11

Procesamiento de archivos

Objetivos

- Ser capaz de crear, leer, escribir y actualizar archivos.
- Familiarizarse con el proceso de archivos de acceso secuencial.
- Familiarizarse con el proceso de archivos de acceso directo.

Leo una parte en su totalidad.
Samuel Goldwyn

¡Saluden!
La bandera está pasando.
Henry Holcomb Bennett

La conciencia ... no se presenta a sí misma dividida en pedazos ... Los símiles mediante los cuales se le puede describir más naturalmente serían un "río" o una "corriente".
William James

Debo suponer que un documento "No archivar" se archivará en un archivo "No archivar".
Senador Frank Church
Senate Intelligence Subcommittee Hearing, 1975

Sinopsis

Resumen • Terminología • Errores comunes de programación • Prácticas sanas de programación • Sugerencias de rendimiento • Sugerencias de portabilidad • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios.

11.1 Introducción

El almacenamiento de datos en variables y en arreglos es temporal; al terminar un programa todos estos datos se pierden. Para la conservación permanente de grandes cantidades de datos se utilizan los *archivos*. Las computadoras almacenan los archivos en dispositivos de almacenamiento secundario, especialmente en dispositivos de almacenamiento en disco. En este capítulo, explicaremos como los programas en C crean, actualizan y procesan los archivos de datos. Analizaremos tanto archivos de acceso secuencial como archivos de acceso directo.

11.2 La jerarquía de datos

En último término, todos los elementos de datos procesados por una computadora se reducen a combinaciones de ceros y de unos. Es así porque es simple y económico construir dispositivos electrónicos que puedan asumir dos estados estables —uno de los estados representando `0` y el otro `1`. Es verdaderamente asombroso que las impresionantes funciones ejecutadas por las computadoras sólo involucren el manejo más básico de `0`s y de `1`s.

En una computadora el elemento de datos más pequeño puede asumir el valor `0` o el valor `1`. Este elemento de datos se conoce como un *bit* (abreviatura de *"binary digit"* —un dígito que puede asumir uno de dos valores). Los circuitos de la computadora ejecutan varias manipulaciones simples de bits, como es determinar el valor de un bit, establecer el valor de un bit, e invertir un bit (de `1` a `0` o de `0` a `1`).

Para los programadores resulta muy engorroso trabajar con datos bajo su forma de nivel bajo, es decir los bits. En vez de ello, los programadores prefieren trabajar con datos en forma de *dígitos decimales* (es decir 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9), *letras* (es decir, de la A a la Z, y a hast a la z) y *símbolos especiales* (es decir, $, @, %, &, *, (,), -, +, ", :, ?, /, y muchos otros). Los *dígitos, las letras y los símbolos especiales se conocen como caracteres*. El conjunto de todos los caracteres que pudieran

ser utilizados para escribir programas y representar elementos de datos de una computadora en particular se llama el *conjunto de caracteres* de dicha computadora. Dado que las computadoras sólo pueden procesar `1`s y `0`s, todo carácter de un conjunto de caracteres en una computadora es representado como un patrón de `1`s y `0`s (llamados un *byte*). Hoy día, los bytes están comúnmente formados por ocho bits. Los programadores crean programas y elementos de datos como caracteres; a continuación, las computadoras manipulan y procesan dichos caracteres como patrones de bits.

Al igual que los caracteres están formados por bits, los *campos* se componen de caracteres. Un campo es un grupo de caracteres que contiene un significado. Por ejemplo, un campo que consista únicamente de letras mayúsculas y minúsculas, puede ser utilizado para representar el nombre de una persona.

Los elementos de datos procesados por las computadoras forman una *jerarquía de datos,* en la cual los elementos de datos se convierten en más grandes y más complejos en cuanto a estructura conforme progresamos desde los bits, hacia los caracteres (bytes), hacia los campos y así sucesivamente.

