1 {
  int c;
public:
  D2(int x, int y, int z) : D1(y, z) // paso de args a D1
  {
   c = x;
  }

  /* Puesto que las bases se heredan como públicas, D2 tiene
   acceso a los elementos públicos de B1 y D1. */
  void show() {
  cout << geta() << ' ' << getb() << ' ';
  cout << c << '\n';
  }
};

main()
{
  D2 ob(1, 2, 3);

  ob.show();
  // geta() y getb() aún son públicos aquí
  cout << ob.geta() << ' ' << ob.getb() << '\n';

  return 0;
}
```

La llamada a **ob.show( )** muestra como salida **3 2 1**. En este ejemplo, **B1** es una clase base indirecta de **D2**. Observe que **D2** tiene acceso a los atributos públicos de **D1** y de **B1**. Como recordará, cuando se heredan los atributos públicos de una clase base, éstos se transforman en atributos públicos de la clase derivada. Por consiguiente, cuando **D1** hereda **B1**, **geta( )** pasa a ser un miembro público de **D1** y de **D2**.

Como muestra el programa, cada clase dentro de una jerarquía de clases debe pasar todos los argumentos necesitados por su clase base precedente. Si así no fuera, se generaría un error en tiempo de compilación.

La jerarquía de clases creada por este programa es la siguiente:

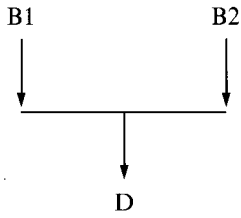


2. A continuación se presenta una nueva versión del programa anterior, en la que una clase derivada hereda directamente dos clases base:

```
#include <iostream.h>

// Creación de la primera clase base.
class B1 {
  int a;
public:
  B1(int x) { a = x; }
  int geta() { return a; }
};

// Creación de la segunda clase base.
class B2 {
  int b;
public:
  B2(int x)
  {
   b = x;
  }
  int getb() { return b; }
};
// Herencia directa de dos clases base.
class D : public B1, public B2 {
  int c;
public:
// aquí, z e y se pasan directamente a B1 y B2.
  D(int x, int y, int z) : B1(z), B2(y)
  {
   c = x;
  }

  /* Puesto que las bases son heredadas como públicas, D
     tiene acceso a los elementos públicos de B1 y B2. */
  void show() {
    cout << geta() << ' ' << getb() << ' ';
    cout << c << '\n';
  }
};
```

```
main()
{
  D ob(1, 2, 3);

  ob.show();

  return 0;
}
```

En esta versión, **D** pasa individualmente los argumentos a las clases **B1** y **B2**. Este programa crea una clase con el siguiente aspecto:



3. El siguiente programa muestra el orden de llamada de las funciones constructoras y destructoras cuando una clase derivada hereda directamente múltiples clases base:

```
#include <iostream.h>

class B1 {
public:
  B1() { cout << "Construcción de B1\n"; }
  ~B1() { cout << "Destrucción de B1\n"; }
};

class B2 {
  int b;
public:
  B2() { cout << "Construcción de B2\n"; }
  ~B2() { cout << "Destrucción de B2\n"; }
};

// Herencia de dos clases base.
class D : public B1, public B2 {
public:
  D() { cout << "Construcción de D\n"; }
  ~D() { cout << "Destrucción de D\n"; }
};

main()
{
  D ob;

  return 0;
}
```

Este programa muestra lo siguiente:

```
Construcción de B1
Construcción de B2
Construcción de D
Destrucción de D
Destrucción de B2
Destrucción de B1
```

Como ya habrá observado, cuando se heredan múltiples clases base de modo directo, los constructores son llamados en el orden, de izquierda a derecha, especificado en la lista de herencia. Los destructores son llamados en orden inverso.

EJERCICIOS

1. ¿Cuál es la salida del siguiente programa? (Intente saberlo sin ejecutar el programa.)

```
#include <iostream.h>

class A {
public:
  A() { cout << "Construcción de A\n"; }
  ~A() { cout << "Destrucción de A\n"; }
};

class B {
public:
  B() { cout << "Construcción de B\n"; }
  ~B() { cout << "Destrucción de B\n"; }
};

class C : public A, public B {
public:
  C() { cout << "Construcción de C\n"; }
  ~C() { cout << "Destrucción de C\n"; }
};

main()
{
  C ob;

  return 0;
}
```

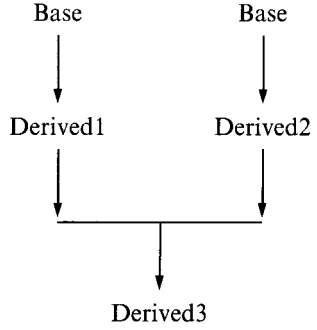
2. Utilizando la siguiente jerarquía de clases, elabore el constructor de **C**, de modo que inicialice **k** y pase los argumentos a **A( )** y a **B( )**.

```
#include <iostream.h>

class A {
```

```
  int i;
public:
  A(int a) { i = a; }
};

class B {
  int j;
public:
  B(int a) { j = a; }
};

class C : public A, public B {
  int k;
public:
  /* Cree C() de modo que inicialice k
  y pase los argumentos a A() y a B() */
};
```

7.5. CLASES BASE VIRTUALES

Cuando una clase derivada hereda múltiples clases base puede surgir algún problema. Para entender el problema, consideremos la siguiente jerarquía de clases:



En este ejemplo, la clase *Base* es heredada por *Derived1* y *Derived2*. *Derived3* hereda directamente *Derived1* y *Derived2*. Sin embargo, esto, en realidad, implica que *Base* es heredada dos veces por *Derived3* —la primera, a través de *Derived1* y, la segunda, a través de *Derived2*. Esto conduce a ambigüedad siempre que *Derived3* haga referencia a un atributo de *Base*. Puesto que se han incluido dos copias de *Base* en *Derived3*, una referencia a un atributo de *Base* ¿se refiere a la *Base* heredada indirectamente a través de *Derived1* o a la heredada indirectamente a través de *Derived2*? Para solucionar este problema de ambigüedad C++ incluye un mecanismo mediante el cual *Derived3* sólo incorpora una copia de *Base*. Esta característica se denomina *clase base virtual.*

En situaciones como las descritas, en las que una clase derivada hereda indirectamente la misma clase base más de una vez, es posible impedir que las

dos copias de la base formen parte del objeto derivado, permitiendo que la clase base sea heredada como **virtual** por cualquier clase derivada. Esto evita que se incluyan dos (o más) copias de la clase base en una clase derivada que herede indirectamente la clase base. Cuando una clase derivada hereda la clase base, la palabra clave **virtual** precede al especificador de acceso de la clase base.

EJEMPLOS

1. Este es un ejemplo que utiliza una clase base virtual para evitar que estén presentes en **derived3** dos copias de **base**:

```
// Este programa utiliza una clase base virtual.
#include <iostream.h>

class base {
public:
  int i;
};

// Herencia de base como virtual.
class derived1 : virtual public base {
public:
  int j;
};

// Aquí también se hereda base como virtual.
  class derived2 : virtual public base {
public:
  int k;
};

/* Aquí, derived3 hereda derived1 y derived2.
  No obstante, sólo aparece una copia de base.
*/
class derived3 : public derived1, public derived2 {
public:
  int product() { return i * j * k; }
};

main()
{
  derived3 ob;

  ob.i = 10; // sin ambiguedad ya que sólo aparece una copia
  ob.j = 3;
  ob.k = 5;

  cout << "El producto es " << ob.product() << '\n';

  return 0;
}
```

Si **derived1** y **derived2** no hubieran heredado **base** como virtual, entonces la sentencia

```
ob.i = 10;
```

sería ambigua y se produciría un error en tiempo de compilación. (Veáse el próximo Ejercicio 1.)

2. Es importante entender que cuando una clase derivada hereda una clase base como virtual, esa clase base todavía existe dentro de la clase derivada. Por ejemplo, partiendo del programa previo, este fragmento de código es correcto:

```
derived1 ob;
ob.i = 100;
```

La única diferencia entre una clase base normal y una virtual se presenta cuando un objeto hereda la base más de una vez. Si se emplean clases base virtuales, entonces sólo se incluye una copia de la clase base en el objeto. En cualquier otro caso, aparecerán múltiples copias.

EJERCICIOS

1. Utilizando el programa del Ejemplo 1, elimine la palabra clave **virtual** e intente compilar el programa. Observe el tipo de errores que se producen.
2. Explique por qué puede ser necesaria una clase base virtual.

COMPROBACION DE APTITUD SUPERIOR

Al llegar a este punto, debería ser capaz de responder a estas preguntas y de realizar estos ejercicios.

1. Construya una clase base genérica llamada **building** que almacene el número de plantas que tiene un edificio, el número de habitaciones y su superficie total. Cree una clase derivada llamada **house** que herede **building** y que almacene también lo siguiente: el número de dormitorios y de baños. Cree, también otra clase derivada, llamada **office** que herede **building** y que almacene además el número de extintores y de teléfonos. Nota: Su solución puede diferir de la respuesta dada en la parte final de este libro. No obstante, si funcionalmente es la misma, considérela correcta.
2. Cuando se hereda un atributo público de una clase base por una clase derivada como público, ¿qué sucede con sus atributos públicos?, ¿qué sucede con sus atributos privados?. Si la clase derivada hereda la clase base como privada, ¿qué sucede con sus atributos públicos y privados?
3. Explique qué significa **protected**. (Hágalo cuando se refiere a atributos de una clase y cuando se utiliza como especificador de acceso de herencia.)
4. Cuando una clase hereda a otra, ¿cuál es el orden de llamada de los constructores de las clases?, ¿y el de los destructores?

5. Dado este esquema, complete lo que falta según indican los comentarios:

```
#include <iostream.h>

class planet {
protected:
  double distance; // millas desde el sol
  int revolve; // en días
public:
  planet(double d, int r) { distance = d; revolve = r; }
};

class earth : public planet {
  double circumference; // perímetro de la órbita
public:
  /* Cree earth(double d, int r). La función debe pasar la
     distancia y los días de giro en su vuelta al planeta.
     Calcule dentro de ella el perímetro de la órbita.
     (Truco: perímetro = 2r*3.1416.)
  */
  /* Cree una función llamada show() que muestre la
   información. */
};

main()
{
  earth ob(93000000, 365);

  ob.show();

  return 0;
}
```

6. Complete el siguiente programa:

```
/* Variación de la jerarquía de vehículos. Pero este
   programa contiene un error. Márquelo. Sugerencia:
   intente compilarlo como está y analice los mensajes de error.
*/
#include <iostream.h>

// Una clase base para varios tipos de vehículos.
class vehicle {
  int num_wheels;
  int range;
public:
  vehicle(int w, int r)
  {
    num_wheels = w; range = r;
  }
  void showv()
  {
    cout << "Ruedas: " << num_wheels << '\n';
```

```
    cout << "Autonomía: " << range << '\n';
  }
};

enum motor {gas, electric, diesel};

class motorized : public vehicle {
  enum motor mtr;
public:
  motorized(enum motor m, int w, int r) : vehicle(w, r)
  {
    mtr = m;
  }
  void showm() {
    cout << "Motor: ";
    switch(mtr) {
      case gas : cout << "Gas\n";
      break;
    case electric : cout << "Eléctrico\n";
      break;
    case diesel : cout << "Diesel\n";
      break;
    }
  }
};

class road_use : public vehicle {
  int passengers;
public:
  road_use(int p, int w, int r) : vehicle(w, r)
  {
    passengers = p;
  }
  void showr()
  {
    cout << "Pasajeros: " << passengers << '\n';
  }
};

enum steering { power, rack_pinion, manual };

class car : public motorized, public road_use {
  enum steering strng;
public:
  car(enum steering s, enum motor m, int w, int r, int p) :
    road_use(p, w, r), motorized(m, w, r), vehicle(w, r)
  {
    strng = s;
  }
  void show() {
    showv(); showr(); showm();
    cout << "Giro: ";
    switch(strng) {
    case power : cout << "Potencia\n";
```

```
        break;
      case rack_pinion : cout << "Rack y Piñón\n";
        break;
      case manual : cout << "Manual\n";
        break;
    }
  }
};

main()
{
  car c(power, gas, 4, 500, 5);

  c.show();

  return 0;
}
```

COMPROBACION DE APTITUD

Esta sección comprueba cómo ha asimilado el contenido de este capítulo con el de los anteriores.

1. En el Ejercicio 6 de la anterior sección Comprobación de aptitud superior, podría haberse producido un mensaje de advertencia (o quizá un mensaje de error) relacionado con el uso de la sentencia **switch** dentro de **car( )** y de **motor( )**. ¿Por qué?
2. Como ya se vió en el capítulo anterior, la mayoría de los operadores sobrecargados dentro de una clase base pueden ser utilizados por la clase derivada. ¿Cuáles no pueden serlo?, ¿Puede dar una razón de ello?
3. Lo siguiente es una versión nueva de la clase **coord** del capítulo anterior. En esta ocasión, se utiliza como base de otra clase llamada **quad**, que contiene también el cuadrante específico en el que se encuentra el punto. Ejecute este programa e intente comprender su salida:

```
/* Sobrecarga de +, - e = con relación a la clase coord.
   A continuación, uso de coord como base para quad.*/
#include <iostream.h>

class coord {
public:
  int x, y; // valores coodenados
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  coord operator+(coord ob2);
  coord operator-(coord ob2);
  coord operator=(coord ob2);
};

// Sobrecarga de + con relación a la clase coord.
coord coord::operator+(coord ob2)
```

```
{
  coord temp;

  cout << "Uso de operator+() de coord\n";

  temp.x = x + ob2.x;
  temp.y = y + ob2.y;

  return temp;
}

// Sobrecarga de - con relación a la clase coord.
coord coord::operator-(coord ob2)
{
  coord temp;

  cout << "Uso de operator-() de coord\n";

  temp.x = x - ob2.x;
  temp.y = y - ob2.y;

  return temp;
}

// Sobrecarga de = con relación a coord.
coord coord::operator=(coord ob2)
{
  cout << "Uso de operator=() de coord\n";

  x = ob2.x;
  y = ob2.y;

  return *this; // devolución del objeto al que ha sido
                // asignado
}

class quad : public coord {
  int quadrant;
public:
  quad() { x = 0; y = 0; quadrant = 0; }
  quad(int x, int y) : coord(x, y)
  {
    if(x>=0 && y>=0) quadrant = 1;
    else if(x<0 && y>=0) quadrant = 2;
    else if(x<0 && y<0) quadrant = 3;
    else quadrant = 4;
  }
  void showq()
  {
   cout << "Punto en cuadrante: " << quadrant << '\n';
  }
  quad operator=(coord ob2);
};
```

```
quad quad::operator=(coord ob2)
{
  cout << "Uso de operator=() de quad\n";

  x = ob2.x;
  y = ob2.y;
  if(x>=0 && y>=0) quadrant = 1;
  else if(x<0 && y>=0) quadrant = 2;
  else if(x<0 && y<0) quadrant = 3;
  else quadrant = 4;

  return *this;
}

main()
{
  quad o1(10, 10), o2(15, 3), o3;
  int x, y;

 o3 = o1 + o2; // suma de dos objetos - llamada a // operator+()

  o3.get_xy(x, y);
  o3.showq();
  cout << "(o1+o2) X: " << x << ", Y: " << y << "\n";

  o3 = o1 - o2; // resta de dos objetos
  o3.get_xy(x, y);
  o3.showq();
  cout << "(o1-o2) X: " << x << ", Y: " << y << "\n";

  o3 = o1; // asignación de un objeto
  o3.get_xy(x, y);
  o3.showq();
  cout << "(o3=o1) X: " << x << ", Y: " << y << "\n";

  return 0;
}
```

4. Transforme el programa mostrado en el Ejercicio 3, de modo que utilice funciones operadoras amigas.

8 *Introducción al sistema de E/S de C++*

OBJETIVOS DEL CAPITULO

Aunque se ha estado empleando E/S de C++ desde el primer capítulo de este libro, es el momento de analizarla con mayor profundidad. Al igual que su predecesor, C, el lenguaje C++ incluye un amplio sistema de E/S, que es, a la vez, flexible y potente. Es importante saber que C++ aún admite el sistema completo de E/S de C. Sin embargo, C++ contiene un conjunto completo de rutinas de E/S orientadas a objeto. La ventaja principal del sistema de E/S de C++ es que puede sobrecargarse con relación a las clases creadas. Dicho de otro modo, el sistema de E/S de C++ permite la integración sencilla dentro del mismo de los nuevos tipos que se creen.

Como en C, el sistema de E/S de C++ orientado a objeto hace una pequeña distinción entre la E/S por consola y la E/S por archivo. La E/S por consola y la E/S por archivo son sólo diferentes perspectivas del mismo mecanismo. Los ejemplos de este capítulo utilizan E/S por consola, pero la información dada también es aplicable a E/S por archivo. (La E/S por archivos se examina en detalle en el Capítulo 9.)

Este capítulo revisa varios aspectos del sistema de E/S de C++, incluyendo E/S formateada, manipuladores de E/S y la creación de insertores y extractores de E/S propios. Como se verá, el sistema de E/S de C++ comparte muchas de las características del sistema de E/S de C.

COMPROBACION DE APTITUD

Antes de continuar, debería ser capaz de responder a estas preguntas y de realizar estos ejercicios.

1. Construya una jerarquía de clases que almacene información sobre aeronaves. Comience con una clase base general, llamada **airship**, que almacene el número de pasajeros que pueden ser transportados y la cantidad de carga (en libras) que puede llevar. A continuación, cree dos clases derivadas, **airplane** y **balloon**, a partir de **airship**. **airplane** debe almacenar el tipo de motor empleado (propulsión o jet) y su rango, en millas. **balloon** debe almacenar información sobre el tipo de combustible utilizado para la ascensión del globo (hidrógeno o helio) y su altitud máxima (en pies). Construya un pequeño programa que muestre esta jerarquía de clases.

   Nota *Su solución, sin lugar a dudas, será ligeramente diferente de la dada en la parte final de este libro. Si funcionalmente es similar, considérela correcta.*

2. ¿Para qué se utiliza **protected**?
3. Dada la siguiente jerarquía de clases, ¿cuál es el orden de llamada de las funciones constructoras? ¿En qué orden se llama a las funciones destructoras?

```
#include <iostream.h>

class A {
public:
```

```
  A() { cout << "Construcción de A\n"; }
  ~A() { cout << "Destrucción de A\n"; }
};

class B : public A {
public:
  B() { cout << "Construcción de B\n"; }
  ~B() { cout << "Destrucción de B\n"; }
};

class C : public B {
public:
  C() { cout << "Construcción de C\n"; }
  ~C() { cout << "Destrucción de C\n"; }
};

main()
{
  C ob;

  return 0;
}
```

4. Dado el siguiente fragmento de código, ¿cuál es el orden de llamada de las funciones constructoras?

```
class myclass : public A, public B, public C { ...
```

5. Complete las funciones constructoras que faltan en este programa.

```
#include <iostream.h>

class base {
  int i, j;
public:
  // aquí se necesita el constructor
  void showij() { cout << i << ' ' << j << '\n'; }
};

class derived : public base {
  int k;
public:
  // aquí se necesita el constructor
  void show() { cout << k << ' '; showij(); }
};

main()
  {
  derived ob(1, 2, 3);
  ob.show();

  return 0;
}
```

6. Generalmente, cuando se define una jerarquía de clases, se comienza con la clase más _____ hasta llegar a la clase más _____. (Rellene las palabras que faltan.)

8.1. ALGUNOS PRINCIPIOS DE E/S EN C++

Antes de comenzar la descripción de la E/S en C++ es necesario realizar algunos comentarios generales.

El sistema de E/S de C++, como el sistema de E/S de C, funciona mediante *flujos*. Dada su experiencia en programación en C, usted ya debe saber lo que es un flujo. No obstante, lo resumiremos en unas pocas frases. Un flujo es un dispositivo lógico que consume o produce información. Un flujo se enlaza con un dispositivo físico mediante el sistema de E/S de C++. Todos los flujos se comportan del mismo modo, el sistema de E/S puede operar virtualmente sobre cualquier tipo de dispositivo. Por ejemplo, puede utilizarse el mismo método para escribir en pantalla que para escribir en un archivo de disco o en la impresora.

Como es sabido, cuando comienza la ejecución de un programa en C, se abren automáticamente tres flujos predefinidos: **stdin**, **stdout** y **stderr**. Lo mismo sucede cuando comienza a ejecutarse un programa en C++. Cuando comienza un programa en C++, se abren automáticamente estos cuatro flujos:

Flujos	Significado	Dispositivo implícito
cin	Entrada estándar	Teclado
cout	Salida estándar	Pantalla
cerr	Error estándar	Pantalla
clog	Versión búfer de cerr	Pantalla

Como seguramente ya habrá adivinado, los flujos **cin**, **cout** y **cerr** se corresponden con los flujos de C **stdin**, **stdout** y **stderr**. **cin** y **cout** ya han sido utilizados. El flujo **clog** es sencillamente una versión en búfer de **cerr**.

Los flujos estándar son usados, por omisión, para la comunicación con la consola. Sin embargo, en entornos que permitan redireccionamiento de E/S, estos flujos pueden redirigirse a otros dispositivos.

De acuerdo a lo aprendido en el Capítulo 1, C++ proporciona el soporte para su sistema de E/S en el archivo cabecera **iostream.h**. En este archivo, se definen las jerarquías de clase para que admitan las operaciones de E/S. Existen dos jerarquías de clase de E/S relacionadas, pero diferentes. La primera deriva de una clase de E/S de bajo nivel llamada **streambuf**. Esta clase admite las operaciones básicas de entrada y salida de bajo nivel y proporciona el substrato para el sistema completo de E/S. A menos que se realice programación avanzada de E/S, no será necesario emplear directamente **streambuf**. La jerarquía de clase con la que se trabaja más normalmente deriva de **ios**. Esta es una clase de E/S

de alto nivel que proporciona formateado, comprobación de errores e información del estado relativas al flujo de E/S. **ios** se emplea como una base para varias clases derivadas, incluyendo **istream**, **ostream** y **iostream**. Estas clases se utilizan para crear flujos que lleven a cabo entrada, salida y entrada/salida, respectivamente.

La clase **ios** contiene muchos miembros y variables que controlan o monitorizan las operaciones fundamentales de un flujo. Se hará, frecuentemente, referencia a ellas. Sólo queda por recordar que, para tener acceso a esta importante clase, debe incluirse **iostream.h** en el programa.

8.2. E/S FORMATEADA

Hasta ahora, todos los ejemplos vistos en este libro han presentado la información en la pantalla, utilizando los formatos implícitos de C++. Sin embargo, es posible presentar la información de múltiples formas. De hecho, puede darse formato a los datos empleando el sistema de E/S de C++, del mismo modo que se utiliza la función **printf( )** de C. También es posible modificar algunos aspectos en la entrada de la información.

Cada flujo de C++ está asociado con un número de indicadores de formato, que determinan la forma de presentación de los datos. Estos indicadores se codifican en un entero largo. (El estándar ANSI C++ sugiere que el nombre del tipo de la variables que contienen los indicadores de formato sea **fmtflags**, pero, actualmente, los compiladores no admiten este tipo.) El nombre y el valor de estos indicadores viene dado en la clase **ios**, que usa normalmente una lista como la siguiente:

```
// indicadores de formato en ios
enum {
  skipws = 0x0001,
  left = 0x0002,
  right = 0x0004,
  internal = 0x0008,
  dec = 0x0010,
  oct = 0x0020,
  hex = 0x0040,
  showbase = 0x0080,
  showpoint = 0x0100,
  uppercase = 0x0200,
  showpos = 0x0400,
  scientific = 0x0800,
  fixed = 0x1000,
  unitbuf = 0x2000,
  stdio = 0x4000
};
```

Generalmente, cuando se fija un indicador de formato, se activa esa característica. Cuando se borra un indicador, se utiliza el formato implícito. A continuación, se describen estos indicadores.

Cuando se fija el indicador **skipws**, al realizar la entrada en un flujo, no se consideran los primeros caracteres de espacio en blanco (espacios, tabulaciones y nuevas líneas). Cuando se borra el valor de **skipws**, se tienen en cuenta los espacios en blanco. En general, sólo será necesario cambiar el valor de este indicador cuando se leen ciertos tipos de archivos de disco.

Cuando se fija el indicador **left**, la salida se ajusta a la izquierda. Cuando se fija el indicador **internal**, se introduce un valor numérico para completar un campo, mediante la inserción de espacios entre cualquier signo o caracteres base. (Describiremos brevemente cómo se especifica la anchura de un campo.) Si no se fija ninguno de estos indicadores, la salida se ajusta implícitamente a la derecha.

Los valores enteros, por omisión, se presentan en decimal. Sin embargo, es posible cambiar la base numérica. Si se da valor al indicador **oct** la salida se presenta en octal. Si se hace con el indicador **hex**, la salida aparece en hexadecimal. Para que la salida vuelva a presentarse en decimal debe fijarse el indicador **dec**.

El indicador **showcase** muestra la base de los valores numéricos. Por ejemplo, si la base de conversión es hexadecimal, el valor 1F aparecerá como 0x1F.

El indicador **showpoint** muestra, para todas las salidas en coma flotante, el punto decimal y un grupo de ceros a continuación —sea o no necesario.

Cuando se utiliza notación científica, por omisión, la «e» se presenta en minúsculas. También, cuando el valor es hexadecimal la «x» aparece en minúsculas. Cuando se fija **uppercase**, estos caracteres aparecen en mayúsculas.

Si se fija el indicador **showpos** aparecerá delante de los valores decimales positivos el signo +.

El indicador **scientific** da lugar a que los valores numéricos en coma flotante se presenten en notación científica. El indicador **fixed** permite que los valores en coma flotante aparezcan en notación normal. Cuando **fixed** se fija, por omisión, se presentan seis posiciones decimales. Cuando no se da valor a ningún indicador, el compilador elige el método apropiado.

Si se fija **unitbuf**, el sistema de E/S de C++ elimina sus flujos de salida después de cada operación de salida. (Compruebe su compilador para conocer los detalles de este indicador de formato.)

Cuando se fija **stdio**, se eliminan **stdout** y **stderr** después de cada operación de salida. Sin embargo, este indicador no está definido por el estándar ANSI C++ y puede no ser aplicable en todos los compiladores.

Para dar un valor a un indicador de formato, debe utilizarse la función **setf( )**. Esta función es un miembro de **ios**. Su forma normal es la siguiente:

```
long setf(long flags);
```

Esta función devuelve los valores previos de los indicadores de formato y activa sólo aquellos especificados mediante *flags*. (El resto de indicadores permanece inalterado.) Por ejemplo, para activar el indicador **showpos** se puede emplear esta sentencia:

```
stream.setf(ios::showpos);
```

stream es el flujo que se desea modificar. Es importante destacar el uso del operador de alcance de resolución. No debe olvidarse que **showpos** es una constante listada dentro de la clase **ios**. Por tanto, es necesario comunicárselo al compilador, precediendo **showpos** con el nombre de la clase y el operador de alcance de resolución. Si no se hiciera así, sencillamente no se reconocería a la constante **showpos**.

Es importante entender que **setf( )** es un miembro de la clase **ios** y afecta a los flujos creados por esa clase. De esa manera, cualquier llamada a **setf( )** se hace en relación con un flujo específico. No existe en sí mismo el concepto de llamada a **setf( )**. Dicho de otro modo, no existe en C++ el concepto de estado de formato global. Cada flujo mantiene su propia información sobre el estado del formato de forma individual.

Es posible fijar más de un indicador en una sola llamada a **setf( )**, en lugar de emplear múltiples llamadas. Para hacerlo, se aplica el operador OR a los valores de los indicadores que se quieran fijar. Por ejemplo, esta única llamada fija para **cout** los indicadores **showbase** y **hex**:

```
cout.setf(ios::showbase | ios::hex);
```

Recuerde *Puesto que los indicadores de formato se definen en la clase* **ios**, *el acceso a sus valores debe hacerse utilizando* **ios** *y el operador de alcance de resolución. Por ejemplo, sólo* **showbase** *no sería reconocido; debe especificarse* **ios::showbase**.

El complementario de **setf( )** es **unsetf( )**. Este miembro de **ios** borra uno o más indicadores de formato. Su prototipo es el siguiente:

```
long unsetf(long flags);
```

Se borran los indicadores especificados por *flags*. (El resto de indicadores permanece inalterado.) Se activan los valores previos de los indicadores.

Existirá alguna ocasión en la que sólo se desee conocer el valor actual de los valores del formato, sin modificar ninguno. Puesto que **setf( )** y **unsetf( )** modifican los valores de uno o más indicadores, en la clase **ios** también se incluye el miembro **flags( )**, que simplemente devuelve el valor de cada indicador de formato codificado dentro de un **long int**. Su prototipo es:

```
long flags( );
```

La función **flags( )** tiene otra forma que permite fijar *todos* los indicadores de formato asociados con un flujo a los valores especificados en el argumento de **flags( )**. El prototipo de esta versión de **flags( )** es el siguiente:

```
long flags(long f);
```

Cuando se utiliza esta versión, el bit patrón encontrado en *f* se copia en la variable utilizada para almacenar los indicadores de formato asociados con el flujo, anulando, de este modo, todos los valores previos de los indicadores. La función devuelve dichos valores previos.

EJEMPLOS

1. A continuación se presenta un ejemplo que muestra el uso de **setf( )**:

```
#include <iostream.h>

main()
{

  // presentación usando los valores implícitos
  cout << 123.23 << " hola " << 100 << '\n';
  cout << 10 << ' ' << -10 << '\n';
  cout << 100.0 << "\n\n";

  // ahora, cambio de formatos
  cout.setf(ios::hex | ios::scientific);
  cout << 123.23 << " hola " << 100 << '\n';

  cout.setf(ios::showpos);
  cout << 10 << ' ' << -10 << '\n';

  cout. setf(ios::showpoint | ios::fixed);
  cout << 100.0;

  return 0;
}
```

Este programa da lugar a la siguiente salida:

```
123.23 hola 100
10 -10
100

1.232300e+02 hola 64
a fffffff6
+100.000000
```

Observe que el indicador **showpos** sólo afecta a la salida decimal. No influye sobre el valor 10 cuando la salida es hexadecimal.

2. El siguiente programa explica el uso de **unsetf( )**. Primero fija los indicadores **uppercase**, **showbase** y **hex**. A continuación muestra 88 utilizando notación científica. En este caso, la «X» empleada en la notación hexadecimal aparece en mayúsculas. Después, borra el indicador **uppercase** utilizando **unsetf( )** y, de nuevo, muestra 88 en hexadecimal. Esta vez la «x» aparece en minúsculas:

```
#include <iostream.h>

main()
{
  cout.setf(ios::uppercase | ios::showbase | ios::hex);
```

```
  cout << 88 << '\n';

  cout.unsetf(ios::uppercase);

  cout << 88 << '\n';

  return 0;
}
```

3. El siguiente programa utiliza **flags( )** para presentar el valor de los indicadores de formato con respecto a **cout**. Preste una atención especial a la función **showflags( )**. Puede que le sea útil en sus propios programas.

```
#include <iostream.h>

void showflags();
main()
{
  // presentación de la condición implícita de los
     indicadores de formato showflags();

  cout.setf(ios::oct | ios::showbase | ios::fixed);

  showflags();

  return 0;
}

// Esta función muestra el estado de los indicadores
   de formato.
void showflags()
{
  long f, i;
  int j;

  char flgs[15][12] = {
    "skipws",
    "left",
    "right",
    "internal",
    "dec",
    "oct",
    "hex",
    "showbase",
    "showpoint",
    "uppercase",
    "showpos",
    "scientific",
    "fixed",
    "unitbuf",
    "stdio"
  };

  f = cout.flags(); // obtención de los valores del indicador
```

```
  // comprobación de cada indicador
  for(i=1, j=0; i<=0x4000; i = i<<1, j++)
    if(i & f) cout << flgs[j] << " activado\n";
    else cout << flgs[j] << " desactivado\n";

  cout << "\n";
}
```

La salida de este programa es:

```
skipws activado
left desactivado
right desactivado
internal desactivado
dec desactivado
oct desactivado
hex desactivado
showbase desactivado
showpoint desactivado
uppercase desactivado
showpos desactivado
scientific desactivado
fixed desactivado
unitbuf desactivado
stdio desactivado

skipws activado
left desactivado
right desactivado
internal desactivado
dec desactivado
oct activado
hex desactivado
showbase activado
showpoint desactivado
uppercase desactivado
showpos desactivado
scientific desactivado
fixed activado
unitbuf desactivado
stdio desactivado
```

4. El próximo programa ilustra la segunda versión de **flags( )**. Primero construye una máscara de indicadores que activa **showpos**, **showcase**, **oct** y **right**. Estos indicadores toman los valores 0x0400, 0x0080, 0x0020 y 0x0004. Cuando se utilizan conjuntamente generan el valor usado en el programa 0x04A4. El resto de los indicadores están desactivados. A continuación, utiliza **flags( )** para almacenar en la variable de los indicadores asociada con **cout** esos valores. La función **showflags( )** comprueba que los indicadores toman los valores previstos. (Es igual que parte de lo realizado en el programa anterior.)

```
#include <iostream.h>

void showflags() ;
```

```
main()
{
  // presentación de la condición implícita de los indicadores
  //de formato showflags();

  // showpos, showbase, oct, right activados, el resto no
  long f = 0x04A4;
  cout.flags(f); // fija todos los indicadores
                 .
  showflags();

  return 0;
}
```

EJERCICIOS

1. Construya un programa que fije los indicadores de **cout** de modo que los enteros lleven delante el signo + cuando tomen valores positivos. Demuestre que ha colocado correctamente los valores de los indicadores de formato.
2. Construya un programa que fije los indicadores de **cout** de modo que se muestre el punto decimal siempre que se presenten valores en coma flotante. También deben mostrarse todos los valores en coma flotante en notación científica, con la E en mayúsculas.
3. ¿Cómo debe ser la llamada a **flags( )** para que todos los indicadores de formato tomen sus valores implícitos?
4. Construya un programa que almacene el estado actual de los indicadores de formato, fije **showbase** y **hex** y que muestre el valor 100. A continuación, los indicadores deben tomar sus valores previos.

8.3. USO DE width( ), precision( ) Y fill( )

Además de los indicadores de formato, hay tres funciones miembro definidas por **ios** que fijan estos parámetros de formato: la anchura del campo, la precisión y el carácter de relleno. Las funciones que llevan esto a cabo son **width( )**, **precision( )** y **fill( )**, respectivamente.

Cuando se presenta un valor, por omisión, éste sólo ocupa el espacio correspondiente al número de caracteres destinados a mostrarlo. Sin embargo, puede especificarse una anchura mínima de campo utilizando la función **width( )**. Su prototipo es:

```
int width(int w);
```

w adopta la nueva anchura del campo y se desestima la anchura de campo anterior. En algunas realizaciones, cada vez que se efectúa una operación de salida, la anchura del campo vuelve a su valor implícito, de modo que puede ser necesario fijar la anchura mínima de campo antes de cada sentencia de escritura.

Tras haber fijado una mínima anchura de campo, cuando un valor utiliza menos espacio que el especificado, el campo se completa con el carácter de relleno actual (por omisión, el espacio) de manera que se cubra la anchura del campo. Sin embargo, no debe olvidarse que si el tamaño del valor presentado excede la mínima anchura de campo, el campo se ampliará. No se trunca ningún valor.

Se utilizan, implícitamente, seis dígitos de precisión. No obstante, este número puede modificarse empleando la función **precision( )**. Su prototipo es el siguiente:

```
int precision(int p);
```

La precisión se fija en *p*, anulándose su valor previo.

Cuando es necesario completar un campo, éste se rellena, por omisión, mediante la función **fill( )**.

```
char fill(char ch);
```

Después de llamar a **fill( )** *ch* pasa a ser el nuevo carácter de relleno y el valor anterior es desestimado.

EJEMPLOS

1. A continuación se presenta un programa que explica el uso de estas funciones:

```
#include <iostream.h>

main()
{
  cout.width(10); // se fija la mínima anchura de campo
  cout << "hola" << '\n'; // ajuste a la derecha, por omisión
  cout.fill('%'); // se fija el carácter de relleno
  cout.width(10); // se fija la anchura
  cout << "hola" << '\n'; // ajuste a la derecha, por omisión
  cout.setf(ios::left); // ajuste a la izquierda
  cout.width(10); // se fija la anchura
  cout << "hola" << '\n'; // salida ajustada a la izquierda

  cout.width(10); // se fija la anchura
  cout.precision(10); // se fijan 10 dígitos de precisión
  cout << 123.234567 << '\n';
  cout.width(10); // se fija la anchura
  cout.precision(6); // se fijan seis dígitos de precisión
  cout << 123.234567 << '\n';

  return 0;
}
```

Este programa muestra la siguiente salida:

```
     hola
%%%%%hola
hola%%%%%
123.234567
123.235%%%
```

Observe que la anchura de campo se fija antes de cada sentencia de salida.

2. El siguiente programa muestra cómo utilizar las funciones de formato de C++ para crear una tabla alineada de números:

```
// Creación de una tabla de raíces cuadradas y de cuadrados.
#include <iostream.h>
#include <math.h>


main()
{
  double x;


  cout.precision(4);
  cout << " x sqrt(x) x^2\n\n";


  for(x = 2.0; x <= 20.0; x++) {
    cout.width(7);
    cout << x << "   ";
    cout.width(7);
    cout << sqrt(x) << "   ";
    cout.width(7);
    cout << x*x << '\n';
  }


  return 0;
}
```

El programa crea la siguiente tabla:

```
      x    sqrt(x)      x^2


      2     1.414        4
      3     1.732        9
      4         2       16
      5     2.236       25
      6     2.449       36
      7     2.646       49
      8     2.828       64
      9         3       81
     10     3.162      100
     11     3.317      121
     12     3.464      144
     13     3.606      169
     14     3.742      196
     15     3.873      225
     16         4      256
     17     4.123      289
     18     4.243      324
     19     4.359      361
     20     4.472      400
```

EJERCICIOS

1. Construya un programa que imprima el logaritmo neperiano y el logaritmo en base 10 de los números 2 hasta el 100. El formato de la tabla debe dejar que los números queden ajustados a la derecha dentro de una anchura de campo de 10, utilizando una precisión de 5 posiciones decimales.
2. Cree una función denominada **center( )** que tenga el siguiente prototipo:

```
void center(char *s);
```

   Esta función debe centrar la cadena especificada en la pantalla. Para realizarlo, utilice la función **width( )**. La pantalla tiene una anchura de 80 caracteres. (Por simplicidad, puede asumirse que ninguna cadena exceda los 80 caracteres.) Construya un programa para comprobar el buen funcionamiento de la función.
3. Haga diferentes pruebas con los indicadores de formato y con las funciones de formato. Una vez que se familiarice con el sistema de E/S de C++, no tendrá ninguna dificultad en darle a la salida el formato que desee.

8.4. USO DE LOS MANIPULADORES DE E/S

Existe otro modo de dar formato a la información utilizando el sistema de E/S. Este método emplea unas funciones especiales denominadas *manipuladores de E/S*. En algunas situaciones, estos manipuladores son más fáciles de utilizar que los indicadores y funciones de formato de **ios**.

Los manipuladores de E/S son funciones especiales de formato de E/S que, a diferencia de los miembros de **ios**, pueden aparecer *dentro* de una sentencia de E/S. Los manipuladores estándar se presentan en la Tabla 8.1. Como puede observarse, muchos de estos manipuladores son semejantes a los miembros de la clase **ios**.

Tabla 8.1. *Manipuladores de E/S en C++*

Manipulador	Propósito	Entrada/Salida
dec	Formato decimal para datos numéricos	Salida
endl	Saca un carácter de nueva línea y borra el flujo	Salida
ends	Saca un nulo	Salida
flush	Borra un flujo	Salida
hex	Formato hexadecimal para datos numéricos	Salida
oct	Formato octal para datos numéricos	Salida
resetiosflags (long *f*)	Desactiva los indicadores especificados en *f*	Entrada y salida
setbase (int *base*)	Fija el número base a *base*	Salida
setfill (int *ch*)	Fija el carácter de relleno a *ch*	Salida
setiosflags (long *f*)	Activa los indicadores especificados en *f*	Entrada y salida
setprecision (int *p*)	Fija el número de dígitos de precisión	Salida
setw (int *w*)	Fija la anchura de campo a *w*	Salida
ws	Salto, espacio en blanco inicial	Entrada

Nota *Para acceder a los manipuladores que tienen parámetros (como* **setw( )***, debe incluirse en el programa* **iomaip.h***. Esto no es necesario cuando el manipulador utilizado no necesita un argumento.*

Como se indicó más arriba, los manipuladores pueden estar en la cadena de las operaciones de E/S. Por ejemplo:

```
cout << oct << 100 << hex << 100;
cout << setw(10) << 100;
```

La primera sentencia le indica a **cout** que muestre enteros en octal y después muestra el número 100 en octal. A continuación, le comunica al flujo que muestre enteros en hexadecimal y después muestra el 100 en formato hexadecimal. La segunda sentencia pone el valor de la anchura de campo a 10 y muestra, de nuevo, el número 100 en formato hexadecimal. Obsérvese que cuando un manipulador no tiene argumentos, como es el caso de **oct** en el ejemplo, no va seguido de paréntesis. Esto se debe a que la dirección del manipulador se pasa al operador sobrecargado <<.

No debe olvidarse que un manipulador de E/S afecta sólo al flujo del que forma parte la expresión de E/S. Los manipuladores de E/S *no* influyen sobre todos los flujos que estén abiertos.

Como sugiere el anterior ejemplo, las principales ventajas del uso de manipuladores frente a los miembros del **ios** está en que, a menudo, son más fáciles de emplear y a que permiten que el código escrito sea más compacto.

Si se desean fijar los indicadores de formato específicos utilizando un manipulador, debe emplearse la función **setiosflags( )**. Este manipulador lleva a cabo la misma función que la función miembro **setf( )**. Para desactivar los indicadores debe utilizarse el manipulador **resetiosflags( )**. Este manipulador es equivalente a **unsetf( )**.

EJEMPLOS

1. Este programa muestra varios de los manipuladores de E/S:

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

main()
 {
  cout << hex << 100 << endl;
  cout << oct << 10 << endl;

  cout << setfill('X') << setw(10);
  cout << 100 << " hi " << endl;

  return 0;
}
```

Este programa obtiene la siguiente salida:

```
64
12
XXXXXXX144 hi
```

2. A continuación se presenta otra versión del programa que muestra una tabla de las potencias de dos y de las raíces cuadradas de los números 2 hasta 20. Esta versión emplea manipuladores de E/S en lugar de miembros e indicadores de formato.

```
/* Esta versión emplea manipuladores de E/S para presentar
   la tabla de cuadrados y de raíces cuadradas. */
#include <iostream.h>
#include <iomanip.h>
#include <math.h>

main()
{
  double x;

  cout << setprecision(4);
  cout << " x sqrt(x) x^2\n\n";

  for(x = 2.0; x <= 20.0; x++) {
    cout << setw(7) << x << "  ";
    cout << setw(7) << sqrt(x) << "  ";
    cout << setw(7) << x*x << '\n';
  }

  return 0;
}
```

EJERCICIOS

1. Repita los Ejercicios 1 y 2 de la Sección 8.3, utilizando, sólo por esta vez, los manipuladores de E/S en lugar de los miembros y los indicadores de formato.
2. Construya la sentencia que escriba el número 100 en hexadecimal, mostrando el indicador de base (0x). Para realizar esto, utilice el manipulador **setiosflags( )**.

8.5. CREACION DE INSERTORES PROPIOS

Como ya se ha indicado en este libro, una de las razones para emplear sentencias de E/S de C++ es que los operadores de E/S pueden sobrecargarse para las clases creadas. La realización de ello permite incorporar las clases creadas en los programas en C++ sin demasiadas añadiduras. En esta sección se enseña a sobrecargar el operador de salida << de C++.

En el lenguaje C++, la operación de salida se denomina *inserción* y a << se le denomina *operador de inserción*. Cuando se sobrecarga para salida el <<, se está creando una *función insertora* o, abreviado, un *insertor*. La razón para utilizar estos términos proviene del hecho de que un operador de salida *inserta* información en un flujo.

Todas las funciones insertoras tienen la siguiente forma general:

```
ostream &operator <<(ostream &stream, class-name ob)
{
   // cuerpo del insertor
   return stream;
}
```

El primer parámetro es una referencia a un objeto de tipo **ostream**. Esto significa que *stream* debe ser un flujo de salida. (Recuerde que **ostream** está definida dentro de la clase **ios**.) El segundo parámetro recibe el objeto que será la salida. (También puede ser un parámetro por referencia, si se adapta mejor a su aplicación.) Observe que la función insertora devuelve una referencia a *stream*, que es del tipo **ostream**. Esto es necesario puesto que el operador sobrecargado << va a ser empleado en expresiones complejas de E/S, como

```
cout << ob1 << ob2 << ob2;
```

Dentro de un insertor puede realizarse cualquier tipo de procedimiento. Lo que haga un insertor depende completamente del programador. Sin embargo, para que el insertor sea consistente con la práctica correcta de la programación, debería limitar sus operaciones a sacar información en un flujo.

Aunque en principio pueda parecer sorprendente, un insertor no puede ser un miembro de la clase en la que ha sido diseñada para trabajar. Esta es la razón: Cuando una función operadora de cualquier tipo es el miembro de una clase, el operando de la izquierda, que se pasa implícitamente mediante el puntero **this**, es el objeto que genera la llamada a la función operadora. Esto significa que el operando de la izquierda es un objeto de esa clase. Por tanto, si una función operadora sobrecargada es un miembro de una clase, el operando izquierdo debe ser un objeto de esa clase. Sin embargo, cuando se crea un insertor, el operando izquierdo es un flujo, no el objeto de una clase, y el operando derecho es el objeto que se quiere obtener como salida.

Por consiguiente, un insertor no puede ser una función miembro.

El hecho de que un insertor no pueda ser un miembro puede parecer una seria debilidad de C++, porque significa que todos los datos de una clase que se obtengan utilizando un insertor serán públicos, violándose de esta manera el principio básico del encapsulamiento. Sin embargo, no es esto lo que sucede. Aún cuando los insertores no pueden ser miembros de la clase para la que han sido diseñados, pueden ser amigos de esa clase. De hecho, en la mayoría de los programas, un insertor sobrecargado será amigo de la clase para la que fue creado.

EJEMPLOS

1. Como primer ejemplo sencillo, este programa contiene un insertor para la clase **coord**, desarrollado en el capítulo anterior:

```
// Uso de un insertor amigo para objetos de tipo coord.
#include <iostream.h>

class coord {
  int x, y;
public:
  coord() { x = 0; y = 0; }
  coord(int i, int j) { x = i; y = j; }
  friend ostream &operator<<(ostream &stream, coord ob);
};

ostream &operator<<(ostream &stream, coord ob)
{
  stream << ob.x << ", " << ob.y << '\n';
  return stream;
}

main()
{
  coord a(1, 1), b(10, 23);

  cout << a << b;

  return 0;
}
```

Este programa muestra los siguiente:

```
1, 1
10, 23
```

El insertor de este programa demuestra un punto importante sobre la creación de insertores: hacerlos tan generales como sea posible. En este caso, la sentencia de E/S del insertor coloca los valores **x** e **y** en **stream**, que es cualquier flujo pasado a la función.

Como se describirá en el próximo capítulo, el mismo insertor que imprime en pantalla puede emplearse para imprimir en *cualquier flujo*. A veces los neófitos están tentados de escribir el insertor **coord** de este modo:

```
ostream &operator<<(ostream &stream, coord ob)
{
  cout << ob.x << ", " << ob.y << '\n';
  return stream;
}
```

En este caso, la sentencia de salida está codificada de forma fija para presentar la información en el dispositivo de salida estándar conectado a **cout**. Sin embargo, esto

impide que el insertor sea utilizado por otros flujos. Lo fundamental es que deben construirse los insertores de la manera más general posible porque no existe ningún inconveniente derivado de ello.

2. Con el fin de ilustrar, a continuación se presenta el programa anterior revisado, de modo que el insertor no es amigo de la clase **coord**. Debido a que el insertor no tiene acceso a las áreas privadas de **coord**, las variables **x** e **y** tienen que hacerse públicas:

```
/* Creación de un insertor para objetos de tipo coord,
   usando un insertor no-amigo. */

#include <iostream.h>

class coord {
public:
  int x, y; // deben ser públicas
  coord() { x = 0; y = 0; }
  coord(int i, int j) { x = i; y = j; }
};

// Un insertor para la clase coord.
ostream &operator<<(ostream &stream, coord ob)
{
  stream << ob.x << ", " << ob.y << '\n';
  return stream;
}

main()
{
  coord a(1, 1), b(10, 23);
  cout << a << b;
  return 0;
}
```

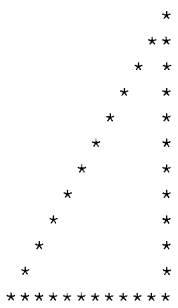
3. Un insertor no está limitado a mostrar sólo información textual. Un insertor puede efectuar cualquier operación o conversión requeridas por un dispositivo o una situación particular. Por ejemplo, es perfectamente correcto crear un insertor que envíe información a un trazador gráfico. En este caso, el insertor necesitará enviar al trazador gráfico, además de la información, los códigos apropiados. Para conocer este tipo de insertor, el siguiente programa crea una clase, llamada **triangle**, que almacena la anchura y la altura de un triángulo rectángulo. El insertor de esta clase muestra el triángulo en la pantalla.

```
// Este programa dibuja triángulos rectángulos
#include <iostream.h>

class triangle {
  int height, base;
public:
  triangle(int h, int b) { height = h; base = b; }
  friend ostream &operator<<(ostream &stream, triangle ob);
};
```

```
// Dibujo de un triángulo.
ostream &operator<<(ostream &stream, triangle ob)
{
  int i, j, h, k;
  i = j = ob.base-1;
  for(h=ob.height-1; h; h--) {
    for(k=i; k; k--)
      stream << ' ';
    stream << '*';

    if(j!=i) {
      for(k=j-i-1; k; k--)
        stream << ' ';
      stream << '*';
    }

    i--;
    stream << '\n';
  }
  for(k=0; k<ob.base; k++) stream << '*';
  stream << '\n';

  return stream;
}

main()
{
  triangle t1(5, 5), t2(10, 10), t3(12, 12);

  cout << t1;
  cout << endl << t2 << endl << t3;

  return 0;
}
```

Observe que este programa muestra cómo un insertor diseñado apropiadamente puede integrarse completamente en una expresión «normal» de E/S. Este programa da lugar a lo siguiente:

```
    *
   **
  * *
 *  *
*****

          *
         **
        * *
       *  *
      *   *
     *    *
    *     *
   *      *
  *       *
**********
```



EJERCICIOS

1. Dada la siguiente clase **strtype** y este trozo de programa, construya un insertor que muestre una cadena:

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

class strtype {
  char *p;
  int len;
public:
  strtype(char *ptr);
  ~strtype();
  friend ostream &operator<<(ostream &stream, strtype &ob);
};

strtype::strtype(char *ptr)
{
  len = strlen(ptr);
  p = new char [len+1];
  if(!p) {
    cout << "Error de asignación\n";
    exit(1);
  }
  strcpy(p, ptr);
}

strtype::~strtype()
{
  delete p;
}

// Aquí debe crearse la función insertora de operator<<.

main()
{
  strtype s1("Esto es una prueba"), s2("Me gusta C++");

  cout<< s1 << '\n' << s2;

  return 0;
}
```

2. Sustituya la función **show( )** del siguiente programa por una función insertora:

```
#include <iostream.h>
class planet {
protected:
  double distance; // millas desde el sol
  int revolve; // en días
public:
  planet(double d, int r) { distance = d; revolve = r; }
};

class earth : public planet {
  double circumference; // perímetro de la órbita
public:
  earth(double d, int r) : planet(d, r) {
    circumference = 2*distance*3.1416;
  }

  /*
    Escriba esto de nuevo para que muestre la información
    usando una función insertora. */
  void show() {
    cout << "Distancia desde el sol: " << distance << '\n';
    cout << "Días en órbita: " << revolve << '\n';
    cout << "Perímetro de la órbita: " << circumference << '\n';
  }
};

main()
{
  earth ob(93000000, 365);

  cout << ob;

  return 0;
}
```

3. Explique por qué un insertor no puede ser una función miembro.

8.6. CREACION DE EXTRACTORES

Del mismo modo que puede sobrecargarse el operador de salida <<, puede sobrecargarse el operador de entrada >>. En C++, el operador >> se denomina *operador de extracción* y la función que lo sobrecarga se denomina *extractor*. La razón del uso de este término está en el hecho de que la introducción de información en un flujo elimina (esto es, extrae) los datos de ella.

La forma general de una función extractora es:

```
istream &operator>>(istream &stream, class-name &ob)
{
 // cuerpo del extractor
 return stream;
}
```

Los extractores devuelven una referencia a **istream**, que es un flujo de entrada. El primer parámetro debe ser una referencia al flujo de entrada. El segundo parámetro es una referencia al objeto que recibe la entrada.

Por la misma razón por la que un insertor no puede ser un miembro, un extractor tampoco puede serlo. A pesar de que dentro de un extractor puede realizarse cualquier función, lo más aconsejable es limitar su actividad a la de introducir información.

EJEMPLOS

1. Este programa añade un extractor a la clase **coord**:

```
// Adición de un extractor amigo para un objeto de tipo coord.
#include <iostream.h>

class coord {
  int x, y;
public:
  coord() { x = 0; y = 0; }
  coord(int i, int j) { x = i; y = j; }
  friend ostream &operator<<(ostream &stream, coord ob);
  friend istream &operator>>(istream &stream, coord &ob);
};

ostream &operator<<(ostream &stream, coord ob)
{
  stream << ob.x << ", " << ob.y << '\n';
  return stream;
}

istream &operator>>(istream &stream, coord &ob)
{
  cout << "Introduzca las coordenadas: ";
  stream >> ob.x >> ob.y;
  return stream;
}

main()
{
  coord a(1, 1), b(10, 23);

  cout << a << b;

  cin >> a;
  cout << a;

  return 0;
}
```

Observe cómo el extractor también indica al usuario la entrada de información. Mientras no se necesite (o incluso no se desee) en la mayoría de las ocasiones, esta función muestra como un extractor hecho a medida puede simplificar el código para obtener un mensaje indicativo.

2. En este ejemplo se crea una clase inventario que almacena el nombre de un elemento, el número de existencias y su precio. El programa incluye un insertor y un extractor para esta clase.

```
#include <iostream.h>
#include <string.h>
class inventory {
  char item[40]; // nombre del elemento
  int onhand; // número de existencias
  double cost; // precio del elemento
public:
  inventory(char *i, int o, double c)
  {
    strcpy(item, i);
    onhand = o;
    cost = c;
  }
  friend ostream &operator<<(ostream &stream, inventory ob);
  friend istream &operator>>(istream &stream, inventory &ob);
};

ostream &operator<<(ostream &stream, inventory ob)
{
  stream << ob.item << ": " << ob.onhand;
  stream << " existencias en $" << ob.cost << '\n';

  return stream;
}

istream &operator>>(istream &stream, inventory &ob)
{
  cout << "Introduzca el nombre del elemento: ";
  stream >> ob.item;
  cout << "Introduzca el número de existencias: ";
  stream >> ob.onhand;
  cout << "Introduzca el precio: ";
  stream >> ob.cost;

  return stream;
}

main()
{
  inventory ob("martillo", 4, 12.55);

  cout << ob;

  cin >> ob;

  cout << ob;

  return 0;
}
```

EJERCICIOS

1. Añada un extractor a la clase **strtype** del Ejercicio 1 de la sección anterior.
2. Cree una clase que almacene un valor entero y su mínimo factor. Construya un insertor y un extractor para esta clase.

COMPROBACION DE APTITUD SUPERIOR

En este punto, debería ser capaz de realizar estos ejercicios y de responder a estas preguntas.

1. Escribe un programa que muestre el número 100 en decimal, hexadecimal y octal. (Utilice los indicadores de formato de **ios**.)
2. Escriba un programa que muestre el valor 1000.5364 en un campo de 20 caracteres, ajustado a la izquierda, con dos posiciones decimales y utilizando el * como carácter de relleno. (Utilice los indicadores de formato de **ios**.)
3. Resuelva de nuevo los Ejercicios 1 y 2 de manera que empleen los manipuladores de E/S.
4. Muestre el modo de eliminar los indicadores de formato para **cout** y el modo de restablecerlos. Utilice cualquier función miembro o manipulador.
5. Construya un insertor y un extractor para esta clase:

```
class pwr {
  int base;
  int exponent;
  double result; // base para el exponente
public:
  pwr(int b, int e);
};

pwr::pwr(int b, int e)
{
  base = b;
  exponent = e;

  result = 1;
  for( ; e; e--) result = result * base;
}
```

6. Construya una clase llamada **box** que almacene las dimensiones de un cuadrado. Cree un insertor que muestre un cuadro en pantalla. (Utilice el método que desee para mostrar el cuadro.)

COMPROBACION DE APTITUD INTEGRADA

Esta sección comprueba la integración del contenido de este capítulo con el de los anteriores.

1. Utilizando la clase de **stack** mostrada aquí, construya un insertor que muestre los contenidos de la pila. Pruebe dicho insertor.

```
#include <iostream.h>

#define SIZE 10

// Declaración de una clase pila para caracteres
class stack {
  char stck[SIZE]; // guarda la pila
  int tos; // índice de la cabeza de la pila
public:
  stack();
  void push(char ch); // mete caracteres en la pila
  char pop(); // saca caracteres de la pila
};

// Inicialización de la pila
stack::stack()
{
  tos = 0;
}

// Introducción de un carácter.
void stack::push(char ch)
{
  if(tos==SIZE) {
    cout << "La pila está llena";
      return;
  }
  stck[tos] = ch;
  tos++;
}

// Extracción de un carácter.
char stack::pop()
{
  if(tos==0) {
    cout << "La pila está vacía";
    return 0; // devuelve nulo si la pila está vacía
  }
  tos--;
  return stck[tos];
}
```

2. Escriba un programa que contenga una clase llamada **watch**. Utilizando las funciones temporales estándar, el constructor de esta clase debe leer la hora del sistema y almacenarla. Construya un insertor que muestre la hora.
3. Haciendo uso de la siguiente clase, que convierte pies en pulgadas, cree un extractor que le pida al usuario introducir pies. Construya, también, un insertor que muestre el número de pies y de pulgadas. Añada un programa que demuestre que el insertor y el extractor funcionan.

```
class ft_to_inches {
  double feet;
  double inches;
public:
  void set(double f) {
    feet = f;
    inches = f * 12;
  }
};
```

9 E/S avanzada en C++

OBJETIVOS DEL CAPITULO

Este capítulo continúa con el análisis del sistema de E/S. A lo largo del mismo aprenderá a crear sus propios manipuladores de E/S y a realizar E/S en archivo. No olvide que el sistema de E/S de C++ es muy rico y flexible y contiene muchas funciones. Ya que está fuera del alcance de este libro incluir todas ellas, se describen algunas de las más importantes. Puede encontrarse una descripción completa del sistema de E/S de C++ en el libro C++: *La referencia completa* (Berkeley: Osborne/McGraw-Hill, 1991).

Nota *El sistema de E/S de C++ descrito en este capítulo refleja el estándar ANSI C++ propuesto y es compatible con la mayoría de los compiladores de C++. Si se dispone de una versión antigua o no compatible, el sistema de E/S no contará con todas las capacidades descritas aquí.*

COMPROBACION DE APTITUD

Antes de continuar, debería ser capaz de responder a estas preguntas y de realizar estos ejercicios.

1. Escriba un programa que muestre las frase: «C++ es divertido», en un campo con una anchura de 40 caracteres y utilizando como carácter de relleno los dos puntos (:).
2. Escriba un programa que muestre el valor de 10/3 con cuatro posiciones decimales. Para hacerlo, utilice los miembros de **ios**.
3. Escriba de nuevo el programa anterior utilizando manipuladores de E/S.
4. ¿Qué es un insertor? ¿Qué es un extractor?
5. Dada la siguiente clase, construya para ella un insertor y un extractor.

```
class date {
  char date[9]; // almacenamiento de la fecha como la
  cadena: mm/dd/aa
public:
  // adición del insertor y del extractor
};
```

6. ¿Qué archivo cabecera debe incluirse para utilizar manipuladores de E/S que tengan parámetros?
7. ¿Qué flujos predefinidos se crean cuando comienza a ejecutarse un programa en C++?

9.1. CREACION DE MANIPULADORES PROPIOS

Además de sobrecargar los operadores de inserción y de extracción, cualquiera puede confeccionarse su propio sistema de E/S en C++ mediante la creación de funciones manipuladoras propias. Los manipuladores hechos a medida son importantes por dos razones fundamentales. La primera es que un manipulador puede agrupar, dentro de él, una secuencia de varias operaciones de E/S inde-

pendientes. Por ejemplo, no es inusual encontrar situaciones en las que se repite frecuentemente la misma secuencia de operaciones de E/S dentro de un programa. En estos casos puede utilizarse un manipulador a medida para realizar estas operaciones, simplificándose de este modo el código fuente e impidiendo que se produzcan errores accidentales. La segunda razón es que un manipulador a medida puede utilizarse cuando se necesite llevar a cabo operaciones de E/S en un dispositivo que no sea estándar. Por ejemplo, podría emplearse un manipulador para enviar códigos de control a un tipo especial de impresora o a un sistema de reconocimiento óptico.

Los manipuladores hechos a medida son una característica de C++ que admite POO, pero de ellos también pueden beneficiarse otros programas que no están orientados a objeto. Como se verá, los manipuladores a medida pueden ayudar a construir de una forma sencilla y eficiente programas que hagan un uso intensivo de operaciones de E/S.

Como ya es sabido, hay dos tipos básicos de manipuladores: aquellos que operan sobre flujos de entrada y los que lo hacen sobre flujos de salida. Sin embargo, además de estas dos amplias categorías, existe una división secundaria: aquellos manipuladores que tienen argumentos y los que no lo tienen. Existen algunas diferencias significativas entre la forma de crear un manipulador sin parámetros y un manipulador parametrizado. La creación de manipuladores parametrizados es mucho más difícil que la de los no parametrizados y está fuera del objetivo de este libro. Sin embargo, la construcción de manipuladores no parametrizados es bastante sencilla y se examina en esta sección.

Todas las funciones de los manipuladores de salida no parametrizados responden a este esquema:

```
ostream &manip-name(ostream &stream)
{
   // aquí introduce el programador su código

   return stream;
}
```

manip-name es el nombre del manipulador. La función devuelve una referencia a un flujo de tipo **ostream**. Esto es necesario si se utiliza un manipulador como parte de una expresión de E/S más extensa. Es importante entender que aún cuando el manipulador tenga como único argumento una referencia al flujo sobre el que opera, en una operación de salida no se emplea ningún argumento en la llamada al manipulador.

Todas las funciones de los manipuladores de entrada no parametrizados presentan el siguiente esquema:

```
istream &manip-name(istream &stream)
   {
   // aquí introduce el programador su código

   return stream;
}
```

Un manipulador de entrada recibe una referencia al flujo para el que ha sido invocado. El manipulador debe devolver este flujo.

Es fundamental que los manipuladores devuelvan *stream*. Si así no fuera, no podrían utilizarse en una secuencia de operaciones de entrada o de salida.

EJEMPLOS

1. Este primer ejemplo es un pequeño programa que crea un manipulador llamado **setup( )** que fija la anchura del campo a 10, la precisión a 4 y el carácter de relleno a *:

```
#include <iostream.h>

ostream &setup(ostream &stream)
{
  stream.width(10);
  stream.precision(4);
  stream.fill('*');

  return stream;
}

main()
{
  cout << setup << 123.123456;

  return 0;
}
```

   Como puede verse, **setup** es utilizado como parte de una expresión de E/S, del mismo modo que se utiliza cualquiera de los manipuladores ya existentes.

2. Es necesario que los manipuladores hechos a medida no sean complejos para que sean útiles. Por ejemplo, los manipuladores **atn( )** y **note( )** proporcionan un mecanismo sencillo para presentar palabras o frases utilizadas frecuentemente:

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

// Atención:
ostream &atn(ostream &stream)
{
  stream << "Atención: ";
  return stream;
}

// Tome nota, por favor:
  ostream &note(ostream &stream)
{
  stream << "Tome nota, por favor: ";
  return stream;
```

```
}

main()
{
  cout << atn << "Circuito de alto voltaje\n";
  cout << note << "Apague todas las luces\n";

  return 0;
}
```

Aunque sea sencillo, si algo se repite con mucha frecuencia, estos manipuladores ahorran el esfuerzo de escribirlo continuamente.

3. Este programa crea el manipulador de entrada **getpass( )** que hace sonar una alarma y pide, a continuación, una palabra clave:

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

// Un sencillo manipulador de entrada
istream &getpass(istream &stream)
{
  cout << '\a'; // sonido de alarma
  cout << "Introduzca la palabra clave: ";

  return stream;
}

main()
{
  char pw[80];

  do {
    cin >> getpass >> pw;
  } while (strcmp(pw, "palabra clave"));

  cout << "Completado el inicio de la sesión\n";

  return 0;
}
```

EJERCICIOS

1. Construya un manipulador de salida que muestre la hora y la fecha actual del sistema. Llámelo **td( )**.
2. Construya un manipulador de salida llamado **sethex( )** que muestre la salida en hexadecimal y que desactive los indicadores **uppercase** y **showbase**. Cree, también, un manipulador de salida llamado **reset( )** que deshaga lo realizado por **sethex( )**.
3. Construya un manipulador de entrada llamado **skipchar( )** que lea e ignore los primeros diez caracteres del flujo de entrada.

9.2. PRINCIPIOS DE E/S EN ARCHIVOS

Para realizar E/S en archivos debe incluirse en el programa el archivo cabecera **fstream.h**. Este archivo define varias clases, incluyendo a **ifstream**, **ofstream** y **fstream**. Recuerde que **istream** y **ostream** derivan de **ios**, por lo cual **ifstream**, **ofstream** y **fstream** también tienen acceso a todas las operaciones definidas por **ios** (discutidas en el capítulo anterior).

En C++, un archivo se abre mediante el enlace a un flujo. Hay tres tipos de flujos: de entrada, de salida y de entrada/salida. Antes de que pueda abrirse un archivo debe obtenerse un flujo. Para crear un flujo de entrada, éste debe declararse de la clase **ifstream**. Para crear un flujo de salida, éste debe ser declarado de la clase **ofstream**.

Los flujos que realizan tanto operaciones de entrada como de salida deben ser declarados de la clase **fstream**. Por ejemplo, este fragmento de código crea un flujo de entrada, uno de salida y un flujo que efectúa tanto entrada como salida:

```
ifstream in; // entrada
ofstream out; // salida
fstream io; // entrada y salida
```

Una vez creado un flujo, una forma de asociarle a un archivo es utilizar la función **open( )**. Esta función es una función miembro de las clases de los tres flujos. Su prototipo es éste:

```
void open(const char *filename, int mode, int access);
```

filename es el nombre del archivo, que puede incluir un especificador del camino. El valor de *mode* determina el modo de abrir el archivo. Debe tomar uno (o más) de estos valores (heredados de **fstream.h**):

```
ios::app
ios::ate
ios::binary
ios::in
ios::nocreate
ios::noreplace
ios::out
ios::trunc
```

Pueden combinarse dos o más valores, aplicando sobre ellos el operador OR. Veamos qué significa cada uno de ellos.

La inclusión de **ios::ap** da lugar a que toda la salida al archivo se añada al final del mismo. Este valor sólo puede utilizarse con archivos de salida. El valor **ios::ate** provoca, cuando se abre el archivo, una búsqueda del final del mismo. Aunque **ios::ate** busca el final de archivo, las operaciones de E/S pueden producirse en cualquier parte de él.

El valor **ios::in** especifica que el archivo es de entrada. El valor **ios::out** especifica que el archivo es de salida. Sin embargo, la creación de un flujo utilizando **ifstream** implica que el archivo sea de entrada y la creación de un flujo utilizando **ofstream** implica que sea de salida, de modo que en estos casos no es necesario especificar esos valores.

El valor **ios::binary** da lugar a que un archivo sea abierto en modo binario. Por omisión, todos los archivos se abren en modo texto. En este modo, puede tener lugar la transformación de diversos caracteres, como el retorno de carro, convirtiéndose las secuencias de saltos de línea en nuevas líneas. No obstante, cuando se abre un archivo en modo binario, no se produce ninguna transformación de caracteres. No hay que olvidar que cualquier archivo, contenga texto formateado o datos, puede abrirse en modo binario o en modo texto. La única diferencia estriba en la transformación de caracteres.

La inclusión de **ios::nocreate** da lugar a que se produzca un error en la función **open( )** si el archivo todavía no existía. El valor **ios::noreplace** da lugar a un error en la función **open( )** si el archivo ya existía.

El valor **ios::trunc** ocasiona que se destruya el contenido de un archivo ya existente y que el archivo vea reducido su tamaño a cero.

El valor de *access* determina el modo de acceso al archivo. Su valor implícito es **filebuf::openprot** (**filebuf** es la clase padre de las clases flujo), que para entornos UNIX es 0x644 y significa que es un archivo normal. En entornos DOS/Windows, el valor de *access* corresponde generalmente a códigos de atributos de archivos del DOS/Windows. Estos son:

Atributo	Significado
0	archivo normal: acceso por open
1	archivo de sólo lectura
2	archivo oculto
4	archivo del sistema
8	conjunto de bits del archivo

Puede hacerse el OR de dos o más de estos valores. Para el DOS/Windows, un archivo normal tiene un valor de *access* nulo. Para otros sistemas operativos, es necesario comprobar el manual del usuario del compilador para conocer los valores correctos de *access*.

El siguiente fragmento de código abre un archivo normal de salida en un entorno DOS/Windows:

```
ofstream out;
out.open("test", ios::out, 0);
```

Sin embargo, pocas veces (probablemente ninguna) se verá una llamada a **open( )**, como la mostrada, porque los parámetros *mode* y *access* toman valores implícitos. Para **ifstream**, el valor de *mode* está en **ios::in** y para **ofstream** en **ios::out.** *access* toma, por omisión, un valor que crea un archivo normal. Por tantó, la sentencia anterior normalmente aparecerá así:

```
out.open("test"); // Implícitamente archivo normal de salida
```

Para abrir un flujo de entrada y de salida deben especificarse en *mode* los valores **ios::in** y **ios::out**, como puede verse en el ejemplo. (En esta situación *mode* no toma ningún valor implícito.)

```
fstream mystream;
mystream.open("test", ios::in | ios::out);
```

Si se produce un error en **open( )**, el flujo será nulo. Por consiguiente, antes de utilizar un archivo, sería necesario comprobar que la operación de apertura del archivo ha tenido éxito, con una sentencia como ésta:

```
if(!mystream) {
  cout << "El archivo no puede abrirse\n";
  // error del descriptor
}
```

Aunque parezca lo más adecuado abrir un archivo utilizando la función **open( )**, en la mayor parte de las ocasiones no debe hacerse porque las clases **ifstream**, **ofstream** y **fstream** poseen funciones constructoras que abren el archivo automáticamente. Las funciones constructoras tienen los mismos parámetros y valores implícitos que la función **open( )**. Por ello, la forma más normal de encontrar un archivo abierto es la presentada en este ejemplo:

```
ifstream mystream("miarchivo"); // se abre un archivo de entrada
```

Como ya se ha dicho, si por alguna razón no puede abrirse un archivo, el valor de la variable flujo asociada será cero. De este modo, si utiliza una función constructora para abrir el archivo o una llamada explícita a **open( )**, lo más conveniente es confirmar que el archivo ha sido abierto comprobando el valor del flujo.

Para cerrar un archivo hay que emplear la función miembro **close( )**. Por ejemplo, para cerrar el archivo enlazado al flujo **mystream** hay que emplear esta sentencia:

```
mystream.close( );
```

La función **close( )** no tiene parámetros y no devuelve ningún valor.

Puede conocerse dónde está el final de un archivo usando la función miembro **eof( )**, que tiene este prototipo:

```
int eof();
```

Devuelve un valor distinto de cero cuando encuentra el final del archivo y cero en cualquier otro caso.

Una vez que se ha abierto un archivo es muy fácil leer los datos textuales del mismo o escribir datos textuales formateados en él. Simplemente hay que utilizar los operadores << y >> del mismo modo que se utilizan cuando se realiza

E/S por consola excepto que, en lugar de emplear **cin** y **cout**, se sustituye un flujo enlazado por un archivo. De alguna manera, leer y escribir archivos mediante >> y << es igual que utilizar las funciones **fprintf( )** y **fscanf( )** de C. En el archivo se almacena la información en el mismo formato que cuando va a ser mostrado por pantalla. Por tanto, un archivo generado mediante << es un archivo de texto formateado y cualquier archivo leído mediante >> debe ser un archivo de texto formateado.

Recuerde *Cuando se realiza E/S en archivos de texto puede tener lugar alguna transformación de caracteres. Por ejemplo, los caracteres de nueva línea pueden pasar a ser una combinación de retorno de carro/avance de línea. Sin embargo, la apertura de un archivo para operaciones binarias impide que se produzcan estas transformaciones.*

EJEMPLOS

1. Este programa crea un archivo de salida, escribe información en él, lo cierra y lo abre de nuevo como archivo de entrada y lee la información que contiene:

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

main()
{
  ofstream fout("test"); // creación de un archivo normal de
  //salida

  if(!fout) {
    cout << "El archivo de salida no puede abrirse.\n";
    return 1;
  }

  fout << "¡Hola!\n";
  fout << 100 << ' ' << hex << 100 << endl;

  fout.close();

  ifstream fin("test"); // se abre un archivo normal de
  entrada

  if(!fin) {
    cout << "El archivo de entrada no puede abrirse.\n";
    return 1;
  }

  char str[80];
  int i;

  fin >> str >> i;
  cout << str << ' ' << i << endl;
```

```
  fin.close();
  return 0;
}
```

Después de ejecutar este programa, examine el contenido de test. Contendrá lo siguiente:

```
¡Hola!
100 64
```

Como ya se indicó anteriormente, cuando se utilizan los operadores < < y > > para llevar a cabo E/S en archivos, la información está formateada exactamente igual que si fuera a aparecer en pantalla.

2. Este es un ejemplo de E/S en disco rígido. Este programa lee cadenas introducidas mediante el teclado y las escribe en el disco. El programa se detiene cuando el usuario introduce una línea en blanco. Para utilizar el programa, especifique el nombre de archivo de salida en la línea de órdenes:

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

main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "Uso: WRITE <filename>\n";
    return 1;
  }

  ofstream out(argv[1]); // archivo normal de salida

  if(!out) {
    cout << "El archivo de salida no puede abrirse.\n";
    return 1;
  }

  char str[80];
  cout << "Escritura de cadenas en disco, RETURN para
  detenerlo\n";

  do {
    cout << ": ";
    gets(str);
    out << str << endl;
  } while (*str);

  out.close();
  return 0;
}
```

3. El siguiente programa copia un archivo de texto y, durante el proceso, transforma todos los espacios en el símbolo \. La función **eof( )** se utiliza para conocer el final

del archivo de entrada. El flujo de entrada **fin** tiene desactivado **skipws**. Esto evita que se salten los primeros espacios:

```
// Conversión de espacios a \.
#include <iostream.h>
#include <fstream.h>

main(int argc, char *argv[])
{
  if(argc!=3) {
    cout << "Uso: CONVERT <input> <output>\n";
    return 1;
  }

  ifstream fin(argv[1]); // apertura del archivo de entrada
    ofstream fout(argv[2]); // creación del archivo de salida
  if(!fout) {
    cout << "El archivo de salida no puede abrirse.\n";
    return 1;
  }
  if(!fin) {
    cout << "El archivo de entrada no puede abrirse.\n";
    return 1;
  }

  char ch;

  fin.unsetf(ios::skipws); // no hay salto de espacios
  while(!fin.eof()) {
    fin >> ch;
    if(ch==' ') ch = '\';
    fout << ch;
  }

  return 0;
}
```

EJERCICIOS

1. Escriba un programa que copie un archivo de texto. El programa debe contar el número de caracteres copiados y mostrar el resultado. ¿Por qué el número de caracteres copiados difiere del que aparece cuando se lista el archivo de salida en el directorio?
2. Construya un programa que escriba la siguiente tabla de información en un archivo llamado **phone**.

```
Issac Newton, 415 555-3423
Robert Goddard, 213 555-2312
Enrico Fermi, 202 555-1111
```

3. Escriba un programa que cuente el número de palabras de un archivo.

9.3. E/S BINARIA SIN FORMATO

Aunque los archivos de texto con formato, como los generados en los ejemplos previos, son útiles en un gran número de ocasiones, no tienen la flexibilidad de los archivos binarios sin formato. Por esta razón, C++ admite una amplia gama de funciones de E/S en archivos binarios (o «naturales»).

Las funciones de E/S binarias de más bajo nivel son **get( )** y **put( )**. Se puede escribir un byte utilizando la función miembro **put( )** y leerlo con la función miembro **get( )**. La función **get( )** tiene muchas formas, pero la versión usada más frecuentemente de ella, junto con la de **put( )**, es la siguiente:

```
istream &get(char &ch);
ostream &put(char ch);
```

La función **get( )** lee un sólo carácter del flujo asociado y coloca ese valor en *ch*. Devuelve una referencia al flujo, que será nula si encuentra el carácter EOF. La función **put( )** escribe *ch* en el flujo y devuelve el flujo.

Para leer y escribir bloques de datos binarios deben utilizarse las funciones **read( )** y **write( )**. Sus prototipos son:

```
istream &read(unsigned char *buf, int num);
ostream &write(const unsigned char *buf, int num);
```

La función **read( )** lee *num* bytes del flujo asociado y los coloca en el búfer apuntado por *buf*. La función **write( )** escribe *num* bytes del búfer apuntado por *buf* en el flujo asociado.

Si se encuentra el final del archivo antes de que se hayan leído *num* caracteres, la función **read( )** simplemente se detiene y el búfer queda con los caracteres que pudieron leerse. Puede averiguarse el número de caracteres leídos utilizando otro miembro llamado **gcount( )**, con este prototipo:

```
int gcount( );
```

Devuelve el número de caracteres leídos en la última operación de entrada binaria.

Nota *No es necesario abrir un archivo utilizando* **ios::binary** *para emplear las funciones de archivo binarias sin formato. Estas funciones se pueden emplear para acceder a cualquier tipo de archivo. El hecho de especificar* **ios::binary** *simplemente impide que tengan lugar transformaciones de caracteres. Puesto que en los ejemplos de este libro no se considera la transformación de caracteres, no se necesita* **ios::binary**.

EJEMPLOS

1. El próximo programa muestra en pantalla el contenido de cualquier archivo. Utiliza la función **get( )**:

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

main(int argc, char *argv[])
{
  char ch;

  if(argc!=2) {
    cout << "Uso: PR <filename>\n";
    return 1;
  }
  ifstream in(argv[1]);
  if(!in) {
    cout << "El archivo no puede abrirse";
    return 1;
  }

  while(!in.eof()) {
    in.get(ch);
    cout << ch;
  }

  return 0;
}
```

2. Este programa utiliza **put( )** para escribir caracteres en un archivo hasta que el usuario introduzca el símbolo $:

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

main(int argc, char *argv[])
{
  char ch;

  if(argc!=2) {
    cout << "Uso: WRITE <filename>\n";
    return 1;
  }

  ofstream out(argv[1]);
  if(!out) {
    cout << "El archivo no puede abrirse";
    return 1;
  }

  cout << "Para detenerlo introduzca $\n";
```

```
  do {
    cout << ": ";
    cin.get(ch);
    out.put(ch);
  } while (ch!='$');

  out.close();

  return 0;
}
```

Observe que el programa utiliza **get( )** para leer caracteres de **cin**. Esto es necesario porque el uso de >> conduciría a que no fueran considerados los primeros espacios en blanco. Esto no sucede utilizando **get( )**.

3. Este programa emplea **write( )** para escribir un **double** y una cadena en un archivo llamado **test**:

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

main()
{
  ofstream out("test");
  if(!out) {
    cout << "El archivo de salida no puede abrirse.\n";
    return 1;
  }

  double num = 100.45;
  char str[] = "Esto es una prueba";

  out.write((char *) &num, sizeof(double));
  out.write(str, strlen(str));
  out.close();

  return 0;
}
```

Nota *Cuando se genera un búfer que no está definido como un array de caracteres, es necesario el molde de tipo* **(char *)** *dentro de la llamada a* **write( )**. *Debido a que C++ efectúa una comprobación rígida de los tipos, un puntero de un tipo no se convierte automáticamente en un puntero de otro tipo.*

4. Este programa utiliza **read( )** para leer el archivo creado por el programa del Ejemplo 3:

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

```
main()
{
  ifstream in("test");
  if(!in) {
    cout << "El archivo de entrada no puede abrirse.\n";
    return 1;
  }

  double num;
  char str[80];
  in.read((char *) &num, sizeof(double));
  in.read(str, 15);

  cout << num << ' ' << str;

  in.close();

  return 0;
}
```

Como sucede con el programa anterior, el molde de tipo dentro de **read( )** es necesario porque C++ no transforma un puntero de un tipo en otro.

5. El siguiente programa escribe un array de valores **double** en un archivo y después los lee. También informa sobre el número de caracteres leídos.

```
// Demostración de gcount().
#include <iostream.h>
#include <fstream.h>

main()
{
  ofstream out("test");
  if(!out) {
    cout << "El archivo de salida no puede abrirse.\n";
    return 1;
  }

  double nums[4] = {1.1, 2.2, 3.3, 4.4 };

  out.write((char *) nums, sizeof(nums));
  out.close();

  ifstream in("test");
  if(!in) {
    cout << "El archivo de entrada no puede abrirse.\n";
    return 1;
  }

  in.read((char *) &nums, sizeof(nums));

  int i;
  for(i=0; i<4; i++)
```

```
    cout << nums[i] << ' ';

  cout << '\n';

  cout << in.gcount() << " caracteres leídos\n";
  in.close();

  return 0;
}
```

EJERCICIOS

1. Resuelva nuevamente los Ejercicios 1 y 3 de la anterior sección (9.2) de manera que utilicen **get( )**, **put( )**, **read( )** y/o **write( )**. (Utilice de estas funciones aquellas que considere más adecuadas.)
2. Dada la siguiente clase, construya un programa que vuelque el contenido de la clase en un archivo. Para este fin, cree una función insertora.

```
class account {
  int custnum;
  char name[80];
  double balance;
public:
  account(int c, char *n, double b)
  {
    custnum = c;
    strcpy(name, n);
    balance = b;
  }
  // introduzca aquí el insertor
};
```

9.4. MAS SOBRE FUNCIONES DE E/S BINARIAS

Además de la forma mostrada arriba, la función **get( )** puede sobrecargarse de diferentes modos. El prototipo de las dos formas sobrecargadas más frecuentemente utilizadas es:

```
istream &get(char *buf, int num, char delim ='\n');
int get( );
```

La primera forma sobrecargada lee caracteres del array apuntado por *buf* hasta que hayan sido leídos *num* caracteres o hasta que se encuentre el carácter especificado por *delim*. **get( )** coloca el carácter nulo al final del array apuntado por *buf*. Si no se especifica el parámetro *delim*, actúa como delimitador por omisión el carácter nueva línea. Si se encuentra en el flujo de entrada el carácter delimitador, *no* se elimina. Se queda en el flujo hasta la próxima operación de entrada.

La segunda forma sobrecargada de **get( )** devuelve el siguiente carácter del flujo. Si encuentra el final del archivo devuelve EOF. Esta forma de **get( )** es similar a la función **getc( )** de C.

Otra función miembro encargada de realizar la entrada de información es **getline( )**. Su prototipo es el siguiente:

```
istream &getline(char *buf, int num, char delim='\n');
```

Como puede verse, esta función es virtualmente idéntica a la versión **get(buf, num, delim)** de **get( )**. Lee caracteres del flujo de entrada, hasta que se han leído *num* caracteres hasta que se encuentra el carácter especificado por *delim*, y los coloca en el array apuntado por *buf*. Si no se especifica nada, *delim* es, por omisión, el carácter de nueva línea. El array apuntado por *buf* termina con el carácter nulo. La diferencia entre **get(buf, num, delim)** y **getline( )** está en que **getline( )** lee y elimina el delimitador del flujo de entrada.

Puede obtenerse el siguiente carácter del flujo de entrada, sin eliminarlo de la misma, utilizando **peek( )**. Este es su prototipo:

```
int peek( );
```

Devuelve el próximo carácter del flujo o, si encuentra el final de archivo, devuelve EOF.

Puede devolverse el último carácter leído del flujo al mismo utilizando **putback( )**. Su prototipo es:

```
istream &putback(char c);
```

donde *c* es el último carácter leído.

Cuando se realiza una operación de salida, los datos no se escriben inmediatamente en el dispositivo físico vinculado al flujo. En lugar de ello, la información se almacena en un búfer interno hasta que éste se completa. Sólo entonces se escribe en el disco el contenido del búfer. Sin embargo, se puede lograr que la información se escriba físicamente en el disco antes de que el búfer esté lleno llamando a **flush( )**. Su prototipo es:

```
ostream &flush( );
```

Cuando se va a utilizar un programa en entornos hostiles (en situaciones en las que se producen, por ejemplo, frecuentes cortes de energía), deberían garantizarse las llamadas a **flush( )**.

EJEMPLOS

1. Como ya sabe, cuando se utiliza >> para leer una cadena, la lectura se detiene al encontrar el primer carácter de espacio en blanco. Esto lo convierte en un operador poco útil cuando se trata de leer una cadena que contiene espacios en blanco. Este problema puede eliminarse usando **getline( )**, como lo demuestra este programa:

```
// Uso de getline() para leer una cadena que contiene
// espacios en blanco.
#include <iostream.h>
#include <fstream.h>

    main()
{
  char str[80];

  cout << "Introduzca su nombre: ";
  cin.getline(str, 79);

  cout << str << '\n';

  return 0;
}
```

En este caso se permite que el parámetro final para **getline( )** sea, por omisión, el de nueva línea. Esto da lugar a que **getline( )** se comporte como la función **gets( )**.

2. En los casos de programación reales, las funciones **peek( )** y **putback( )** son especialmente útiles porque le permitirán gestionar de un modo sencillo aquellas situaciones en las que desconozca qué tipo de información va ser introducida en cualquier momento. El siguiente programa da una idea acerca de esto. Lee cadenas o enteros de un archivo. Las cadenas y los enteros pueden estar en cualquier orden:

```
// Demostración de peek().
#include <iostream.h>
#include <fstream.h>
#include <ctype.h>
#include <stdlib.h>
main()
{
  char ch;
  ofstream out("test");
  if(!out) {
    cout << "El archivo de salida no puede abrirse.\n";
    return 1;
  }

  char str[80], *p;

  out << 123 << "esto es una prueba" << 23;
  out << "¡Hola gente!" << 99 << "sdf" << endl;
  out.close();

  ifstream in("test");
  if(!in) {
    cout << "El archivo de entrada no puede abrirse.\n";
    return 1;
  }

  do {
    p = str;
    ch = in.peek(); // ve el tipo del próximo carácter
    if(isdigit(ch)) {
```

```
        while(isdigit(*p=in.get())) p++; // lectura de enteros
        in.putback(*p); // devolución de un carácter a la
        // cadena
        *p = '\0'; // la cadena termina en el carácter nulo
        cout << "Entero: " << atoi(str);
      }
      else if(isalpha(ch)) { // lectura de una cadena
        while(isalpha(*p=in.get())) p++;
        in.putback(*p); // devolución de un carácter a la
        // cadena
        *p = '\0'; // la cadena termina en el carácter nulo
        cout << "Cadena: " << str;
      }
      else in.get(); // ignorarlo
      cout << '\n';
    } while(!in.eof());

    in.close();
    return 0;
}
```

EJERCICIOS

1. Escriba de nuevo el programa del Ejemplo 1 de forma que utilice **get( )** en vez de **getline( )**. ¿Funciona el programa de modo diferente?
2. Escriba un programa que lea una línea de un archivo de texto y que la muestre en la pantalla. Utilice **getline( )**.
3. Analice los casos en los que sea apropiada una llamada a **flush( )**.

9.5. ACCESO ALEATORIO

En el sistema de E/S de C++, puede realizarse acceso aleatorio utilizando las funciones **seekg( )** y **seekp( )**. La forma más habitual de ambas es:

```
istream &seekg(streamoff offset, seek_dir origin);
ostream &seekp(streamoff offset, seek_dir origin);
```

streamoff es un tipo definido en **iostream.h**, que es capaz de contener el mayor valor correcto que puede tomar *offset*. **seek_dir** es una lista que toma estos valores:

Valor	Significado
ios::beg	búsqueda desde el principio
ios::cur	búsqueda desde la posición actual
ios::end	búsqueda desde el final

El sistema de E/S de C++ gestiona dos punteros asociados con el archivo. Uno es el *puntero get*, que especifica dónde tendrá lugar la próxima operación de entrada. El otro es el *puntero put*, que especifica el lugar en el que tendrá lugar la siguiente operación de salida. Cada vez que se produce una operación de entrada o de salida, el puntero correspondiente avanza automáticamente una posición. Sin embargo, el uso de las funciones **seekg( )** y **seekp( )** permite el acceso al archivo de modo no secuencial.