Un *registro* (es decir, un `struct` en C) se compone de varios campos. En un sistema de nómina, por ejemplo, un registro para un empleado en particular pudiera estar formado por los campos siguientes:

1. Número de seguridad social
2. Nombre
3. Dirección
4. Tasa horaria de salario
5. Número de excepciones reclamadas
6. Ganancias acumuladas año a la fecha
7. Cantidades retenidas de impuestos federales, etcétera

Entonces, un registro es un grupo de campos relacionados. En el ejemplo anterior, cada uno de los campos corresponde al mismo empleado. Naturalmente, una compañía particular pudiera tener muchos empleados, y para cada uno de ellos tendrá un registro de nómina. Un *archivo* es un grupo de registros relacionados. El archivo de nóminas de una empresa normalmente contiene un registro para cada empleado. Entonces, un archivo de nóminas para una pequeña compañía pudiera contener únicamente 22 registros, en tanto que un archivo de nómina para una compañía grande pudiera contener 100,000 registros. No es desusado para una organización tener cientos e inclusive miles de archivos, muchos de ellos conteniendo millones y aún miles de millones de caracteres de información. Con la creciente popularidad de los discos laser ópticos y la tecnología de multimedios, pronto inclusive serán comunes archivos de billones de bytes. En la figura 11.1 se ilustra la jerarquía de los datos.

Para facilitar la recuperación de registros específicos a partir de un archivo, por lo menos un campo de cada registro es seleccionado como *registro clave.* Un registro clave identifica a un registro como perteneciente a una persona o entidad en particular. Por ejemplo, en el registro de nóminas descrito en esta sección, el número de seguridad social normalmente sería seleccionado como registro clave.

Existen muchas formas de organizar los registros dentro de un archivo. El tipo más popular de organización se conoce como *archivo secuencial,* en el cual típicamente los registros se almacenan en orden, en relación con el campo de registro clave. En un archivo de nóminas, los

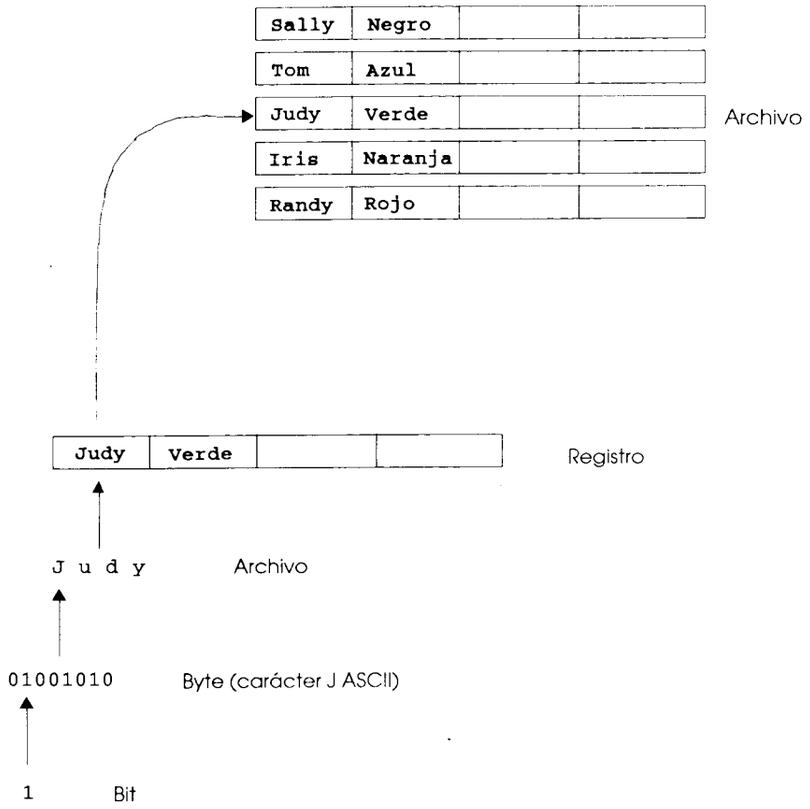


Fig. 11.1 La jerarquía de datos.

registros normalmente se ordenan por número de seguridad social. El primer registro de empleado del archivo contiene el número más bajo de seguridad social, y los registros subsiguientes contienen números de seguridad social cada vez más altos.