La función **seekg( )** mueve el actual puntero get, asociado al archivo, un número de bytes igual a *offset* desde el valor de *origin* especificado. La función **seekp( )** mueve el actual puntero put, asociado al archivo, un número *offset* de bytes desde el valor de *origin* especificado.

Es posible determinar la posición actual de cada puntero del archivo utilizando estas funciones:

```
streampos tellg();
streampos tellp();
```

streampos es un tipo definido en **iostream.h**, capaz de almacenar el mayor valor que cualquier función pueda devolver.

EJEMPLOS

1. El siguiente programa muestra el uso de la función **seekp( )**. Permite cambiar un carácter específico de un archivo. Hay que especificar el nombre del archivo en la línea de órdenes, seguido por el número del byte del archivo que se desea cambiar, seguido del nuevo carácter. Observe que el archivo está abierto para operaciones de E/S.

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

main(int argc, char *argv[])
{
  if(argc!=4) {
    cout << "Uso: CHANGE <filename> <byte> <char>\n";
    return 1;
  }

  fstream out(argv[1], ios::in|ios::out);
  if(!out) {
    cout << "El archivo no puede abrirse";
    return 1;
  }

  out.seekp(atoi(argv[2]), ios::beg);
  out.put(*argv[3]);
  out.close();

  return 0;
}
```

2. El próximo programa utiliza **seekg( )** para colocar el puntero get en la mitad de un archivo y a continuación muestra el contenido del archivo desde ese punto. El nombre del archivo y la posición de comienzo de lectura se especifican en la línea de órdenes:

```
// Demuestra el uso de seekg().
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
  char ch;

  if(argc!=3) {
    cout << "Uso: LOCATE <filename> <loc>\n";
    return 1;
  }

  ifstream in(argv[1]);
  if(!in) {
    cout << "El archivo de entrada no puede abrirse.\n";
    return 1;
  }

  in.seekg(atoi(argv[2]), ios::beg);

  while(!in.eof()) {
    in.get(ch);
    cout << ch;
  }

  in.close();

  return 0;
}
```

EJERCICIOS

1. Escriba un programa que muestre un archivo de texto invertido. Sugerencia: piénselo bien antes de crear el programa. La solución es más sencilla de lo que imagina.
2. Construya un programa que intercambie todos los pares de caracteres de un archivo de texto. Por ejemplo, si el archivo contiene «1234», después de ejecutar el programa, el archivo deberá contener «2143». (Puede asumir, por simplicidad, que el archivo contiene un número par de caracteres.)

9.6. COMPROBACION DEL ESTADO DE E/S

El sistema de E/S de C++ mantiene información acerca del estado de los resultados de cada operación de E/S. El estado actual del sistema de E/S se almacena en un entero, en el que se codifican los siguientes indicadores:

Nombre	Significado
goodbit	0 cuando no se producen errores 1 cuando se indica un error
eofbit	0 cuando se encuentra el fin de archivo 1 en cualquier otro caso
failbit	0 cuando ha sucedido un error no fatal de E/S 1 en cualquier otro caso
badbit	0 cuando ha sucedido un error fatal de E/S 1 en cualquier otro caso

Estos indicadores están listados dentro de **ios**.

Existen dos formas de obtener información del estado de E/S. La primera, mediante la llamada al miembro **rdstate( )**. Su prototipo es:

```
int rdstate( );
```

Devuelve el estado actual de los indicadores de error codificados en un entero. Como ya habrá adivinado después de mirar la lista previa de indicadores, **rdstate( )** devuelve cero cuando no se ha producido ningún error. En cualquier otro caso, se activa un bit de error.

El otro modo de determinar si se ha producido un error es mediante la utilización de una o más de estas funciones:

```
int bad( )
int eof( )
int fail( )
int good( )
```

La función **eof( )** se ha discutido anteriormente. La función **bad( )** devuelve verdadero si **badbit** vale 1. La función **fail( )** devuelve verdadero si **failbit** vale 1. La función **good( )** devuelve verdadero si no hay errores. En cualquier otro caso devuelve falso.

Una vez que se ha producido un error, puede ser necesario borrar los indicadores antes de que el programa continúe. Para hacerlo, debe utilizarse la función **clear( )**, cuyo prototipo es:

```
void clear(int flags=0);
```

Si *flag* es cero (valor que toma por omisión), todos los indicadores de error son borrados (puestos a cero). En cualquier otro caso, déle a *flags* el valor que desee.

Nota *El estándar ANSI C++ propuesto define el tipo* **iostate,** *que es un tipo entero capaz de almacenar el estado del sistema de E/S. Este estándar utiliza este tipo, en vez de* **int,** *cuando define funciones como* **rdstate()** *y* **clear( )** *para representar el estado de E/S. Sin embargo, en el momento de escribir este libro, no existe ningún compilador que admita este tipo. Desde un punto de vista práctico, esta discusión es irrelevante.*

EJEMPLOS

1. El siguiente programa ilustra el uso de **rdstate( )**. Muestra el contenido de un archivo de texto. Si se produce un error, la función lo indica utilizando **checkstatus( )**.

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

void checkstatus(ifstream &in);

main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "Uso: DISPLAY <filename>\n";
    return 1;
  }

  ifstream in(argv[1]);

  if(!in) {
    cout << "El archivo de entrada no puede abrirse.\n";
    return 1;
  }

  char c;
  while(in.get(c)) {
    cout << c;
    checkstatus(in);
  }

  checkstatus(in); // comprobación del estado final
    in.close();
    return 0;
}

void checkstatus(ifstream &in)
{
  int i;

  i = in.rdstate();

  if(i & ios::eofbit)
```

```
    cout << "Se ha encontrado EOF\n";
  else if(i & ios::failbit)
    cout << "Error no fatal de E/S\n";
  else if(i & ios::badbit)
    cout << "Error fatal de E/S\n";
}
```

El programa anterior siempre informa de un «error». Tras finalizar el bucle **while**, la llamada final a **checkstatus( )** indica, como se esperaba, que se ha encontrado el carácter EOF.

2. Este programa utiliza **good( )** para detectar un error en un archivo:

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

main(int argc, char *argv[])
{
  char ch;

  if(argc!=2) {
    cout << "PR: <filename>\n";
    return 1;
  }

  ifstream in(argv[1]);
  if(!in) {
    cout << "El archivo de entrada no puede abrirse.\n";
    return 1;
  }

  while(!in.eof()) {
    in.get(ch);
    // comprobación de error
    if(!in.good() && !in.eof()) {
      cout << "Error de E/S...Detención del programa\n";
      return 1;
    }
    cout << ch;
  }

  in.close();

  return 0;
}
```

EJERCICIOS

1. Añada a la respuesta de las preguntas de la sección anterior la comprobación de errores.

9.7. E/S Y ARCHIVOS A MEDIDA

En el capítulo anterior se describió el modo de sobrecargar los operadores de inserción y de extracción con relación a las clases creadas por el programador. En ese capítulo sólo se llevaba a cabo E/S por consola. No obstante, puesto que todos los flujos de C++ son iguales, puede utilizarse, por ejemplo, la misma función insertora sobrecargada, sin apenas cambio alguno, para generar información en pantalla o en un archivo. Esta es una de las características más importantes y útiles del enfoque de C++ en la E/S.

Como se indicó en dicho capítulo, los insertores y extractores sobrecargados, así como los manipuladores de E/S, pueden emplearse con cualquier flujo ya que están desarrollados de una manera general. Si su «código fijo» presenta un flujo específico en una función de E/S, su uso está, obviamente, limitado a ese flujo. Esta es una razón evidente para generalizar, siempre que sea posible, las funciones de E/S construidas.

EJEMPLOS

1. En el siguiente programa, la clase **coord** sobrecarga los operadores << y >>. Observe que puede usar las funciones operadoras para escribir tanto en pantalla como en un archivo:

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

class coord {
  int x, y;
public:
  coord(int i, int j) { x = i; y = j; }
  friend ostream &operator<<(ostream &stream, coord ob);
  friend istream &operator>>(istream &stream, coord &ob);
};

ostream &operator<<(ostream &stream, coord ob)
{
  stream << ob.x << ' ' << ob.y << '\n';

  return stream;
}

istream &operator>>(istream &stream, coord &ob)
{
  stream >> ob.x >> ob.y;

  return stream;
}

main()
```

```
{
  coord o1(1, 2), o2(3, 4);
  ofstream out("test");

  if(!out) {
    cout << "El archivo de salida no puede abrirse\n";
    return 1;
  }

  out << o1 << o2;

  out.close();

  ifstream in("test");
  if(!in) {
    cout << "El archivo de entrada no puede abrirse\n";
    return 1;
  }

  coord o3(0, 0), o4(0, 0);
  in >> o3 >> o4;

  cout << o3 << o4;

  return 0;
}
```

2. Todos los manipuladores de E/S pueden ser utilizados con archivos. Por ejemplo, en esta versión reconstruida de un programa previo de este capítulo, el mismo manipulador que escribe en la pantalla también escribe en un archivo:

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

// Atención:
ostream &atn(ostream &stream)
{
  stream << "Atención: ";
  return stream;
}

// Tome nota, por favor:
ostream &note(ostream &stream)
{
  stream << "Tome nota, por favor: ";
  return stream;
}

main()
{
```

```
  ofstream out("test");
  if(!out) {
    cout << "El archivo de salida no puede abrirse\n";
    return 1;
  }

  // escritura en pantalla
  cout << atn << "Circuito de alto voltaje\n";
  cout << note << "Apague todas las luces\n";

  // escritura en archivo
  out << atn << "Circuito de alto voltaje\n";
  out << note << "Apague todas las luces\n";

  out.close();

  return 0;
}
```

EJERCICIOS

1. Analice los programas del capítulo anterior e intente desarrollarlos para un disco rígido.

COMPROBACION DE APTITUD SUPERIOR

Al llegar a este punto, debería ser capaz de realizar estos ejercicios y de responder a estas preguntas.

1. Construya un manipulador de salida que escriba tres tabulaciones y que después fije la anchura del campo a 20. Demuestre que funciona.
2. Construya un manipulador de entrada que lea y descarte todos los caracteres no alfabéticos. Cuando lea el primer carácter alfabético, el manipulador debe devolverlo al flujo de entrada y finalizar. Llame a este manipulador **findalpha**.
3. Escriba un programa que copie un archivo de texto. Durante la copia, el programa debe invertir el estado mayúsculas/minúsculas de las letras.
4. Desarrolle un programa que lea un archivo de texto y que informe del número de veces que aparece cada letra del alfabeto.
5. Añada a las soluciones de los Ejercicios 3 y 4, si no lo ha hecho, un mecanismo completo de comprobación de errores.
6. ¿Qué función establece el puntero get? ¿Qué función establece el puntero put?

COMPROBACION DE APTITUD INTEGRADA

Esta sección comprueba cómo ha asimilado el contenido de este capítulo con el de los anteriores.

1. El siguiente fragmento de código es una nueva versión de la clase **inventory**, presentada en el capítulo anterior. Desarrolle un programa que complete las funciones **store( )** y **retrieve( )**. A continuación, cree en el disco un pequeño archivo inventario que contenga unos pocos ejemplos. Después, utilizando E/S aleatoria, permita que el usuario obtenga información sobre cada elemento, especificando su número de registro.

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

#define SIZE 40

class inventory {
  char item[SIZE]; // nombre del elemento
  int onhand; // número de existencias
  double cost; // precio del elemento
public:
  inventory(char *i, int o, double c)
  {
    strcpy(item, i);
    onhand = o;
    cost = c;
  }
  void store(fstream &stream);
  void retrieve(fstream &stream);
  friend ostream &operator<<(ostream &stream, inventory ob);
  friend istream &operator>>(istream &stream, inventory &ob);
};

ostream &operator<<(ostream &stream, inventory ob)
{
  stream << ob.item << ": " << ob.onhand;
  stream << " existencias en $" << ob.cost << '\n';

  return stream;
}

istream &operator>>(istream &stream, inventory &ob)
{
  cout << "Introduzca el nombre del elemento: ";
  stream >> ob.item;
  cout << "Introduzca el número de existencias: ";
  stream >> ob.onhand;
  cout << "Introduzca el precio: ";
  stream >> ob.cost;

  return stream;
}
```

2. Le proponemos como desafío particular crear una clase **stack** para caracteres, que almacene los mismos en un disco rígido en vez de en un array de memoria.

10 Funciones virtuales

OBJETIVOS DEL CAPITULO

Este capítulo examina otro aspecto importante de C++: las funciones virtuales. La importancia conferida a las funciones virtuales radica en que son utilizadas para permitir polimorfismo en tiempo de ejecución. El polimorfismo está presente en C++ de dos formas. La primera, en tiempo de compilación, mediante el uso de operadores y funciones sobrecargadas. La segunda, en tiempo de ejecución, mediante el uso de funciones virtuales. Este segundo tipo de polimorfismo proporciona mayor flexibilidad.

La base de las funciones virtuales y del polimorfismo en tiempo de ejecución son los punteros a las clases derivadas. Por esta razón, este capítulo comienza con una discusión de los punteros a las clases derivadas.

COMPROBACION DE APTITUD

Antes de continuar, debería ser capaz de responder a estas preguntas y de realizar estos ejercicios.

1. Cree un manipulador que permita mostrar números en notación científica, utilizando la letra mayúscula E.
2. Escriba un programa que copie un archivo de texto. Durante el proceso de copia, transforme todas las tabulaciones en sus correspondientes espacios en blanco.
3. Escriba un programa que busque en un archivo de texto la palabra especificada en la línea de órdenes. El programa debe obtener el número de veces que ha encontrado dicha palabra.
4. Confeccione la sentencia que fija el puntero put al byte 234 de un archivo enlazado al flujo **out**.
5. ¿Qué funciones informan sobre el estado del sistema de E/S?
6. Mencione alguna de las ventajas de emplear las funciones de E/S de C++ en vez del sistema de E/S de C.

10.1. PUNTEROS A CLASES DERIVADAS

Aunque ya fueron ampliamente discutidos los punteros en el Capítulo 4, se ha pospuesto hasta aquí un aspecto particular de los mismos debido a la estrecha relación guardada con las funciones virtuales.

La característica es ésta: Un puntero declarado como un puntero a una clase base puede también ser utilizado para apuntar a cualquier clase derivada de base. Asumamos, por ejemplo, dos clases, denominadas **base** y **derived**, donde **derived** hereda a **base**. Dada esta situación, las siguientes sentencias son correctas:

```
base *p; // puntero a la clase base

base base_ob; // objeto del tipo base
derived derived_ob; // objeto del tipo derivado

// p puede, por supuesto, apuntar a los objetos base
p = &base_ob; // p apunta a un objeto base
```

```
// p también puede apuntar sin producirse errores a los objetos
   //derivados
p = &derived_ob; // p apunta a un objeto derivado
```

Como sugieren los comentarios, un puntero base puede apuntar a un objeto de cualquier clase derivada de la base sin producirse errores de equivalencia entre tipos.

Aunque puede utilizarse un puntero base para apuntar a un objeto derivado, sólo se puede acceder a los miembros del objeto derivado que fueron heredados de la base. Esto se debe a que el puntero base tiene sólo conocimiento de la clase base. No sabe nada acerca de los miembros añadidos por la clase derivada.

Mientras que está permitido que un puntero base apunte a un objeto derivado, lo contrario no es válido. Un puntero de un tipo derivado no puede ser utilizado para acceder a un objeto de la clase base. (Puede utilizarse un molde de tipo para solventar esta restricción, pero no es una práctica recomendable.)

Un punto final: Recuerde que el puntero aritmético es relativo al tipo de dato al que apunta. Además, si un puntero base apunta a un objeto derivado y a continuación se incrementa el puntero, éste no apuntará al siguiente objeto derivado. Apuntará (al menos eso cree) al próximo objeto base. Tenga cuidado con esto.

EJEMPLO

1. Este pequeño programa ilustra cómo se puede emplear un puntero a una clase base para acceder a una clase derivada:

```
// Demostración del puntero a la clase derivada
#include <iostream.h>
class base {
  int x;
public:
  void setx(int i) { x = i; }
  int getx() { return x; }
};

class derived : public base {
  int y;
public:
  void sety(int i) { y = i; }
  int gety() { return y; }
};

main()
{
  base *p; // puntero al tipo base
  base b_ob; // objeto de base
  derived d_ob; // objeto de derived
  // uso de p para acceder al objeto base
  p = &b_ob;
```

```
  p->setx(10); // acceso al objeto base
  cout << "Objeto base x: " << p->getx() << '\n';

  // uso de p para acceder al objeto derivado
  p = &d_ob; // apunta al objeto derivado
  p->setx(99); // acceso al objeto derivado

  // no se puede utilizar p para fijar y, por lo que lo
  hace directamente d_ob.sety(88);
  cout << "Objeto derivado x: " << p->getx() << ' ';
  cout << "Objeto derivado y: " << d_ob.gety() << '\n';

  return 0;
}
```

Aparte de mostrar los punteros a las clases derivadas, no tiene sentido utilizar un puntero a una clase base de la forma vista en este ejemplo. Sin embargo, la próxima sección describirá por qué son tan importantes los punteros de la clase base para las clases derivadas.

EJERCICIO

1. Ponga en funcionamiento el anterior ejemplo y haga diversas pruebas con él. Por ejemplo, pruebe a declarar un puntero derivado que intente acceder a un objeto de la clase base.

10.2. INTRODUCCION A LAS FUNCIONES VIRTUALES

Una *función virtual* es un miembro de una clase que se declara dentro de una clase base y se redefine en una clase derivada. Para crear una función virtual, debe preceder a la declaración de la función la palabra clave **virtual**. Cuando una clase, que contiene una función virtual, es heredada, la clase derivada redefine la función virtual con respecto a la clase derivada. En el fondo, las funciones virtuales desarrollan la filosofía subyacente en el polimorfismo «una interfaz, múltiples miembros». La función virtual dentro de la clase base define la *forma* de la *interfaz* a esa función. Cada redefinición de la función virtual en una clase derivada expresa el funcionamiento específico de la misma con respecto a esa clase derivada. Esto es, la redefinición crea un *miembro específico*. Cuando se redefine una función virtual en una clase derivada, no es necesaria la palabra clave **virtual**.

Una función virtual puede llamarse exactamente igual que otro miembro. Sin embargo, lo que hace diferente a una función virtual —y capaz de admitir polimorfismo en tiempo de ejecución— es lo que ocurre cuando se llama a la función virtual utilizando un puntero. En la sección anterior se ha visto que un

puntero a la clase base puede ser usado para apuntar a un objeto de una clase derivada. Cuando un puntero base apunta a un objeto derivado que contiene una función virtual y esa función es llamada a través de ese puntero, el compilador determina la versión de la función a la que llamar, dependiendo del tipo de objeto apuntado por el puntero. Dicho de otro modo, es el tipo del objeto apuntado el que determina la versión de la función virtual que será ejecutada. Por tanto, si de una clase base, que contiene una función virtual, derivan dos o más clases derivadas, cuando el puntero base apunta a diferentes objetos, se ejecutan diferentes versiones de la función virtual. Mediante este proceso se logra el polimorfismo en tiempo de ejecución. De hecho, una clase que contiene una función virtual se denomina *clase polimórfica.*

EJEMPLOS

1. Este es un breve ejemplo que utiliza una función virtual:

```
// Sencillo ejemplo de uso de una función virtual.
#include <iostream.h>

class base {
public:
  int i;
  base(int x) { i = x; }
  virtual void func()
  {
    cout << "Uso de la versión de func() de base: ";
    cout << i << '\n';
  }
};

class derived1 : public base {
public:
  derived1(int x) : base(x) {}
  void func()
  {
    cout << "Uso de la versión de func() de derived1: ";
    cout << i*i << '\n';
  }
};

class derived2 : public base {
public:
  derived2(int x) : base(x) {}
  void func()
  {
    cout << "Uso de la versión de func() de derived2: ";
    cout << i+i << '\n';
  }
};

  main()
```

```
{
  base *p;
  base ob(10);
  derived1 d_ob1(10);
  derived2 d_ob2(10);

  p = &ob;
  p->func(); // uso de func() de base

  p = &d_ob1;
  p->func(); // uso de func() de derived1
  p = &d_ob2;
  p->func(); // uso de func() de derived2

  return 0;
}
```

Este programa presenta la siguiente salida:

```
Uso de la versión de func() de base: 10
Uso de la versión de func() de derived1: 100
Uso de la versión de func() de derived2: 20
```

La redefinición de una función virtual dentro de una clase derivada, a priori, puede parecer similar a la sobrecarga de funciones. Sin embargo, ambos procesos son sustancialmente diferentes. En primer lugar, una función sobrecargada debe diferir en el tipo y/o en el número de parámetros, mientras que una función virtual redefinida debe tener exactamente el mismo tipo y número de parámetros y devolver el mismo tipo. (De hecho, si se cambia el tipo o el número de parámetros cuando se redefine una función virtual, se transforma en una función sobrecargada, perdiendo su naturaleza virtual.) Además, las funciones virtuales deben ser miembros de una clase. Esto no tiene que cumplirlo una función sobrecargada. Las funciones destructoras pueden ser virtuales mientras que las constructoras no. Debido a las diferencias entre las funciones sobrecargadas y las funciones virtuales redefinidas, se utiliza el término *supeditación* para describir la redefinición de una función virtual.

Como puede observarse, el programa del ejemplo crea tres clases. La clase **base** define la función virtual **func( )**. Esta clase es heredada después por **derived1** y **derived2**. Cada una de estas clases redefine a **func( )** con su funcionamiento individual. Dentro de **main( )**, el puntero a la clase base **p** se declara junto con los objetos de tipo **base**, **derived1** y **derived2**. En primer lugar, **p** se asigna a la dirección de **ob** (un objeto de tipo **base**). Cuando se llama a **func( )** utilizando **p**, se emplea la versión de **base**. A continuación se asigna **p** a la dirección de **d_ob1** y se llama a **func( )** de nuevo. Puesto que es el tipo del objeto apuntado el que determina la función virtual a la que llamar, esta vez se ejecuta la versión redefinida por **derived1**. Por último, **p** se asigna a la dirección de **d_ob2** y se vuelve a llamar a **func( )**. En esta ocasión, se ejecuta la versión de **func( )** definida dentro de **derived2**.

Los punto críticos, que hay que entender del ejemplo anterior, son que el tipo del objeto apuntado es el que determina la versión de la función virtual redefinida que será ejecutada cuando se acceda a través del puntero de la clase base, y que esta decisión se toma en tiempo de ejecución.

2. Las funciones virtuales son jerárquicas de acuerdo al orden de herencia. Además, cuando una clase derivada *no* redefine una función virtual, se utiliza la función definida dentro de su clase base. En este ejemplo se muestra una versión ligeramente diferente del programa anterior:

```
// Las funciones virtuales son jerárquicas.
#include <iostream.h>

class base {
public:
  int i;
  base(int x) { i = x; }
  virtual void func()
  {
    cout << "Uso de la versión de func() de base: ";
    cout << i << '\n';
  }
};

class derived1 : public base {
public:
  derived1(int x) : base(x) {}
  void func()
  {
    cout << "Uso de la versión de func() de derived1: ";
    cout << i*i << '\n';
  }
};

class derived2 : public base {
public:
  derived2(int x) : base(x) {}
  // derived2 no redefine func()
};

main()
{
  base *p;
  base ob(10);
  derived1 d_ob1(10);
  derived2 d_ob2(10);
  p = &ob;
  p->func(); // uso de func() de base

  p = &d_ob1;
  p->func(); // uso de func() de derived1
  p = &d_ob2;
  p->func(); // uso de func() de base

  return 0;
}
```

Este programa presenta la siguiente salida:

```
Uso de la versión de func() de base: 10
Uso de la versión de func() de derived1: 100
Uso de la versión de func() de base: 10
```

En esta versión, **derived2** no redefine **func( )**. Cuando **p** se asigna a **d_ob2** y se llama a **func( )**, se utiliza la versión de **base** porque es la siguiente en la jerarquía de clases. En general, cuando una clase derivada no redefine una función virtual, se utiliza la versión de la clase base.

3. El próximo ejemplo muestra el modo de reaccionar de una función virtual cuando, en tiempo de ejecución, se producen sucesos aleatorios. Este programa selecciona entre **d_ob1** y **d_ob2** en base al valor devuelto por el generador estándar de números aleatorios **rand( )**. No olvide que la versión de **func( )** que se ejecutará se determina en tiempo de ejecución. (En realidad, es imposible resolver las llamadas a **func( )** en tiempo de compilación.)

```
/* Este ejemplo muestra cómo puede utilizarse una función
   virtual para responder a los sucesos aleatorios producidos
   en tiempo de ejecución.
*/
#include <iostream.h>
#include <stdlib.h>

class base {
public:
  int i;
  base(int x) { i = x; }
  virtual void func()
  {
    cout << "Uso de la versión de func() de base: ";
    cout << i << '\n';
  }
};

class derived1 : public base {
public:
  derived1(int x) : base(x) {}
  void func()
  {
    cout << "Uso de la versión de func() perteneciente a
    derived1: ";
    cout << i*i << '\n';
  }
};

class derived2 : public base {
  public:
    derived2(int x) : base(x) {}
    void func()
    {
    cout << "Uso de la versión de func() perteneciente
    a derived2: ";
```

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

main()
{
  base *p;
  derived1 d_ob1(10);
  derived2 d_ob2(10);
  int i, j;

  for(i=0; i<10; i++) {
    j = rand();
    if((j%2)) p = &d_ob1; // si es un uso impar de d_ob1
    else p = &d_ob2; // si es un uso par de d_ob2
    p->func(); // llamada a la función apropiada
  }

  return 0;
}
```

4. A continuación se presenta un ejemplo más práctico sobre el uso de una función virtual. Este programa crea una clase base genérica llamada **area** que almacena dos dimensiones de una figura. También declara una función virtual llamada **getarea( )** que, cuando es redefinida por las clases derivadas, devuelve el área del tipo de la figura definida por la clase derivada. En este caso, la declaración de **getarea( )** dentro de la clase base determina la naturaleza de la interfaz. El desarrollo, propiamente dicho, se deja para las clases que la heredan. En este ejemplo se calcula el área de un triángulo y el de un rectángulo:

```
// Uso de una función virtual para definir una interfaz.
#include <iostream.h>

class area {
  double dim1, dim2; // dimensiones de la figura
public:
  void setarea(double d1, double d2)
  {
    dim1 = d1;
    dim2 = d2;
  }
  void getdim(double &d1, double &d2)
  {
    d1 = dim1;
    d2 = dim2;
  }
  virtual double getarea()
  {
    cout << "Usted debe redefinir esta función\n";
    return 0.0;
  }
};

class rectangle : public area {
```

```
public:
  double getarea()
  {
    double d1, d2;
    getdim(d1, d2);
    return d1 * d2;
  }
};

class triangle : public area {
public:
  double getarea()
  {
    double d1, d2;

    getdim(d1, d2);
    return 0.5 * d1 * d2;
  }
};

main()
{
  area *p;
  rectangle r;
  triangle t;

  r.setarea(3.3, 4.5);
  t.setarea(4.0, 5.0);

  p = &r;
  cout << "El área del rectángulo es: " << p->getarea()
  << '\n';

  p = &t;
  cout << "El área del triángulo es: " << p->getarea()
  << '\n';

  return 0;
}
```

Observe que la definición de **getarea( )** dentro de **area** es sólo un resguardo y no lleva a cabo ninguna función real. Puesto que **area** no está vinculado a ningún tipo específico de figura, no puede darse una definición significativa de **getarea( )** dentro de **area**. De hecho, para que **getarea( )** sea útil debe ser redefinida por una clase derivada. En la próxima sección se profundizará en ello.

EJERCICIOS

1. Construya un programa que cree una clase base denominada **num**. Esta clase debe almacenar un valor entero y debe contener una función virtual denominada **showNum( )**. Cree dos clases derivadas llamadas **outhex** y **outoct** que hereda **num**. Las

clases derivadas redefinen **shownum( )** de manera que esta función muestre el valor en hexadecimal y en octal, respectivamente.

2. Construya un programa que cree una clase base llamada **distance**, que almacene la distancia entre dos puntos en una variable **double**. En **distance**, cree una función virtual llamada **trav_time( )** que obtenga el tiempo que cuesta recorrer esa distancia, asumiendo que la distancia está en millas y que la velocidad es de 60 millas por hora. En una clase derivada llamada **metric**, se redefine **trav_time( )** de modo que obtenga el tiempo del recorrido asumiendo que la distancia está en kilómetros y que la velocidad es de 100 kilómetros por hora.

10.3. MAS SOBRE FUNCIONES VIRTUALES

Como muestra el Ejemplo 4 de la sección anterior, a veces cuando una función virtual se declara en la clase base, no hay ninguna operación significativa que llevar a cabo. Esta situación es normal porque, a menudo, una clase base no define por sí misma una clase completa. En su lugar, simplemente ofrece un conjunto central de miembros y variables completados por las clases derivadas. Cuando no existe ninguna acción significativa a llevar a cabo por una función virtual de una clase base, la consecuencia es que la clase derivada debe redefinir esta función. Para asegurarse de que esto tiene lugar, C++ permite *funciones virtuales puras.*

Una función virtual pura no tiene definición con respecto a la clase base. Sólo se incluye el prototipo de la función. Para construir una función virtual pura debe usarse la siguiente forma general:

virtual *type func-name*(*parameter-list*) = 0;

La parte fundamental de esta declaración reside en hacer que la función sea cero. Esto le indica al compilador que no existe cuerpo para esta función con respecto a la clase base. Cuando se construye una función virtual pura, se obliga a que sea redefinida por cualquier clase derivada. Si la clase derivada no lo hace, tiene lugar un error en tiempo de compilación. Además, la confección de una función virtual pura es una forma de garantizar que una clase derivada proporcione su propia redefinición.

Cuando una clase contiene como mínimo una función virtual pura, se denomina *clase abstracta.* Una clase abstracta contiene como mínimo una función para la que no existe cuerpo, lo que da lugar, técnicamente, a un tipo incompleto y a que no puedan crearse objetos de esa clase. Además, las clases abstractas sólamente existen para ser heredadas. No pretenden ni tienen capacidad alguna por sí mismas. Es importante entender, sin embargo, que se puede crear un puntero a una clase abstracta, puesto que el polimorfismo en tiempo de ejecución se consigue mediante el uso de punteros a la clase base. (También se permite una referencia a una clase abstracta.)

Cuando se hereda una función virtual, conserva su naturaleza virtual. Esto significa que cuando una clase derivada hereda una función virtual de una clase

base y después se utiliza la clase derivada como clase base de otra nueva clase derivada, la función virtual puede ser redefinida por la clase final derivada (así como por la primera clase derivada). Por ejemplo, si la clase base B contiene una función virtual llamada **f( )** y D1 hereda B y D2 hereda D1, tanto D1 como D2 pueden redefinir **f( )** en relación a sus respectivas clases.

EJEMPLOS

1. En este ejemplo puede verse una versión mejorada del programa mostrado en el Ejemplo 4 de la sección previa. En esta versión, la función **getarea( )** se declara como pura en la clase base **area**:

```
// Creación de una clase abstracta.
#include <iostream.h>

class area {
  double dim1, dim2; // dimensiones de la figura
public:
  void setarea(double d1, double d2)
  {
    dim1 = d1;
    dim2 = d2;
  }
  void getdim(double &d1, double &d2)
  {
    d1 = dim1;
    d2 = dim2;
  }
  virtual double getarea() = 0; // función virtual pura
};

class rectangle : public area {
public:
  double getarea()
  {
    double d1, d2;

    getdim(d1, d2);
    return d1 * d2;
  }
};

class triangle : public area {
public:
  double getarea()
  {
    double d1, d2;

    getdim(d1, d2);
    return 0.5 * d1 * d2;
  }
};
```

```
main()
{
  area *p;
  rectangle r;
  triangle t;

  r.setarea(3.3, 4.5);
  t.setarea(4.0, 5.0);

  p = &r;
  cout << "El área del rectángulo es: " << p->getarea() << '\n';

  p = &t;
  cout << "El área del triángulo es: " << p->getarea() << '\n';

  return 0;
}
```

Ahora que **getarea( )** es pura, se ha asegurado la redefinición de la misma en cada clase derivada.

2. El siguiente programa presenta cómo se conserva la naturaleza de una función virtual cuando es heredada:

```
// Las funciones virtuales conservan su naturaleza virtual
   //cuando son heredadas.
#include <iostream.h>

class base {
public:
  virtual void func()
  {
    cout << "Uso de la versión de func() de base\n";
  }
};

class derived1 : public base {
public:
  void func()
  {
    cout << "Uso de la versión de func() de derived1\n";
  }
};

// Derived2 hereda a derived1.
   class derived2 : public derived1 {
public:
  void func()
  {
    cout << "Uso de la versión de func() de derived2\n";
  }
};

main()
```

```
{
  base *p;
  base ob;
  derived1 d_ob1;
  derived2 d_ob2;

  p = &ob;
  p->func(); // uso de func() de base

  p = &d_ob1;
  p->func(); // uso de func() de derived1

  p = &d_ob2;
  p->func(); // uso de func() de derived2

  return 0;
}
```

En este programa, primero, la función virtual **func( )** es heredada por **derived1**, que la redefine con respecto a ella. A continuación, **derived2** hereda **derived1**. En **derived2**, se redefine nuevamente **func( )**.

Puesto que las funciones virtuales son jerárquicas, si **derived2** no hubiera redefinido a **func( )**, se habría hecho uso de **func( )** de **derived1** en el acceso a **d_ob2**. Si **derived1** y **derived2** no hubieran redefinido **func( )**, todas las referencias a ella se habrían dirigido a la función definida en **base**.

EJERCICIOS

1. Compruebe los dos programas anteriores. En particular, intente crear un objeto a partir de **area** del Ejemplo 1 y observe el mensaje de error. En el Ejemplo 2, intente extraer la definición de **func( )** de **derived2**. Confirme que se utiliza, realmente, la versión incluida en **derived1**.
2. ¿Por qué no puede crearse un objeto utilizando una clase abstracta?
3. En el Ejemplo 2, ¿qué sucede si se extrac sólo la redefinición de **func( )** de **derived1**? ¿Puede compilarse y ejecutarse el programa? Si así fuera ¿por qué?

10.4. APLICACION DE POLIMORFISMO

Ahora que sabe cómo utilizar una función virtual para conseguir polimorfismo en tiempo de ejecución, es el momento de considerar cómo y por qué usarlo. Como se ha repetido muchas veces en este libro, el polimorfismo es el proceso mediante el cual una interfaz común se aplica a dos o más situaciones similares (pero técnicamente diferentes), llevándose a cabo además la filosofía de «una interfaz, múltiples miembros».

El polimorfismo es importante porque puede simplificar enormemente sistemas complejos. Una sencilla y bien definida interfaz se emplea para acceder a

un número de acciones diferentes pero relacionadas, eliminándose de este modo la complejidad artificial. En el fondo, el polimorfismo permite que la relación lógica entre acciones similares llegue a ser cierta; además, el programa es más fácil de entender y de mantener. Cuando se accede a acciones relacionadas a través de una interfaz común, es menor lo que hay que recordar.

Existen dos términos que se vinculan a menudo con la POO, en general, y con C++, en particular. Son la *vinculación anticipada* y la *vinculación postergada.* Es importante saber lo que significan. La vinculación anticipada se refiere a aquellos sucesos que pueden conocerse en tiempo de compilación. En particular, se refiere a las llamadas a función que pueden resolverse durante la compilación. Las entidades con vinculación anticipada incluyen funciones «normales», funciones sobrecargadas y miembros y funciones amigas que no sean virtuales. Cuando se compila este tipo de funciones, toda la información necesaria sobre su dirección de llamada se conoce en tiempo de compilación. La ventaja principal de la vinculación anticipada (y la razón por la que es ampliamente utilizada) es que es muy eficaz. Las llamadas a funciones ligadas en tiempo de compilación son los tipos de llamadas a función más rápidas. La desventaja fundamental es la ausencia de flexibilidad.

La vinculación postergada se refiere a los sucesos que ocurren en tiempo de ejecución. La llamada a una función de vinculación postergada es aquella en la que la dirección de la función llamada no se conoce hasta que el programa se ejecute. En C++, una función virtual es un objeto de vinculación postergada. Cuando se accede a una función virtual a través de un puntero a una clase base, el programa debe determinar en tiempo de ejecución a qué tipo de objeto está apuntando y después seleccionar la versión de la función redefinida que se va a ejecutar. La ventaja principal de la vinculación postergada es la flexibilidad en tiempo de ejecución. El programa es libre de responder a los sucesos aleatorios sin la necesidad de contener cantidades enormes de «código para imprevistos». Su desventaja fundamental es la existencia de una sobrecarga superior asociada a la llamada a la función. Esto provoca que las llamadas sean, en general, más lentas que las que tienen lugar con la vinculación anticipada.

Debido a los registros de eficacia potencial, el programador debe decidir cuándo es apropiado utilizar vinculación anticipada o postergada.

EJEMPLOS

1. Este programa ilustra la filosofía «una interfaz, múltiples miembros». Desarrolla una clase genérica de lista enlazada para valores enteros. También declara la naturaleza de la interfaz a una lista. Para almacenar un valor, puede llamarse a la función **store( )**. Para extraer un valor de la lista, puede llamarse a **retrieve( )**. La clase base **list** no define miembros implícitos para estas acciones. En su lugar, cada clase derivada define exactamente el tipo de lista que mantendrá. En el programa se crean dos tipos de listas: una cola y una pila. Sin embargo, aunque las dos listas trabajan de forma diferente, se accede a ellas a través de la misma interfaz. Debe estudiar este programa con detalle:

```
// Creación de una clase genérica lista para enteros.
#include <iostream.h>
#include <stdlib.h>
#include <ctype.h>

class list {
public:
  list *head; // puntero al principio de la lista
  list *tail; // puntero al final de la lista
  list *next; // puntero al siguiente elemento
  int num; // valor que va a ser almacenado
  list() { head = tail = next = NULL; }
  virtual void store(int i) = 0;
  virtual int retrieve() = 0;
};

// Creación de una cola de tipo lista.
class queue : public list {
public:
  void store(int i);
  int retrieve();
};

void queue::store(int i)
{
  list *item;

  item = new queue;
  if(!item) {
    cout << "Error de asignación\n";
    exit(1);
  }
  item->num = i;

  // colocación por el final de la lista
  if(tail) tail->next = item;
  tail = item;
  item->next = NULL;
  if(!head) head = tail;
}

int queue::retrieve()
{
  int i;
  list *p;

  if(!head) {
    cout << "Lista vacía\n";
    return 0;
  }

  // extracción por el comienzo de la lista
  i = head->num;
  p = head;
```

```
  head = head->next;
  delete p;

  return i;
}

// Creación de una pila de tipo lista.
class stack : public list {
public:
  void store(int i);
  int retrieve();
};

void stack::store(int i)
{
  list *item;

  item = new stack;
  if(!item) {
    cout << "Error de asignación\n";
    exit(1);
  }
  item->num = i;

  // colocación al comienzo de la lista para una operación
  //como en una pila
  if(head) item->next = head;
  head = item;
  if(!tail) tail = head;
}

int stack::retrieve()
{
  int i;
  list *p;

  if(!head) {
    cout << "Lista vacía\n";
    return 0;
  }

  // extracción del comienzo de la lista
  i = head->num;
  p = head;
  head = head->next;
  delete p;

  return i;
  }

main()
{
  list *p;
```

```
  // demostración de la cola
  queue q_ob;

  p = &q_ob; // apunta a cola

  p->store(1);
  p->store(2);
  p->store(3);

  cout << "Cola: ";
  cout << p->retrieve();
  cout << p->retrieve();
  cout << p->retrieve();

  cout << '\n';

  // demostración de pila
  stack s_ob;

  p = &s_ob; // apunta a pila

  p->store(1);
  p->store(2);
  p->store(3);

  cout << "Pila: ";
  cout << p->retrieve();
  cout << p->retrieve();
  cout << p->retrieve();

  cout << '\n';

  return 0;
}
```

2. La función **main( )**, del programa sobre listas recién presentado, sólo muestra que las clases lista realmente funcionan. Sin embargo, para comenzar a ver por qué es tan potente el polimorfismo intente, en su lugar, utilizar este **main( )**:

```
main()
{
  list *p;
  stack s_ob;
  queue q_ob;
  char ch;
  int i;

  for(i=0; i<10; i++) {
    cout << "¿Pila o Cola? (P/C): ";
    cin >> ch;
    ch = tolower(ch);
    if(ch=='q') p = &q_ob;
    else p = &s_ob;
    p->store(i);
  }
```

```
  cout << "Introduzca F para finalizar\n";
  for(;;) {
    cout << "¿Extracción de la pila o de la cola? (P/C): ";
    cin >> ch;
    ch = tolower(ch);
    if(ch=='t') break;
    if(ch=='q') p = &q_ob;
    else p = &s_ob;
    cout << p->retrieve() << '\n';
  }

  cout << '\n';

  return 0;
}
```

Este **main( )** manifiesta cómo los sucesos aleatorios, que tienen lugar en tiempo de ejecución, puede gestionarse fácilmente utilizando funciones virtuales y polimorfismo en tiempo de ejecución. El programa ejecuta un bucle **for** desde 0 hasta 9. En cada iteración del bucle le pide elegir el tipo de lista —pila o cola— en la que se va a colocar el valor. De acuerdo a la respuesta obtenida, se fija el puntero a base **p** para que apunte al objeto correcto y para que el valor sea almacenado. Una vez que finaliza el bucle, comienza otro, que pide el tipo de lista del que extraer un valor. De nuevo, es su respuesta la que determina la lista seleccionada.

A pesar de que este ejemplo es trivial, es necesario que usted sepa cómo el polimorfismo puede simplificar un programa que debe responder ante sucesos aleatorios. Como ya sabe, la mayoría de los sistemas operativos basados en ventanas, como Windows y OS/2, realizan la interfaz con un programa mediante el envío de mensajes. Estos mensajes se generan aleatoriamente y su programa debe responder a ellos a medida que los recibe. Como puede adivinarse, el polimorfismo en tiempo de ejecución es muy útil en los programas escritos para estos sistemas operativos.

EJERCICIOS

1. Añada otro tipo de lista al programa del Ejemplo 1. Haga que esa versión mantenga una lista ordenada (en orden ascendente). Llame a la lista **stored**.
2. Piense en diferentes formas en las que puede aplicarse polimorfismo en tiempo de ejecución para simplificar la solución a ciertos tipos de problemas.

COMPROBACION DE APTITUD SUPERIOR

Al llegar a este punto, debería ser capaz de realizar estos ejercicios y de responder a estas preguntas.

1. ¿Qué es una función virtual?
2. ¿Qué tipos de funciones no pueden hacerse virtuales?

3. ¿Cómo facilita una función virtual el logro de polimorfismo en tiempo de ejecución?
4. ¿Qué es una función virtual pura?
5. ¿Qué es una clase abstracta? ¿Qué es una clase polimórfica?
6. ¿Es correcto el siguiente fragmento de código? Si no es así, ¿por qué?