Para almacenar datos la mayor parte de los negocios utilizan muchos archivos distintos. Por ejemplo, las empresas pueden tener archivos de nóminas, archivos de cuentas por cobrar (enlistando el dinero que los clientes les deben), archivos de cuentas por pagar (enlistando dinero que se les debe a los proveedores), archivos de inventarios (enlistando hechos relacionados con todos los elementos manejados por el negocio) y muchos otros tipos de archivos. A veces un grupo de archivos relacionados se conoce como una *base de datos*. Una colección de programas diseñado para crear y administrar bases de datos se conoce como un *sistema de administración de bases de datos* (DBMS, por *database management system*).

11.3 Archivos y flujos

C ve cada uno de los archivos simplemente como un flujo secuencial de bytes (figura 11.2). Cada archivo termina con un *marcador de fin de archivo* o en un número de bytes específico registrado en una estructura administrativa de datos, mantenida por el sistema. Cuando un archivo *se abre*, se asocia un flujo con el archivo. Al empezar la ejecución de un programa automáticamente se

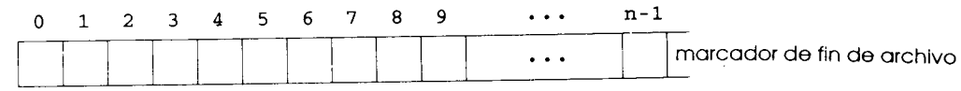


Fig. 11.2 Vista en C de un archivo de *n* bytes.

abren tres archivos y sus flujos asociados —la *entrada estándar*, la *salida estándar* y el *error estándar*. Los flujos proporcionan canales de comunicación entre archivos y programas. Por ejemplo, el flujo de entrada estándar permite que un programa lea datos del teclado, el flujo de salida estándar permite que un programa imprima datos a la pantalla. Abrir un archivo regresa un apuntador a una estructura **FILE** (definida en **<stdio.h>**) que contiene información utlizada para procesar dicho archivo. Esta estructura incluye un *descriptor de archivo*, es decir un índice a un arreglo del sistema operativo, conocido como una *tabla de archivo abierto*. Cada elemento del arreglo contiene un *bloque de control de archivo* (*FCB*, por *file control block*) utilizado por el sistema operativo para administrar el archivo particular. La entrada estándar, salida estándar y error estándar son manejados utilizando los apuntadores de archivo **stdin**, **stdout** y **stderr**.

La biblioteca estándar proporciona muchas funciones para leer datos de los archivos y para escribir datos a los archivos. La función **fgetc**, al igual que **getchar**, lee un carácter de un archivo. La función **fgetc** recibe como un argumento un apuntador **FILE** para el archivo del cual se leerá un carácter. La llamada **fgetc(stdin)** lee un carácter de **stdin** —la entrada estándar. Esta llamada es equivalente a la llamada **getchar()**. La función **fputc**, al igual que **putchar**, escribe un carácter a un archivo. La función **fputc** recibe como argumentos un carácter para ser escrito, y un apuntador al archivo hacia el cual el carácter será escrito. La llamada de función **fputc('a', stdout)** escribe el carácter **'a'**, a **stdout** —la salida estándar. Esta llamada es equivalente a **putchar('a')**.

Varias otras funciones, utilizadas para leer datos de la entrada estándar y para escribir datos a la salida estándar, tienen funciones de procesamiento de archivo similarmente identificados. Las funciones **fgets** y **fputs**, por ejemplo, pueden ser utilizadas para leer una línea de un archivo y para escribir una línea a un archivo, respectivamente. Sus contrapartidas para leer de la estrada estándar y para escribir a la salida estándar, **gets** y **puts**, ya fueron analizadas en el capítulo 8. En varias de las siguientes secciones, presentamos los equivalentes en procesamiento de archivo de las funciones **scanf** y **printf**—**fscanf** y **fprintf**. Más adelante en el capítulo analizaremos las funciones **fread** y **fwrite**.

11.4 Cómo crear un archivo de acceso secuencial

C no impone estructuras a un archivo. Por lo tanto, como parte del lenguaje C no existen conceptos como registro de un archivo. Por lo tanto, para que cumpla con los requisitos de cada aplicación en particular, el programador deberá proporcionar alguna estructura de archivo. En el ejemplo siguiente, vemos como un programador puede imponer una estructura de registro en un archivo.

El programa de la figura 11.3 crea un archivo simple de acceso secuencial, que podría ser utilizado en cualquier sistema de cuentas por cobrar para ayudar a llevar control de las cantidades que deben los clientes a crédito de una empresa. Para cada cliente, el programa obtiene un número de cuenta, el nombre del cliente y el saldo del mismo (es decir, la cantidad que el cliente le debe a la empresa debido a bienes y servicios recibidos en el pasado). Los datos obtenidos de cada cliente constituyen un "registro" para cada uno de estos clientes. En esta aplicación el número de cuenta se utiliza como registro clave —el archivo será creado y mantenido por orden de número

```
/* Create a sequential file */
#include <stdio.h>

main()
{
   int account;
   char name[30];
   float balance;
   FILE *cfPtr;   /* cfPtr = clients.dat file pointer */

   if ((cfPtr = fopen("clients.dat", "w")) == NULL)
      printf("File could not be opened\n");
   else {
      printf("Enter the account, name, and balance.\n");
      printf("Enter EOF to end input.\n");
      printf("? ");
      scanf("%d%s%f", &account, name, &balance);

      while (!feof(stdin)) {
         fprintf(cfPtr, "%d %s %.2f\n",
                 account, name, balance);
         printf("? ");
         scanf("%d%s%f", &account, name, &balance);
      }

      fclose(cfPtr);
   }

   return 0;
}
```

```
Enter the account, name, and balance.
Enter the EOF character to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
?
```

Fig. 11.3 Cómo crear un archivo secuencial.

de cuenta. Este programa supone que el usuario escribe los registros en orden por número de cuenta. En un sistema de cuentas por cobrar más completo, se incluiría una capacidad de ordenamiento, de tal forma que el usuario pudiera introducir los registros en cualquier orden. Los registros a continuación serían clasificados y escritos en el archivo.

Examinemos ahora este programa. El enunciado

```
FILE *cfPtr;
```

establece que **cfPtr** es un apuntador a una estructura **FILE**. El programa C administra cada archivo con una estructura **FILE** por separado. Para utilizar archivos el programador no necesita

saber los detalles específicos de la estructura **FILE**. Pronto veremos precisamente cómo la estructura **FILE** lleva indirectamente al bloque de control de archivos del sistema operativo (FCB) correspondiente a un archivo.

Sugerencia de portabilidad 11.1

La estructura **FILE** *depende del sistema operativo (es decir, los miembros de la estructura varían de un sistema a otro, según la forma en que cada sistema maneja sus archivos).*

Cada archivo abierto debe tener un apuntador declarado por separado del tipo **FILE**, que es utilizado para referirse al archivo. La línea

```
if ((cfPtr = fopen("clients.dat", "w")) == NULL)
```

nombra el archivo —**"clients.dat"**— para ser utilizado por el programa y establece una "línea de comunicación" con el archivo. El apuntador de archivo **cfPtr** es asignado a un apuntador a la estructura **FILE** para el archivo abierto con **fopen**. La función **fopen** toma dos argumentos: un nombre de archivo y un *modo de archivo abierto*. El modo de archivo abierto **"w"** indica que el archivo debe de ser abierto para *escritura*. Si el archivo no existe y es abierto para escritura, **fopen** crea el archivo. Si un archivo existente es abierto para escritura, el contenido del archivo es descartado sin advertencia. En el programa, la estructura **if** se utiliza para determinar si el apuntador al archivo **cfPtr** es **NULL** (es decir, el archivo no está abierto). Si es **NULL**, se imprime un mensaje de error y el programa termina. De lo contrario, la entrada es procesada y escrita al archivo.