```
class base {
public:
  virtual int f(int a) = 0;
  // ...
};

class derived : public base {
public:
  int f(int a, int b) { return a*b; }
  // ...
};
```

7. ¿Se hereda la cualidad de virtual?
8. Experimente ahora con las funciones virtuales. Es un concepto importante y debería dominar la técnica.

COMPROBACION DE APTITUD INTEGRADA

Esta sección comprueba cómo ha asimilado el contenido de este capítulo con el de los anteriores.

1. Mejore la lista ejemplo de la Sección 10.4, Ejemplo 1, de modo que sobrecargue los operadores + y −. Haga que el + almacene un elemento y que el − extraiga un elemento.
2. ¿En qué se diferencian las funciones virtuales de las funciones sobrecargadas?
3. Examine de nuevo algunos de los ejemplos de sobrecarga de funciones presentadas a lo largo del libro. Determine cuál de ellas puede convertirse en una función virtual. Piense, también, en la manera en que una función virtual puede resolver algunos de sus propios problemas de programación.

11 Plantillas y manejo de excepciones

OBJETIVOS DEL CAPITULO

Se han añadido recientemente a C++ dos nuevas características: *plantillas* y *manejo de excepciones*. A pesar de que ninguna de ellas formó parte de la especificación original de C++, ambas están definidas en el estándar ANSI C++ propuesto y están admitidas en la mayoría de los compiladores actuales de C++. Sin embargo, si se dispone de un compilador antiguo, éste puede no contemplar alguna de estas características. (Es necesario comprobar el manual de usuario.)

Mediante el uso de plantillas es posible crear funciones y clases genéricas. En una función o una clase genérica el tipo de datos que se utiliza se especifica como un parámetro. Además, una función o una clase puede utilizarse con diferentes tipos de datos, sin tener que codificar de nuevo explícitamente una versión específica para cada tipo de datos. En este capítulo se describen tanto las funciones como las clases genéricas.

El manejo de excepciones es el subsistema de C++ que permite la gestión estructurada y controlada de los errores acaecidos en tiempo de ejecución. La utilización del manejo de excepciones de C++ permite que un programa pueda llamar automáticamente a una rutina de manejo de excepciones cuando se produce un error. La ventaja principal del manejo de excepciones es que automatiza el código de tratamiento de errores, que anteriormente tenía que ser codificado «a mano» dando lugar a enormes programas.

COMPROBACION DE APTITUD

Antes de continuar, debería ser capaz de responder a estas preguntas y de realizar estos ejercicios.

1. ¿Qué es una función virtual?
2. ¿Qué es una función virtual pura? Si la declaración de una clase contiene la declaración de una función virtual pura , ¿cómo es la llamada a esa clase ? y ¿qué restricciones se aplican para su uso?
3. El polimorfismo en tiempo de ejecución se consigue mediante el uso de funciones _____ y punteros a clases _____. (Complete las palabras que faltan.)
4. Si en una jerarquía de clases una clase derivada no redefine una función virtual (no pura), ¿qué sucede cuando un objeto de la clase derivada referencia a esa función?
5. ¿Cuál es la principal ventaja del polimorfismo en tiempo de ejecución? ¿Cuál es su desventaja potencial?

11.1. FUNCIONES GENERICAS

Una función genérica define un conjunto general de operaciones que se aplicarán a diferentes tipos de datos. Una función genérica tiene como parámetro el tipo de datos que le son pasados para que opere sobre ellos. La utilización de este mecanismo permite que pueda aplicarse el mismo procedimiento general sobre

un amplio rango de datos. Como ya sabe, muchos algoritmos son, desde el punto de vista de la lógica, iguales independientemente del tipo de datos sobre los que trabajan. Por ejemplo, el algoritmo Quicksort es el mismo se aplique sobre un array de enteros o de flotantes. Sólo son diferentes los tipos de datos que van a ser clasificados. La creación de una función genérica permite definir, independientemente de los datos, la naturaleza del algoritmo. Una vez hecho esto, el compilador genera automáticamente el código adecuado para cada tipo de datos que, en realidad, se utiliza cuando se ejecuta la función. Cuando se crea una función genérica se crea realmente una función que puede sobrecargarse a sí misma automáticamente.

Una función genérica se crea utilizando la palabra clave **template**. El significado común de la palabra «plantilla» refleja exactamente su uso en C++. Se utiliza para crear una plantilla (o esquema) que describe lo qué hará una función, dejando que el compilador complete los detalles, a medida que los necesite. La forma general de la definición de una función **template** es ésta:

```
template <class Ttype> ret-type func-name(parameter list)
{
   // cuerpo de la función
}
```

Ttype es el nombre resguardo para el tipo de datos usado por la función. Se puede utilizar este nombre dentro de la definición de la función. Sin embargo, sólo es un resguardo que el compilador reemplazará automáticamente con un tipo de datos real cuando cree la versión específica de la función.

EJEMPLOS

1. El siguiente programa crea una función genérica que intercambia el valor de las dos variables con las que se la llama. Debido a que el proceso general de intercambio de dos valores es independiente del tipo de variables, este proceso es una buena elección para ser efectuada por una función genérica:

```
// Ejemplo de una función plantilla.
#include <iostream.h>

// Esta es una función plantilla.
template <class X> void swap(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
}

main()
```

```
{
  int i=10, j=20;
  float x=10.1, y=23.3;

  cout << "Original i, j: " << i << ' ' << j << endl;
  cout << "Original x, y: " << x << ' ' << y << endl;

  swap(i, j); // intercambia enteros
  swap(x, y); // intercambia flotantes

  cout << "Original i, j: " << i << ' ' << j << endl;
  cout << "Original x, y: " << x << ' ' << y << endl;

  return 0;
}
```

La palabra clave **template** se utiliza para definir una función genérica. La línea:

```
template <class X> void swap(X &a, X &b)
```

le indica al compilador dos cosas: que se ha creado una plantilla y que está comenzando una definición genérica. En este caso, X es un tipo genérico que se utiliza como resguardo. Después del fragmento **template** se declara la función **swap( )**, donde X es el tipo de datos de los valores que serán intercambiados. En **main( )** se llama a **swap( )** utilizando dos tipos diferentes de datos: enteros y flotantes. Puesto que **swap( )** es una función genérica, el compilador crea automáticamente dos versiones de **swap( )** —una que intercambia valores enteros y otra que intercambia valores flotantes. Ahora, debería compilar y ejecutar este programa.

Veamos otros términos que a veces se utilizan cuando se habla de plantillas y que pueden encontrarse en otra literatura sobre C++. El primero es que una función genérica (esto es, la definición de una función precedida por la palabra **template**) también se denomina *función plantilla*. Cuando un compilador crea una versión específica de esta función se dice que ha creado una *función generada*. La acción de generar una función se referencia como *creación de una instancia* de ella. Dicho de otro modo, una función generada es una instancia específica de una función plantilla.

2. Técnicamente, la parte **template** de la definición de una función genérica no tiene que estar en la misma línea que el nombre de la función. Por ejemplo, lo que se muestra a continuación es también una forma normal de dar formato a la función **swap( )**:

```
template <class X>
   void swap(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
}
```

Si usa esta forma, es importante entender que no puede aparecer ninguna otra sentencia entre la sentencia **template** y el comienzo de la definición de la función genérica. Por ejemplo, el siguiente fragmento de código no compilará correctamente:

```
// Esto no compilará.
template <class X>
int i; // esto es un error
void swap(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
}
```

Como indican los comentarios, la especificación **template** debe preceder directamente a la definición de la función.

3. Puede definirse más de un tipo genérico de datos utilizando en la sentencia **template**, una lista separada por comas. Por ejemplo, este programa crea una función genérica que tiene dos tipos genéricos de datos:

```
#include <iostream.h>

template <class type1, class type2>
void myfunc(type1 x, type2 y)
{
  cout << x << ' ' << y << endl;
}
main()
{
  myfunc(10, "hola");

  myfunc(0.23, 10L);

  return 0;
}
```

En este ejemplo, cuando el compilador genera las instancias específicas de **myfunc( )** reemplaza el resguardo de los tipos **type1** y **type2** por los tipos de datos **int** y **char*** y **double** y **long**, respectivamente.

Recuerde *Cuando cree una función genérica, en el fondo, se le está permitiendo al compilador que genere tantas versiones diferentes de esa función, como sean necesarias para gestionar las distintas llamadas que el programa hará a dicha función.*

4. Las funciones genéricas son equivalentes a las funciones sobrecargadas, excepto que son más restrictivas. Cuando las funciones se sobrecargan, dentro del cuerpo de cada función pueden suceder acciones diferentes. Pero una función genérica debe

realizar la misma función general para todas las versiones. Por ejemplo, las siguientes funciones sobrecargadas *no* podrían ser reemplazadas por una función genérica porque no realizan la misma tarea:

```
void outdata(int i)
{
  cout << i;
}

void outdata(double d)
{
  cout << setprecision(10) << setfill('#');
  cout << d;
  cout << setprecision(6) << setfill(' ');
}
```

5. Aunque una función plantilla puede sobrecargarse a sí misma, si es necesario, también es posible sobrecargarla explícitamente. Si sobrecarga una función genérica, la función sobrecargada redefine (u «oculta») la función genérica relativa a esa versión específica. Considere esta versión del Ejemplo 1:

```
// Redefinición de una función plantilla.
#include <iostream.h>

template <class X> void swap(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
}

// Redefinición de la versión genérica de swap().
  void swap(int a, int b)
{
  cout << "esto está dentro de swap(int,int)\n";
}

main()
{
  int i=10, j=20;
  float x=10.1, y=23.3;


  cout << "Original i, j: " << i << ' ' << j << endl;
  cout << "Original x, y: " << x << ' ' << y << endl;

  swap(i, j); // llamada a la versión de swap() sobrecargada
  explícitamente
  swap(x, y); // intercambio de flotantes

  cout << "Original i, j: " << i << ' ' << j << endl;
  cout << "Original x, y: " << x << ' ' << y << endl;


  return 0;
}
```

Como indican los comentarios, cuando se llama a **swap(i,j)**, se invoca a la versión de **swap( )** sobrecargada explícitamente en el programa. Además, el compilador no genera esta versión de la función genérica **swap( )** porque la función genérica está redefinida por la sobrecarga explícita.

La sobrecarga manual de una plantilla, como se ha mostrado en este ejemplo, permite confeccionar una versión de una función genérica para que se adapte a una situación especial. Sin embargo, en general, si necesita tener diferentes versiones de una función para diferentes tipos de datos, debería usar funciones sobrecargadas en lugar de plantillas.

EJERCICIOS

1. Pruebe los programas anteriores, si aún no lo ha hecho.
2. Escriba una función genérica, llamada **min( )**, que devuelve el menor de sus dos argumentos. Por ejemplo, **min(3,4)** devolverá **3** y **min('c','a')** devolverá **a**. Compruebe el buen funcionamiento de la función dentro de un programa.
3. Un buen candidato a función plantilla es **find( )**. Esta función busca un array para un objeto. Devuelve el índice del objeto equiparado (si lo encuentra) o -1 si no se produce equiparación. Aquí se da el prototipo de una versión específica de **find( )**. Convierta **find( )** en una función genérica y compruebe que funciona. (El parámetro **size** especifica el número de elementos del array.)

```
int find(int object, int *list, int size)
{
  // ...
}
```

4. Explique, con sus propios términos, el gran valor de las funciones genéricas y su contribución a la hora de simplificar el código fuente de los programas creados.

11.2. CLASES GENERICAS

Además de las funciones genéricas, también puede definirse una clase genérica. Esto significa crear una clase que define todos los algoritmos usados por ella, pero el tipo de datos que, realmente, va a ser manipulado se especificará como un parámetro cuando se creen los objetos de esa clase.

Las clases genéricas son útiles cuando una clase contiene una lógica generalizable. Por ejemplo, el mismo algoritmo que almacena una cola de enteros también funciona para una cola de caracteres. El mismo mecanismo que almacena una lista enlazada de direcciones postales podría almacenar una lista enlazada con información sobre piezas de vehículos. Mediante una clase genérica se puede crear una clase que almacenará una cola, una lista enlazada y cualquier tipo de datos. El compilador generará automáticamente el tipo adecuado de objeto en función del tipo que se especifique cuando se crea el objeto.

La forma general de la declaración de una clase genérica se muestra aquí:

```
template <class Ttype> class class-name {
  .
  .
  .
}
```

Ttype es el nombre del tipo resguardo que se especificará cuando se instancie la clase. Si es necesario, puede definirse más de un tipo de datos genéricos utilizando una lista separada por comas.

Una vez que se ha creado una clase genérica se puede crear una instancia específica de esa clase, utilizando la siguiente forma general:

```
class-name <type> ob;
```

type es el nombre del tipo de datos sobre el que trabajará la clase.

Los miembros de una clase genérica son, por sí mismos, genéricos. No es necesario utilizar **template** para especificarlos como tales.

EJEMPLOS

1. Este programa crea una sencilla clase genérica de lista enlazada. A continuación muestra la clase creando una lista enlazada que almacena caracteres:

```
// Una sencilla lista enlazada genérica.
#include <iostream.h>

template <class data_t> class list {
  data_t data;
  list *next;
  public:
    list(data_t d);
    void add(list *node) {node->next = this; next = 0; }
    list *getnext() { return next; }
    data_t getdata() { return data; }
};

template <class data_t> list<data_t>::list(data_t d)
{
  data = d;
  next = 0;
}

main()
{
  list<char> start('a');
  list<char> *p, *last;
  int i;
```

```
  // construcción de una lista
  last = &start;
  for(i=1; i<26; i++) {
    p = new list<char> ('a' + i);
    p->add(last);
    last = p;
  }
  // sigue la lista
  p = &start;
  while(p) {
    cout << p->getdata();
    p = p->getnext();
  }

  return 0;
}
```

Como puede observarse, la declaración de una clase genérica es similar a la de una función genérica. El tipo real de datos almacenado por la lista, en la declaración de la clase, es genérico. Cuando se declara un objeto de la lista se determina el tipo real de los datos. En este ejemplo, los objetos y punteros se declaran dentro de **main( )**, que especifica que el tipo de datos de la lista será **char**. Preste especial atención a esta declaración:

```
list<char> start('a');
```

Observe que el tipo de datos deseado se pasa dentro de los paréntesis angulares.

Usted debería escribir y ejecutar este programa. Contiene una lista enlazada que almacena los caracteres del alfabeto y después presenta dicha lista. Sin embargo, cambiando simplemente el tipo de datos especificado cuando se crean los objetos **list**, se modifica el tipo de datos almacenados en la lista. Por ejemplo, podría crear otro objeto que almacene enteros usando simplemente esta declaración:

```
list<int> int_start(1);
```

También puede utilizar **list** para almacenar sus propios tipos de datos. Por ejemplo, si desea almacenar información sobre direcciones, utilice esta estructura:

```
struct addr {
  char name[40];
  char street[40];
  char city[30];
  char state[3];
  char zip[12];
}
```

Para utilizar **list** después con el fin de generar objetos que almacenen objetos de tipo **addr**, emplee la siguiente declaración (asuma que **structvar** contiene una estructura **addr**):

```
list<addr> obj(structvar);
```

2. Este es otro ejemplo de una clase genérica. Es una versión modificada de la primera clase **stack** introducida en el Capítulo 1. Sin embargo, en este caso, **stack** se ha construido como una clase genérica. Además, puede almacenar cualquier tipo de objetos. En este ejemplo, se crean una pila de caracteres y una pila de números en coma flotante:

```
// Esta función ilustra una pila genérica.
#include <iostream.h>

#define SIZE 10

// Creación de una clase pila genérica
template <class StackType> class stack {
  StackType stck[SIZE]; // contiene la pila
  int tos; // índice de la cabecera de la pila

public:
  void init() { tos = 0; } // inicialización de la pila
  void push(StackType ch); // introducción de objetos en la pila
  StackType pop(); // extracción de objetos de la pila
};

// Colocación de un objeto.
template <class StackType> void stack<StackType>::push(StackType ob)
{
 if(tos==SIZE) {
  cout << "La pila está llena";
  return;
  }
  stck[tos] = ob;
  tos++;
}

// Extracción de un objeto.
template <class StackType> StackType stack<StackType>::pop()
{
  if(tos==0) {
    cout << "La pila está vacía";
    return 0; // si la pila está vacía devuelve cero
  }
  tos--;
  return stck[tos];
}

main()
{
  // Demostración de pilas de caracteres.
  stack<char> s1, s2; // creación de dos pilas
  int i;

  // inicialización de las pilas
  s1.init();
  s2.init();
```

```
  s1.push('a');
  s2.push('x');
  s1.push('b');
  s2.push('y');
  s1.push('c');
  s2.push('z');

 for(i=0; i<3; i++) cout << "Coloca s1: " << s1.pop() << "\n";
 for(i=0; i<3; i++) cout << "Coloca s2: " << s2.pop() << "\n";
  // demostración de pilas dobles
  stack<double> ds1, ds2; // creación de dos pilas

  // inicialización de las pilas
  ds1.init();
  ds2.init();

  ds1.push(1.1);
  ds2.push(2.2);
  ds1.push(3.3);
  ds2.push(4.4);
  ds1.push(5.5);
  ds2.push(6.6);

 for(i=0; i<3; i++) cout << "Extrae ds1: " << ds1.pop() << "\n";
 for(i=0; i<3; i++) cout << "Extrae ds2: " << ds2.pop() << "\n";


  return 0;
}
```

Como se manifiesta en la clase **stack** (y la anterior clase **list**), las funciones y las clases genéricas proporcionan una potente herramienta, que puede usarse para aprovechar al máximo el tiempo de programación, porque le permite definir la forma general de un algoritmo utilizable con cualquier tipo de datos. Queda usted liberado del tedio de reprogramar un algoritmo para cada tipo de datos.

3. Una clase plantilla puede tener más de un tipo genérico de datos. Simplemente hay que declarar todos los tipos de datos requeridos por la clase en una lista separada por comas dentro de la especificación **template**. El siguiente pequeño ejemplo crea una clase que utiliza dos tipos de datos genéricos:

```
/* Este ejemplo usa dos tipos de datos genéricos en la
   definición de una clase.
*/
#include <iostream.h>

template <class Type1, class Type2> class myclass
{
  Type1 i;
  Type2 j;
  public:
  myclass(Type1 a, Type2 b) { i = a; j = b; }
  void show() { cout << i << ' ' << j << '\n'; }
};
```

```
main()
{
  myclass<int, double> ob1(10, 0.23);
  myclass<char, char *> ob2('X', "Esto es una prueba");

  ob1.show(); // muestra int, double
  ob2.show(); // muestra char, char *

  return 0;
}
```

Este programa produce la siguiente salida:

```
10 0.23
X Esto es una prueba
```

El programa declara dos tipos de objetos. **ob1** utiliza datos enteros y **dobles**. **ob2** utiliza un carácter y un puntero a carácter. Para ambos casos, el compilador genera automáticamente los datos y las funciones apropiadas para adaptarse a la creación de los objetos.

EJERCICIOS

1. Si aún no lo ha hecho, compile y ejecute los dos ejemplos sobre clases genéricas. Intente declarar listas y/o pilas de diferentes tipos de datos.
2. Cree y compruebe si funciona una clase genérica cola.
3. Cree una clase genérica, llamada **input**, que cuando llame a su constructor, haga lo siguiente:

   - le pide al usuario una entrada
   - almacena los datos introducidos por el usuario y
   - vuelve a pedir los datos si no están dentro del rango predeterminado.

   Los objetos de tipo **input** se deberían declarar así:

   input ob(*«mensaje indicativo», min-value, max-value*)

   En este caso, el *mensaje indicativo* es el mensaje que solicita la entrada. Los valores mínimo y máximo aceptados se especifican mediante *min-value* y *max-value*, respectivamente. (Nota: el tipo de datos introducidos por el usuario será igual al de *min-value* y *max-value*.)

11.3. MANEJO DE EXCEPCIONES

C++ posee un mecanismo de gestión de errores incorporado que se denomina *manejo de excepciones*. La utilización del manejo de excepciones permite gestionar y responder a los errores en tiempo de ejecución. El manejo de excepciones

de C++ está construido a partir de tres palabras claves: **try**, **catch** y **throw**. En términos generales, las sentencias de un programa que se quiere monitorizar, para vigilar las excepciones, están contenidas en un bloque **try**. Si se produce una excepción (un error) en un bloque **try**, éste se eleva (utilizando **throw**). Usando **catch** se captura la excepción y se procesa. A continuación se profundiza en esta descripción general.

Como ya se ha indicado, cualquier sentencia que provoque una excepción debe haber sido ejecutada desde un bloque **try**. (Las funciones llamadas desde un bloque **try** pueden ocasionar también una excepción.) Cualquier excepción debe ser capturada por una sentencia **catch** que sigue, inmediatamente, a la sentencia **try**, causante de la excepción. La forma general de **try** y **catch** es:

```
try {
// bloque try
}
catch (type1 arg) {
// bloque catch
}
catch (type2 arg) {
// bloque catch
}
catch (type3 arg) {
// bloque catch
}
.
.
.
catch (typeN arg) {
   // bloque catch
}
```

El bloque **try** debe contener la porción de programa que se quiere monitorizar. Esto puede ser tan corto como unas pocas sentencias dentro de una función o tan grande como el código de la función **main( )** (que da lugar a que se monitorice todo el programa).

Cuando se genera una excepción, la sentencia **catch** correspondiente la captura y la procesa. Puede haber más de una sentencia **catch** asociada a un bloque **try**. La sentencia catch utilizada depende del tipo de excepción. Esto es, si el tipo de datos especificado por **catch** coincide con el de la excepción, se ejecuta esa sentencia **catch**. (Las otras sentencias no se consideran.)

Cuando se captura una excepción *arg* recibe un valor. Puede capturarse cualquier tipo de datos, incluidas las clases creadas por el programador.

La forma general de la sentencia **throw** es la siguiente:

```
throw exception;
```

throw debe ejecutarse dentro del bloque **try**, que es lo más adecuado, o desde cualquier función que la llame (directa o indirectamente). *exception* es el valor generado.

Nota *Si se produce una excepción para la que no existe una sentencia* **catch** *aplicable, el programa puede terminar de forma anormal. Si su compilador admite el estándar ANSI C++ propuesto, la generación de una excepción no gestionada provoca una llamada a la función* **terminate( )**. *Por omisión,* **terminate( )** *llama a* **abort( )** *para que su programa se detenga, pero, si lo desea, puede especificar su propio gestor de finalización. Es necesario consultar la biblioteca de referencias de su compilador para conocer más detalles.*

EJEMPLOS

1. Este sencillo ejemplo muestra la forma de funcionamiento de el manejo de excepciones en C++:

```
// Un ejemplo sencillo de manejo de excepciones.
#include <iostream.h>

main()
{
  cout << "inicio\n";

  try { // inicio de un bloque try
    cout << "Dentro del bloque try\n";
    throw 10; // generación de un error
    cout << "Esto no se ejecutará correctamente";
  }
  catch (int i) { // captura de un error
    cout << "¡Capturado un error! Su número es: ";
    cout << i << "\n";
  }

  cout << "fin";

  return 0;
}
```

Este programa da lugar a la siguiente salida:

```
inicio
Dentro del bloque try
¡Capturado un error! Su número es: 10
fin
```

Observe este programa cuidadosamente. En él hay un bloque **try** que contiene tres sentencias y una sentencia **catch(int i)** que procesan una excepción entera. Dentro del bloque **try**, sólo se ejecutarán dos de estas sentencias: la primera sentencia **cout** y la sentencia **throw**. Una vez lanzada una excepción, el control pasa a la expresión **catch** y el bloque **try** termina. Esto es, *no* se llama a **catch**. En lugar de ello, se le transfiere la ejecución del programa. (La pila se inicializa de nuevo para llevar esto a cabo.) De este modo, la sentencia **cout**, que sigue **throw**, no se ejecutará.

Cuando se ejecuta la sentencia **catch** el programa continúa con las sentencias que siguen a **catch**. Sin embargo, un bloque **catch** finalizará normalmente con una llamada a **exit( )**, **abort( )**, etc., porque el manejo de excepciones se utiliza a menudo para gestionar errores fatales.

2. Como ya se ha indicado, el tipo de una excepción debe coincidir con el tipo especificado en una sentencia **catch**. En el ejemplo anterior, si se cambia a **double** el tipo de la sentencia **catch**, no podrá capturarse la excepción y el programa terminará de un modo anormal. Este programa refleja este cambio:

```
// Este ejemplo no va a funcionar.
#include <iostream.h>

main()
{
  cout << "inicio\n";

  try { // inicio de un bloque try
    cout << "Dentro del bloque try\n";
    throw 10; // generación de un error
    cout << "Esto no se ejecutará correctamente";
  }
  catch (double i) { // No funcionará para una excepción
  entera
    cout << "¡Capturado uno! Su número es: ";
    cout << i << "\n";
  }

  cout << "end";

  return 0;
}
```

Este programa da lugar a la siguiente salida, ya que la sentencia **double catch** no capturará la excepción entera:

```
inicio
Dentro del bloque try
Detención anormal del programa
```

3. Se puede generar una excepción desde una sentencia que no está dentro del bloque **try** siempre que esté incluida en una función que esté, a su vez, dentro del bloque **try**. Este programa, por ejemplo, es correcto:

```
/* Generación de una excepción desde una función exterior
   al bloque try.
*/
#include <iostream.h>

void Xtest(int test)
{
  cout << "Dentro de Xtest, test vale: " << test << "\n";
  if(test) throw test;
}
```

```
main()
{
  cout << "inicio\n";

  try { // inicio de un bloque try
    cout << "Dentro del bloque try\n";
    Xtest(0);
    Xtest(1);
    Xtest(2);
  }
  catch (int i) { // captura de un error
    cout << "¡Capturado uno! Su número es: ";
    cout << i << "\n";
  }

  cout << "fin";

  return 0;
}
```

Este programa muestra la siguiente salida:

```
inicio
Dentro del bloque try
Dentro de Xtest, test vale: 0
Dentro de Xtest, test vale: 1
¡Capturado uno! Su número es: 1
fin
```

4. Un bloque **try** puede encontrarse dentro de una función. Cuando se da este caso, cada vez que se introduce la función, se redefine el manejo de excepciones con respecto a esa función.

```
#include <iostream.h>

// Un bloque try/catch puede incluirse en una función que
   no sea main().
void Xhandler(int test)
{
  try{
    if(test) throw test;
  }
  catch(int i) {
    cout << "¡Capturado uno! Ex.#: " << i << '\n';
  }
}

main()
{
  cout << "inicio\n";

  Xhandler(1);
  Xhandler(2);
```

```
  Xhandler(0);
  Xhandler(3);

  cout << "fin";

  return 0;
}
```

Este programa presenta la salida:

```
inicio
¡Capturado uno! Ex.#: 1
¡Capturado uno! Ex.#: 2
¡Capturado uno! Ex.#: 3
fin
```

Como puede verse, se han producido tres excepciones. Después de cada excepción, la función finaliza. Cuando se llama de nuevo a la función, se redefine el manejo de excepciones.

5. Como ya se comentó anteriormente, puede haber más de una sentencia **catch** asociada con un bloque **try**. De hecho, lo normal es que sea así. Sin embargo, cada sentencia **catch** debe capturar un tipo diferente de excepción. Por ejemplo, el siguiente programa captura enteros y cadenas:

```
#include <iostream.h>

// Pueden capturarse diferentes tipos de excepciones.
void Xhandler(int test)
{
  try{
    if(test) throw test;
    else throw "El valor es cero";
  }
  catch(int i) {
    cout << "¡Capturado uno! Ex.#: " << i << '\n';
  }
  catch(char *str) {
    cout << "Capturada una cadena: ";
    cout << str << '\n';
  }
}

main()
{
  cout << "inicio\n";

  Xhandler(1);
  Xhandler(2);
  Xhandler(0);
  Xhandler(3);

  cout << "fin";

  return 0;
}
```

Este programa produce la siguiente salida:

```
inicio
¡Capturado uno! Ex.#: 1
¡Capturado uno! Ex.#: 2
Capturada una cadena: El valor es cero
¡Capturado uno! Ex.#: 3
fin
```

Como se observa, cada sentencia **catch** responde sólo a su propio tipo.

En general, las expresiones **catch** se comprueban de acuerdo a su orden de aparición dentro del programa. Sólo se ejecuta la sentencia que coincida. El resto de bloques **catch** se ignoran.

EJERCICIOS

1. La mejor forma para entender cómo funciona el manejo de excepciones en C++ es practicando con él. Introduzca, compile y ejecute los programas anteriores. A continuación, experimente con ellos cambiando partes de los mismos y analizando los resultados.

2. ¿Qué hay incorrecto en este fragmento?

```
main()
{
  throw 12.23;
```

3. ¿Qué hay incorrecto en este fragmento?

```
try {
  // ...
  throw 'a';
  // ...
}
catch(char *) {
  // ...
}
```

4. ¿Qué ocurriría si se genera una excepción, para la que no existe la correspondiente sentencia **catch**?

11.4. MAS SOBRE MANEJO DE EXCEPCIONES

Existen diversos detalles y matices del manejo de excepciones en C++ que lo hacen fácil de utilizar.

En algunas circunstancias se necesitará un gestor de excepciones que capture todas ellas en lugar de capturar una de un determinado tipo. Esto es sencillo de llevar a cabo. Basta con utilizar simplemente esta forma de **catch**:

```
catch(...) {
   // procesamiento de todas las excepciones
}
```

En este caso, los puntos suspensivos coinciden con cualquier tipo de datos.

Cuando se llama a una función desde un bloque **try**, el tipo de excepciones que la función puede producir puede limitarse. De hecho, también puede impedirse que la función genere cualquier tipo de excepción. Para cumplir estas restricciones debe añadirse una cláusula **throw** a la definición de la función. La forma general es ésta:

```
ret-type func-name(arg-list) throw(type-list)
{
   // ...
}
```

La función sólo puede generar los tipos de datos contenidos en la lista *type-list*. La producción de cualquier otro tipo de excepción dará lugar a una terminación anormal del programa. Si no desea una función que sea capaz de generar *todo tipo* de excepciones, utilice una lista vacía.

Nota *Si su compilador cuenta con el estándar ANSI C++ propuesto, el intento de generar una excepción que no esté admitida por una función provocará una llamada a la función* **unexpected( )**. *Por omisión, esto da lugar a una llamada a* **abort( )**, *que ocasiona una terminación anormal del programa. Sin embargo, si lo desea,puede especificar su propio gestor de terminación. Será necesario consultar la biblioteca de referencias de su compilador para mayores detalles.*

Si se desea relanzar una excepción desde el interior de un gestor de excepciones, basta simplemente con llamar a **throw** sin ninguna excepción. Esto da lugar a que la excepción actual sea pasada a una secuencia **try/catch** más externa.

EJEMPLOS

1. El siguiente programa presenta **catch(...)**:

```
// Este ejemplo captura todas las excepciones.
#include <iostream.h>

void Xhandler(int test)
{
  try{
    if(test==0) throw test; // genera un entero
    if(test==1) throw 'a'; // genera un carácter
    if(test==2) throw 123.23; // genera un doble
  }
  catch(...) { // captura todas las excepciones
    cout << "¡Capturada una!\n";
  }
}
```

```
main()
{
  cout << "inicio\n";

  Xhandler(0);
  Xhandler(1);
  Xhandler(2);

  cout << "fin";
  return 0;
}
```

Este programa muestra la siguiente salida:

```
inicio
¡Capturada una!
¡Capturada una!
¡Capturada una!
fin
```

Los tres **throw** han sido capturados utilizando una única sentencia **catch**.

2. Un buen uso para **catch(...)** es el de última sentencia de una agrupación de capturas. Se trata de una útil sentencia implícita o una sentencia de «captura de todo». Por ejemplo, esta versión, ligeramente diferente del programa anterior, captura explícitamente excepciones enteras dejando que **catch(...)** capture todas las demás:

```
// Este ejemplo usa catch(..) implícitamente.
#include <iostream.h>

void Xhandler(int test)
{
  try{
    if(test==0) throw test; // se produce un entero
    if(test==1) throw 'a'; // se produce un carácter
    if(test==2) throw 123.23; // se produce un doble
  }
  catch(int i) { // captura de una excepción entera
    cout << "Captura " << i << '\n';
  }
  catch(...) { // captura del resto de las excepciones
    cout << "¡Capturada una!\n";
  }
}

main()
{
  cout << "inicio\n";

  Xhandler(0);
  Xhandler(1);
  Xhandler(2);

  cout << "fin";

  return 0;
}
```

La salida producida por este programa es:

```
inicio
Capturada 0
¡Capturada una!
¡Capturada una!
fin
```

Como sugiere el ejemplo, el uso de **catch(...)** por omisión es una buena forma de capturar todas las excepciones que no desee gestionar explícitamente. La captura de todas las excepciones impide que una excepción incontrolada provoque una terminación anormal de un programa.

3. El siguiente programa muestra cómo restringir el tipo de excepciones que puede producir una función:

```
// Restricción de los tipos producidos por una función.
#include <iostream.h>

// Esta función sólo genera enteros, caracteres y dobles.
void Xhandler(int test) throw(int, char, double)
{
  if(test==0) throw test; // se produce un entero
  if(test==1) throw 'a'; // se produce un carácter
  if(test==2) throw 123.23; // se produce un doble
}

main()
{
  cout << "inicio\n";
  try{
    Xhandler(0); // se intenta pasar 1 y 2 a Xhandler()
  }
  catch(int i) {
    cout << "Capturado el entero\n";
  }
  catch(char c) {
    cout << "Capturado el carácter\n";
  }
  catch(double d) {
    cout << "Capturado el doble\n";
  }

  cout << "fin";

  return 0;
}
```

En este programa, la función **Xhandler( )** sólo puede generar excepciones enteras, caracteres y dobles. Si intenta generar otro tipo de excepción, el programa terminará de modo anormal. (Esto es, se llamará a **unexpected( )**.) Para ver un ejemplo de esto, extraiga **int** de la lista de excepciones y ejecute el programa de nuevo.

Es importante entender que sólo se puede restringir a una función en el tipo de las excepciones que devuelve al bloque **try** que la llamó. Esto es, un bloque **try** *dentro* de una función puede generar cualquier tipo de excepción mientras ésta sea capturada *dentro* de la función. La restricción es aplicable sólo cuando la excepción se genera fuera de la función.

4. El siguiente cambio en **Xhandler( )** impide que genere cualquier tipo de excepción:

```
// ¡Esta función NO puede generar excepciones!
void Xhandler(int test) throw()
{
  /* Las siguientes sentencias no funcionan. En su lugar,
     provocarán una terminación anormal del programa. */
  if(test==0) throw test;
  if(test==1) throw 'a';
  if(test==2) throw 123.23;
}
```

5. Como ya se ha dicho, se puede relanzar una excepción. La razón más probable para hacerlo es la de permitir que accedan a la excepción múltiples gestores. Quizá un gestor de excepciones, por ejemplo, controle un aspecto de la excepción y un segundo gestor puede hacerse cargo del resto. Una excepción sólo puede ser relanzada desde un bloque **catch** (o desde cualquier función llamada dentro del bloque). Cuando se genere de nuevo una excepción, ésta no será capturada por la misma sentencia **catch**. Se propagará a una sentencia **catch** más externa. El siguiente programa muestra el relanzamiento de una excepción. La excepción es del tipo **char ***:

```
// Ejemplo de "relanzamiento" de una excepción.
#include <iostream.h>

void Xhandler()
{
  try {
    throw "hola"; // se produce un char *
  }
  catch(char *) { // se captura un char *
    cout << "Se captura char * dentro de Xhandler\n";
    throw ; // se relanza char * desde fuera de la función
  }
}

main()
{
  cout << "inicio\n";

  try{
    Xhandler();
  }
  catch(char *) {
    cout << "Se captura char * dentro de main\n";
  }

  cout << "fin";

  return 0;
}
```

Este programa produce la siguiente salida:

```
inicio
Se captura char * dentro de Xhandler
Se captura char * dentro de main
fin
```

EJERCICIOS

1. Antes de proseguir, compile y ejecute todos los ejemplos de esta sección. Asegúrese de entender la salida de cada programa.
2. ¿Qué hay incorrecto en este fragmento de código?

```
try {
  // ...
  throw 10;
}
catch(int *p) {
  // ...
}
```

3. Muestre un modo de corregir el fragmento anterior.
4. ¿Qué expresión de **catch** captura todo tipo de excepciones?
5. Esta es una plantilla de una función llamada **divide( )**:

```
double divide(double a, double b)
{
  // añada la gestión de errores
  return a/b;
}
```

Esta función devuelve el resultado de dividir **a** y **b**. Añada a esta función un mecanismo de comprobación de errores utilizando el manejo de excepciones de C++. En concreto, impida el error resultante de dividir por cero. Compruebe que su solución funciona construyendo un programa.

COMPROBACION DE APTITUD SUPERIOR

Al llegar a este punto, debería ser capaz de realizar estos ejercicios y de responder a estas preguntas.

1. Cree una función genérica que devuelva el modo de un array de valores. (El *modo* de un conjunto es el valor que más se repite.)
2. Cree una función genérica que devuelva la suma de un array de valores.
3. Construya un algoritmo de ordenación genérico por el método de la burbuja (o haga lo mismo con cualquier otro algoritmo de ordenación).
4. Construya de nuevo la clase **stack** para que almacene pares de diferentes tipos de objetos en la pila. Compruebe la solución.

5. Muestre las formas generales de **try**, **catch** y **throw**. Describa su funcionamiento.
6. Rehaga, de nuevo, la clase **stack**, de modo que los hechos de sobrepasar o de no alcanzar los límites de la pila sean tratados como excepciones.
7. Compruebe el manual de usuario de su compilador. Vea si admite las funciones **unexpected( )** y **terminate( )**. Generalmente, estas funciones pueden configurarse para llamar a cualquier función. Si esto es lo que ocurre con su compilador, intente crear su propio conjunto de funciones de terminación para gestionar cualquier excepción no controlada.

COMPROBACION DE APTITUD INTEGRADA

Esta sección comprueba cómo ha asimilado el contenido de este capítulo con el de los anteriores.

1. En el Capítulo 4 se mostró una clase de array limitado. Conviértalo en un array limitado genérico.
2. En el Capítulo 1 se crearon las versiones sobrecargadas de la función **abs( )**. Para obtener una solución mejor, cree una función genérica **abs( )** que devuelva el valor absoluto de cualquier objeto numérico.

12 Temas diversos

OBJETIVOS DEL CAPITULO

¡FELICIDADES! Ha recorrido un largo camino desde el Capítulo 1. Cuando finalice este capítulo usted ya será un programador de C++. Este capítulo trata un grupo de temas y de características especiales. También discute algunas diferencias importantes entre C y C++.

Este libro proporciona una introducción completa de C++, pero no da alcance a todos sus matices y detalles. Si desea aprender más sobre C++, incluyendo las bibliotecas de clases normalmente incorporadas, más sobre la E/S en C++, algunas técnicas de propósito especial y un mayor número de ejemplos, puede leer mi libro C++: *La referencia completa* (Berkeley: Osborne/McGraw-Hill, 1991).

COMPROBACION DE APTITUD

Antes de continuar, debería ser capaz de responder a estas preguntas y de realizar estos ejercicios.

1. ¿Qué es una función genérica y cuál es su forma general?
2. ¿Qué es una clase genérica y cuál es su forma general?
3. Construya una función genérica llamada **gexp( )** que devuelva el valor de uno de sus argumentos elevado a la potencia del otro.
4. En el Capítulo 9, Sección 7, Ejemplo 1, se creó una clase **coord** que almacenaba coordenadas enteras y se demostró que funcionaba. Construya una versión genérica de la clase **coord** de modo que pueda almacenar coordenadas de cualquier tipo. Cree un programa que pruebe que la solución es correcta.
5. Explique brevemente cómo funcionan en conjunto **try**, **catch** y **throw** para llevar a cabo la gestión de excepciones en C++.
6. ¿Puede utilizarse **throw** si la ejecución no ha pasado a través de un bloque **try**?
7. ¿Cuál es la finalidad de **terminate( )** y **unexpected( )**?
8. ¿Qué forma de **catch** gestiona todo tipo de excepciones?

12.1. ATRIBUTOS ESTATICOS

Es posible declarar una variable atributo de una clase como **static**. (También es posible declarar un método como **static**, pero no se utiliza normalmente y no se examina en este libro.) El empleo de atributos **static** impide que se produzcan determinados problemas.

En concreto, cuando se declara una variable atributo como **static** sólo se genera una copia de esa variable —independientemente del número de objetos de esa clase que se creen. Cada objeto, simplemente, comparte esa variable. Recuerde que para atributos normales, cada vez que se crea un objeto se crean copias nuevas de esa variable que sólo son accesibles a ese objeto. (Esto es, para variables normales, cada objeto posee sus propias copias.) Sin embargo, sólo hay una copia de la variable **static** y todos los objetos de su clase la comparten. La misma variable estática también puede ser utilizada por cualquier clase derivada de la clase que contiene el atributo **static**.

Aunque en un principio puede resultar extraño no es así, ya que un atributo **static** existe *antes* de que se cree un objeto de su clase. Un atributo **static** es una variable global con validez restringida a la clase en la que se declara. De hecho, como se verá en los próximos ejemplos, en realidad es posible acceder a una variable **static** independientemente de cualquier objeto.

Cuando se declara dentro de una clase un atributo **static**, éste no queda definido. En lugar de ello, debe definirse el mismo fuera de la clase. Para llevarlo a cabo, se declara dos veces la variable **static** utilizando el operador de alcance de resolución, que permite identificar la clase a la que pertenece.

Todos los atributos **static** se inicializan, por omisión, a cero. Sin embargo, es posible fijar un valor inicial en un atributo **static**.

No debe olvidarse que la principal razón por la que se admiten en C++ las variables **static** es para impedir el uso de variables globales. Como puede suponerse, las clases que contienen variables globales casi siempre violan el principio de encapsulamiento, que es fundamental para POO y para C++.

EJEMPLOS

1. A continuación se muestra un ejemplo que utiliza un atributo **static**:

```
// Ejemplo de atributo estático.
#include <iostream.h>
class myclass {
  static int i;
public:
  void seti(int n) { i = n; }
  int geti() { return i; }
};

// Definición de myclass::i. i aún es privada para myclass.
int myclass::i;

main()
{
  myclass o1, o2;

  o1.seti(10);

  cout << "o1.i: " << o1.geti() << '\n'; // muestra 10
  cout << "o2.i: " << o2.geti() << '\n'; // también muestra 10

  return 0;
}
```

Este programa obtiene la siguiente salida:

```
o1.i: 10
o2.i: 10
```

En este programa sólo el objeto **o1** fija en realidad el valor del atributo **static i**. Sin embargo, ya que **i** es compartida por **o1** y **o2** (y, en particular, por cualquier objeto de tipo **myclass**), las dos llamadas a **geti( )** muestran el mismo resultado.

Observe cómo se declara **i** dentro de **myclass,** pero se define fuera de ella. Este segundo paso asegura la disposición de espacio para **i**. Desde un punto de vista técnico, la declaración de una clase es sólo una declaración. No se reserva memoria alguna por ello. Ya que un atributo **static** *implica* que se asigna memoria a ese atributo, se necesita una definición aparte para realizar la asignación de la misma.

Nota *En algunos desarrollos de C++ si, simplemente, se permite que el valor inicial de la variable* **static** *sea implícitamente cero, no existe la necesidad de definirla fuera de la clase. Sin embargo, de acuerdo al estándar ANSI C++ propuesto, se produce un anacronismo. Por esta razón, es probable que se deseen incluir definiciones separadas para todos los atributos estáticos.*

2. Debido a que un atributo **static** existe antes de que se cree un objeto de la clase, se puede acceder al atributo dentro de un programa con independencia de los objetos. Por ejemplo, la siguiente variación del programa anterior fija el valor de **i** a 100, sin referencia alguna a un objeto específico. Observe el uso del operador de resolución de alcance para acceder a **i**.

```
// Referencia a una variable static fuera de un objeto.
#include <iostream.h>

class myclass {
public:
  static int i;
  void seti(int n) { i = n; }
  int geti() { return i; }
};

int myclass::i;

main()
  {
  myclass o1, o2;

  // fija i directamente
  myclass::i = 100; // no se referencia ningún objeto.

  cout << "o1.i: " << o1.geti() << '\n'; // muestra 100
  cout << "o2.i: " << o2.geti() << '\n'; // también muestra 100

  return 0;
}
```

Ya que **i** se fija a 100, la salida obtenida es:

```
o1.i: 100
o2.i: 100
```

3. Un uso frecuente de una variable de una clase como **static** es el de coordinar el acceso a un recurso compartido, como un archivo del disco, una impresora o un servidor de red. Como probablemente habrá conocido en su experiencia como programador, la coordinación del acceso a un recurso compartido requiere de algún medio de secuenciamiento de sucesos. Para tener una idea de cómo los atributos **static** pueden ser empleados para controlar el acceso a un recurso compartido, examinemos el siguiente programa. Crea una clase llamada **output**, que contiene un búfer normal de salida llamado **outbuf**, que es un array de caracteres **static**. Este búfer se utiliza para recibir la salida enviada por el método **outbuf( )**. Esta función envía el contenido de la cadena **str**, carácter a carácter. Esto lo realiza adquiriendo, primero, acceso al búfer y, a continuación, enviando todos los caracteres de **str**. Impide el acceso al búfer mientras no se complete la salida. Debería ser capaz de seguir su operativa estudiando el código y leyendo los comentarios.

```
// Ejemplo sobre un recurso compartido.
#include <iostream.h>
#include <string.h>

class output {
  static char outbuf[255]; // este es el recurso compartido
  static int inuse; // si vale 0, se puede acceder al búfer;
                    // ocupado, en cualquier otro caso
  static int oindex; // índice de outbuf
  char str[80];
  int i; // índice del próximo carácter de str
  int who; // identifica el objeto, debe ser > 0
public:
  output(int w, char *s) { strcpy(str, s); i = 0; who = w; }

  /* Esta función devuelve -1 si el búfer está ocupado,
    devuelve 0 si se ha completado la salida, y
    devuelve who si todavía está usando el búfer.
  */
  int putbuf()
  {
    if(!str[i]) { // realizada la salida
      inuse = 0; // se elimina el búfer
      return 0; // señal de terminación
    }
    if(!inuse) inuse = who; // acceso al búfer
    if(inuse != who) return -1; // utilizado por otra
    if(str[i]) { // todavía hay caracteres que extraer
      outbuf[oindex] = str[i];
      i++; oindex++;
      outbuf[oindex] = '\0'; // mantiene siempre la
      terminación nula
      return 1;
    }
  }
  void show() { cout << outbuf << '\n'; }
};

char output::outbuf[255]; // éste es el recurso compartido
```

```
int output::inuse = 0; // si vale 0, se puede acceder
//al búfer;
                       // ocupado, en cualquier otro caso
int output::oindex = 0; // índice de outbuf

main()
{
  output o1(1, "Esta es una prueba"), o2(2, " sobre
  atributos estáticos");

  while(o1.putbuf() | o2.putbuf()) ; // salida de caracteres

  o1.show();

  return 0;
}
```

EJERCICIOS

1. Rehaga el Ejemplo 3 de modo que muestre el objeto que extrae los caracteres y los objetos bloqueados porque el búfer está siendo utilizado por otro.
2. Un uso interesante de un atributo **static** es el de conocer el número de objetos de una clase existentes en un determinado instante de tiempo. La forma de hacerlo es incrementando el atributo **static**, cada vez que se llama al constructor de la clase, y decrementándolo, cada vez que se llama al destructor de la clase. Desarrolle este esquema y demuestre que funciona.

12.2. E/S BASADA EN ARRAYS

Además de la E/S en archivos y la E/S por consola, C++ admite un completo conjunto de funciones que usan arrays de caracteres como dispositivo de entrada y de salida. Aunque la E/S basada en arrays de C++ conceptualmente es similar a la E/S basada en arrays de C (en particular, las funciones **sscanf( )** y **sprintf( )** de C), la E/S basada en arrays de C++ es más flexible y útil, porque permite la integración de tipos definidos por el usuario. En algún tipo de literatura sobre C++ la E/S basada en arrays se denomina *E/S central* o *E/S basada en RAM*. Este libro utiliza el término «basada en arrays» porque es más descriptivo. Ya que no es posible abordar todos los aspectos de la E/S basada en arrays aquí, examinaremos las características más importantes y las más empleadas.

Hay que entender que la E/S basada en arrays trabaja con cadenas. Todo lo aprendido sobre E/S en C++ en los Capítulos 8 y 9 es aplicable a la E/S basada en arrays. De hecho, sólo aparecerán unas pocas nuevas funciones que aportan las ventajas de la E/S basada en arrays. Estas funciones enlazan una cadena con una región de memoria. Una vez que esto se ha realizado, toda la E/S se lleva a cabo mediante las funciones de E/S ya estudiadas.

Antes de utilizar la E/S basada en arrays hay que incluir en el programa el archivo cabecera **strstream.h**. En este archivo están definidas las clases **istrtream**,

ostrstream y **strstream**. Estas clases crean entrada, salida y entrada/salida basada en arrays, respectivamente. Estas clases tienen como base a **ios**, de manera que todas las funciones y manipuladores incluidos en **istream**, **ostream** y **iostream** también son accesibles en **istrstream**, **ostrstream** y **strstream**.

Para utilizar un array de caracteres de salida, debe usarse esta forma general del constructor **ostrstream**:

```
ostrstream ostr (char *buf, int size, int mode = ios::out);
```

ostr será la cadena asociada con el array *buf*. El tamaño del array se especifica mediante *size*. Generalmente, *mode* toma el valor implícito de salida, pero puede utilizarse cualquier indicador del modo de salida.

Una vez que se ha abierto un array de salida, se almacenarán en él los caracteres hasta que se complete. No se sobrepasarán sus límites. Cualquier intento de ello generará un error de E/S. Para averiguar el número de caracteres escritos en el array, puede emplearse la función **pcount( )**, mostrada aquí:

```
int pcount( );
```

Esta función debe llamarse junto con una cadena y devolverá el número de caracteres escritos en el array, incluyendo la terminación en nulo.

Para abrir un array de entrada, hay que utilizar esta forma general de constructor **istrstream**:

```
istrstream istr (const char *buf );
```

buf es un puntero al array de entrada. La cadena de entrada se llamará *istr*. Cuando se lea la entrada desde un array, **eof( )** devolverá verdadero al encontrar el final del array.

Para abrir un array para operaciones de entrada/salida, debe utilizarse la forma general del constructor **strstream**:

```
strstream iostr (char *buf, int size, int mode);
```

iostr será una cadena de entrada/salida que utilice el array apuntado por *buf*, que tiene *size* caracteres. Para operaciones de entrada/salida, *mode* debe tomar el valor **ios::in | ios::out**.

Es importante recordar que todas las funciones de E/S descritas anteriormente trabajan con E/S basada en arrays, incluidas las funciones de E/S binaria y las funciones de acceso aleatorio.

EJEMPLOS

1. Este es un breve ejemplo que muestra cómo abrir un array de salida y cómo escribir datos en él:

```
// Un breve ejemplo de salida basada en arrays .
#include <iostream.h>
#include <strstream.h>

main()
{
  char buf[255]; // búfer de salida

  ostrstream ostr(buf, sizeof buf); // apertura del array
  //de salida

  ostr << "La E/S basada en arrays utiliza cadenas como
  las de";
  ostr << "la E/S 'normal'\n" << 100;
  ostr << ' ' << 123.23 << '\n';

  // También pueden utilizarse manipuladores
  ostr << hex << 100 << ' ';
  // o indicadores de formato
  ostr << ostr.setf(ios::scientific) << 123.23 << '\n';
  ostr << ends; // se asegura que el búfer termina en nulo

  // presentación de la cadena resultante
  cout << buf;

  return 0;
}
```

Este programa muestra

```
La E/S basada en arrays utiliza cadenas como las de
la E/S 'normal'
100 123.23
64 01.2323e+02
```

Como puede verse, en la E/S basada en arrays pueden emplearse operadores de E/S sobrecargados, manipuladores de E/S incorporados, métodos e indicadores de formato. (Esto también se cumple para los manipuladores o los operadores de E/S sobrecargados que se crean en relación con sus propias clases.)

Este programa coloca de modo manual la terminación en nulo en el array de salida usando el manipulador **ends**. Tanto si la terminación en nulo se coloca automáticamente en el array como si no depende de la implementación, si la terminación en nulo es importante para su aplicación, es conveniente colocarla manualmente.

2. Este es un ejemplo de entrada basada en arrays:

```
// Un ejemplo que utiliza entrada basada en arrays.
#include <iostream.h>
#include <strstream.h>

main()
{
```

```
  char buf[] = "Hola 100 123.125 a";

  istrstream istr(buf); // apertura del array de entrada

  int i;
  char str[80];
  float f;
  char c;

  istr >> str >> i >> f >> c;

  cout << str << ' ' << i << ' ' << f;
  cout << ' ' << c << '\n';
  return 0;
}
```

Este programa lee y muestra los valores contenidos en el array de entrada **buf**.

3. Recuerde que un array de entrada, una vez que se ha enlazado a una cadena, se comportará como un archivo. Por ejemplo, este programa utiliza E/S binaria y la función **eof( )** para leer el contenido de **buf**:

```
/* Demostración de que eof() funciona con E/S basada
en arrays.
  La E/S basada en arrays también funciona con
  funciones de E/S binarias.
*/
#include <iostream.h>
#include <strstream.h>

main()
{
  char buf[] = "Hola 100 123.125 a";

  istrstream istr(buf);
  char c;

  while(!istr.eof()) {
    istr.get(c);
    cout << c;
  }

  return 0;
}
```

4. Este programa realiza entrada y salida en un array:

```
// Demostración de un array de entrada/salida.
#include <iostream.h>
#include <strstream.h>

main()
{
```

```
  char iobuf[255];

  strstream iostr(iobuf, sizeof iobuf, ios::in | ios::out);

  iostr << "Esto es una prueba\n";
  iostr << 100 << hex << 100 << ends;

  char str[80];
  int i;

  iostr.getline(str, 79); // lee la cadena hasta encontrar \n
  iostr >> i; // read 100

  cout << str << ' ' << i << ' ';

  iostr >> i;
  cout << i;
  return 0;
}
```

Este programa, primero, escribe la salida en **iobuf**. Después lee el búfer. Lee la línea completa «Esto es una prueba» utilizando la función **getline( )**. A continuación lee el valor decimal 100 y el valor hexadecimal 0x64.

EJERCICIOS

1. Modifique el Ejemplo 1 de modo que muestre el número de caracteres escritos en **buf** anteriores al carácter de terminación.
2. Programe una aplicación que utilice E/S basada en arrays para copiar el contenido de un array en otro. (Esta no es, por supuesto, la manera más eficiente de realizar esta tarea.)
3. Haciendo uso de E/S basada en arrays, escriba un programa que transforma una cadena que contiene un valor en coma flotante en su representación interna.

12.3. USO DE ESPECIFICACIONES DE ENLACE Y DE LA PALABRA CLAVE asm

C++ proporciona dos importantes mecanismos que facilitan el enlace de C++ con otros lenguajes. Uno es el *especificador de enlace*, que indica al compilador que una o más funciones de su programa C++ se enlazarán con otro lenguaje, que puede tener, entre otras diferencias, distintos convenios de paso de parámetros. El segundo es la palabra clave **asm**, que permite incluir instrucciones de un lenguaje ensamblador en el código fuente C++. Ambos mecanismos son analizados en este apartado.

Por omisión, todas las funciones de un programa C++ se compilan y enlazan como funciones C++. Sin embargo, se le puede indicar al compilador

C++ que enlace una función que sea compatible con otro tipo de lenguaje. Todos los compiladores de C++ permiten el enlace de funciones C o C++. También admite que se enlacen funciones de lenguajes como Pascal, Ada o FORTRAN. Para enlazar una función de un lenguaje diferente, hay que utilizar esta forma general en la especificación del enlace:

```
extern «language» function-prototype;
```

language es el nombre del lenguaje al que pertenece la función especificada para enlazar. Si se necesita enlazar más de una función, debe utilizarse esta forma en la especificación del enlace:

```
extern «language» {
   function-prototypes
}
```

Todas las especificaciones de enlace deben ser globales; no pueden utilizarse dentro de una función. También, aunque no es lo normal, puede especificarse el enlace de objetos.

El uso más normal de las especificaciones de enlace tiene lugar cuando se enlazan programas C++ con paquetes de rutinas de terceros, que han sido compilados utilizando otro lenguaje. Es totalmente posible, sin embargo, que nunca sea necesario emplear una especificación de enlace.

Aunque, generalmente, es posible enlazar rutinas de un lenguaje ensamblador con un programa C++, existe un modo más sencillo de hacerlo. C++ admite una palabra clave especial **asm**, que facilita la inclusión de instrucciones de un lenguaje ensamblador dentro de una función C++. Estas instrucciones se compilan normalmente. La ventaja de utilizar código ensamblador insertado es que el programa queda definido completamente como un programa C++ y no existe la necesidad de enlazarlo con archivos del lenguaje ensamblador. La forma general de la palabra clave **asm** es:

```
asm («op-code»);
```

donde *op-code* es la instrucción del lenguaje ensamblador que se incluye en el programa.

Es importante anotar que Turbo C++ y Borland C++ aceptan estas formas, ligeramente diferentes, de la sentencia **asm**:

```
asm op-code;

asm op-code newline

asm {
   instruction sequence
}
```

op-code, en este caso, no va entrecomillado. Puesto que la instrucción del lenguaje ensamblador depende de la implementación, es conveniente revisar el manual del usuario del compilador.

EJEMPLOS

1. Este programa enlaza **func( )** como una función de C en vez de C++:

```
// Ejemplo del especificador de enlace.
#include <iostream.h>

extern "C" int func(int x); // enlace como una función C

// Esta función se enlaza como una función C.
int func(int x)
{
  return x/3;
}
```

   Esta función puede enlazarse con el código compilado mediante un compilador C.

2. El siguiente fragmento de código le indica al compilador que **f1( )**, **f2( )** y **f3( )** deberían ser enlazadas como funciones C:

```
extern "C" {
  void f1();
  int f2(int x);
  double f3(double x, int *p);
}
```

3. Este fragmento incluye en **func( )** varias instrucciones de lenguaje ensamblador:

```
// ¡No pruebe esta función!
void func()
{
  asm (mov bp, sp);
  asm (push ax);
  asm (mov cl, 4);
  // ...
}
```

Recuerde *Es necesario ser un programador experimentado en lenguaje ensamblador para insertar el código correctamente. También hay que comprobar el manual del usuario del compilador para conocer los detalles acerca del uso del lenguaje ensamblador.*

EJERCICIOS

1. Estudie la sección del manual del usuario del compilador que se refiere a las especificaciones de enlace y a la interfaz con el lenguaje ensamblador.

12.4. CREACION DE UNA FUNCION DE CONVERSION

A veces es útil convertir un objeto de un tipo en otro tipo diferente. Mientras que es posible utilizar una función operadora sobrecargada para llevar a cabo tal conversión, existe un método más sencillo (y mejor), llamado función de conversión. Una *función de conversión* transforma un objeto en un valor compatible con otro tipo que, normalmente, es uno de los tipos incorporados en C++. En concreto, una función de conversión transforma automáticamente un objeto en un valor que es compatible con el tipo de la expresión en la que se utiliza el objeto.

La forma general de una función de conversión es:

operator *type*( ) { return *value*; }

type es el tipo resultante de la conversión y *value* es el valor del objeto después de que la conversión se ha llevado a cabo. La función de conversión devuelve un valor de tipo *type*. No puede especificarse parámetro alguno y la función de conversión debe ser un método de la clase para la que se efectúa la conversión.

Como se mostrará en los ejemplos, una función de conversión normalmente ofrece un enfoque más claro, para convertir el valor de un objeto en otro tipo, que cualquier otro método existente en C++, porque permite la inclusión directa del objeto en la expresión que contiene el tipo resultante.

EJEMPLOS

1. En el siguiente programa, la clase **coord** contiene una función de conversión a enteros. En este caso, la función devuelve el producto de dos valores coordenados; sin embargo, está permitida cualquier conversión que se adecúe a su aplicación:

```
// Un ejemplo sencillo de función de conversión.
#include <iostream.h>

class coord {
  int x, y;
public:
  coord(int i, int j) { x = i; y = j; }
  operator int() { return x*y; } // función de conversión
};

main()
{
  coord o1(2, 3), o2(4, 3);
  int i;

  i = o1; // se convierte automáticamente a entero
```

```
  cout << i << '\n';

  i = 100 + o2; // conversión de o2 a entero
  cout << i << '\n';

  return 0;
}
```

Este programa muestra 6 y 112.

En este ejemplo se observa que se llama a la función de conversión cuando **o1** se asigna a un entero y cuando **o2** se utiliza como parte de una expresión entera más extensa. Como ya se ha dicho, el uso de una función de conversión permite que las clases creadas puedan integrarse en expresiones C++ «normales» sin tener que crear una compleja serie de funciones operadoras sobrecargadas.

2. El próximo es otro ejemplo de función de conversión. En este caso, transforma una cadena de tipo **strtype** en un puntero a la cadena **str**.

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

class strtype {
  char str[80];
  int len;
public:
  strtype(char *s) { strcpy(str, s); len = strlen(s); }
  operator char *() { return str; } // conversión a char*
};

main()
{
  strtype s("Esto es una prueba\n");
  char *p, s2[80];

  p = s; // conversión a char *
  cout << "La cadena es ésta: " << p << '\n';

  // conversión a char * en la llamada a la función
  strcpy(s2, s);
  cout << "Esta es la copia de la cadena: " << s2 << '\n';
  return 0;
}
```

Este programa presenta la siguiente salida:

```
La cadena es ésta: Esto es una prueba
Esta es la copia de la cadena: Esto es una prueba
```

Como puede verse, no sólo se llama a la función de conversión cuando se asigna un objeto **s** a **p** (que es del tipo **char***), sino que también se usa como parámetro de **strcpy( )**. Recuerde que **strcpy( )** tiene el siguiente prototipo:

```
char *strcpy(char *s1, char *s2);
```

Puesto que el prototipo especifica que *s2* es del tipo **char***, se llama automáticamente a la función de conversión a **char***. Esto demuestra cómo una función de conversión puede ayudarle a integrar, sin añadiduras, sus clases en la biblioteca de funciones estándar de C++.

EJERCICIOS

1. Haciendo uso de la clase **strtype** del Ejemplo 2, cree una función de conversión a tipo entero. Esta función debe devolver la longitud de la cadena almacenada en **str**. Demuestre que funciona.
2. Dada esta clase:

```
class pwr {
  int base;
  int exp;
public:
  pwr(int b, int e) { base = b; exp = e; }
// creación de una función de conversión a enteros
};
```

cree una función de conversión que transforme un objeto de tipo **pwr** al tipo entero. a función debe devolver el resultado de $base^{exp}$.

12.5. *DIFERENCIAS ENTRE C Y C++*

Como es sabido, C++ se ha desarrollado a partir de C. Esto significa que, en general, cualquier programa en C puede convertirse automáticamente en un programa en C++ (no orientado a objeto). Sin embargo, ya que C++ admite programación orientada a objetos, y debido a las implicaciones de ello, existen algunas leves diferencias entre C y C++. Algunas de ellas impedirán que un programa en C sea compilado en un compilador de C++. En esta sección se detallan estas diferencias. (Algunas ya se han mencionado a lo largo de este libro, pero se incluyen de nuevo aquí por cuestiones de integridad.)

Nota *Recuerde que las características de C y de C++ que no están relacionadas con la POO difieren en pocos aspectos. En la mayoría de los casos, cualquier programa en C es automáticamente un programa correcto en C++.*

Una de las más importantes y sutiles diferencias entre C y C++ es el hecho de que en C la declaración de una función como ésta:

```
int f();
```

no indica nada sobre los parámetros de la función. Esto es, en C, cuando no se especifica nada entre los paréntesis, que siguen al nombre de la función, significa que no se establece nada sobre los parámetros de esa función. Podría tener parámetros o no tenerlos. Sin embargo, en C++, la declaración de una función

como ésta implica que la función no tiene parámetros. Esto es, en C++, estas dos declaraciones son equivalentes:

```
int f();
int f(void);
```

En C++, **void** es opcional. Muchos programadores de C++ incluyen **void** como un medio de dejar completamente claro, al que lea el programa, que la función no tiene parámetros. Pero, técnicamente, es innecesario.

En C++, todas las funciones deben tener un prototipo. Esto en C es opcional (aunque la buena práctica de la programación sugiere que en C se utilicen siempre los prototipos).

Una pequeña diferencia, que en alguna situación puede ser importante, es que en C una constante carácter se transforma automáticamente en un entero. Esto no sucede en C++.

En C, no es un error declarar una variable varias veces, aunque no sea una buena práctica de programación. En C++, esto conduce a un error.

En C, sólo los 31 primeros caracteres de un identificador son significativos. En C++, son, como mínimo, los 1.024 primeros. Sin embargo, desde un punto de vista práctico, los identificadores excesivamente grandes no son manejables y apenas se emplean.

En C, se puede llamar a **main( )** dentro del programa(no es lo normal). C++ no lo permite.

En C, no puede obtenerse la dirección de una variable **register**. En C++, sí que está permitido. Sin embargo, ésta es una característica que puede no emplearse debido a las restricciones en la portabilidad del programa.

EJERCICIOS

1. Compruebe el manual del usuario de su compilador para ver si describe diferencias adicionales entre C y C++.

COMPROBACION DE APTITUD SUPERIOR

Al llegar a este punto, debería ser capaz de realizar estos ejercicios y de responder a estas preguntas.

1. ¿En qué se diferencia un atributo **static** de otro atributo?
2. ¿Qué archivo cabecera debe incluirse en un programa para utilizar E/S basada en arrays?
3. Aparte del hecho de que la E/S basada en arrays utiliza memoria como un dispositivo de entrada y/o salida, ¿existe alguna otra diferencia entre ella y la E/S «normal» en C++?
4. Dada una función llamada **counter( )**, muestre la sentencia que da lugar a que se compile esta función para enlazarla con el lenguaje C.
5. ¿Qué hace una función de conversión?
6. ¿Son opcionales los prototipos en C++?
7. ¿Cuántos caracteres son significativos en un identificador de C++?

COMPROBACION DE APTITUD INTEGRADA

Esta sección comprueba cómo ha asimilado el contenido de este capítulo con el de los anteriores.

1. Ha recorrido un largo camino desde el Capítulo 1. Tómese un tiempo para hojear nuevamente el libro. Piense en la manera de mejorar los ejemplos (especialmente los de los seis primeros capítulos) para que adopten las ventajas de todas las características de C++ aprendidas.
2. Se aprende a programar programando. Escriba muchos programas en C++. Haga ejercicios con aquellas características que identifican a C++.
3. Por último, recuerde: C++ aporta unos medios sin precedentes. Es importante que aprenda a usarlo con moderación. C++ no determina los límites de su capacidad de programación. Sin embargo, si se emplea incorrectamente, se pueden generar programas muy difíciles de entender, imposibles de seguir y extremadamente difíciles de mantener. C++ es una herramienta muy potente. Pero, como cualquier otra herramienta, su gran capacidad es función de la persona que la utilice.

A Palabras clave extendidas de C++

Como se mencionó en el Capítulo 1, hay varias palabras clave que el comité de ANSI C++ ha considerado oportuno incluir en el estándar de C++. Sin embargo, en el momento de escribir este libro no existían compiladores que admitiesen estas palabras o alguna garantía de que todas (o algunas de ellas) estuvieran incluidas en el último estándar de C++. Ninguna de estas palabras claves formaron parte de la especificación original de C++, ni de alguna implementación existente de C++ y no se necesitaban para programar en este lenguaje o no se conocía completamente las ventajas que aportaban. Por estas razones, no se han discutido en este libro las palabras claves extendidas. Sin embargo, en este apéndice se ofrece una lista de las mismas junto con una breve explicación. Será conveniente que compruebe en el manual del usuario de su compilador si están admitidas.

bool

Un especificador de tipo. Los valores de tipo **bool** sólo son **verdadero** o **falso**.

const_cast

Puede utilizarse para redefinir **const** y/o **volatile** cuando se realiza una conversión de tipos.

dynamic_cast

Lleva a cabo un molde en tiempo de ejecución para tipos polimórficos de clases.

false

Veáse **bool**.

mutable

Permite redefinir el valor **const** de un atributo de un objeto. Esto es, un atributo **mutable** de un objeto **const** no es **const** y puede modificarse.

namespace

Declara un bloque en el que pueden declararse otros identificadores. Además, un identificador declarado dentro de **namespace** se convierte en un «subidentificador» enlazado al identificador externo **namespace**. (En concreto, **namespace** crea un ámbito con nombre.)

reinterpret_cast

Cambia el tipo de un valor por otro. Por ejemplo, convierte un puntero en un tipo entero.

static_cast

Un molde no polimórfico. Por ejemplo, un puntero a la clase base puede cambiar su molde por el de un puntero a una clase derivada.

true

Veáse **bool**.

typeid

Obtiene el tipo de una expresión.

using

Especifica un calificador de resolución de alcance implícito.

wchar_t

Admite caracteres anchos(i.e., caracteres de 16 bits). **wchar_t** permite que C++ ajuste los conjuntos de caracteres anchos requeridos por algunos lenguajes.

B Respuestas

1.2. EJERCICIOS

1.
```
#include <iostream.h>

main()
{
  double hours, wage;

  cout << "Introduzca horas trabajadas: ";
  cin >> hours;

  cout << "Introduzca salario por hora: ";
  cin >> wage;

  cout << "La paga es: $" << wage * hours;

  return 0;
}
```

2.
```
#include <iostream.h>

main()
{
  double feet;

  do {
    cout << "Introduzca pies (0 para salir): ";
    cin >> feet;

    cout << feet * 12 << " pulgadas\n";
  } while (feet != 0.0);

  return 0;
}
```

3.
```
// Este programa calcula el mínimo común denominador.
#include <iostream.h>

main()
{
  int a, b, d, min;

  cout << "Introduzca dos números: ";
  cin >> a >> b;

  min = a > b ? b : a;

  for(d = 2; d<min; d++)
    if(((a%d)==0) && ((b%d)==0)) break;
  if(d==min) {
    cout << "No hay común denominador\n";
    return 0;
  }
  cout << "El mínimo común denominador es " << d << ".\n";

  return 0;
}
```

1.3. *EJERCICIO*

1. El comentario es válido, aunque es extraño.

1.4. *EJERCICIOS*

2.
```
#include <iostream.h>
#include <string.h>


class card {
  char title[80]; // título
  char author[40]; // autor
  int number; // número en la biblioteca
public:
  void store(char *t, char *name, int num);
  void show();
};


void card::store(char *t, char *name, int num)
{
  strcpy(title, t);
  strcpy(author, name);
  number = num;
}


void card::show()
{
  cout << "Título: " << title << "\n";
  cout << "Autor: " << author << "\n";
  cout << "Número : " << number << "\n";
}


main()
{
  card book1, book2, book3;


  book1.store("Dune", "Frank Herbert", 2);
  book2.store("The Foundation Trilogy", "Isaac Asimov", 2);
  book3.store("The Rainbow", "D. H. Lawrence", 1);


  book1.show();
  book2.show();
  book3.show();


  return 0;
}
```

3.
```
#include <iostream.h>

#define SIZE 100

class q_type {
  int queue[SIZE]; // guarda la cola
  int head, tail; // índices de la cabeza y la cola
public:
  void init(); // inicializar
  void q(int num); // almacenar
  int deq(); // recuperar
};

// Inicializar
void q_type::init()
{
  head = tail = 0;
}

// Poner valor en la cola.
void q_type::q(int num)
{
  if(tail+1==head || (tail+1==SIZE && !head)) {
    cout << "La cola esta llena\n";
    return;
  }
  tail++;
  if(tail==SIZE) tail = 0; // recorrer cíclicamente
  queue[tail] = num;
}

// Quitar un valor de la cola.
int q_type::deq()
{
  if(head == tail) {
    cout << "La cola está vacía\n";
    return 0; // o algún otro indicador de error
  }
  head++;
  if(head==SIZE) head = 0; // recorrer cíclicamente
  return queue[head];
}

main()
{
  q_type q1, q2;
  int i;

  q1.init();
  q2.init();

  for(i=1; i<=10; i++) {
    q1.q(i);
    q2.q(i*i);
  }
```

```
  for(i=1; i<=10; i++) {
    cout << "Cola 1: " << q1.deq() << "\n";
    cout << "Cola 2: " << q2.deq() << "\n";
  }

  return 0;
}
```

1.5. *EJERCICIO*

1. La función de la biblioteca estándar **strlen( )** no tiene prototipo. El programa tiene que incluir el archivo de cabecera estándar **string.h** para arreglar el problema.

1.6. *EJERCICIOS*

1.
```
#include <iostream.h>
#include <math.h>

// Sobrecarga sroot() para enteros, largos y dobles.

int sroot(int i);
long sroot(long i);
double sroot(double i);

main()
{
  cout << "La raíz cuadrada de 90.34 es : " << sroot(90.34);
  cout << "\n";
  cout << "La raíz cuadrada de 90L es : " << sroot(90L);
  cout << "\n";
  cout << "La raíz cuadrada de 90 es : " << sroot(90);

  return 0;
}

// Devuelve la raíz cuadrada de un entero.
int sroot(int i)
{
  cout << "calculando raíz entera \n";
  return (int) sqrt((double) i);
}

// Devuelve la raíz cuadrada de un largo.
long sroot(long i)
{
  cout << "calculando la raíz de un largo\n";
  return (long) sqrt((double) i);
}

// Devuelve la raíz cuadrada de un double.
```

```
double sroot(double i)
{
  cout << "calculando la raíz de un double\n";
  return sqrt(i);
}
```

2. Las funciones **atof( )**, **atoi( )**, y **atol( )** no se pueden sobrecarcar porque sólo varían en el tipo de los datos que devuelven. La sobrecarga de funciones requiere que varíe el número o el tipo de los argumentos.

3.
```
// Sobrecarga la función min().

#include <iostream.h>
#include <ctype.h>

char min(char a, char b);
int min(int a, int b);
double min(double a, double b);

main()
{
  cout << "El mínimo es: " << min('x', 'a') << "\n";
  cout << "El mínimo es: " << min(10, 20) << "\n";
  cout << "El mínimo es: " << min(0.2234, 99.2) << "\n";

  return 0;
}

// min() para caracteres
char min(char a, char b)
{
  return tolower(a)<tolower(b) ? a : b;
}

// min() para enteros
int min(int a, int b)
{
  return a<b ? a : b;
}

// min() para doubles
double min(double a, double b)
{
  return a<b ? a : b;
}
```

4.
```
#include <iostream.h>
#include <stdlib.h>

// Sobrecarga sleep para que tome un argumento entero o char *
void sleep(int n);
void sleep(char *n);
```

```
// Cambie este valor para que se ajuste a la velocidad de
   //su procesador.
#define DELAY 100000


main()
{
  cout << '.';
  sleep(3);
  cout << '.';
  sleep("2");
  cout << '.';


  return 0;
}


// Sleep() con argumento entero.
void sleep(int n)
{
  long i;


  for( ; n; n--)
    for(i=0; i<DELAY; i++) ;
}


// Sleep() con argumento char *.
void sleep(char *n)
{
  long i;
  int j;


  j = atoi(n);


  for( ; j; j--)
    for(i=0; i<DELAY; i++) ;
}
```

1. *COMPROBACION DE APTITUD SUPERIOR*

1. El polimorfismo es el mecanismo mediante el cual una interfaz general se puede utilizar para acceder a cualquier implementación específica. La encapsulación proporciona un enlace protegido entre el código y sus datos relacionados. El acceso a rutinas encapsuladas se puede controlar estrechamente, evitando así intromisiones no deseadas. La herencia es el proceso mediante el cual un objeto puede adquirir los rasgos de otro. La herencia se utiliza para soportar un sistema de clasificación jerárquica.
2. En un programa en C++ los comentarios se pueden incluir utilizando el comentario normal de C o el comentario específico de una sóla línea de C++.

3.
```
#include <iostream.h>

main()
{
  int b, e, r;
  cout << "Introduzca base: ";
  cin >> b;
  cout << "Introduzca exponente: ";
  cin >> e;

  r = 1;
  for( ; e; e--) r = r * b;

  cout << "Resultado: " << r;

  return 0;
}
```

4.
```
#include <iostream.h>
#include <string.h>

// Sobrecarga la función de inversión de cadena.
void rev_str(char *s); // invierte la cadena
void rev_str(char *in, char *out); // pone la inversión en la salida

main()
{
  char s1[80], s2[80];

  strcpy(s1, "Esto es una prueba");

  rev_str(s1, s2);
  cout << s2 << "\n";

  rev_str(s1);
  cout << s1 << "\n";

  return 0;
}

// Invierte la cadena, pone el resultado en s.
void rev_str(char *s)
{
  char temp[80];
  int i, j;

  for(i=strlen(s)-1, j=0; i>=0; i--, j++)
    temp[j] = s[i];

  temp[j] = '\0'; // resultado terminado en nulo

  strcpy(s, temp);
}
```

```
// Invierte la cadena, pone el resultado en la salida.
void rev_str(char *in, char *out)
{
  int i, j;

  for(i=strlen(in)-1, j=0; i>=0; i--, j++)
    out[j] = in[i];

  out[j] = '\0'; // resultado terminado en nulo
}
```

2. COMPROBACION DE APTITUD

1.
```
#include <iostream.h>
#include <string.h>

main()
{
  char s[80];

  cout << "Introduzca una cadena: ";
  cin >> s;

  cout << "Longitud: " << strlen(s) << "\n";

  return 0;
}
```

2.
```
#include <iostream.h>
#include <string.h>

class addr {
  char name[40];
  char street[40];
  char city[30];
  char state[3];
  char zip[10];
public:
  void store(char *n, char *s, char *c, char *t,
             char *z);
  void display();
};

void addr::store(char *n, char *s, char *c, char *t,
                 char *z)
{
  strcpy(name, n);
  strcpy(street, s);
  strcpy(city, c);
  strcpy(state, t);
  strcpy(zip, z);
}
```

```
void addr::display()
{
  cout << name << "\n";
  cout << street << "\n";
  cout << city << "\n";
  cout << state << "\n";
  cout << zip << "\n\n";
}

main()
{
  addr a;

  a.store("C. B. Turkle", "11 Pinetree Lane", "Wausau",
          "In", "46576");

  a.display();

  return 0;
}
```

3.
```
#include <iostream.h>

int rotate(int i);
long rotate(long i);

main()
{
  int a;
  long b;

  a = 0x8000;
  b = 8;

  cout << rotate(a);
  cout << "\n";
  cout << rotate(b);

  return 0;
}

int rotate(int i)
{
  int x;

  if(i & 0x8000) x = 1;
  else x = 0;

  i = i << 1;
  i += x;

  return i;
}
```

```
long rotate(long i)
{
  int x;

  if(i & 0x80000000) x = 1;
  else x = 0;

  i = i << 1;
  i += x;

  return i;
}
```

4. El entero **i** es privado a **myclass** y no se puede acceder a él dentro de **main( )**.

2.1. *EJERCICIOS*

1.
```
#include <iostream.h>

#define SIZE 100

class q_type {
  int queue[SIZE]; // guarda la cola
  int head, tail; // indices de cabeza y cola
public:
  q_type(); // constructor
  void q(int num); // guardar
  int deq(); // recuperar
};

// Constructor
q_type::q_type()
{
  head = tail = 0;
}

// Poner valor en la cola.
void q_type::q(int num)
{
  if(tail+1==head || (tail+1==SIZE && !head)) {
    cout << "Cola llena\n";
    return;
  }
  tail++;
  if(tail==SIZE) tail = 0; // recorrer cíclicamente
  queue[tail] = num;
}

// Quitar valor de la cola.
int q_type::deq()
```

```
{
  if(head == tail) {
    cout << "La cola está vacía\n";
    return 0; // o algún otro indicador de error
  }
  head++;
  if(head==SIZE) head = 0; // recorrer cíclicamente
  return queue[head];
}

main()
{
  q_type q1, q2;
  int i;

  for(i=1; i<=10; i++) {
    q1.q(i);
    q2.q(i*i);
  }

  for(i=1; i<=10; i++) {
    cout << "De la cola 1: " << q1.deq() << "\n";
    cout << "De la cola 2: " << q2.deq() << "\n";
  }

  return 0;
}
```

2.
```
// Emulador de un cronógrafo
#include <iostream.h>
#include <time.h>

class stopwatch {
  double begin, end;
public:
  stopwatch();
  ~stopwatch();
  void start();
  void stop();
  void show();
};

stopwatch::stopwatch()
{
  begin = end = 0.0;
}

stopwatch::~stopwatch()
{
  cout << "El objeto cronógrafo se está destruyendo...";
  show();
}

void stopwatch::start()
```

```
{
  begin = (double) clock() / CLK_TCK;
}

void stopwatch::stop()
{
  end = (double) clock() / CLK_TCK;
}

void stopwatch::show()
{
  cout << "Tiempo transcurrido: " << end - begin;
  cout << "\n";
}

main()
{
  stopwatch watch;
  long i;

  watch.start();
  for(i=0; i<320000; i++) ; // tiempo que dura el bucle
  watch.stop();

  watch.show();

  return 0;
}
```

3. Un constructor no puede tener un tipo de retorno.

2.2. *EJERCICIOS*

1.
```
// Pila asignada dinámicamente.
#include <iostream.h>
#include <stdlib.h>

// Declarar una clase pila para caracteres
class stack {
  char *stck; // guarda la pila
  int tos; // índice de la cabeza de la pila
  int size; // tamaño de la pila
public:
  stack(int s); // constructor
  ~stack(); // destructor
  void push(char ch); // mete carácter en la pila
  char pop(); // saca carácter de la pila
};

// inicializar la pila
stack::stack(int s)
```

```
{
  cout << "Construyendo una pila\n";
  tos = 0;
  stck = (char *) malloc(s);
  if(!stck) {
    cout << "Error de asignación...";
    exit(1);
  }
  size = s;
}

stack::~stack()
{
  free(stck);
}

// Mete un carácter.
void stack::push(char ch)
{
  if(tos==size) {
    cout << "La pila está llena\n";
    return;
  }
  stck[tos] = ch;
  tos++;
}

// Saca un carácter.
char stack::pop()
{
  if(tos==0) {
    cout << "La pila está vacía\n";
    return 0; // devuelve nulo cuando la pila está vacía
  }
  tos--;
  return stck[tos];
}

main()
{
  // crea dos pilas que se inicializan automáticamente.
  stack s1(10), s2(10);
  int i;

  s1.push('a');
  s2.push('x');
  s1.push('b');
  s2.push('y');
  s1.push('c');
  s2.push('z');

  for(i=0; i<3; i++) cout << "Saca de s1: " << s1.pop() << "\n";
  for(i=0; i<3; i++) cout << "Saca de s2: " << s2.pop() << "\n";
  return 0;
}
```

2.
```
#include <iostream.h>
#include <time.h>

class t_and_d {
  time_t systime;
public:
  t_and_d(time_t t); // constructor
  void show();
};

t_and_d::t_and_d(time_t t)
{
  systime = t;
}

void t_and_d::show()
{
  cout << ctime(&systime);
}

main()
{
  time_t x;

  x = time(NULL);

  t_and_d ob(x);

  ob.show();

  return 0;
}
```

3.
```
#include <iostream.h>

class box {
  double l, w, h;
  double volume;
public:
  box(double a, double b, double c);
  void vol();
};

box::box(double a, double b, double c)
{
  l = a;
  w = b;
  h = c;

  volume = l * w * h;
}

void box::vol()
```

```
{
  cout << "El volumen es: " << volume << "\n";
}

main()
{
  box x(2.2, 3.97, 8.09), y(1.0, 2.0, 3.0);

  x.vol();
  y.vol();
  return 0;
}
```

2.3. EJERCICIO

1.
```
#include <iostream.h>

class area_cl {
public:
  double height;
  double width;
};

class box : public area_cl {
public:
  box(double h, double w);
  double area();
};

class isosceles : public area_cl {
public:
  isosceles(double h, double w);
  double area();
};

box::box(double h, double w)
{
  height = h;
  width = w;
}

isosceles::isosceles(double h, double w)
{
  height = h;
  width = w;
}

double box::area()
{
  return width * height;
}

double isosceles::area()
```

```
{
  return 0.5 * width * height;
}

main()
{
  box b(10.0, 5.0);
  isosceles i(4.0, 6.0);

  cout << "Cuadro: " << b.area() << "\n";
  cout << "Triángulo: " << i.area() << "\n";

  return 0;
}
```

2.5. *EJERCICIOS*

1.
```
// Clase pila utilizando una estructura.
#include <iostream.h>

#define SIZE 10

// Declara una clase pila para caracteres utilizando una
//estructura.
struct stack {
  stack(); // constructor
  void push(char ch); // mete carácter en la pila
  char pop(); // saca carácter de la pila
private:
  char stck[SIZE]; // guarda la pila
  int tos; // índice de la cabeza de la pila
};

// inicializar la pila.
stack::stack()
{
  cout << "Construyendo una pila\n";
  tos = 0;
}

// Mete un carácter.
void stack::push(char ch)
{
  if(tos==SIZE) {
  cout << "La pila está llena";
  return;
  }
  stck[tos] = ch;
  tos++;
}

// Saca un carácter.
char stack::pop()
```

```
{
  if(tos==0) {
  cout << "La pila está vacía";
  return 0; // devuelve nulo cuando la pila está vacía
  }
  tos--;
  return stck[tos];
}

main()
{
  // crea dos pilas que se inicializan automáticamente.
  stack s1, s2;
  int i;

  s1.push('a');
  s2.push('x');
  s1.push('b');
  s2.push('y');
  s1.push('c');
  s2.push('z');

  for(i=0; i<3; i++) cout << "Saca de s1: " << s1.pop() << "\n";
  for(i=0; i<3; i++) cout << "Saca de s2: " << s2.pop() << "\n";

  return 0;
}
```

2.
```
#include <iostream.h>

union swap {
  unsigned char c[2];
  unsigned i;
  swap(unsigned x);
  void swp();
};

swap::swap(unsigned x)
{
  i = x;
}

void swap::swp()
{
  unsigned char temp;

  temp = c[0];
  c[0] = c[1];
  c[1] = temp;
}

main()
{
  swap ob(1);

  ob.swp();
  cout << ob.i;

  return 0;
}
```

2.6. EJERCICIOS

1.
```
#include <iostream.h>

// Sobrecarga abs() de tres formas:

// abs() para enteros
inline int abs(int n)
{
  cout << "En entero abs()\n";
  return n<0 ? -n : n;
}

// abs() para longs
inline long abs(long n)
{
  cout << "En long abs()\n";
  return n<0 ? -n : n;
}

// abs() para doubles
inline double abs(double n)
{
  cout << "En double abs()\n";
  return n<0 ? -n : n;
}

main()
{
  cout << "Valor absoluto de -10: " << abs(-10) << "\n";
  cout << "Valor absoluto de -10L: " << abs(-10L) << "\n";
  cout << "Valor absoluto de -10.01: " << abs(-10.01) << "\n";

  return 0;
}
```

2. Podría ser posible que la función no se insertase porque contiene un bucle **for**. Muchos compiladores no insertan funciones que contienen bucles.

2.7. EJERCICIOS

1.
```
#include <iostream.h>

#define SIZE 10

// Declara una clase pila para caracteres.
class stack {
  char stck[SIZE]; // guarda la pila
  int tos; // índice de la cabeza de la pila
public:
```

```
  stack() { tos = 0; }
  void push(char ch)
  {
    if(tos==SIZE) {
    cout << "La pila está llena";
    return;
    }
    stck[tos] = ch;
    tos++;
  }

  char pop()
  {
    if(tos==0) {
    cout << "La pila está vacía";
    return 0; // devuelve nulo cuando la pila está vacía
    }
    tos--;
    return stck[tos];
  }
};

main()
{
  // crea dos pilas que se inicializan automáticamente.
  stack s1, s2;
  int i;

  s1.push('a');
  s2.push('x');
  s1.push('b');
  s2.push('y');
  s1.push('c');
  s2.push('z');

  for(i=0; i<3; i++) cout << "Saca de s1: " << s1.pop() << "\n";
  for(i=0; i<3; i++) cout << "Saca de s2: " << s2.pop() << "\n";
  return 0;
}
```

2.
```
#include <iostream.h>
#include <malloc.h>
#include <string.h>
#include <stdlib.h>

class strtype {
  char *p;
  int len;
public:
  strtype(char *ptr)
  {
    len = strlen(ptr);
    p = (char *) malloc(len+1);
  if(!p) {
    cout << "Error de asignación\n";
    exit(1);
  }
```

```
  strcpy(p, ptr);
  }
  ~strtype() { cout << "Liberando p\n"; free(p); }

  void show()
  {
    cout << p << " - Longitud: " << len;
    cout << "\n";
  }
};

main()
{
  strtype s1("Esto es una prueba"), s2("Me gusta C++");

  s1.show();
  s2.show();

  return 0;
}
```

2. COMPROBACION DE APTITUD SUPERIOR

1. Un constructor es la función que se llama cuando se crea un objeto. Un destructor es la función que se llama cuando se destruye un objeto.
2.
```
#include <iostream.h>

class line {
  int len;
public:
  line(int l);
};

line::line(int l)
{
  len = l;

  int i;
  for(i=0; i<len; i++) cout << '*';
}

main()
{
  line l(10);

  return 0;
}
```

3. 10 1000000 −0.009
4.
```
#include <iostream.h>

class area_cl {
```

```
public:
  double height;
  double width;
};

class box : public area_cl {
public:
  box(double h, double w) { height = h; width = w; }
  double area() { return height * width; }
};

class isosceles : public area_cl {
public:
  isosceles(double h, double w) { height = h; width = w; }
  double area() { return 0.5 * width * height; }};

class cylinder : public area_cl {
public:
  cylinder(double h, double w) { height = h; width = w; }
  double area()
  {
    return (2 * 3.1416 * (width/2) * (width/2)) +
           (3.1416 * width * height);
  }
};

main()
{
  box b(10.0, 5.0);
  isosceles i(4.0, 6.0);
  cylinder c(3.0, 4.0);

  cout << "Cuadro: " << b.area() << "\n";
  cout << "Triángulo: " << i.area() << "\n";
  cout << "Cilindro: " << c.area() << "\n";

  return 0;
}
```

5. El código de una función insertada se expande directamente. Esto significa que no se llama realmente a la función. Esto evita lo que está asociado con la llamada a la función y el mecanismo de retorno. La ventaja es que incrementa la velocidad de ejecución. La desventaja es que puede incrementar el tamaño del programa.