Error común de programación 11.1

*Abrir un archivo existente para escritura (***"w"***) cuando, de hecho, el usuario desea conservar el archivo; el contenido del archivo se descartará sin advertencia.*

Error común de programación 11.2

Olvidar abrir un archivo antes de intentar hacer referencia a él en un programa.

El programa le solicita al usuario que introduzca los varios campos de cada registro, o que introduzca fin de archivo cuando esté completa la entrada de datos. La figura 11.4 enlista las combinaciones de teclas para introducir fin de archivo en varios sistemas de computación.

La línea

```
while (!feof(stdin))
```

utiliza la función **feof** para determinar si el *indicador de fin de archivo* está definido para el archivo al que se refiere **stdin**. El indicador de fin de archivo le informa al programa que ya no hay más datos a procesarse. En el programa de la figura 11.3, el indicador de fin de archivo está

Sistema de computación	Combinación de teclas
Sistemas UNIX	*<return><ctrl>d*
IBM PC y compatibles	*<ctrl>z*
Macintosh	*<ctrl>d*
VAX (VMS)	*<ctrl>z*

Fig. 11.4 Combinaciones de teclas de fin de archivo correspondientes a varios sistemas populares de computación.

definido para la entrada estándar cuando el usuario introduce la combinación de teclas de fin de archivo. El argumento de la función **feof** es un apuntador al archivo bajo prueba en relación con el indicador de fin de archivo (en este caso **stdin**). La función regresa un valor no cero (verdadera) una vez definido el indicador de fin de archivo; de lo contrario regresa cero. La estructura **while**, que en este programa incluye la llamada **feof**, se continuará ejecutando en tanto no se defina el indicador de fin de archivo.

El enunciado

```
fprintf(cfPtr, "%d %s %.2f\n", account, name, balance);
```

escribe datos al archivo **clients.dat**. Los datos pueden ser recuperados más tarde mediante un programa diseñado para leer el archivo (vea la sección 11.5). La función **fprintf** es equivalente a **printf**, excepto que **fprintf** también recibe como argumento un apuntador de archivo para el archivo al cual se escribirán los datos.

Error común de programación 11.3

Usar el apuntador de archivo incorrecto para referirse a un archivo.

Práctica sana de programación 11.1

Asegúrese que en un programa las llamadas a las funciones de procesamiento de archivos contienen los apuntadores de archivo correctos.

Después de que el usuario haya introducido el fin de archivo, el programa cierra el archivo **clients.dat** utilizando **fclose** y termina. La función **fclose** también recibe el apuntador de archivo (en vez del nombre del archivo) como un argumento. Si no se llama a la función **fclose** en forma explícita, normalmente cuando la ejecución del programa termine el sistema operativo cerrará el archivo. Esto es un ejemplo de "buena administración interna" del sistema operativo.

Práctica sana de programación 11.2

Cierre cada archivo en forma explícita, tan pronto sepa que el programa ya no hará otra vez referencia al archivo.

Sugerencia de rendimiento 11.1

Cerrar un archivo puede liberar recursos que están siendo esperados por otros usuarios o programas.

En la ejecución de muestra del programa de la figura 11.3, el usuario introduce información correspondiente a cinco cuentas, y a continuación escribe el fin de archivo, para indicar que la entrada de datos está completa. La ejecución de muestra no pone de manifiesto cómo aparecen realmente los registros de datos dentro del archivo. A fin de verificar que el archivo haya sido creado exitosamente, en la siguiente sección introducimos un programa que lee el archivo e imprime su contenido.

En la figura 11.5 se ilustra la relación entre los apuntadores **FILE**, las estructuras **FILE** y los FCB en memoria. Cuando se abre el archivo **"clients.dat"**, se copia un FCB para dicho archivo en la memoria. La figura muestra la conexión entre el apuntador de archivo regresado por **fopen** y el FCB utilizado por el sistema operativo para la administración del archivo.