6.
```
#include <iostream.h>

class myclass {
  int i, j;
public:
  myclass(int x, int y) { i = x; j = y; }
  void show() { cout << i << " " << j; }
};

main()
```

```
{
  myclass count(2, 3);

  count.show();

  return 0;
}
```

7. En una clase, los miembros son privados por omisión. En una estructura, los miembros son públicos por omisión.

2. COMPROBACION DE APTITUD INTEGRADA

1.
```
#include <iostream.h>

class prompt {
  int count;
public:
  prompt(char *s) { cout << s; cin >> count; };
  ~prompt();
};

prompt::~prompt() {
  int i, j;

  for(i=0; i<count; i++) {
    cout << '\a';
    for(j=0; j<32000; j++) ; // retardo
  }
}

main()
{
  prompt ob("Introduzca un número: ");

  return 0;
}
```

2.
```
#include <iostream.h>

class ftoi {
  double feet;
  double inches;
public:
  ftoi(double f);
};

ftoi::ftoi(double f)
{
  feet = f;
```

```
  inches = feet * 12;
  cout << feet << " son " << inches << " pulgadas.\n";
}

main()
{
  ftoi a(12.0), b(99.0);

  return 0;
}
```

3.
```
#include <iostream.h>
#include <stdlib.h>

class dice {
  int val;
public:
  void roll();
};

void dice::roll()
{
  val = (rand() % 6) + 1; // genera del 1 al 6
  cout << val << "\n";
}

main()
{
  dice one, two;

  one.roll();
  two.roll();
  one.roll();
  two.roll();
  one.roll();
  two.roll();

  return 0;
}
```

3. COMPROBACION DE APTITUD

1. El constructor se llama **widgit( )** y el destructor se llama **~widgit( )**.
2. A la función constructor se le llama cuando se crea un objeto (es decir, empieza a existir). Al destructor se le llama cuando se destruye un objecto.

3.
```
class Mars : public planet {
// ...
};
```

4. Una función se puede expandir dentro del código precediendo su definición con el especificador **inline** o incluyendo su definición dentro de una declaración de clase.

5. Una función insertada se tiene que definir antes de usarla por primera vez. Otras restricciones comunes incluyen las siguientes: No puede contener bucles. No tiene que ser recursiva. No puede contener una sentencia **goto** o **switch**. Por último, no puede contener ninguna variable **static**.

6. `sample ob(100, 'X');`

3.1. *EJERCICIOS*

1. La sentencia de asignación **x = y** es errónea porque **cl1** y **cl2** son dos tipos de clases diferentes, y no se pueden asignar objetos de tipos de clase diferentes.

2.
```
#include <iostream.h>

#define SIZE 100

class q_type {
  int queue[SIZE]; // guarda la cola
  int head, tail; // indices de cabeza y cola
public:
  q_type(); // constructor
  void q(int num); // guardar
  int deq(); // recuperar
};

// Constructor
q_type::q_type()
{
  head = tail = 0;
}

// Poner valor en la cola.
void q_type::q(int num)
{
  if(tail+1==head || (tail+1==SIZE && !head)) {
    cout << "Cola llena\n";
    return;
  }
  tail++;
  if(tail==SIZE) tail = 0; // recorrer cíclicamente
  queue[tail] = num;
}

// Quitar valor de la cola.
int q_type::deq()
{
  if(head == tail) {
    cout << "La cola está vacía\n";
    return 0; // o algún otro indicador de error
  }
  head++;
  if(head==SIZE) head = 0; // recorrer cíclicamente
  return queue[head];
}
```

```
main()
{
  q_type q1, q2;
  int i;

  for(i=1; i<=10; i++) {
    q1.q(i);
  }

  // asigna una cola a otra
  q2 = q1;

  // mostrar que ambas tienen los mismos contenidos
  for(i=1; i<=10; i++)
    cout << "De la cola 1: " << q1.deq() << "\n";

  for(i=1; i<=10; i++)
    cout << "De la cola 2: " << q2.deq() << "\n";

  return 0;
}
```

3. Si la memoria que va a guardar la cola se asigna dinámicamente, entonces la asignación de una cola a otra hace que se pierda la memoria dinámica asignada a la cola de la parte izquierda de la sentencia de asignación y la memoria asignada a la cola de la parte derecha se libera dos veces cuando se destruyen los objetos. Cualquiera de estas dos condiciones es un error inaceptable.

3.2. *EJERCICIOS*

1.
```
#include <iostream.h>

#define SIZE 10

// Declara una clase pila para caracteres
class stack {
  char stck[SIZE]; // guarda la pila
  int tos; // índice de la cabeza de la pila
public:
  stack(); // constructor
  void push(char ch); // mete carácter en la pila
  char pop(); // saca carácter de la pila
};

// inicializar la pila
stack::stack()
{
  cout << "Construyendo una pila\n";
  tos = 0;
}
```

```
// Mete un carácter.
void stack::push(char ch)
{
  if(tos==SIZE) {
    cout << "La pila está llena";
    return;
  }
  stck[tos] = ch;
  tos++;
}

// Saca un carácter.
char stack::pop()
{
  if(tos==0) {
    cout << "La pila está vacía";
    return 0; // devuelve nulo cuando la pila está vacía
  }
  tos--;
  return stck[tos];
}

void showstack(stack o);

main()
{
  stack s1;
  int i;

  s1.push('a');
  s1.push('b');
  s1.push('c');

  showstack(s1);

  // s1 en main todavía existe
  cout << "la pila s1 todavía contiene esto: \n";
  for(i=0; i<3; i++) cout << s1.pop() << "\n";

  return 0;
}

// Muestra los contenidos de una pila.
void showstack(stack o)
{
  char c;

  // cuando termina esta sentencia, la pila o está vacía
  while(c=o.pop()) cout << c << "\n";
  cout << "\n";
}
```

Este programa muestra lo siguiente:

```
Construyendo una pila
c
b
a
La pila está vacía
La pila s1 todavía contiene esto:
c
b
a
```

2. La memoria utilizada para contener el entero al que apunta **p** en el objeto **o** utilizado para llamar a **neg( )** se libera cuando se destruye la copia de o al terminar **neg( )** aún considerando que esta memoria la sigue necesitando **o** dentro de **main( )**.

3.3. *EJERCICIOS*

1.
```
#include <iostream.h>

class who {
  char name;
public:
  who(char c) {
    name = c;
    cout << "Construyendo la clase who #";
    cout << name << "\n";
  }
  ~who() { cout << "Destruyendo la clase who #" << name << "\n"; }
};

who makewho()
{
  who temp('B');
  return temp;
}

main()
{
  who ob('A');

  makewho();

  return 0;
}
```

2. Hay varias situaciones en las que sería inadecuado devolver un objeto. Esta es una: si un objeto abre un archivo de disco cuando se crea y cierra ese archivo cuando se destruye.

3.4. *EJERCICIO*

1.
```
#include <iostream.h>

class pr2; // referencia adelantada

class pr1 {
  int printing;
  // ...
public:
  pr1() { printing = 0; }
  void set_print(int status) { printing = status; }
  // ...
  friend int inuse(pr1 o1, pr2 o2);
};

class pr2 {
  int printing;
  // ...
public:
  pr2() { printing = 0; }
  void set_print(int status) { printing = status; }
  // ...
  friend int inuse(pr1 o1, pr2 o2);
};

// Devuelve verdadero si la impresora está en uso.
int inuse(pr1 o1, pr2 o2)
{
  if(o1.printing || o2.printing) return 1;
  else return 0;
}

main()
{
  pr1 p1;
  pr2 p2;

  if(!inuse(p1, p2)) cout << "Impresora sin utilizar\n";

  cout << "Poniendo p1 a imprimir...\n";
  p1.set_print(1);
  if(inuse(p1, p2)) cout << "Ahora, la impresora está
  en uso.\n";

  cout << "Desactivar p1...\n";
  p1.set_print(0);
  if(!inuse(p1, p2)) cout << "Impresora sin utilizar\n";

  cout << "Activar p2...\n";
  p2.set_print(1);
  if(inuse(p1, p2)) cout << "Ahora, la impresora está
  en uso.\n";

  return 0;
}
```

3. COMPROBACION DE APTITUD SUPERIOR

1. Para poder asignar un objeto a otro, ambos tienen que ser del mismo tipo de clase.
2. El problema con la asignación de **ob1** a **ob2** es que la memoria a la que apunta el valor inicial **ob2** de **p** ahora se ha perdido porque este valor se ha sobreescrito al hacer la asignación. Esta memoria es imposible liberarla y la memoria a la que apunta el valor **ob2** de **p** se libera dos veces cuando se destruye —posiblemente produciendo daños en el sistema de asignación dinámica.

3.
```
int light(planet p)
{
  return p.get_miles() / 186000;
}
```

4. Si.

5.
```
// Carga una pila con el alfabeto.
#include <iostream.h>

#define SIZE 27

// Declara una clase pila para caracteres
class stack {
  char stck[SIZE]; // guarda la pila
  int tos; // índice de la cabeza de la pila
public:
  stack(); // constructor
  void push(char ch); // mete carácter en la pila
  char pop(); // saca carácter de la pila
};

// inicializar la pila
stack::stack()
{
  cout << "Construyendo una pila\n";
  tos = 0;
}

// Mete un carácter.
void stack::push(char ch)
{
  if(tos==SIZE) {
    cout << "La pila está llena";
    return;
  }
  stck[tos] = ch;
  tos++;
}
```

```
// Saca un carácter.
char stack::pop()
{
  if(tos==0) {
    cout << "La pila está vacía";
    return 0; // devuelve nulo cuando la pila está vacía
  }
  tos--;
  return stck[tos];
}

void showstack(stack o);
stack loadstack();

main()
{
  stack s1;

  s1 = loadstack();
  showstack(s1);

  return 0;
}

// Muestra los contenidos de una pila.
void showstack(stack o)
{
  char c;

  // cuando termina esta sentencia, la pila o está vacía
  while(c=o.pop()) cout << c << "\n";
  cout << "\n";
}

// Carga una pila con las letras del alfabeto.
stack loadstack()
{
  stack t;
  char c;
  for(c = 'a'; c<='z'; c++) t.push(c);
  return t;
}
```

6. Cuando se pasa un objeto a una función o cuando se devuelve un objeto de una función, se crean objetos temporales que se destruyen cuando la función termina. Cuando se destruye un objeto temporal, la función destructor puede destruir algo que puede ser necesario en alguna parte del programa.
7. Una función amiga es una función no miembro que tiene acceso a las partes privadas de la clase de la que es amiga. Es decir, una función amiga tiene acceso a las partes privadas de la clase de la que es amiga, pero no es miembro de esa clase.

3. COMPROBACION DE APTITUD INTEGRADA

1.
```
// Carga una pila con el alfabeto.
#include <iostream.h>
#include <ctype.h>

#define SIZE 27

// Declara una clase pila para caracteres
class stack {
  char stck[SIZE]; // guarda la pila
  int tos; // índice de la cabeza de la pila
public:
  stack(); // constructor
  void push(char ch); // mete carácter en la pila
  char pop(); // saca carácter de la pila
};

// Inicializar la pila
stack::stack()
{
  cout << "Construyendo una pila\n";
  tos = 0;
}

// Mete un carácter.
void stack::push(char ch)
{
  if(tos==SIZE) {
    cout << "La pila está llena";
    return;
  }
  stck[tos] = ch;
  tos++;
}

// Saca un carácter.
char stack::pop()
{
  if(tos==0) {
    cout << "La pila está vacía";
    return 0; // devuelve nulo cuando la pila está vacía
  }
  tos--;
  return stck[tos];
}

void showstack(stack o);
stack loadstack();
stack loadstack(int upper);
```

```
main()
{
  stack s1, s2, s3;

  s1 = loadstack();
  showstack(s1);

  // obtiene letras mayúsculas
  s2 = loadstack(1);
  showstack(s2);

  // utiliza letras minúsculas
  s3 = loadstack(0);
  showstack(s3);

  return 0;
}

// Muestra los contenidos de una pila.
void showstack(stack o)
{
  char c;

  // cuando termina esta sentencia, la pila o está vacía
  while(c=o.pop()) cout << c << "\n";
  cout << "\n";
}

// Carga una pila con las letras del alfabeto.
stack loadstack()
{
  stack t;
  char c;

  for(c = 'a'; c<='z'; c++) t.push(c);
  return t;
}

/* Carga una pila con las letras del alfabeto. Letras
   mayúsculas si upper es 1; minúsculas en caso contrario. */
stack loadstack(int upper)
{
  stack t;
  char c;

  if(upper) c = 'A';
  else c = 'a';

  for(; toupper(c)<='Z'; c++) t.push(c);
  return t;
}
```

2.
```
#include <iostream.h>
#include <malloc.h>
#include <string.h>
#include <stdlib.h>

class strtype {
  char *p;
  int len;
public:
  strtype(char *ptr);
  ~strtype();
  void show();
  friend char *get_string(strtype *ob);
};

strtype::strtype(char *ptr)
{
  len = strlen(ptr);
  p = (char *) malloc(len+1);
  if(!p) {
    cout << "Error de asignación\n";
    exit(1);
  }
  strcpy(p, ptr);
}

strtype::~strtype()
{
  cout << "Liberando p\n";
  free(p);
}

void strtype::show()
{
  cout << p << " - longitud: " << len;
  cout << "\n";
}

char *get_string(strtype *ob)
{
  return ob->p;
}

main()
{
  strtype s1("Esto es una prueba");

  char *s;

  s1.show();

  // obtener puntero a cadena
  s = get_string(&s1);
  cout << "Aquí está la cadena contenida en s1: ";
  cout << s << "\n";


  return 0;
}
```

3. El resultado del experimento es el siguiente: Si, la información de la clase base también se copia cuando un objeto de una clase derivada se asigna a otro. El siguiente programa demuestra esto:

```
#include <iostream.h>

class base {
  int a;
public:
  void load_a(int n) { a = n; }
  int get_a() { return a; }
};

class derived : public base {
  int b;
public:
  void load_b(int n) { b = n; }
  int get_b() { return b; }
};

main()
{
  derived ob1, ob2;

  ob1.load_a(5);
  ob1.load_b(10);

  // asigna ob1 a ob2
  ob2 = ob1;

  cout << "Aquí está a y b de ob1: ";
  cout << ob1.get_a() << ' ' << ob1.get_b() << "\n";

  cout << "Aquí está a y b de ob2: ";
  cout << ob2.get_a() << ' ' << ob2.get_b() << "\n";

  /* Como puede suponer, la salida es la misma para cada objeto. */

  return 0;
}
```

4. *COMPROBACION DE APTITUD*

1. Cuando un objeto se asigna a otro del mismo tipo, los valores actuales de todos los miembros del objeto de la parte derecha se asignan a los miembros correspondientes de la *izquierda*.
2. El problema puede surgir cuando al asignar un objeto a otro la asignación sobreescribe información importante ya existente en el objeto al que se está asignando. Por ejemplo, un puntero a memoria dinámica o a un archivo abierto se puede sobreescribir y por lo tanto perder.

3. Cuando se pasa un objeto a una función se hace una copia. Sin embargo, no se llama a la función constructor de la copia. Se llama al destructor de la copia cuando se destruye el objeto al terminar la función.
4. La violación de la separación entre un argumento y su copia cuando se pasa a un parámetro se puede producir por muchas situaciones. Por ejemplo, si la memoria dinámica la libera un destructor, entonces esa memoria también se perderá para el argumento. En general, si la función destructor destruye cualquier cosa requerida por el argumento original, se producirán daños en el argumento.

5.
```
#include <iostream.h>

class summation {
  int num;
  long sum; // suma de num
public:
  void set_sum(int n);
  void show_sum() {
    cout << num << " la suma es " << sum << "\n";
  }
};

void summation::set_sum(int n)
{
  int i;

  num = n;

  sum = 0;
  for(i=1; i<=n; i++)
  sum += i;
}

summation make_sum()
{
  int i;
  summation temp;

  cout << "Introduzca número: ";
  cin >> i;

  temp.set_sum(i);

  return temp;
}

main()
{
  summation s;

  s = make_sum();

  s.show_sum();

  return 0;
}
```

6. Para muchos compiladores, las funciones insertadas no pueden contener bucles.

7.
```
#include <iostream.h>

class myclass {
  int num;
public:
  myclass(int x) { num = x; }
  friend int isneg(myclass ob);
};

int isneg(myclass ob)
{
  return (ob.num < 0) ? 1 : 0;
}

main()
{
  myclass a(-1), b(2);

  cout << isneg(a) << ' ' << isneg(b);
  cout << "\n";

  return 0;
}
```

8. Sí, una función amiga puede ser amiga de más de una clase.

4.1. *EJERCICIOS*

1.
```
#include <iostream.h>

class letters {
  char ch;
public:
  letters(char c) { ch = c; }
  char get_ch() { return ch; }
};

main()
{
  letters ob[10] = { 'a', 'b', 'c','d', 'e', 'f',
                     'g', 'h', 'i', 'j' };

  int i;

  for(i=0; i<10; i++)
    cout << ob[i].get_ch() << ' ';

  cout << "\n";

  return 0;
}
```

2.
```
#include <iostream.h>

class squares {
  int num, sqr;
public:
  squares(int a, int b) { num = a; sqr = b; }
  void show() {cout << num << ' ' << sqr << "\n"; }
};

main()
{
  squares ob[10] = {
    squares(1, 1),
    squares(2, 4),
    squares(3, 9),
    squares(4, 16),
    squares(5, 25),
    squares(6, 36),
    squares(7, 49),
    squares(8, 64),
    squares(9, 81),
    squares(10, 100)
  };
  int i;

  for(i=0; i<10; i++) ob[i].show();

  return 0;
}
```

3.
```
#include <iostream.h>

class letters {
  char ch;
public:
  letters(char c) { ch = c; }
  char get_ch() { return ch; }
};

main()
{
  letters ob[10] = {
    letters('a'),
    letters('b'),
    letters('c'),
    letters('d'),
    letters('e'),
    letters('f'),
    letters('g'),
    letters('h'),
    letters('i'),
    letters('j')
  };

  int i;
```

```
  for(i=0; i<10; i++)
    cout << ob[i].get_ch() << ' ';

  cout << "\n";

  return 0;
}
```

4.2. EJERCICIOS

1.
```
// Muestra en orden inverso.
#include <iostream.h>

class samp {
  int a, b;
public:
  samp(int n, int m) { a = n; b = m; }
  int get_a() { return a; }
  int get_b() { return b; }
};

main()
{
  samp ob[4] = {
    samp(1, 2),
    samp(3, 4),
    samp(5, 6),
    samp(7, 8)
  };
  int i;

  samp *p;

  p = &ob[3]; // obtener la dirección del último elemento

  for(i=0; i<4; i++) {
    cout << p->get_a() << ' ';
    cout << p->get_b() << "\n";
    p--; // pasa al objeto anterior
  }

  cout << "\n";

  return 0;
}
```

2.
```
/* Crea un array de objetos bidimensional.
   El acceso es mediante puntero. */
#include <iostream.h>

class samp {
```

```
  int a;
public:
  samp(int n) { a = n; }
  int get_a() { return a; }
};

main()
{
  samp ob[4][2] = {
    1, 2,
    3, 4,
    5, 6,
    7, 8
  };
  int i;

  samp *p;

  p = (samp *) ob;

  for(i=0; i<4; i++) {
    cout << p->get_a() << ' ';
    p++;
    cout << p->get_a() << "\n";
    p++;
  }

  cout << "\n";

  return 0;
}
```

4.3. *EJERCICIO*

1.
```
// Utiliza el puntero this.
#include <iostream.h>

class myclass {
  int a, b;
public:
  myclass(int n, int m) { this->a = n; this->b = m; }
  int add() { return this->a + this->b; }
  void show();
};

void myclass::show()
{
  int t;

  t = this->add(); // llama a la función miembro
  cout << t << "\n";
}
```

```
main()
{
  myclass ob(10, 14);

  ob.show();

  return 0;
}
```

4.4. EJERCICIOS

1.
```
#include <iostream.h>

main()
{
  float *f;
  long *l;
  char *c;

  f = new float;
  l = new long;
  c = new char;

  if(!f || !l || !c) {
    cout << "Error de asignación.";
    return 1;
  }

  *f = 10.102;
  *l = 100000;
  *c = 'A';

  cout << *f << ' ' << *l << ' ' << *c;
  cout << '\n';

  delete f; delete l; delete c;

  return 0;
}
```

2.
```
#include <iostream.h>
#include <string.h>

class phone {
  char name[40];
  char number[14];
public:
  void store(char *n, char *num);
  void show();
};

void phone::store(char *n, char *num)
```

```
{
  strcpy(name, n);
  strcpy(number, num);
}

void phone::show()
{
  cout << name << ": " << number;
  cout << "\n";
}

main()
{
  phone *p;

  p = new phone;

  if(!p) {
    cout << "Error de asignación.";
    return 1;
  }

  p->store("Issac Newton", "111 555-2323");

  p->show();

  delete p;

  return 0;
}
```

4.5. *EJERCICIOS*

1.
```
char *p;

p = new char [100];
// ...
strcpy(p, "Esto es una prueba");
```

2.
```
#include <iostream.h>

main()
{
  double *p;

  p = new double (-123.0987);

  cout << *p << '\n';

  return 0;
}
```

4.6. EJERCICIOS

1.
```
#include <iostream.h>

void rneg(int &i);
void pneg(int *i);

main()
{
  int i = 10;
  int j = 20;

  rneg(i);
  pneg(&j);

  cout << i << ' ' << j << '\n';

  return 0;
}

void rneg(int &i)
{
  i = -i;
}

void pneg(int *i)
{
  *i = - *i;
}
```

2. Cuando se llama a **triple( )**, la dirección de **d** se obtiene explícitamente utilizando el operador **&**. Esto ni es necesario ni es legal. Cuando se utiliza un parámetro por referencia, el argumento no está precedido por el operador **&**.
3. La dirección de un parámetro por referencia se pasa automáticamente a la función. No hace falta obtener la dirección manualmente. El paso por referencia es más rápido que el paso por valor. No se genera copia del argumento. Por lo tanto, no hay posibilidad de que se produzcan efectos laterales porque se llama al destructor de la copia.

4.7. EJERCICIO

1. En el programa original el objeto se pasa a **show( )** por valor. De este modo se hace una copia. Cuando se vuelve de **show( )**, la copia se destruye y se llama a su destructor. Esto hace que se libere **p**, pero la memoria a la que apunta la siguen necesitando los argumentos de **show( )**. Aquí se muestra una versión corregida que usa un parámetro por referencia para evitar que se haga una copia cuando se llama a la función.

```
// Este programa ahora está corregido.
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
```

```
class strtype {
  char *p;
public:
  strtype(char *s);
  ~strtype() { delete p; }
  char *get() { return p; }
};

strtype::strtype(char *s)
{
  int l;

  l = strlen(s);

  p = new char [l];
  if(!p) {
    cout << "Error de asignación\n";
    exit(1);
  }

  strcpy(p, s);
}

// Se corrige utilizando un parámetro por referencia.
void show(strtype &x)
{
  char *s;

  s = x.get();
  cout << s << "\n";
}

main()
{
  strtype a("Hola"), b("Gente");

  show(a);
  show(b);

  return 0;
}
```

4.8. *EJERCICIOS*

1.
```
// Un ejemplo de array de dos dimensiones con límite.
#include <iostream.h>
#include <stdlib.h>

class array {
  int isize, jsize;
  int *p;
```

```
public:
  array(int i, int j);
  &put(int i, int j);
  int get(int i, int j);
};

array::array(int i, int j)
{
  p = new int [i*j];
  if(!p) {
    cout << "Error de asignación\n";
    exit(1);
  }
  isize = i;
  jsize = j;
}

// Poner algo en el array.
int &array::put(int i, int j)
{
  if(i<0 || i>=isize || j<0 || j>=jsize) {
    cout << "¡¡¡Error de límites!!!\n";
    exit(1);
  }
  return p[i*jsize + j]; // devuelve referencia a p[i]
}

// Obtener algo del array.
int array::get(int i, int j)
{
  if(i<0 || i>=isize || j<0 || j>=jsize) {
    cout << "¡¡¡Error de límites!!!\n";
    exit(1);
  }
  return p[i*jsize +j]; // devuelve carácter
}

main()
{
  array a(2, 3);
  int i, j;

  for(i=0; i<2; i++)
    for(j=0; j<3; j++)
      a.put(i, j) = i+j;

  for(i=0; i<2; i++)
    for(j=0; j<3; j++)
      cout << a.get(i, j) << ' ';

  // fuera de límites
  a.put(10, 10);

  return 0;
}
```

2. No. Una referencia devuelta por una función no se puede asignar a un puntero.

4. COMPROBACION DE APTITUD SUPERIOR

1.
```
#include <iostream.h>

class a_type {
  double a, b;
public:
  a_type(double x, double y) {
    a = x;
    b = y;
  }
  void show() { cout << a << ' ' << b << '\n'; }
};

main()
{
  a_type ob[2][5] = {
    a_type(1, 1), a_type(2, 2),
    a_type(3, 3), a_type(4, 4),
    a_type(5, 5), a_type(6, 6),
    a_type(7, 7), a_type(8, 8),
    a_type(9, 9), a_type(10, 10)
  };

  int i, j;

  for(i=0; i<2; i++)
    for(j=0; j<5; j++)
      ob[i][j].show();

  cout << '\n';

  return 0;
}
```

2.
```
#include <iostream.h>

class a_type {
  double a, b;
public:
  a_type(double x, double y) {
    a = x;
    b = y;
  }
  void show() { cout << a << ' ' << b << '\n'; }
};

main()
```

```
{
  a_type ob[2][5] = {
    a_type(1, 1), a_type(2, 2),
    a_type(3, 3), a_type(4, 4),
    a_type(5, 5), a_type(6, 6),
    a_type(7, 7), a_type(8, 8),
    a_type(9, 9), a_type(10, 10)
  };

  a_type *p;

  p = (a_type *) ob;

  int i, j;

  for(i=0; i<2; i++)
    for(j=0; j<5; j++) {
      p->show();
      p++;
    }

  cout << '\n';

  return 0;
}
```

3. El puntero **this** es un puntero que se pasa automáticamente a una función miembro que apunta al objeto que generó la llamada.
4. Las formas generales de **new** y **delete** son

   p-var = new *tipo;*
   delete *p-var;*

   Cuando se utiliza **new** no es necesario utilizar un molde de tipo. El tamaño del objeto se determina automáticamente; no hace falta utilizar **sizeof**. Además, no es necesario incluir **malloc.h** en el programa.
5. Una referencia es básicamente una constante puntero implícita que en realidad es un nombre diferente para otra variable o argumento. Una ventaja de utilizar un parámetro por referencia es que no se realiza copia del argumento.

6.
```
#include <iostream.h>

void recip(double &d);

main()
{
  double x = 100.0;

  cout << "x es " << x << '\n';

  recip(x);
```

```
  cout << "El recíproco es " << x << '\n';

  return 0;
}

void recip(double &d)
{
  d = 1/d;
}
```

4. COMPROBACION DE APTITUD INTEGRADA

1. Cuando se accede al miembro de un objecto mediante un puntero, se utiliza el operador flecha (→).

2.
```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>

class strtype {
  char *p;
  int len;
public:
  strtype(char *ptr);
  ~strtype();
  void show();
};

strtype::strtype(char *ptr)
{
  len = strlen(ptr);
  p = new char [len+1];
  if(!p) {
    cout << "Error de asignación\n";
    exit(1);
  }
  strcpy(p, ptr);
}

strtype::~strtype()
{
  cout << "Liberando p\n";
  delete [] p;
}

void strtype::show()
{
  cout << p << " - longitud: " << len;
  cout << "\n";
}

main()
```

```
{
  strtype s1("Esto es una prueba"), s2("Me gusta C++");

  s1.show();
  s2.show();

  return 0;
}
```

5. COMPROBACION DE APTITUD

1. Una referencia es un tipo especial de puntero al que se deja de referenciar automáticamente y que se puede usar intercambiándolo con el objeto al que está apuntando.

2.
```
#include <iostream.h>

main()
{
  float *f;
  int *i;

  f = new float;
  i = new int;

  if(!f || !i) {
    cout << "Error de asignación\n";
    return 1;
  }

  *f = 10.101;
  *i = 100;

  cout << *f << ' ' << *i << '\n';

  delete f;
  delete i;

  return 0;
}
```

3. Aquí se muestra la forma general de **new** que incluye un inicializador:

   p-var = new *tipo* (*inicializador*);

   Por ejemplo, esto asigna espacio para un entero y le da el valor 10:

```
int *p;

p = new int (10);
```

4.
```
#include <iostream.h>

class samp {
  int x;
public:
  samp(int n) { x = n; }
  int getx() { return x; }
};

main()
{
  samp A[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
  int i;

  for(i=0; i<10; i++) cout << A[i].getx() << ' ';

  cout << "\n";

  return 0;
}
```

5. Ventajas: Un parámetro por referencia no hace una copia del objeto utilizado en la llamada a realizar. La referencia es a menudo más rápida de pasar que un valor. El parámetro por referencia hace más eficiente la sintaxis y procedimiento de llamada por referencia, reduciendo las posibilidades de error.

   Desventajas: Los cambios en un parámetro por referencia modifican la variable utilizada en la llamada. Un parámetro por referencia abre la posibilidad de efectos laterales en la rutina llamante.

6. No.

7.
```
#include <iostream.h>

void mag(long &num, long order);

main()
{
  long n = 4;
  long o = 2;

  cout << "4 elevado a la 2ª magnitud es ";
  mag(n, o);
  cout << n << '\n';

  return 0;
}

void mag(long &num, long order)
{
  for( ; order; order--) num = num * 10;
}
```

5.1. *EJERCICIOS*

1.
```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>

class strtype {
  char *p;
  int len;
public:
  strtype();
  strtype(char *s, int l);
  char *getstring() { return p; }
  int getlength() { return len; }
};

strtype::strtype()
{
  p = new char [255];
  if(!p) {
    cout << "Error de asignación\n";
    exit(1);
  }
  *p = '\0'; // cadena nula
  len = 255;
}

strtype::strtype(char *s, int l)
{
  if(strlen(s) >= l) {
    cout << "¡Asignando memoria insuficiente!\n";
    exit(1);
  }

  p = new char [l];
  if(!p) {
    cout << "Error de asignación\n";
    exit(1);
  }
  strcpy(p, s);
  len = l;
}


main()
{
  strtype s1;
  strtype s2("Esto es una prueba", 100);

  cout << "s1: " << s1.getstring() << " - Longitud: ";
  cout << s1.getlength() << '\n';

  cout << "s2: " << s2.getstring() << " - Longitud: ";
```

```
  cout << s2.getlength() << '\n';

  return 0;
}
```

2.
```
// Emulador de un cronógrafo
#include <iostream.h>
#include <time.h>

class stopwatch {
  double begin, end;
public:
  stopwatch();
  stopwatch(clock_t t);
  ~stopwatch();
  void start();
  void stop();
  void show();
};

stopwatch::stopwatch()
{
  begin = end = 0.0;
}

stopwatch::stopwatch(clock_t t)
{
  begin = (double) t / CLK_TCK;
  end = 0.0;
}

stopwatch::~stopwatch()
{
  cout << "Objeto stopwatch destruyéndose...";
  show();
}

void stopwatch::start()
{
  begin = (double) clock() / CLK_TCK;
}

void stopwatch::stop()
{
  end = (double) clock() / CLK_TCK;
}

void stopwatch::show()
{
  cout << "Tiempo transcurrido: " << end - begin;
  cout << "\n";
}

main()
```

```
{
  stopwatch watch;
  long i;

  watch.start();
  for(i=0; i<320000; i++) ; // tiempo que dura el bucle
  watch.stop();
  watch.show();

  // crear objecto utilizando valor inicial
  stopwatch s2(clock());
  for(i=0; i<250000; i++) ; // tiempo que dura el bucle
  s2.stop();
  s2.show();

  return 0;
}
```

5.4. *EJERCICIOS*

1. Los objetos **obj** y **temp** se construyen normalmente. Sin embargo, cuando **temp** vuelve de **f( )**, se hace un objeto temporal que genera la llamada al constructor de la copia.

2. Tal y como está escrito el programa, cuando se pasa un objeto a **getval( )**, se hace una copia a nivel de bits. Cuando se vuelve de **getval( )** y se destruye esa copia, se libera la memoria asignada a ese objeto (al que apunta **p**). Sin embargo, esta es la misma memoria que todavía requiere el objeto utilizado en la llamada a **getval( )**. Aquí se muestra la versión correcta del programa. Utilice un constructor de copia para evitar este problema.

```
// Este programa ahora está corregido.
#include <iostream.h>
#include <stdlib.h>

class myclass {
  int *p;
public:
  myclass(int i);
  myclass(const myclass &o); // constructor de copia
  ~myclass() { delete p; }
  friend int getval(myclass o);
};

myclass::myclass(int i)
{
  p = new int;

  if(!p) {
    cout << "Error de asignación\n";
```

```
    exit(1);
  }
  *p = i;
}

// Constructor de copia
myclass::myclass(const myclass &o)
{
  p = new int; // asignar la memoria de la copia

  if(!p) {
    cout << "Error de asignación\n";
    exit(1);
  }
  *p = *o.p;
}

int getval(myclass o)
{
  return *o.p; // obtención del valor
}

main()
{
  myclass a(1), b(2);

  cout << getval(a) << " " << getval(b);
  cout << "\n";
  cout << getval(a) << " " << getval(b);

  return 0;
}
```

3. Un constructor de copias se llama cuando se utiliza un objeto para inicializar otro. Un constructor normal se llama cuando se crea un objeto.

5.4. *EJERCICIOS*

1.
```
#include <iostream.h>
#include <stdlib.h>

long mystrtol(const char *s, char **end, int base = 10)
{
  return strtol(s, end, base);
}

main()
{
  long x;
  char *s1 = "100234";
  char *p;
```

```
  x = mystrtol(s1, &p, 16);
  cout << "Base 16: " << x << '\n';

  x = mystrtol(s1, &p, 10);
  cout << "Base 10: " << x << '\n';

  x = mystrtol(s1, &p); // utiliza base 10 implícita
  cout << "Base 10 por omisión: " << x << '\n';

  return 0;
}
```

2. Todos los parámetros que toman argumentos implícitos tienen que aparecer a la derecha de los que no toman argumentos implícitos. Es decir, cuando se empieza a pasar parámetros implícitos todos los parámetros posteriores también tienen que ser implícitos. en la pregunta, **q** no se pasa como implícito.

3.
```
// Nota: Este programa es específico de Turbo/Borland C++.
#include <iostream.h>
#include <conio.h>

void myclreol(int len = -1);

main()
{
  int i;

  gotoxy(1, 1);
  for(i=0; i<24; i++)
    cout << "abcdefghijklmnopqrstuvwxyz1234567890\n";

  gotoxy(1, 2);
  myclreol();
  gotoxy(1, 4);
  myclreol(20);

  return 0;
}
/* Borra hasta el final de la línea a no ser que se
   especifique el parametro len */.
void myclreol(int len)
{
  int x, y;

  x = wherex(); // obtener la posición x
  y = wherey(); // obtener la posición y

  if(len == -1) len = 80-x;

  int i = x;

  for( ; i<=len; i++) cout << ' ';

  gotoxy(x, y); // inicializa el cursor
}
```

4. Un argumento implícito no puede ser otro parámetro o una variable local.

5.6. EJERCICIO

1.
```
#include <iostream.h>

int dif(int a, int b)
{
  return a-b;
}

float dif(float a, float b)
{
  return a-b;
}

main()
{
  int (*p1)(int, int);
  float (*p2)(float, float);

  p1 = dif; // dirección de dif(int, int)
  p2 = dif; // dirección de dif(float, float);

  cout << p1(10, 5) << ' ';
  cout << p2(10.5, 8.9) << '\n';

  return 0;
}
```

5. COMPROBACION DE APTITUD SUPERIOR

1.
```
// Sobrecarga date() para time_t.
#include <iostream.h>
#include <stdio.h> // incluído para sscanf()
#include <time.h>

class date {
  int day, month, year;
public:
  date(char *str);
  date(int m, int d, int y) {
    day = d;
    month = m;
    year = y;
  }
  // sobrecarga para el parámetro de tipo time_t
  date(time_t t);
  void show() {
    cout << month << '/' << day << '/';
    cout << year << '\n';
  }
};
```

```
date::date(char *str)
{
  sscanf(str, "%d%*c%d%*c%d", &month, &day, &year);
}

date::date(time_t t)
{
  struct tm *p;

  p = localtime(&t); // convertir a hora desglosada
  day = p->tm_mday;
  month = p->tm_mon;
  year = p->tm_year;
}

main()
{
  // construye el objeto date utilizando una cadena
  date sdate("11/1/95");

  // construye el objeto date utilizando enteros
  date idate(11, 1, 95);

  /* construye el objeto date utilizando time_t - esto crea
     un objeto utilizando la fecha del sistema */
  date tdate(time(NULL));

  sdate.show();
  idate.show();
  tdate.show();

  return 0;
}
```

2. La clase **samp** sólo define un constructor, y este constructor requiere un inicializador. Por lo tanto, es inadecuado declarar un objeto de tipo **samp** sin un inicializador. (Es decir, **samp x** es una declaración no válida.)
3. Una razón para sobrecargar un constructor es para proporcionar flexibilidad, permitiendo elegir el constructor más apropiado en la instancia específica. Otra razón es permitir declarar objetos tanto inicializados como no inicializados. Puede que quiera sobrecargar un constructor para poder asignar espacio a arrays dinámicos.
4. La forma general de un constructor de copia es la siguiente:

```
nombreclase (const nombreclase &obj) {
 // cuerpo del constructor
}
```

5. Un constructor de copia se llama cuando se realiza una inicialización. Concretamente, cuando se utiliza explícitamente un objeto para inicializar otro, cuando un

objeto se pasa como parámetro a una función y cuando se crea un objeto temporal cuando una función devuelve un objeto.

6. La palabra clave **overload** le indica al compilador que se va a sobrecargar una función. Esto es anacrónico y no se requiere cuando se sobrecargan funciones.
7. Un argumento implícito es un valor que se le da a un parámetro de función cuando no aparece argumento correspondiente al llamar a la función.

8.
```
#include <iostream.h>
#include <string.h>

void reverse(char *str, int count = 0);

main()
{
  char *s1 = "Esto es una prueba";
  char *s2 = "Me gusta C++";

  reverse(s1); // invierte toda la cadena
  reverse(s2, 7); // invierte los 7 primeros caracteres

  cout << s1 << '\n';
  cout << s2 << '\n';

  return 0;
}

void reverse(char *str, int count)
{
  int i, j;
  char temp;

  if(!count) count = strlen(str)-1;

  for(i=0, j=count; i<j; i++, j--) {
    temp = str[i];
    str[i] = str[j];
    str[j] = temp;
  }
}
```

9. Todos los parámetros que reciben argumentos implícitos tienen que aparecer a la derecha de los que no los reciben.
10. La ambigüedad puede aparecer por las conversiones de un tipo implícito, parámetros por referencia y argumentos implícitos.
11. Es ambiguo porque el compilador no puede saber a qué versión de **compute( )** llamar. ¿Es la primera versión con el parámetro implícito **divisor**? ¿O es la segunda versión, que sólo toma un parámetro?
12. Cuando se obtiene la dirección de una función sobrecargada, lo que determina qué función se utiliza es la especificación de tipo del puntero.

5. COMPROBACION DE APTITUD INTEGRADA

1.
```
#include <iostream.h>

void order(int &a, int &b)
{
  int t;

  if(a<b) return;
  else { // intercambia a y b
    t = a;
    a = b;
    b = t;
  }
}

main()
{
  int x=10, y=5;

  cout << "x: " << x << ", y: " << y << "\n";

  order(x, y);
  cout << "x: " << x << ", y: " << y << "\n";

  return 0;
}
```

2. La sintaxis para llamar a una función que toma un parámetro por referencia es idéntica a la de un parámetro por valor.
3. Un argumento implícito es básicamente un método abreviado de sobrecargar una función porque el resultado final es el mismo. Por ejemplo,

```
int f(int a, int b = 0);
```

es funcionalmente equivalente a estas dos funciones sobrecargadas:

```
int f(int a);

int f(int a, int b);
```

4.
```
#include <iostream.h>
class samp {
  int a;
public:
  samp() { a= 0; }
  samp(int n) { a = n; }
  int get_a() { return a; }
};

main()
```

```
{
  samp ob(88);
  samp obarray[10];

  // ...
}
```

5. Los constructores de copia son necesarios cuando el programador tiene que controlar de forma precisa cómo se realiza la copia de un objeto. Esto sólo es importante cuando la copia a nivel de bits implícita crea efectos laterales no deseados.

6. COMPROBACION DE APTITUD

1.
```
class myclass {
  int x, y;
public:
  myclass(int i, int j) { x=i; y=j; }
  myclass() { x=0; y=0; }
};
```

2.
```
class myclass {
  int x, y;
public:
  myclass(int i=0, int j=0) { x=i; y=j; }
};
```

3. Una vez que se empieza con argumentos implícitos, no puede existir un parámetro no implícito.
4. Una función no se puede sobrecargar cuando la única diferencia es que una toma un parámetro por valor y la otra toma un parámetro por referencia. (El compilador no puede separarlas.)
5. Es apropiado utilizar argumentos implícitos cuando hay dos o más valores que aparecerán con frecuencia. Es inapropiado cuando no hay ningún valor o valores que tengan probabilidad de aparecer.
6. No, porque no hay forma de inicializar un array dinámico. Esta clase sólo tiene un constructor y requiere inicializadores.
7. Un constructor de copia es un constructor especial que se llama cuando un objeto inicializa a otro. Esta circunstancia ocurre en los tres casos siguientes: Cuando un objeto se utiliza explícitamente para inicializar otro, cuando se pasa un objeto a una función o cuando se crea un objeto temporal como valor de retorno de la función.

6.2. EJERCICIOS

1.
```
// Sobrecarga el * y / respecto a la clase coord.
#include <iostream.h>
```

```
class coord {
  int x, y; // valores de coordenadas
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  coord operator*(coord ob2);
  coord operator/(coord ob2);
};

// Sobrecarga * respecto a la clase coord.
coord coord::operator*(coord ob2)
{
  coord temp;

  temp.x = x * ob2.x;
  temp.y = y * ob2.y;

  return temp;
}

// Sobrecarga / respecto a la clase coord.
coord coord::operator/(coord ob2)
{
  coord temp;

  temp.x = x / ob2.x;
  temp.y = y / ob2.y;

  return temp;
}

main()
{
  coord o1(10, 10), o2(5, 3), o3;
  int x, y;

  o3 = o1 * o2;
  o3.get_xy(x, y);
  cout << "(o1*o2) X: " << x << ", Y: " << y << "\n";

  o3 = o1 / o2;
  o3.get_xy(x, y);
  cout << "(o1/o2) X: " << x << ", Y: " << y << "\n";

  return 0;
}
```

2. La sobrecarga del operador % es inapropiada porque su funcionamiento no está relacionado con el uso tradicional.

6.3. EJERCICIO

1.
```
// Sobrecarga el < y > respecto a la clase coord.
#include <iostream.h>

class coord {
  int x, y; // valores de coordenadas
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  int operator<(coord ob2);
  int operator>(coord ob2);
};

// Sobrecarga el operador < para coord.
int coord::operator<(coord ob2)
{
  if(x<ob2.x && y<ob2.y) return 1;
  else return 0;
}

// Sobrecarga el operador > para coord.
int coord::operator>(coord ob2)
{
  return ((x > ob2.x) && (y > ob2.y));
}

main()
{
  coord o1(10, 10), o2(5, 3);

  if(o1>o2) cout << "o1 > o2\n";
  else cout << "o1 <= o2 \n";

  if(o1<o2) cout << "o1 < o2\n";
  else cout << "o1 >= o2\n";

  return 0;
}
```

6.4. EJERCICIOS

1.
```
// Sobrecarga el -- respecto a la clase coord.
#include <iostream.h>

class coord {
  int x, y; // valores de coordenadas
public:
  coord() { x=0; y=0; }
```

```
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  coord operator--(); // prefijo
  coord operator--(int notused); // postfijo
};

// Sobrecarga el prefijo -- para la clase coord.
coord coord::operator--()
{
  x--;
  y--;
  return *this;
}

// Sobrecarga el postfijo -- para la clase coord.
coord coord::operator--(int notused)
{
  x--;
  y--;
  return *this;
}

main()
{
  coord o1(10, 10);
  int x, y;

  o1--; // decrementa un objeto
  o1.get_xy(x, y);
  cout << "(o1--) X: " << x << ", Y: " << y << "\n";

  --o1; // decrementa un objeto
  o1.get_xy(x, y);
  cout << "(--o1) X: " << x << ", Y: " << y << "\n";
  return 0;
}
```

2.
```
// Sobrecarga el + respecto a la clase coord.
#include <iostream.h>

class coord {
  int x, y; // valores de coordenadas
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  coord operator+(coord ob2); // mas binario
  coord operator+(); // mas unario
};

// Sobrecarga el + respecto a la clase coord.
coord coord::operator+(coord ob2)
{
  coord temp;
```

```
  temp.x = x + ob2.x;
  temp.y = y + ob2.y;

  return temp;
}

// Sobrecarga el + unario para la clase coord.
coord coord::operator+()
{
  if(x<0) x = -x;
  if(y<0) y = -y;

  return *this;
}

main()
{
  coord o1(10, 10), o2(-2, -2);
  int x, y;
  o1 = o1 + o2; // suma
  o1.get_xy(x, y);
  cout << "(o1+o2) X: " << x << ", Y: " << y << "\n";

  o2 = +o2; // valor absoluto
  o2.get_xy(x, y);
  cout << "(+o2) X: " << x << ", Y: " << y << "\n";

  return 0;
}
```

6.5. *EJERCICIOS*

1.
```
/* Sobrecarga - y / respecto a la clase coord
   utilizando funciones amigas. */
#include <iostream.h>

class coord {
  int x, y; // valores de coordenadas
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  friend coord operator-(coord ob1, coord ob2);
  friend coord operator/(coord ob1, coord ob2);
};

/* Sobrecarga - respecto a la clase coord utilizando funciones
   amigas. */
coord operator-(coord ob1, coord ob2)
{
  coord temp;
```

```
  temp.x = ob1.x - ob2.x;
  temp.y = ob1.y - ob2.y;

  return temp;
}

/* Sobrecarga / respecto a la clase coord utilizando funciones
   amigas. */
coord operator/(coord ob1, coord ob2)
{
  coord temp;

  temp.x = ob1.x / ob2.x;
  temp.y = ob1.y / ob2.y;

  return temp;
}

main()
{
  coord o1(10, 10), o2(5, 3), o3;
  int x, y;

  o3 = o1 - o2;
  o3.get_xy(x, y);
  cout << "(o1-o2) X: " << x << ", Y: " << y << "\n";

  o3 = o1 / o2;
  o3.get_xy(x, y);
  cout << "(o1/o2) X: " << x << ", Y: " << y << "\n";

  return 0;
}
```

2.
```
// Sobrecarga el * para ob*int y int*ob.
#include <iostream.h>

class coord {
  int x, y; // valores de coordenadas
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  friend coord operator*(coord ob1, int i);
  friend coord operator*(int i, coord ob2);
};

// Sobrecarga * de una forma.
coord operator*(coord ob1, int i)
{
  coord temp;

  temp.x = ob1.x * i;
```

```
  temp.y = ob1.y * i;

  return temp;
}

// Sobrecarga * de otra forma.
coord operator*(int i, coord ob2)
{
  coord temp;

  temp.x = ob2.x * i;
  temp.y = ob2.y * i;

  return temp;
}

main()
{
  coord o1(10, 10), o2;
  int x, y;

  o2 = o1 * 2; // ob * int
  o2.get_xy(x, y);
  cout << "(o1*2) X: " << x << ", Y: " << y << "\n";

  o2 = 3 * o1; // int * ob
  o2.get_xy(x, y);
  cout << "(3*o1) X: " << x << ", Y: " << y << "\n";

  return 0;
}
```

3. Utilizando funciones amigas, es posible tener un tipo incorporado como operando izquierdo. Cuando se utilizan funciones miembro, el operando izquierdo tiene que ser un objeto de la clase para la que está definida el operador.

4.
```
/* Sobrecarga the -- respecto a la clase coord utilizando
   una función amiga */
#include <iostream.h>

class coord {
  int x, y; // valores de coordenadas
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  friend coord operator--(coord &ob); // prefijo
  friend coord operator--(coord &ob, int notused); // postfijo
};

/* Sobrecarga -- (prefijo) para la clase coord utilizando
   una función amiga. */
coord operator--(coord &ob)
```

```
{
  ob.x--;
  ob.y--;
  return ob;
}

/* Sobrecarga -- (postfijo) para la clase coord utilizando una
función amiga. */
coord operator--(coord &ob, int notused)
{
  ob.x--;
  ob.y--;
  return ob;
}

main()
{
  coord o1(10, 10);
  int x, y;

  --o1; // decrementa o1 un objeto
  o1.get_xy(x, y);
  cout << "(--o1) X: " << x << ", Y: " << y << "\n";

  o1--; // decrementa o1 un objeto
  o1.get_xy(x, y);
  cout << "(o1--) X: " << x << ", Y: " << y << "\n";

  return 0;
}
```

6.6. *EJERCICIO*

1.
```
#include <iostream.h>
#include <stdlib.h>

class dynarray {
  int *p;
  int size;
public:
  dynarray(int s);
  int &put(int i);
  int get(int i);
  dynarray &operator=(dynarray &ob);
};

// Constructor
dynarray::dynarray(int s)
{
  p = new int [s];
  if(!p) {
    cout << "Error de asignación\n";
    exit(1);
  }
```

```
  size = s;
}

// Almacena un elemento.
int &dynarray::put(int i)
{
  if(i<0 || i>=size) {
    cout << "¡Error de límites!\n";
    exit(1);
  }

  return p[i];
}

// Obtiene un elemento.
int dynarray::get(int i)
{
  if(i<0 || i>=size) {
    cout << "¡Error de límites!\n";
    exit(1);
  }

  return p[i];
}

// Sobrecarga = para dynarray.
dynarray &dynarray::operator=(dynarray &ob)
{
  int i;

  if(size!=ob.size) {
    cout << "¡No se pueden copiar arrays de tamaños
    diferentes!\n";
    exit(1);
  }

  for(i = 0; i<size; i++) p[i] = ob.p[i];
  return *this;
}

main()
{
  int i;

  dynarray ob1(10), ob2(10), ob3(100);

  ob1.put(3) = 10;
  i = ob1.get(3);
  cout << i << "\n";

  ob2 = ob1;

  i = ob2.get(3);
```

```
  cout << i << "\n";

  // produce un error
  ob1 = ob3; // !!!
  return 0;
}
```

6. COMPROBACION DE APTITUD SUPERIOR

1.
```
// Sobrecarga << y >>.
#include <iostream.h>

class coord {
  int x, y; // valores de coordenadas
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  coord operator<<(int i);
  coord operator>>(int i);
};

// Sobrecarga <<.
coord coord::operator<<(int i)
{
  coord temp;

  temp.x = x << i;
  temp.y = y << i;

  return temp;
}

// Sobrecarga >>.
coord coord::operator>>(int i)
{
  coord temp;
  temp.x = x >> i;
  temp.y = y >> i;

  return temp;
}

main()
{
  coord o1(4, 4), o2;
  int x, y;

  o2 = o1 << 2; // ob << int
  o2.get_xy(x, y);
  cout << "(o1<<2) X: " << x << ", Y: " << y << "\n";
```

```
  o2 = o1 >> 2; // ob >> int
  o2.get_xy(x, y);
  cout << "(o1>>2) X: " << x << ", Y: " << y << "\n";

  return 0;
}
```

2.
```
#include <iostream.h>

class three_d {
  int x, y, z;
public:
  three_d(int i, int j, int k)
  {
  x = i; y = j; z = k;
  }
  three_d() { x=0; y=0; z=0; }
  void get(int &i, int &j, int &k)
  {
  i = x; j = y; k = z;
  }
  three_d operator+(three_d ob2);
  three_d operator-(three_d ob2);
  three_d operator++();
  three_d operator--();
  };

three_d three_d::operator+(three_d ob2)
{
  three_d temp;
  temp.x = x + ob2.x;
  temp.y = y + ob2.y;
  temp.z = z + ob2.z;

  return temp;
}

three_d three_d::operator-(three_d ob2)
{
  three_d temp;

  temp.x = x - ob2.x;
  temp.y = y - ob2.y;
  temp.z = z - ob2.z;

  return temp;
}

three_d three_d::operator++()
{
  x++;
  y++;
  z++;

  return *this;
}
```

```
three_d three_d::operator--()
{
  x--;
  y--;
  z--;

  return *this;
}

main()
{
  three_d o1(10, 10, 10), o2(2, 3, 4), o3;
  int x, y, z;

  o3 = o1 + o2;
  o3.get(x, y, z);
  cout << "X: " << x << ", Y: " << y;
  cout << ", Z: " << z << "\n";

  o3 = o1 - o2;
  o3.get(x, y, z);
  cout << "X: " << x << ", Y: " << y;
  cout << ", Z: " << z << "\n";

  ++o1;
  o1.get(x, y, z);
  cout << "X: " << x << ", Y: " << y;
  cout << ", Z: " << z << "\n";

  --o1;
  o1.get(x, y, z);
  cout << "X: " << x << ", Y: " << y;
  cout << ", Z: " << z << "\n";

  return 0;
}
```

3.
```
#include <iostream.h>

class three_d {
  int x, y, z;
public:
  three_d(int i, int j, int k)
  {
    x = i; y = j; z = k;
  }
  three_d() { x=0; y=0; z=0; }
  void get(int &i, int &j, int &k)
  {
    i = x; j = y; k = z;
  }
  three_d operator+(three_d &ob2);
  three_d operator-(three_d &ob2);
  friend three_d operator++(three_d &ob);
  friend three_d operator--(three_d &ob);
  };
```

```
three_d three_d::operator+(three_d &ob2)
{
  three_d temp;

  temp.x = x + ob2.x;
  temp.y = y + ob2.y;
  temp.z = z + ob2.z;

  return temp;
}

three_d three_d::operator-(three_d &ob2)
{
  three_d temp;

  temp.x = x - ob2.x;
  temp.y = y - ob2.y;
  temp.z = z - ob2.z;

  return temp;
}

three_d operator++(three_d &ob)
{
  ob.x++;
  ob.y++;
  ob.z++;

  return ob;
}

three_d operator--(three_d &ob)
{
  ob.x--;
  ob.y--;
  ob.z--;

  return ob;
}

main()
{
  three_d o1(10, 10, 10), o2(2, 3, 4), o3;
  int x, y, z;

  o3 = o1 + o2;
  o3.get(x, y, z);
  cout << "X: " << x << ", Y: " << y;
  cout << ", Z: " << z << "\n";

  o3 = o1 - o2;
  o3.get(x, y, z);
  cout << "X: " << x << ", Y: " << y;
```

```
  cout << ", Z: " << z << "\n";

  ++o1;
  o1.get(x, y, z);
  cout << "X: " << x << ", Y: " << y;
  cout << ", Z: " << z << "\n";

  --o1;
  o1.get(x, y, z);
  cout << "X: " << x << ", Y: " << y;
  cout << ", Z: " << z << "\n";

  return 0;
}
```

4. Para operadores binarios, a una función operador miembro se le pasa el operando izquierdo implícitamente utilizando **this**. A una función operador amiga binaria se le pasan ambos operandos explícitamente. Las funciones operador unario miembro no tienen parámetros explícitos. Una función operador unario amiga tiene un parámetro.
5. Tendrá que sobrecargar el operator = cuando la copia a nivel de bits implícita sea insuficiente. Por ejemplo, puede tener objetos en los que sólo quiera asignar partes de datos de un objeto a otras.
6. No.

7.
```
#include <iostream.h>

class three_d {
  int x, y, z;
public:
  three_d(int i, int j, int k)
  {
  x = i; y = j; z = k;
  }
  three_d() { x=0; y=0; z=0; }
  void get(int &i, int &j, int &k)
  {
  i = x; j = y; k = z;
  }
  friend three_d operator+(three_d ob, int i);
  friend three_d operator+(int i, three_d ob);
  };

three_d operator+(three_d ob, int i)
{
  three_d temp;
  temp.x = ob.x + i;
  temp.y = ob.y + i;
  temp.z = ob.z + i;

  return temp;
}

three_d operator+(int i, three_d ob)
```

```
{
  three_d temp;

  temp.x = ob.x + i;
  temp.y = ob.y + i;
  temp.z = ob.z + i;

  return temp;
}

main()
{
  three_d o1(10, 10, 10);
  int x, y, z;

  o1 = o1 + 10;
  o1.get(x, y, z);
  cout << "X: " << x << ", Y: " << y;
  cout << ", Z: " << z << "\n";

  o1 = -20 + o1;
  o1.get(x, y, z);
  cout << "X: " << x << ", Y: " << y;
  cout << ", Z: " << z << "\n";

  return 0;
}
```

8.

```
#include <iostream.h>

class three_d {
  int x, y, z;
public:
  three_d(int i, int j, int k)
  {
    x = i; y = j; z = k;
  }
  three_d() { x=0; y=0; z=0; }
  void get(int &i, int &j, int &k)
  {
    i = x; j = y; k = z;
  }
  int operator==(three_d ob2);
  int operator!=(three_d ob2);
  int operator||(three_d ob2);
  };

int three_d::operator==(three_d ob2)
{
  return (x==ob2.x && y==ob2.y && z==ob2.z);
}

int three_d::operator!=(three_d ob2)
```

```
{
  return (x!=ob2.x && y!=ob2.y && z!=ob2.z);
}

int three_d::operator||(three_d ob2)
{
  return (x||ob2.x && y||ob2.y && z||ob2.z);
}

main()
{
  three_d o1(10, 10, 10), o2(2, 3, 4), o3(0, 0, 0);

  if(o1==o1) cout << "o1 == o1\n";

  if(o1!=o2) cout << "o1 != o2\n";

  if(o3 || o1) cout << "o1 o o3 es verdadero\n";

  return 0;
}
```

6. *COMPROBACION DE APTITUD INTEGRADA*

1.
```
/* No se ha utilizado comprobación de errores por claridad.
   Sin embargo debería añadir algo de esto si utiliza este
   código para una aplicación real.
*/
#include <iostream.h>
#include <string.h>

class strtype {
  char s[80];
public:
  strtype() { *s = '\0'; }
  strtype(char *p) { strcpy(s, p); }
  char *get() { return s; }
  strtype operator+(strtype s2);
  strtype operator=(strtype s2);
  int operator<(strtype s2);
  int operator>(strtype s2);
  int operator==(strtype s2);
};

strtype strtype::operator+(strtype s2)
{
  strcat(s, s2.s);

  return *this;
}

strtype strtype::operator=(strtype s2)
```

```
{
  strcpy(s, s2.s);

  return *this;
}

int strtype::operator<(strtype s2)
{
  return strcmp(s, s2.s) < 0;
}

int strtype::operator>(strtype s2)
{
  return strcmp(s, s2.s) > 0;
}

int strtype::operator==(strtype s2)
{
  return strcmp(s, s2.s) == 0;
}

main()
{
  strtype o1("Hola"), o2(" Allí"), o3;

  o3 = o1 + o2;
  cout << o3.get() << "\n";

  o3 = o1;
  if(o1==o3) cout << "o1 igual que o3\n";

  if(o1>o2) cout << "o1 > o2\n";

  if(o1<o2) cout << "o1 < o2\n";

  return 0;
}
```

7. COMPROBACION DE APTITUD

1. No. La sobrecarga de un operador simplemente amplía los tipos de datos con los que puede operar, pero no afecta a las operaciones ya existentes.
2. Sí. No se puede sobrecargar un operador respecto a uno de los tipos incorporados de C++.
3. No, la precedencia no se puede cambiar. No, el número de operandos no se puede modificar.

4.
```
#include <iostream.h>

class array {
  int nums[10];
```

```
public:
  array();
  void set(int n[10]);
  void show();
  array operator+(array ob2);
  array operator-(array ob2);
  int operator==(array ob2);
};

array::array()
{
  int i;
  for(i=0; i<10; i++) nums[i] = 0;
}

void array::set(int *n)
{
  int i;

  for(i=0; i<10; i++) nums[i] = n[i];
}

void array::show()
{
  int i;

  for(i=0; i<10; i++)
    cout << nums[i] << ' ';

  cout << "\n";
}

array array::operator+(array ob2)
{
  int i;
  array temp;

  for(i=0; i<10; i++)
    temp.nums[i] = nums[i] + ob2.nums[i];

  return temp;
}

array array::operator-(array ob2)
{
  int i;
  array temp;

  for(i=0; i<10; i++)
    temp.nums[i] = nums[i] - ob2.nums[i];

  return temp;
}
```

```
int array::operator==(array ob2)
{
  int i;

  for(i=0; i<10; i++)
    if(nums[i]!=ob2.nums[i]) return 0;

  return 1;
}

main()
{
  array o1, o2, o3;

  int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

  o1.set(i);
  o2.set(i);

  o3 = o1 + o2;
  o3.show();

  o3 = o1 - o3;
  o3.show();

  if(o1==o2) cout << "o1 igual a o2\n";
  else cout << "o1 no es igual a o2\n";

  if(o1==o3) cout << "o1 igual a o3\n";
  else cout << "o1 no es igual a o3\n";

  return 0;
}
```

5.
```
#include <iostream.h>

class array {
  int nums[10];
public:
  array();
  void set(int n[10]);
  void show();
  friend array operator+(array ob1, array ob2);
  friend array operator-(array ob1, array ob2);
  friend int operator==(array ob1, array ob2);
};

array::array()
{
  int i;
  for(i=0; i<10; i++) nums[i] = 0;
}

void array::set(int *n)
```

```
{
  int i;

  for(i=0; i<10; i++) nums[i] = n[i];
}

void array::show()
{
  int i;

  for(i=0; i<10; i++)
    cout << nums[i] << ' ';

  cout << "\n";
}

array operator+(array ob1, array ob2)
{
  int i;
  array temp;

  for(i=0; i<10; i++)
    temp.nums[i] = ob1.nums[i] + ob2.nums[i];

  return temp;
}

array operator-(array ob1, array ob2)
{
  int i;
  array temp;

  for(i=0; i<10; i++)
    temp.nums[i] = ob1.nums[i] - ob2.nums[i];

  return temp;
}

int operator==(array ob1, array ob2)
{
  int i;

  for(i=0; i<10; i++)
    if(ob1.nums[i]!=ob2.nums[i]) return 0;

  return 1;
}

main()
{
  array o1, o2, o3;

  int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

```
  o1.set(i);
  o2.set(i);

  o3 = o1 + o2;
  o3.show();

  o3 = o1 - o3;
  o3.show();

  if(o1==o2) cout << "o1 igual a o2\n";
  else cout << "o1 no es igual a o2\n";

  if(o1==o3) cout << "o1 igual a o3\n";
  else cout << "o1 no es igual a o3\n";

  return 0;
}
```

6.

```
#include <iostream.h>

class array {
  int nums[10];
public:
  array();
  void set(int n[10]);
  void show();
  array operator++();
  friend array operator--(array &ob);
};

array::array()
{
  int i;

  for(i=0; i<10; i++) nums[i] = 0;
}

void array::set(int *n)
{
  int i;

  for(i=0; i<10; i++) nums[i] = n[i];
}

void array::show()
{
  int i;

  for(i=0; i<10; i++)
  cout << nums[i] << ' ';

  cout << "\n";
}
```

```
// Sobrecarga el op unario utilizando función miembro.
array array::operator++()
{
  int i;

  for(i=0; i<10; i++)
    nums[i]++;

  return *this;
}

// Utiliza una función amiga.
array operator--(array &ob)
{
  int i;

  for(i=0; i<10; i++)
    ob.nums[i]--;

  return ob;
}

main()
{
  array o1, o2, o3;

  int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

  o1.set(i);
  o2.set(i);

  o3 = ++o1;
  o3.show();

  o3 = --o1;
  o3.show();

  return 0;
}
```

7. No. El operador de asignación se tiene que sobrecargar utilizando una función miembro.

7.1. *EJERCICIOS*

1. Las sentencias correctas son la A y la C.
2. Un miembro público de una clase base se convierte en miembro público de una clase derivada cuando se hereda como público. Cuando un miembro público de una clase base se hereda como privado, se convierte en miembro privado de la clase derivada.

7.2. EJERCICIOS

1. Cuando un miembro protegido de una clase base se hereda como público, se convierte en miembro protegido de la clase derivada. Si se hereda como privado, se convierte en miembro privado de la clase derivada. Si se hereda como protegido, se convierte en miembro protegido de la clase derivada.
2. La categoría protegida es necesaria para permitir que una clase base mantenga ciertos miembros privados mientras sigue permitiendo que una clase derivada tenga acceso a ellos.
3. No.

7.3. EJERCICIOS

1.
```
#include <iostream.h>
#include <string.h>

class mybase {
  char str[80];
public:
  mybase(char *s) { strcpy(str, s); }
  char *get() { return str; }
};

class myderived : public mybase {
  int len;
public:
  myderived(char *s) : mybase(s) {
    len = strlen(s);
  }
  int getlen() { return len; }
  void show() { cout << get() << '\n'; }
};

main()
{
  myderived ob("hola");

  ob.show();
  cout << ob.getlen() << '\n';

  return 0;
}
```

2.
```
#include <iostream.h>

// Una clase base para distintos tipos de vehículos.
class vehicle {
  int num_wheels;
  int range;
```

```
public:
  vehicle(int w, int r)
  {
    num_wheels = w; range = r;
  }
  void showv()
  {
    cout << "Ruedas: " << num_wheels << '\n';
    cout << "Autonomía: " << range << '\n';
  }
};


class car : public vehicle {
  int passengers;
public:
  car(int p, int w, int r) : vehicle(w, r)
  {
    passengers = p;
  }
  void show()
  {
    showv();
    cout << "Pasajeros: " << passengers << '\n';
  }
};


class truck : public vehicle {
  int loadlimit;
public:
  truck(int l, int w, int r) : vehicle(w, r)
  {
    loadlimit = l;
  }
  void show()
  {
    showv();
    cout << "límite de carga " << loadlimit << '\n';
  }
};


main()
{
  car c(5, 4, 500);
  truck t(30000, 12, 1200);


  cout << "Coche: \n";
  c.show();
  cout << "\nCamión:\n";
  t.show();


  return 0;
}
```

7.4. EJERCICIOS

1.
```
Construcción de A
Construcción de B
Construcción de C
Destrucción de C
Destrucción de B
Destrucción de A
```

2.
```
#include <iostream.h>

class A {
  int i;
public:
  A(int a) { i = a; }
};

class B {
  int j;
public:
  B(int a) { j = a; }
};

class C : public A, public B {
  int k;
public:
  C(int c, int b, int a) : A(a), B(b) {
    k = c;
  }
};
```

7.5. EJERCICIOS

2. Una clase base virtual es necesaria cuando una clase derivada hereda dos (o más) clases, cada una de las cuales ha heredado otra clase base.Sin clases base virtuales, existirían dos (o más) copias de la clase base común en la clase derivada final.

7. COMPROBACION DE APTITUD SUPERIOR

1.
```
#include <iostream.h>

class building {
protected:
  int floors;
  int rooms;
  double footage;
};
```

```
class house : public building {
  int bedrooms;
  int bathrooms;
public:
  house(int f, int r, double ft, int br, int bth) {
    floors = f; rooms = r; footage = ft;
    bedrooms = br; bathrooms = bth;
  }
  void show() {
    cout << "pisos: " << floors << '\n';
    cout << "habitaciones: " << rooms << '\n';
    cout << "extensión: " << footage << '\n';
    cout << "dormitorios: " << bedrooms << '\n';
    cout << "cuartos de baño: " << bathrooms << '\n';
  }
};

class office : public building {
  int phones;
  int extinguishers;
public:
  office(int f, int r, double ft, int p, int ext) {
    floors = f; rooms = r; footage = ft;
    phones = p; extinguishers = ext;
  }
  void show() {
    cout << "pisos: " << floors << '\n';
    cout << "habitaciones: " << rooms << '\n';
    cout << "extensión: " << footage << '\n';
    cout << "Teléfonos: " << phones << '\n';
    cout << "extintores de incendio: ";
    cout << extinguishers << '\n';
  }
};

main()
{
  house h_ob(2, 12, 5000, 6, 4);
  office o_ob(4, 25, 12000, 30, 8);

  cout << "Casa: \n";
  h_ob.show();

  cout << "\nOficina: \n";
  o_ob.show();

  return 0;
}
```

2. Cuando una clase base se hereda como pública, los miembros públicos de la clase base se convierten en miembros públicos de la clase derivada y los miembros privados de la clase base siguen siendo privados de la clase base. Si la base se hereda como privada, todos los miembros de la base se convierten en miembros privados de la clase derivada.

3. Los miembros declarados como protegidos son privados a la clase base, pero se pueden heredar (y acceder) mediante cualquier clase derivada. Cuando se utiliza como un especificador de acceso de herencia, hace que todos los miembros públicos y protegidos de la clase base se conviertan en miembros protegidos de la clase derivada.
4. Los constructores se llaman en orden de derivación. Los destructores se llaman en orden inverso.

5.
```
#include <iostream.h>

class planet {
protected:
  double distance; // millas desde el sol
  int revolve; // en días
public:
  planet(double d, int r) { distance = d; revolve = r; }
};

class earth : public planet {
  double circumference; // perímetro de la órbita
public:
  earth(double d, int r) : planet(d, r) {
    circumference = 2*distance*3.1416;
  }
  void show() {
    cout << "Distancia del sol: " << distance << '\n';
    cout << "Dias en órbita: " << revolve << '\n';
    cout << "Perímetro de la órbita: ";
    cout << circumference << '\n';
  }
};

main()
{
  earth ob(93000000, 365);

  ob.show();

  return 0;
}
```

6. Para corregir el programa, hay que hacer que **motorized** y **road_use** hereden **vehicle** como clase base virtual. Consulte también la pregunta 1 de la Comprobación de aptitud integrada de este capítulo.

7. COMPROBACION DE APTITUD INTEGRADA

1. En algunos compiladores, no se puede usar un **switch** en una función insertada. Si este es el caso de su compilador, entonces las funciones se convierten automaticamente en funciones «normales».

2. El operador de asignación es el único operador que no se hereda. La razón de esto es fácil de entender. Ya que una clase derivada contendrá miembros que no están en la clase base, el operador = sobrecargado respecto a la **class** base no tiene conocimiento de los miembros añadidos por la clase derivada y, como tal, no puede copiar de forma apropiada esos nuevos miembros.

8. COMPROBACION DE APTITUD

1.
```
#include <iostream.h>

class airship {
protected:
  int passengers;
  double cargo;
};

class airplane : public airship {
  char engine; // p para propulsión, j para jet
  double range;
public:
  airplane(int p, double c, char e, double r)
  {
    passengers = p;
    cargo = c;
    engine = e;
    range = r;
  }
  void show();
};

class balloon : public airship {
  char gas; // h para hidrógeno, e para hélio
  double altitude;
public:
  balloon(int p, double c, char g, double a)
  {
  passengers = p;
    cargo = c;
    gas = g;
    altitude = a;
  }
  void show();
};

void airplane::show()
{
  cout << "Pasajeros: " << passengers << '\n';
  cout << "Capacidad de carga: " << cargo << '\n';
  cout << "Motor: ";
  if(engine=='p') cout << "Propulsión\n";
  else cout << "Jet\n";
  cout << "Autonomía: " << range << '\n';
}
```

```
void balloon::show()
{
  cout << "Pasajeros: " << passengers << '\n';
  cout << "Capacidad de carga: " << cargo << '\n';
  cout << "Combustible: ";
  if(gas=='h') cout << "Hidrógeno\n";
  else cout << "Hélio\n";
  cout << "Altitud: " << altitude << '\n';
}

main()
{
  balloon b(2, 500.0, 'h', 12000.0);
  airplane b727(100, 40000.0, 'j', 40000.0);

  b.show();
  cout << '\n';
  b727.show();

  return 0;
}
```

2. El especificador de acceso **protected** hace que los miembros de la clase sean privados a esa clase, pero siguen siendo accesibles para cualquier clase derivada.
3. El programa muestra la siguiente salida, que indica cuándo se llama a los constructores y a los destructores.

```
Construcción de A
Construcción de B
Construcción de C
Destrucción de C
Destrucción de B
Destrucción de A
```

4. Los constructores se llaman en el orden ABC, y los destructores en el orden CBA.

5.
```
#include <iostream.h>

class base {
  int i, j;
public:
  base(int x, int y) { i = x; j = y; }
  void showij() { cout << i << ' ' << j << '\n'; }
};

class derived : public base {
  int k;
public:
  derived(int a, int b, int c) : base(b, c) {
    k = a;
  }
  void show() { cout << k << ' '; showij(); }
};

main()
```

```
{
  derived ob(1, 2, 3);

  ob.show();

  return 0;
}
```

6. Las palabras que faltan son «general» y «específica.»

8.2. EJERCICIOS

1.
```
#include <iostream.h>

main()
{
  cout.setf(ios::showpos);

  cout << -10 << ' ' << 10 << '\n';

  return 0;
}
```

2.
```
#include <iostream.h>

main()
{
  cout.setf(ios::showpoint | ios::uppercase
  | ios::scientific);

  cout << 100.0;

  return 0;
}
```

3. Esta sentencia, entre otras, inicializa los indicadores:

```
flags(0L);
```

4.
```
#include <iostream.h>

main()
{
  long f;

  f = cout.flags(); // guardar los indicadores

  cout.setf(ios::showbase | ios::hex);
  cout << 100 << '\n';

  cout.flags(f); // inicializar los indicadores

  return 0;
}
```

8.3. EJERCICIOS

1.
```
// Crea una tabla de log10 y log desde 2 hasta 100.
#include <iostream.h>
#include <math.h>

main()
{
  double x;

  cout.precision(5);
  cout << " x log x ln e\n\n";

  for(x = 2.0; x <= 100.0; x++) {
  cout.width(10);
  cout << x << " ";
  cout.width(10);
  cout << log10(x) << " ";
  cout.width(10);
  cout << log(x) << '\n';
  }

  return 0;
}
```

2.
```
#include <iostream.h>
#include <string.h>

void center(char *s);

main()
{
  center("¡Hola amigos!");
  center("Me gusta C++.");

  return 0;
}

void center(char *s)
{
  int len;

  len = 40+(strlen(s)/2);

  cout.width(len);
  cout << s << '\n';
}
```

8.4. EJERCICIOS

1a.
```
// crea una tabla de log10 y log desde 2 hasta 100.
#include <iostream.h>
#include <math.h>
#include <iomanip.h>
```

```
main()
{
  double x;

  cout << setprecision(5);
  cout << " x log x ln e\n\n";

  for(x = 2.0; x <= 100.0; x++) {
    cout << setw(10) << x << " ";
    cout << setw(10) << log10(x) << " ";
    cout << setw(10) << log(x) << '\n';
  }

  return 0;
}
```

1b.
```
#include <iostream.h>
#include <string.h>
#include <iomanip.h>

void center(char *s);

main()
{
  center("¡Hola amigos!");
  center("Me gusta C++.");

  return 0;
}

void center(char *s)
{
  int len;

  len = 40+(strlen(s)/2);

  cout << setw(len) << s << '\n';
}
```

2.
```
cout << setiosflags(ios::showbase | ios::hex) << 100;
```

8.5. *EJERCICIOS*

1.
```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>

class strtype {
  char *p;
  int len;
public:
```

```
  strtype(char *ptr);
  ~strtype();
  friend ostream &operator<<(ostream &stream, strtype &ob);
};

strtype::strtype(char *ptr)
{
  len = strlen(ptr);
  p = new char [len+1];
  if(!p) {
    cout << "Error de asignación\n";
    exit(1);
  }
  strcpy(p, ptr);
}

strtype::~strtype()
{
  delete p;
}

ostream &operator<<(ostream &stream, strtype &ob)
{
  stream << ob.p;

  return stream;
}

main()
{
  strtype s1("Esto es una prueba"), s2("Me gusta C++");

  cout << s1;
  cout << endl << s2 << endl;

  return 0;
}
```

2.
```
#include <iostream.h>

class planet {
protected:
  double distance; // millas desde el sol
  int revolve; // en dias
public:
  planet(double d, int r) { distance = d; revolve = r; }
};

class earth : public planet {
  double circumference; // perímetro de la órbita
public:
  earth(double d, int r) : planet(d, r) {
    circumference = 2*distance*3.1416;
  }
```

```
  friend ostream &operator<<(ostream &stream, earth ob);
};

ostream &operator<<(ostream &stream, earth ob)
{
  stream << "Distancia desde el sol: " << ob.distance << '\n';
  stream << "Dias en órbita: " << ob.revolve << '\n';
  stream << "Perímetro de la órbita: " << ob.circumference;
  stream << '\n';

  return stream;
}

main()
{
  earth ob(93000000, 365);

  cout << ob;

  return 0;
}
```

3. Un insertor no puede ser una función miembro porque el objeto que genera una llamada al insertor no es un objeto de una clase definida por el usuario.

8.6. *EJERCICIOS*

1.
```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>

class strtype {
  char *p;
  int len;
public:
  strtype(char *ptr);
  ~strtype();
  friend ostream &operator<<(ostream &stream, strtype &ob);
  friend istream &operator>>(istream &stream, strtype &ob);
};

strtype::strtype(char *ptr)
{
  len = strlen(ptr);
  p = new char [len+1];
  if(!p) {
    cout << "Error de asignación\n";
    exit(1);
  }
  strcpy(p, ptr);
}
```

```
strtype::~strtype()
{
  delete p;
}

ostream &operator<<(ostream &stream, strtype &ob)
{
  stream << ob.p;

  return stream;
}

istream &operator>>(istream &stream, strtype &ob)
{
  char temp[255];

  stream >> temp;

  if(strlen(temp) >= ob.len) {
    delete ob.p;
    ob.p = new char [strlen(temp)+1];
    if(!ob.p) {
      cout << "Error de asignación\n";
      exit(1);
    }
  }
  strcpy(ob.p, temp);

  return stream;
}

main()
{
  strtype s1("Esto es una prueba"), s2("Me gusta C++");

  cout << s1;
  cout << '\n' << s2;

  cout << "\nIntroduzca una cadena: ";
  cin >> s1;
  cout << s1;

  return 0;
}
```

2.
```
#include <iostream.h>

class factor {
  int num; // número
  int lfact; // menor factor
public:
  factor(int i);
```

```
  friend ostream &operator<<(ostream &stream, factor ob);
  friend istream &operator>>(istream &stream, factor &ob);
};

factor::factor(int i)
{
  int n;

  num = i;

  for(n=2; n < (i/2); n++)
    if(!(i%n)) break;

  if(n<(i/2)) lfact = n;
  else lfact = 1;
}

istream &operator>>(istream &stream, factor &ob)
{
  stream >> ob.num;

  int n;

  for(n=2; n < (ob.num/2); n++)
    if(!(ob.num%n)) break;
  if(n<(ob.num/2)) ob.lfact = n;
  else ob.lfact = 1;

  return stream;
}

ostream &operator<<(ostream &stream, factor ob)
{
  stream << ob.lfact << " es el menor factor de of ";
  stream << ob.num << '\n';

  return stream;
}

main()
{
  factor o(32);

  cout << o;

  cin >> o;
  cout << o;

  return 0;
}
```

8. COMPROBACION DE APTITUD SUPERIOR

1.
```
#include <iostream.h>

main()
{
  cout << 100 << ' ';

  cout.setf(ios::hex);
  cout << 100 << ' ';

  cout.unsetf(ios::hex); // borra el indicador hex
  cout.setf(ios::oct);
  cout << 100 << '\n';

  return 0;
}
```

2.
```
#include <iostream.h>

main()
{
  cout.setf(ios::left);
  cout.precision(2);
  cout.fill('*');
  cout.width(20);

  cout << 1000.5364 << '\n';

  return 0;
}
```

3.
```
#include <iostream.h>

main()
{
  cout << 100 << ' ';

  cout << hex << 100 << ' ';

  cout << oct << 100 << '\n';

  return 0;
}
```

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

main()
{
  cout << setiosflags(ios::left);
  cout << setprecision(2);
```

```
  cout << setfill('*');
  cout << setw(20);

  cout << 1000.5364 << '\n';

  return 0;
}
```

4.
```
long f;

f = cout.flags(); // guardar

// ...

cout.flags(f); // restaurar
```

5.
```
#include <iostream.h>

class pwr {
  int base;
  int exponent;
  double result; // base a la potencia del exponente
public:
  pwr(int b, int e);
  friend ostream &operator<<(ostream &stream, pwr ob);
  friend istream &operator>>(istream &stream, pwr &ob);
};

pwr::pwr(int b, int e)
{
  base = b;
  exponent = e;

  result = 1;
  for( ; e; e--) result = result * base;
}

ostream &operator<<(ostream &stream, pwr ob)
{
  stream << ob.base << "^" << ob.exponent;
  stream << " es " << ob.result << '\n';

  return stream;
}

istream &operator>>(istream &stream, pwr &ob)
{
  int b, e;

  cout << "Introduzca base y exponente: ";
  stream >> b >> e;

  pwr temp(b, e); // crea uno temporal
```

```
  ob = temp;

  return stream;
}
main()
{
  pwr ob(10, 2);

  cout << ob;

  cin >> ob;
  cout << ob;

  return 0;
}
```

6.
```
// Este program dibuja cuadros.
#include <iostream.h>

class box {
  int len;
public:
  box(int l) { len = l; }
  friend ostream &operator<<(ostream &stream, box ob);
};

// Dibuja un cuadro.
ostream &operator<<(ostream &stream, box ob)
{
  int i, j;

  for(i=0; i<ob.len; i++) stream << '*';
    stream << '\n';
    for(i=0; i<ob.len-2; i++) {
    stream << '*';
  for(j=0; j<ob.len-2; j++) stream << ' ';
  stream << "*\n";
  }
  for(i=0; i<ob.len; i++) stream << '*';
  stream << '\n';

  return stream;
}

main()
{
  box b1(4), b2(7);

  cout << b1 << endl << b2;

  return 0;
}
```

8. COMPROBACION DE APTITUD INTEGRADA

1.
```
#include <iostream.h>

define SIZE 10

// Declara una clase pila para caracteres
class stack {
  char stck[SIZE]; // guarda la pila
  int tos; // índice de la cabeza de la pila
public:
  stack();
  void push(char ch); // introduce carácter en la pila
  char pop(); // extrae carácter de la pila
  friend ostream &operator<<(ostream &stream, stack ob);
};

// Inicialización de la pila
stack::stack()
{
  tos = 0;
}

// Introducción de un carácter.
void stack::push(char ch)
{
  if(tos==SIZE) {
    cout << "La pila está llena";
    return;
  }
  stck[tos] = ch;
  tos++;
}

// Saca un carácter.
char stack::pop()
{
  if(tos==0) {
    cout << "La pila está vacía";
    return 0; // devuelve nulo si la pila está vacía
  }
  tos--;
  return stck[tos];
}

ostream &operator<<(ostream &stream, stack ob)
{
  char ch;

  while(ch=ob.pop()) stream << ch;
  stream << endl;

  return stream;
}
```

```
main()
{
  stack s;

  s.push('a');
  s.push('b');
  s.push('c');

  cout << s;
  cout << s;

  return 0;
}
```

2.
```
#include <iostream.h>
#include <time.h>

class watch {
  time_t t;
public:
  watch() { t = time(NULL); }
  friend ostream &operator<<(ostream &stream, watch ob);
};

ostream &operator<<(ostream &stream, watch ob)
{
  struct tm *localt;

  localt = localtime(&ob.t);
  stream << asctime(localt) << endl;
  return stream;
}

main()
{
  watch w;

  cout << w;

  return 0;
}
```

3.
```
#include <iostream.h>

class ft_to_inches {
  double feet;
  double inches;
public:
  void set(double f) {
    feet = f;
    inches = f * 12;
  }
  friend istream &operator>>(istream &stream,
                             ft_to_inches &ob);
```

```
  friend ostream &operator<<(ostream &stream,
                             ft_to_inches ob);
};

istream &operator>>(istream &stream, ft_to_inches &ob)
{
  double f;

  cout << "Introduzca pies: ";
  stream >> f;
  ob.set(f);

  return stream;
}

ostream &operator<<(ostream &stream, ft_to_inches ob)
{
  stream << ob.feet << " pies son " << ob.inches;
  stream << " pulgadas\n";

  return stream;
}
main()
{
  ft_to_inches x;

  cin >> x;
  cout << x;

  return 0;
}
```

9. *COMPROBACION DE APTITUD*

1.
```
#include <iostream.h>

main()
{
  cout.width(40);
  cout.fill(':');

  cout << "C++ es divertido" << '\n';

  return 0;
}
```

2.
```
#include <iostream.h>

main()
{
  cout.precision(4);
  cout << 10.0/3.0 << '\n';

  return 0;
}
```

3.
```
#include <iostream.h>
#include <iomanip.h>

main()
{
  cout << setprecision(4) << 10.0/3.0 << '\n';

  return 0;
}
```

4. Un insertor es un **operador<<( )** sobrecargado que saca la información de una clase a un flujo de salida. Un extractor es un **operador>>( )** sobrecargado que mete la información de una clase desde un flujo de entrada.

5.
```
#include <iostream.h>

class date {
 char d[9]; // almacenamiento de la fecha como la cadena : mm/dd/aa
public:
 friend ostream &operator<<(ostream &stream, date ob);
 friend istream &operator>>(istream &stream, date &ob);
};

ostream &operator<<(ostream &stream, date ob)
{
 stream << ob.d << '\n';

 return stream;
}

istream &operator>>(istream &stream, date &ob)
{
 cout << "Introduzca fecha (mm/dd/aa): ";
 stream >> ob.d;

 return stream;
}

main()
{
 date ob;

 cin >> ob;
 cout << ob;

 return 0;
}
```

6. Para utilizar un manipulador parametrizado tiene que incluir **iomanip.h** en su programa.
7. Los flujos predefinidos son:

cin
cout
cerr
clog

9.1. *EJERCICIOS*

1.
```
// Mostrar hora y fecha.
#include <iostream.h>
#include <time.h>

// Un manipulador de salida de hora y fecha.
ostream &td(ostream &stream)
{
  struct tm *localt;
  time_t t;

  t = time(NULL);
  localt = localtime(&t);
  stream << asctime(localt) << endl;

  return stream;
}

main()
{
  cout << td << '\n';

  return 0;
}
```

2.
```
#include <iostream.h>

// Activa la salida en hexadecimal con X.
ostream &sethex(ostream &stream)
{
  stream.setf(ios::hex | ios::uppercase | ios::showbase);

  return stream;
}

// Inicializa los indicadores.
ostream &reset(ostream &stream)
{
  stream.unsetf(ios::hex | ios::uppercase | ios::showbase);

  return stream;
}

main()
{
  cout << sethex << 100 << '\n';
  cout << reset << 100 << '\n';

  return 0;
}
```

3.
```
#include <iostream.h>
#include <string.h>

// Salta 10 caracteres.
istream &skipchar(istream &stream)
{
  int i;
  char c;

  for(i=0; i<10; i++) stream >> c;

  return stream;
}

main()
{
  char str[80];

  cout << "Introduzca algunos caracteres: ";
  cin >> skipchar >> str;

  cout << str << '\n';

  return 0;
}
```

9.2. *EJERCICIOS*

1.
```
/* Copia un archivo y muestra el número de caracteres
   copiados. */
#include <iostream.h>
#include <fstream.h>

main(int argc, char *argv[])
{
  if(argc!=3) {
    cout << "Uso: CPY <entrada> <salida>\n";
    return 1;
  }

  ifstream fin(argv[1]); // abrir archivo de entrada
  ofstream fout(argv[2]); // crear archivo de salida

  if(!fin) {
    cout << "No se puede abrir archivo de entrada\n";
    return 1;
  }

  if(!fout) {
    cout << "No se puede abrir archivo de salida\n";
    return 1;
  }
```

```
  char ch;
  unsigned count=0;

  fin.unsetf(ios::skipws); // no salta espacios
  while(!fin.eof()) {
    fin >> ch;
    fout << ch;
    count++;
  }

  cout << "Número de bytes copiados: " << count << '\n';
  return 0;
}
```

La razón por la que este programa puede mostrar un resultado diferente del mostrado cuando se lista el directorio es que se pueden producir algunas conversiones de carácter. Concretamente, cuando se lee una secuencia retorno de carro/avance de línea, se convierte en carácter de nueva línea. En la salida, los caracteres de nueva línea se cuentan como un carácter, pero se convierten de nuevo a la secuencia retorno de carro/avance de línea.

2.
```
#include <iostream.h>
#include <fstream.h>

main()
{
 ofstream pout("phone");

 if(!pout) {
  cout << "No se puede abrir el archivo PHONE\n";
  return 1;
 }

 pout << "Isaac Newton 415 555-3423\n";
 pout << "Robert Goddard 213 555-2312\n";
 pout << "Enrico Fermi 202 555-1111\n";

 pout.close();

 return 0;
}
```

3.
```
// Cuenta de palabras.
#include <iostream.h>
#include <fstream.h>
#include <ctype.h>

main(int argc, char *argv[])
{
 if(argc!=2) {
  cout << "Uso: COUNT <entrada>\n";
  return 1;
 }
```

```
ifstream in(argv[1]);

if(!in) {
  cout << "No se puede abrir archivo de entrada\n";
  return 1;
}

int count=0;
char ch;

in >> ch; // busca el primer carácter distinto de espacio

// después de encontrarlo, no salta espacios
in.unsetf(ios::skipws); // no salta espacios

while(!in.eof()) {
  in >> ch;
  if(isspace(ch)) count++;
}

cout << "Cuenta de palabras: " << count << '\n';

in.close();

return 0;
}
```

9.3. *EJERCICIOS*

1.
```
/* Copia un archivo y muestra el número de caracteres copia-
dos. */
#include <iostream.h>
#include <fstream.h>

main(int argc, char *argv[])
{
  if(argc!=3) {
    cout << "Uso: CPY <entrada> <salida>\n";
    return 1;
  }

  ifstream fin(argv[1]); // abrir archivo de entrada
  ofstream fout(argv[2]); // crear archivo de salida

  if(!fin) {
    cout << "No se puede abrir archivo de entrada\n";
    return 1;
  }

  if(!fout) {
```

```
    cout << "No se puede abrir archivo de salida\n";
    return 1;
  }

  char ch;
  unsigned count=0;

  while(!fin.eof()) {
    fin.get(ch);
    fout.put(ch);
    count++;
  }

  cout << "Número de bytes copiados: " << count << '\n';
  return 0;
}

// Cuenta de palabras.
#include <iostream.h>
#include <fstream.h>
#include <ctype.h>

main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "Uso: COUNT <entrada>\n";
    return 1;
  }

  ifstream in(argv[1]);

  if(!in) {
    cout << "No se puede abrir archivo de entrada\n";
    return 1;
  }

  int count=0;
  char ch;

  // busca el primer carácter distinto de espacio
  do {
    in.get(ch);
  } while(isspace(ch));

  while(!in.eof()) {
    in.get(ch);
    if(isspace(ch)) count++;
  }

  cout << "Cuenta de palabras: " << count << '\n';

  in.close();

  return 0;
}
```

2.
```
/* Vuelca en un archivo la información de cuentas utilizando
   un insertor. */
#include <iostream.h>
#include <fstream.h>
#include <string.h>

class account {
  int custnum;
  char name[80];
  double balance;
public:
  account(int c, char *n, double b)
  {
  custnum = c;
  strcpy(name, n);
  balance = b;
  }
  friend ostream &operator<<(ostream &stream, account ob);
};

ostream &operator<<(ostream &stream, account ob)
{
  stream << ob.custnum << ' ';
  stream << ob.name << ' ' << ob.balance;
  stream << '\n';

  return stream;
}

main()
{
  account Rex(1011, "Ralph Rex", 12323.34);
  ofstream out("cuentas");

  out << Rex;

  out.close();

  return 0;
}
```

9.4. *EJERCICIOS*

1.
```
// Usa get() para leer una cadena que contiene espacios.
#include <iostream.h>
#include <fstream.h>

main()
{
  char str[80];
```

```
  cout << "Introduzca su nombre: ";
  cin.get(str, 79);

  cout << str << '\n';

  return 0;
}
```

El programa funciona igual utilizando **get( )** o **getline( )**.

2.
```
// Usa getline() para mostrar un archivo.
#include <iostream.h>
#include <fstream.h>

main(int argc, char *argv[])
{
 if(argc!=2) {
  cout << "uso: PR <archivo>\n";
  return 1;
 }

 ifstream in(argv[1]);
 if(!in) {
  cout << "No se puede abrir archivo de entrada\n";
  return 1;
 }

 char str[255];

 while(!in.eof()) {
  in.getline(str, 254);
  cout << str << '\n';
 }

 in.close();

 return 0;
}
```

9.5. *EJERCICIOS*

1.
```
// Muestra en pantalla un archivo hacia atrás.
#include <iostream.h>
#include <fstream.h>

main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "uso: REVERSE <archivo>\n";
    return 1;
  }
```

```
  ifstream in(argv[1]);
  if(!in) {
    cout << "No se puede abrir archivo de entrada\n";
    return 1;
  }

  char ch;
  long i;

  // ir al final del archivo (menos el carácter eof)
  in.seekg(0, ios::end);
  i = in.tellg()-2; // ver cúantos bytes hay en el archivo

  for( ;i>=0; i--) {
    in.seekg(i, ios::beg);
    in.get(ch);
    cout << ch;
  }

  in.close();

  return 0;
}
```

2.
```
// Intercambia caracteres en un archivo.
#include <iostream.h>
#include <fstream.h>

main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "uso: SWAP <archivo>\n";
    return 1;
  }

  // abrir archivo para entrada/salida
  fstream io(argv[1], ios::in | ios::out);
  if(!io) {
    cout << "No se puede abrir el archivo\n";
    return 1;
  }

  char ch1, ch2;
  long i;

  for(i=0 ; !io.eof(); i+=2) {
    io.seekg(i, ios::beg);
    io.get(ch1);
    if(io.eof()) continue;
    io.get(ch2);
    if(io.eof()) continue;
    io.seekg(i, ios::beg);
    io.put(ch2);
```

```
    io.put(ch1);
  }

  io.close();

  return 0;
}
```

9.6. *EJERCICIO*

1.
```
/* Muestra en pantalla un archivo hacia atrás, además de
   hacer comprobación de errores. */
#include <iostream.h>
#include <fstream.h>

main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "uso: REVERSE <archivo>\n";
    return 1;
  }

  ifstream in(argv[1]);
  if(!in) {
    cout << "No se puede abrir archivo de entrada\n";
    return 1;
  }

  char ch;
  long i;

  // ir al final del archivo (menos el carácter eof)
  in.seekg(0, ios::end);
  if(!in.good()) return 1;
  i = in.tellg()-2; // ver cuántos bytes hay en el archivo
  if(!in.good()) return 1;

  for( ;i>=0; i--) {
    in.seekg(i, ios::beg);
    if(!in.good()) return 1;
    in.get(ch);
    if(!in.good()) return 1;
    cout << ch;
  }

  in.close();
  if(!in.good()) return 1;

  return 0;
}
```

```
/* Intercambia caracteres en un archivo con comprobación
   de errores. */
#include <iostream.h>
#include <fstream.h>

main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "uso: SWAP <archivo>\n";
    return 1;
  }

  // abrir archivo para entrada/salida
  fstream io(argv[1], ios::in | ios::out);
  if(!io) {
    cout << "No se puede abrir el archivo \n";
    return 1;
  }

  char ch1, ch2;
  long i;

  for(i=0 ;!io.eof(); i+=2) {
    io.seekg(i, ios::beg);
    if(!io.good()) return 1;
    io.get(ch1);
    if(io.eof()) continue;
    io.get(ch2);
    if(!io.good()) return 1;
    if(io.eof()) continue;
    io.seekg(i, ios::beg);
    if(!io.good()) return 1;
    io.put(ch2);
    if(!io.good()) return 1;
    io.put(ch1);
    if(!io.good()) return 1;
  }

  io.close();
  if(!io.good()) return 1;

  return 0;
}
```

9. COMPROBACION DE APTITUD SUPERIOR

1.
```
#include <iostream.h>

ostream &tabs(ostream &stream)
{
  stream << '\t' << '\t' << '\t' ;
  stream.width(20);

  return stream;
}
```

```
main()
{
  cout << tabs << "Probando\n";
  return 0;
}
```

2.
```
#include <iostream.h>
#include <ctype.h>

istream &findalpha(istream &stream)
{
  char ch;

  do {
    stream.get(ch);
  } while(!isalpha(ch));
  return stream;
}

main()
{
  char str[80];

  cin >> findalpha >> str;
  cout << str << '\n';

  return 0;
}
```

3.
```
// Copia un archivo y cambia el tamaño de las letras.
#include <iostream.h>
#include <fstream.h>
#include <ctype.h>

main(int argc, char *argv[])
{
  char ch;

  if(argc!=3) {
    cout << "Uso: COPYREV <origen> <destino>\n";
    return 1;
  }

  ifstream in(argv[1]);
  if(!in) {
    cout << "No se puede abrir archivo de entrada";
    return 1;
  }

  ofstream out(argv[2]);
  if(!out) {
    cout << "No se puede abrir archivo de salida";
    return 1;
  }
```

```
  while(!in.eof()) {
    ch = in.get();
    if(islower(ch)) ch = toupper(ch);
    else ch = tolower(ch);
    out.put(ch);
  };

  in.close();
  out.close();

  return 0;
}
```

4.
```
// Cuenta de letras.
#include <iostream.h>
#include <fstream.h>
#include <ctype.h>

int alpha[26];

main(int argc, char *argv[])
{
  char ch;

  if(argc!=2) {
    cout << "Uso: COUNT <origen>\n";
    return 1;
  }

  ifstream in(argv[1]);
  if(!in) {
    cout << "No se puede abrir archivo de entrada";
    return 1;
  }

  // inicializa alpha[]
  int i;
  for(i=0; i<26; i++) alpha[i] = 0;

  while(!in.eof()) {
    ch = in.get();
    if(isalpha(ch)) { // si encuentra letra, la cuenta
      ch = toupper(ch); // normaliza
      alpha[ch-'A']++; // 'A'-'A' == 0, 'B'-'A' == 1, etc.
    }
  };

  // muestra la cuenta
  for(i=0; i<26; i++) {
    cout << (char) ('A'+ i) << ": " << alpha[i] << '\n';
  }

  in.close();
  return 0;
}
```

5.
```
/* Copia un archivo y cambia el tamaño de las letras
   con comprobación de errores. */
#include <iostream.h>
#include <fstream.h>
#include <ctype.h>

main(int argc, char *argv[])
{
  char ch;

  if(argc!=3) {
    cout << "Uso: COPYREV <origen> <destino>\n";
    return 1;
  }

  ifstream in(argv[1]);
  if(!in) {
    cout << "No se puede abrir archivo de entrada";
    return 1;
  }

  ofstream out(argv[2]);
  if(!out) {
    cout << "No se puede abrir archivo de salida";
    return 1;
  }

  while(!in.eof()) {
    ch = in.get();
    if(!in.good()) return 1;
    if(islower(ch)) ch = toupper(ch);
    else ch = tolower(ch);
    out.put(ch);
    if(!out.good()) return 1;
  };

  in.close();
  out.close();
  if(!in.good() && !out.good()) return 1;

  return 0;
}

// Cuenta de letras con comprobación de errores.
#include <iostream.h>
#include <fstream.h>
#include <ctype.h>

int alpha[26];

main(int argc, char *argv[])
{
  char ch;
```

```
  if(argc!=2) {
    cout << "Uso: COUNT <origen>\n";
    return 1;
  }

  ifstream in(argv[1]);
  if(!in) {
    cout << "No se puede abrir archivo de entrada";
    return 1;
  }

  // inicializa alpha[]
  int i;
  for(i=0; i<26; i++) alpha[i] = 0;

  while(!in.eof()) {
    ch = in.get();
    if(!in.good() && !in.eof()) return 1;
    if(isalpha(ch)) { // si encuentra letra, la cuenta
      ch = toupper(ch); // normaliza
      alpha[ch-'A']++; // 'A'-'A' == 0, 'B'-'A' == 1, etc.
    }
  };

  // muestra la cuenta
  for(i=0; i<26; i++) {
    cout << (char) ('A'+ i) << ": " << alpha[i] << '\n';
  }

  in.close();
  if(!in.good()) return 1;

  return 0;
}
```

6. Para establecer el puntero **get** utilice **seekg( )**. Para establecer el puntero **put** utilice **seekp( )**.

9. COMPROBACION DE APTITUD INTEGRADA

1.
```
#include <fstream.h>
#include <iostream.h>
#include <string.h>

define SIZE 40

class inventory {
  char item[SIZE]; // nombre del elemento
  int onhand; // número de existencias
  double cost; // precio del elemento
public:
```

```
  inventory(char *i, int o, double c)
  {
    strcpy(item, i);
    onhand = o;
    cost = c;
  }
  void store(fstream &stream);
  void retrieve(fstream &stream);
  friend ostream &operator<<(ostream &stream, inventory ob);
  friend istream &operator>>(istream &stream, inventory &ob);
};

ostream &operator<<(ostream &stream, inventory ob)
{
  stream << ob.item << ": " << ob.onhand;
  stream << " existencias en $" << ob.cost << '\n';

  return stream;
}

istream &operator>>(istream &stream, inventory &ob)
{
  cout << "Introduzca el nombre del elemento: ";
  stream >> ob.item;
  cout << "Introduzca el número de existencias: ";
  stream >> ob.onhand;
  cout << "Introduzca el precio: ";
  stream >> ob.cost;

  return stream;
}

void inventory::store(fstream &stream)
{
  stream.write(item, SIZE);
  stream.write((char *) &onhand, sizeof(int));
  stream.write((char *) &cost, sizeof(double));
}

void inventory::retrieve(fstream &stream)
{
  stream.read(item, SIZE);
  stream.read((char *) &onhand, sizeof(int));

  stream.read((char *) &cost, sizeof(double));
}

main()
{
  fstream inv("inv", ios::in | ios::out); //entrada/salida
  int i;

  inventory pliers("alicates", 12, 4.95);
  inventory hammers("martillos", 5, 9.45);
```

```
  inventory wrenches("llaves inglesas", 22, 13.90);
  inventory temp("", 0, 0.0);

  // escribir en el archivo
  pliers.store(inv);
  hammers.store(inv);
  wrenches.store(inv);

  do {
  cout << "Registro # (-1 para salir): ";
  cin >> i;
  if(i == -1) break;
  inv.seekg(i*(SIZE+sizeof(int)+sizeof(double)), ios::beg);
  temp.retrieve(inv);
  cout << temp;
  } while(inv.good());

  inv.close();

  return 0;
}
```

10. COMPROBACION DE APTITUD

1.
```
#include <iostream.h>

ostream &setsci(ostream &stream)
{
  stream.setf(ios::scientific | ios::uppercase);

  return stream;
}

main()
{
  double f = 123.23;

  cout << setsci << f;
  cout << '\n';

  return 0;
}
```

2.
```
// Copia y convierte tabuladores a espacios.
#include <iostream.h>
#include <fstream.h>

main(int argc, char *argv[])
{
  if(argc!=3) {
    cout << "Uso: CPY <entrada> <salida>\n";
    return 1;
  }
```

```
  ifstream in(argv[1]);
  if(!in) {
    cout << "No se puede abrir archivo de entrada\n";
    return 1;
  }

  ofstream out(argv[2]);
  if(!out) {
    cout << "No se puede abrir archivo de salida\n";
    return 1;
  }

  char ch;
  int i = 8;

  while(!in.eof()) {
    in.get(ch);
    if(ch=='\t') for( ; i>=0; i--) out.put(' ');
    else out.put(ch);
    if(i == -1 || ch=='\n') i = 8;
    i--;
  }

  in.close();
  out.close();

  return 0;
}
```

3.
```
// Búsqueda en un archivo.
#include <iostream.h>
#include <fstream.h>
#include <string.h>

main(int argc, char *argv[])
{
  if(argc!=3) {
    cout << "Uso: SEARCH <archivo> <palabra>\n";
    return 1;
  }

  ifstream in(argv[1]);
  if(!in) {
    cout << "No se puede abrir archivo de entrada\n";
    return 1;
  }

  char str[255];
  int count=0;

  while(!in.eof()) {
    in >> str;
    if(!strcmp(str, argv[2])) count++;
  }
```

```
  cout << argv[2] << " encontrado " << count;
  cout << " número de veces.\n";

  in.close();

  return 0;
}
```

4. La sentencia es

```
out.seekp(234, ios::beg);
```

5. Las funciones son **rdstate( )**, **good( )**, **eof( )**, **fail( )**, y **bad( )**.
6. La E/S de C++ se puede personalizar, para que opere sobre las clases que se creen.

10.2. *EJERCICIOS*

1.
```
#include <iostream.h>

class num {
public:
  int i;
  num(int x) { i = x; }
  virtual void shownum() { cout << i << '\n'; }
};

class outhex : public num {
public:
  outhex(int n) : num(n) {}
  void shownum() { cout << hex << i << '\n'; }
};

class outoct : public num {
public:
  outoct(int n) : num(n) {}
  void shownum() { cout << oct << i << '\n'; }
};

main()
{
  outoct o(10);
  outhex h(20);

  o.shownum();
  h.shownum();

  return 0;
}
```

2.
```
#include <iostream.h>


class distance {
public:
  double d;
  distance(double f) { d = f; }
  virtual void trav_time()
  {
    cout << "Duración del viaje a 60 mph: ";
    cout << d / 60 << '\n';
  }
};


class metric : public distance {
public:
  metric(double f) : distance(f) {}
  void trav_time()
  {
    cout << "Duración del viaje a 100 kph: ";
    cout << d / 100 << '\n';
  }
};

main()
{
  distance *p, mph(88.0);
  metric kph(88);


  p = &mph;
  p->trav_time();


  p = &kph;
  p->trav_time();


  return 0;
}
```

10.3. *EJERCICIOS*

2. Por definición, una clase abstracta contiene al menos una función virtual pura. Esto significa que no existe cuerpo para esa función respecto a esa clase. De este modo, no hay forma de poder crear un objeto porque no está completa la definición de clase.
3. Cuando se llama a **func( )** respecto a **derived1**, la que se utiliza es la **func( )** que hay dentro de base. La razón de esto es que las funciones virtuales son jerárquicas.

10.4. EJERCICIO

1.
```
// Crea una clase de lista genérica para enteros.
#include <iostream.h>
#include <stdlib.h>

class list {
public:
  list *head; // puntero al siguiente elemento de la lista
  list *tail;
  list *next;
  int num; // valor a almacenar
public:
  list() { head = tail = next = NULL; }
  virtual void store(int i) = 0;
  virtual int retrieve() = 0;
};

// crea una lista de tipo cola.
class queue : public list {
public:
  void store(int i);
  int retrieve();
};
void queue::store(int i)
{
  list *item;

  item = new queue;
  if(!item) {
    cout << "Error de asignación\n";
    exit(1);
  }
  item->num = i;

  // poner al final de la lista
  if(tail) tail->next = item;
  tail = item;
  item->next = NULL;
  if(!head) head = tail;
}

int queue::retrieve()
{
  int i;
  list *p;

  if(!head) {
    cout << "Lista vacía\n";
    return 0;
  }

  // quitar del principio de la lista
```

```
  i = head->num;
  p = head;
  head = head->next;
  delete p;

  return i;
}
// Crea una lista de tipo pila.
class stack : public list {
public:
  void store(int i);
  int retrieve();
};

void stack::store(int i)
{
  list *item;

  item = new stack;
  if(!item) {
    cout << "Error de asignación\n";
    exit(1);
  }
  item->num = i;

  // pone al principio de la lista al estilo de una pila
  if(head) item->next = head;
  head = item;
  if(!tail) tail = head;
}

int stack::retrieve()
{
  int i;
  list *p;

  if(!head) {
    cout << "Lista vacía\n";
    return 0;
  }

  // quitar del principio de la lista
  i = head->num;
  p = head;
  head = head->next;
  delete p;

  return i;
}

// Crea una lista ordenada.
class sorted : public list {
public:
  void store(int i);
  int retrieve();
};
```

```
void sorted::store(int i)
{
  list *item;
  list *p, *p2;

  item = new sorted;
  if(!item) {
    cout << "Error de asignación\n";
    exit(1);
  }
  item->num = i;

  // buscar dónde poner el siguiente elemento
  p = head;
  p2 = NULL;
  while(p) { // situarse en el medio
    if(p->num > i) {
    item->next = p;
    if(p2) p2->next = item; // no el primer elemento
    if(p==head) head = item; // nuevo primer elemento
    break;
    }
    p2 = p;
    p = p->next;
  }
  if(!p) { // ir al final
    if(tail) tail->next = item;
    tail = item;
    item->next = NULL;
  }
  if(!head) // es el primer elemento
    head = item;
}

int sorted::retrieve()
{
  int i;
  list *p;

  if(!head) {
    cout << "Lista vacía\n";
    return 0;
  }

  // quitar del principio de la lista
  i = head->num;
  p = head;
  head = head->next;
  delete p;

  return i;
}

main()
```

```
{
  list *p;

  // prueba la cola
  queue q_ob;

  p = &q_ob; // apunta a la cola

  p->store(1);
  p->store(2);
  p->store(3);

  cout << "Cola: ";
  cout << p->retrieve();
  cout << p->retrieve();
  cout << p->retrieve();

  cout << '\n';

  // prueba la pila
  stack s_ob;

  p = &s_ob; // apunta a la pila

  p->store(1);
  p->store(2);
  p->store(3);

  cout << "Pila: ";
  cout << p->retrieve();
  cout << p->retrieve();
  cout << p->retrieve();

  cout << '\n';

  // prueba la lista ordenada
  sorted sorted_ob;

  p = &sorted_ob;

  p->store(4);
  p->store(1);
  p->store(3);
  p->store(9);
  p->store(5);

  cout << "Ordenada: ";
  cout << p->retrieve();
  cout << p->retrieve();
  cout << p->retrieve();
  cout << p->retrieve();
  cout << p->retrieve();

  cout << '\n';

  return 0;
}
```

10. COMPROBACION DE APTITUD SUPERIOR

1. Una función virtual es básicamente una función contenedor declarada en una clase base que está redefinida por una clase derivada de esa clase base. Este proceso se llama redefinición.
2. Las funciones no miembro y las funciones constructor no se pueden hacer virtuales.
3. Una función virtual soporta polimorfismo en tiempo de ejecución mediante el uso de punteros a clase base. Cuando un puntero a clase base apunta a una clase derivada que contiene una función virtual, la función específica llamada está determinada por el tipo de objeto al que está apuntando.
4. Una función virtual pura es la que no contiene ninguna definición relativa a la clase base.
5. Una clase abstracta es una clase base que contiene al menos una función virtual. Una clase polimórfica es la que contiene al menos una función virtual.
6. El fragmento es incorrecto porque la redefinición de una función virtual tiene que tener el mismo tipo de retorno y número y tipo de parámetros que la función original. En este caso, la redefinición de **f( )** varía en el número de parámetros.
7. Sí.

10. COMPROBACION DE APTITUD INTEGRADA

1.
```
// Crea una clase de lista genérica para enteros.
#include <iostream.h>
#include <stdlib.h>

class list {
public:
  list *head; // puntero al siguiente elemento de la lista
  list *tail;
  list *next;
  int num; // valor a almacenar
public:
  list() { head = tail = next = NULL; }
  virtual void store(int i) = 0;
  virtual int retrieve() = 0;
};

// Crea una lista de tipo cola.
class queue : public list {
public:
  void store(int i);
  int retrieve();
  queue operator+(int i) { store(i); return *this; }
  int operator--(int unused) { return retrieve(); }
};

void queue::store(int i)
{
  list *item;
```

```
  item = new queue;
  if(!item) {
    cout << "Error de asignación\n";
    exit(1);
  }
  item->num = i;

  // poner al final de la lista
  if(tail) tail->next = item;
  tail = item;
  item->next = NULL;
  if(!head) head = tail;
}

int queue::retrieve()
{
  int i;
  list *p;

  if(!head) {
    cout << "Lista vacía\n";
    return 0;
  }

  // quitar del principio de la lista
  i = head->num;
  p = head;
  head = head->next;
  delete p;

  return i;
}

// Crea una lista de tipo pila.
class stack : public list {
public:
  void store(int i);
  int retrieve();
  stack operator+(int i) { store(i); return *this; }
  int operator--(int unused) { return retrieve(); }
};

void stack::store(int i)
{
  list *item;

  item = new stack;
  if(!item) {
    cout << "Error de asignación\n";
    exit(1);
  }
  item->num = i;

  // pone al principio de la lista al estilo de una pila
```

```
  if(head) item->next = head;
  head = item;
  if(!tail) tail = head;
}

int stack::retrieve()
{
  int i;
  list *p;

  if(!head) {
    cout << "Lista vacía\n";
    return 0;
  }

  // quitar del principio de la lista
  i = head->num;
  p = head;
  head = head->next;
  delete p;

  return i;
}

main()
{
  // prueba la cola
  queue q_ob;

  q_ob + 1;
  q_ob + 2;
  q_ob + 3;

  cout << "Cola: ";
  cout << q_ob--;
  cout << q_ob--;
  cout << q_ob--;

  cout << '\n';

  // prueba la pila
  stack s_ob;

  s_ob + 1;
  s_ob + 2;
  s_ob + 3;

  cout << "Pila: ";
  cout << s_ob--;
  cout << s_ob--;
  cout << s_ob--;

  cout << '\n';

  return 0;
}
```

2. Las funciones virtuales varían de las funciones sobrecargadas en que las funciones sobrecargadas *tienen* que variar en el número de parámetros o en el tipo de los parámetros. Una función virtual redefinida tiene que tener exactamente el mismo prototipo (es decir, el mismo tipo de retorno y número y tipo de parámetros) que la función original.

11. COMPROBACION DE APTITUD

1. Una función virtual es una función que está declarada como **virtual** por una clase base y después redefinida por una clase derivada.
2. Una función virtual pura es la que no tiene cuerpo definido dentro de la clase base. Esto significa que la función tiene que estar redefinida por una clase derivada. Una clase base que contiene al menos una función virtual pura se llama abstracta.
3. El polimorfismo en tiempo de ejecución se consigue mediante el uso de funciones virtuales y punteros a clase base.
4. Si una clase derivada no redefine una función virtual no pura, la clase derivada utilizará la versión de la clase base de la función virtual.
5. La principal ventaja del polimorfismo en tiempo de ejecución es la flexibilidad. La principal desventaja es la pérdida de velocidad de ejecución.

11.1. EJERCICIOS

2.
```
#include <iostream.h>

template <class X> X min(X a, X b)
{
  if(a<=b) return a;
  else return b;
}

main()
{
  cout << min(12.2, 2.0);
  cout << endl;
  cout << min(3, 4);
  cout << endl;
  cout << min('c', 'a');
  return 0;
}
```

3.
```
#include <iostream.h>
#include <string.h>

template <class X> int find(X object, X *list, int size)
{
  int i;

  for(i=0; i<size; i++)
```

```
    if(object == list[i]) return i;
  return -1;
}

main()
{
  int a[] = {1, 2, 3, 4};
  char *c = "Esto es una prueba";
  double d[] = {1.1, 2.2, 3.3};

  cout << find(3, a, 4);
  cout << endl;
  cout << find('a', c, (int) strlen(c));
  cout << endl;
  cout << find(0.0, d, 3);

  return 0;
}
```

4. Las funciones genéricas son valiosas porque permiten definir un algoritmo general que se puede aplicar a diferentes tipos de datos. (Es decir, no es necesario crear versiones específicas del algoritmo.) Las funciones genéricas además ayudan a implementar el concepto de «una interfaz, múltiples métodos», que es algo común en la programación en C++.

11.2. *EJERCICIOS*

2.
```
// Crea una cola genérica.
#include <iostream.h>

define SIZE 100

template <class Qtype> class q_type {
  Qtype queue[SIZE]; // guarda la cola
  int head, tail; // índices de cabeza y cola
public:
  q_type() { head = tail = 0; }
  void q(Qtype num); // guardar
  Qtype deq(); // recuperar
};

// Poner valor en la cola.
template <class Qtype> void q_type<Qtype>::q(Qtype num)
{
  if(tail+1==head || (tail+1==SIZE && !head)) {
    cout << "La cola está llena\n";
    return;
  }
  tail++;
  if(tail==SIZE) tail = 0; // recorrer cíclicamente
  queue[tail] = num;
}
```

```
// Quitar valor de la cola.
template <class Qtype> Qtype q_type<Qtype>::deq()
{
  if(head == tail) {
    cout << "La cola está vacía\n";
    return 0; // o algún otro indicador de error
  }
  head++;
  if(head==SIZE) head = 0; // recorrer cíclicamente
  return queue[head];
}

main()
{
  q_type<int> q1;
  q_type<char> q2;
  int i;

  for(i=1; i<=10; i++) {
    q1.q(i);
    q2.q(i-1+'A');
  }

  for(i=1; i<=10; i++) {
    cout << "De la cola 1: " << q1.deq() << "\n";
    cout << "De la cola 2: " << q2.deq() << "\n";
  }

  return 0;
}
```

3.
```
#include <iostream.h>

template <class X> class input {
  X data;
public:
  input(char *s, X min, X max);
  // ...
};

template <class X>
input<X>::input(char *s, X min, X max)
{
  do {
    cout << s << ": ";
    cin >> data;
  } while( data < min || data > max);
}

main()
{
  input<int> i("introduzca entero", 0, 10);
  input<char> c("introduzca carácter", 'A', 'Z');

  return 0;
}
```

11.3. EJERCICIOS

2. Se llama a **throw** antes de que la ejecución pase por el bloque **try**.
3. Surge una excepción de carácter, pero la sentencia **catch** sólo manejará un puntero a carácter. (Es decir, no hay una sentencia **catch** correspondiente para manejar la excepción de carácter.)
4. Si surge una excepción para la que no hay **catch** correspondiente, se llama a **terminate( )** y puede producirse una terminación del programa anormal.

11.4. EJERCICIOS

2. No hay sentencia **catch** correspondiente para **throw**. Un modo de corregir el problema es crear un manejador **catch(int)**. Otra forma de corregirlo es captar todas las excepciones utilizando un manejador **catch(...)**.
3. Véase la respuesta 2.
4. **catch(...)** captura todas las excepciones.

5.
```
#include <iostream.h>
#include <stdlib.h>

double divide(double a, double b)
{
  try {
    if(!b) throw(b);
  }
  catch(double) {
    cout << "No se puede dividir por cero\n";
    exit(1);
  }
  return a/b;
}

main()
{
  cout << divide(10.0, 2.5) << endl;
  cout << divide(10.0, 0.0);

  return 0;
}
```

11. COMPROBACION DE APTITUD SUPERIOR

1.
```
#include <iostream.h>
#include <string.h>

// Una función genérica de búsqueda de modo.
template <class X> X mode(X *data, int size)
```

```
{
  register int t, w;
  X md, oldmd;
  int count, oldcount;

  oldmd = 0;
  oldcount = 0;
  for(t=0; t<size; t++) {
    md = data[t];
    count = 1;
    for(w = t+1; w < size; w++)
      if(md==data[w]) count++;
    if(count > oldcount) {
      oldmd = md;
      oldcount = count;
    }
  }
  return oldmd;
}

main()
{
  int i[] = { 1, 2, 3, 4, 2, 3, 2, 2, 1, 5};
  char *p = "Esto es una prueba";

  cout << "modo de i: " << mode(i, 10) << endl;
  cout << "modo de p: " << mode(p, (int) strlen(p));

  return 0;
}
```

2.

```
#include <iostream.h>

template <class X> X sum(X *data, int size)
{
  int i;
  X result = 0;

  for(i=0; i<size; i++) result += data[i];

  return result;
}

main()
{
  int i[] = {1, 2, 3, 4};
  double d[] = {1.1, 2.2, 3.3, 4.4};

  cout << sum(i, 4) << endl;
  cout << sum(d, 4) << endl;

  return 0;
}
```

3.
```
#include <iostream.h>

// Algoritmo de ordenación de burbuja genérico.
template <class X> void bubble(X *data, int size)
{
  register int a, b;
  X t;

  for(a=1; a < size; a++)
    for(b=size-1; b >= a; b--)
      if(data[b-1] > data[b]) {
        t = data[b-1];
        data[b-1] = data[b];
        data[b] = t;
      }
}

main()
{
  int i[] = {3, 2, 5, 6, 1, 8, 9, 3, 6, 9};
  double d[] = {1.2, 5.5, 2.2, 3.3};
  int j;

  bubble(i, 10); // ordena enteros
  bubble(d, 4); // ordena dobles


  for(j=0; j<10; j++) cout << i[j] << ' ';
  cout << endl;

  for(j=0; j<4; j++) cout << d[j] << ' ';
  cout << endl;

  return 0;
}
```

4.
```
/* Esta función prueba una pila genérica que guarda dos
   valores. */
#include <iostream.h>

define SIZE 10

// Crea una clase pila genérica
template <class StackType> class stack {
  StackType stck[SIZE][2]; // guarda la pila
  int tos; // índice de la cabeza de la pila

public:
  void init() { tos = 0; }
  void push(StackType ob, StackType ob2);
  StackType pop(StackType &ob2);
};

// poner objetos.
```

```
template <class StackType> void
stack<StackType::push(StackType ob, StackType ob2)
{
  if(tos==SIZE) {
    cout << "La pila está llena";
    return;
  }
  stck[tos][0] = ob;
  stck[tos][1] = ob2;
  tos++;
}

// Sacar objetos.
template <class StackType>
StackType stack<StackType::pop(StackType &ob2)
{
  if(tos==0) {
    cout << "La pila está vacía";
    return 0; // devuelve nulo cuando la pila está vacía
  }
  tos--;
  ob2 = stck[tos][1];
  return stck[tos][0];
}
main()
{
  // Prueba las pilas de caracteres.
  stack<char> s1, s2; // crea dos pilas
  int i;
  char ch;

  // inicializar las pilas
  s1.init();
  s2.init();

  s1.push('a', 'b');
  s2.push('x', 'z');
  s1.push('b', 'd');
  s2.push('y', 'e');
  s1.push('c', 'a');
  s2.push('z', 'x');

  for(i=0; i<3; i++) {
    cout << "Saca de s1: " << s1.pop(ch);
    cout << ' ' << ch << "\n";
  }
  for(i=0; i<3; i++) {
    cout << "Saca de s2: " << s2.pop(ch);
    cout << ' ' << ch << "\n";
  }

  // prueba las pilas de doubles
  stack<double> ds1, ds2; // crea dos pilas
  double d;
```

```
  // inicializar las pilas
  ds1.init();
  ds2.init();

  ds1.push(1.1, 2.0);
  ds2.push(2.2, 3.0);
  ds1.push(3.3, 4.0);
  ds2.push(4.4, 5.0);
  ds1.push(5.5, 6.0);
  ds2.push(6.6, 7.0);

  for(i=0; i<3; i++) {
    cout << "Saca de ds1: " << ds1.pop(d);
    cout << ' '<< d << "\n";
  }

  for(i=0; i<3; i++) {
    cout << "Saca de ds2: " << ds2.pop(d);
    cout << ' '<< d << "\n";
  }

  return 0;
}
```

5. A continuación se muestra la forma general de **try**, **catch** y **throw**:

```
try {
// bloque try
throw exp;
}
catch(arg) {
// ...
}
```

6.
```
/* Esta función prueba una pila genéricá que incluye manejo
   de excepciones. */
#include <iostream.h>

#define SIZE 10

// Crea una clase pila genérica
template <class StackType> class stack {
  StackType stck[SIZE]; // guarda la pila
  int tos; // índice de la cabeza de la pila

public:
  void init() { tos = 0; } // inicializar la pila
  void push(StackType ch); // poner objeto en la pila
  StackType pop(); // sacar objeto de la pila
};

// Pone un objeto.
```

```
template <class StackType> void
stack<StackType::push(StackType ob)
{
 try {
  if(tos==SIZE) throw SIZE;
 }
 catch(int) {
  cout << "La pila está llena";
  return;
 }
 stck[tos] = ob;
 tos++;
}

// Saca un objeto.
template <class StackType> StackType stack<StackType::pop()
{
 try {
  if(tos==0) throw 0;
 }
 catch(int) {
  cout << "La pila está vacía";
  return 0; // devuelve nulo cuando la pila está vacía
 }
 tos--;
 return stck[tos];
}
main()
{
 // Prueba las pilas de caracteres.
 stack<char> s1, s2; // crea dos pilas
 int i;
 // inicializar las pilas
 s1.init();
 s2.init();

 s1.push('a');
 s2.push('x');
 s1.push('b');
 s2.push('y');
 s1.push('c');
 s2.push('z');

 for(i=0; i<3; i++) cout << "Saca de s1: " << s1.pop() << "\n";
 for(i=0; i<4; i++) cout << "Saca de s2: " << s2.pop() << "\n";

 // prueba las pilas de dobles
 stack<double> ds1, ds2; // crea dos pilas

 // inicializar las pilas
 ds1.init();
 ds2.init();

 ds1.push(1.1);
```

```
 ds2.push(2.2);
 ds1.push(3.3);
 ds2.push(4.4);
 ds1.push(5.5);
 ds2.push(6.6);

 for(i=0; i<3; i++) cout << "Saca de ds1: " << ds1.pop() << "\n";
 for(i=0; i<4; i++) cout << "Saca de ds2: " << ds2.pop() << "\n";

 return 0;
}
```

12. COMPROBACION DE APTITUD

1. En C++, una función genérica define un conjunto general de operaciones que se aplicarán a distintos tipos de datos. Se implementa utilizando la palabra clave **template**. Su forma general se muestra aquí:

```
template <class Ttipo> tipo-ret nombre-func(lista-param)
{
 // ...
}
```

2. En C++, una clase genérica define todas las operaciones que se relacionan con esa clase, pero la información real se especifica como parámetro cuando se crea un objeto de esa clase. A continuación se muestra su forma general:

```
   template <class Ttipo> class nombre-clase
{
 // ...
}
```

3.
```
#include <iostream.h>

// Devuelve a elevado a b.
template <class X> X gexp(X a, X b)
{
    X i, result=1;

  for(i=0; i<b; i++) result *= a;

  return result;
}

main()
{
  cout << gexp(2, 3) << endl;
  cout << gexp(10.0, 2.0);

  return 0;
}
```

4.
```
#include <iostream.h>
#include <fstream.h>

template <class CoordType> class coord {
  CoordType x, y;
public:
  coord(CoordType i, CoordType j) { x = i; y = j; }
  void show() { cout << x << ", " << y << endl; }
};

main()
{
  coord<int> o1(1, 2), o2(3, 4);

  o1.show();
  o2.show();

  coord<double> o3(0.0, 0.23), o4(10.19, 3.098);

  o3.show();
  o4.show();

  return 0;
}
```

5. Puede hacer lo siguiente con **try**, **catch**, y **throw**. Ponga todas las sentencias que quiera supervisar para excepciones dentro del bloque **try**. Si se produce una excepción, saque esa excepción utilizando **throw** y gestiónela con la sentencia **catch** correspondiente.
6. No.
7. Se llama a **terminate( )** cuando surge una excepción para la que no hay sentencia **catch** correspondiente. A **unexpected( )** se le llama cuando surge una excepción que no está soportada por una función.
8. **catch(...)**

12.1. *EJERCICIOS*

1.
```
// Un ejemplo de recurso compartido que traza la salida.
#include <iostream.h>
#include <string.h>

class output {
  static char outbuf[255]; // este es el recurso compartido
  static int inuse; // si vale 0, se puede acceder al búfer;
                    //ocupado, en cualquier otro caso
  static int oindex; // índice de outbuf
  char str[80];
  int i; // índice del siguiente carácter de str
  int who; // identifica el objeto, debe ser > 0
public:
  output(int w, char *s) { strcpy(str, s); i = 0; who = w; }
```

```
  /* Esta función devuelve -1 si el búfer está ocupado, se
     ha completado la salida, y devuelve who si todavía está
     usando el búfer.
  */
  int putbuf()
    {
    if(!str[i]) { // realizando la salida
      inuse = 0; // se elimina el búfer
      return 0; // señal de terminación
    }
    if(!inuse) inuse = who; // acceso al búfer
    if(inuse != who) {
      cout << "Proceso " << who << " Utilizado por otra\n";
      return -1; // en uso por alguién más
    }
    if(str[i]) { // todavía hay caracteres que extraer
      outbuf[oindex] = str[i];
      cout << "Proceso " << who << " mandando carácter\n";
      i++; oindex++;
      outbuf[oindex] = '\0'; // mantiene siempre la
      //terminación nula
      return 1;
    }
  }
  void show() { cout << outbuf << '\n'; }
};

char output::outbuf[255]; // este es el recurso compartido
int output::inuse = 0; // si vale 0 se puede acceder al búfer;
                       // ocupado en cualquier otro caso
int output::oindex = 0; // índice de outbuf

main()
{
  output o1(1, "Esto es una prueba"), o2(2, " de atributos
  estáticos");

  while(o1.putbuf() | o2.putbuf()) ; // salida de caracteres

  o1.show();

  return 0;
}
```

2.

```
#include <iostream.h>

class test {
  static int count;
public:
  test() { count++; }
  ~test() { count--; }
  int getcount() { return count; }
};

int test::count = 0;
```

```
main()
{
  test o1, o2, o3;

  cout << o1.getcount() << " objetos en existencia\n";

  test *p;

  p = new test; // asigna espacio a un objeto
  if(!p) {
    cout << "Error de asignación\n";
    return 1;
  }

  cout << o1.getcount();
  cout << " objetos en existencia después de la asignación\n";

  // eliminar objeto
  delete p;

  cout << o1.getcount();
  cout << " objetos en existencia después de la eliminación\n";

  return 0;
}
```

12.2. *EJERCICIOS*

1.
```
/* Esta versión muestra el número de caracteres escritos
   en buf.
*/
#include <iostream.h>
#include <strstream.h>

main()
{
  char buf[255];

  ostrstream ostr(buf, sizeof buf);

  ostr << "La E/S basada en arrays utiliza flujos igual que la ";
  ostr << "E/S 'normal'\n" << 100;
  ostr << ' ' << 123.23 << '\n';

  // se pueden usar también manipuladores
  ostr << hex << 100 << ' ';
  // o indicadores de formato
  ostr << ostr.setf(ios::scientific) << 123.23 << '\n';
  ostr << ends; // asegura que el búfer termina en nulo
```

```
  // muestra cadena resultante
  cout << buf;

  cout << ostr.pcount();

  return 0;
}
```

2.
```
/* Usa E/S basada en arrays para copiar los contenidos
   de un array en otro.
*/
#include <iostream.h>
#include <strstream.h>

char inbuf[] = "Esto es una prueba de E/S basada en arrays
de C++";
char outbuf[255];

main()
{
  istrstream istr(inbuf);
  ostrstream ostr(outbuf, sizeof outbuf);

  char ch;

  while(!istr.eof()) {
    istr.get(ch);
    ostr.put(ch);
  }
  ostr.put('\0'); // terminado en nulo

  cout << "Entrada: " << inbuf << '\n';
  cout << "Salida: " << outbuf << '\n';

  return 0;
}
```

3.
```
// Convertir cadena afloat.
#include <iostream.h>
#include <strstream.h>

main()
{
  float f;
  char s[] = "1234.564"; // flotante representado como cadena

  istrstream istr(s);

  // ¡convertir a representación interna de forma fácil!
  istr >> f;

  cout << "Forma convertida: " << f << '\n';

  return 0;
}
```

12.4. *EJERCICIOS*

1.
```
// Convertir tipo cadena a entero.
#include <iostream.h>
#include <string.h>

class strtype {
  char str[80];
  int len;
public:
  strtype(char *s) { strcpy(str, s); len = strlen(s); }
  operator char *() { return str; }
  operator int() { return len; }
};

main()
{
  strtype s("Las funciones de conversión son convenientes");
  char *p;
  int l;

  l = s; // convertir s a entero - que es la longitud de la
  //cadena
  p = s; // convertir s a carácter * - que es puntero a la
  //cadena

  cout << "La cadena:\n";
  cout << p << "\nes " << l << " caracteres de largo.\n";

  return 0;
}
```

2.
```
#include <iostream.h>
#include <strstream.h>

int p(int base, int exp);

class pwr {
  int base;
  int exp;
public:
  pwr(int b, int e) { base = b; exp = e; }
  operator int() { return p(base, exp); }
};

// Devuelve la base elevada a la potencia exp.
int p(int base, int exp)
{
  int temp;

  for(temp=1; exp; exp--) temp = temp * base;

  return temp;
}
```

```
main()
{
  pwr o1(2, 3), o2(3, 3);
  int result;

  result = o1;
  cout << result << '\n';

  result = o2;
  cout << result << '\n';

  // se puede usar directamente en una sentencia cout como
  //ésta:
  cout << o1+100 << '\n';

  return 0;
}
```

12. COMPROBACION DE APTITUD SUPERIOR

1. A diferencia de las variables miembro normales, para las que cada objeto tiene su propia copia, sólo existe una copia de una variable miembro **static**, y la comparten todos los objetos de esa clase.
2. Para utilizar E/S basada en arrays incluya **strstream.h**.
3. No.

4. `extern "C" int counter();`

5. Una función de conversión simplemente convierte un objeto en un valor compatible con otro tipo. Las funciones de conversión se utilizan normalmente para convertir objetos en valores compatibles con los tipos de datos incorporados.
6. No, los prototipos no son opcionales en C++.
7. El estándar propuesto del ANSI C++ establece que al menos los primeros 1.024 caracteres de un identificador son significativos.

Indice