Los programas no pueden procesar ningún archivo, un archivo o varios archivos. Cada archivo utilizado en un programa deberá tener un nombre único, y deberá tener un apuntador de archivo distinto regresado por **fopen**. Todas las funciones de procesamiento de archivos subsecuentes después de su apertura deberán referirse al archivo utilizando el apuntador de archivo apropiado.

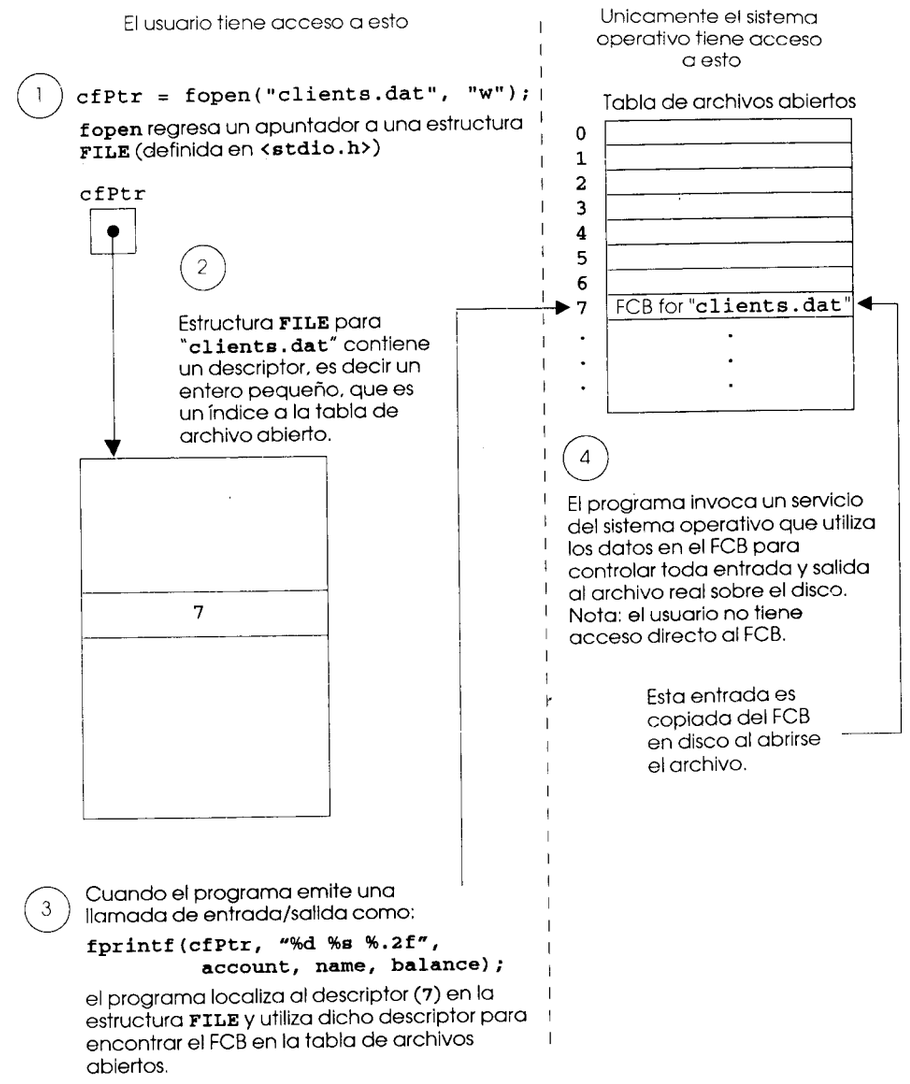


Fig. 11.5 Relación entre los apuntadores **FILE** las estructuras **FILE** y los FCB.

Los archivos pueden ser abiertos en uno de varios modos. Para crear un archivo, o para descartar el contenido de un archivo antes de escribir datos, abra el archivo para escritura (**"w"**). Para leer un archivo existente, ábralo para lectura (**"r"**). Para añadir registros al final de un archivo existente, abra el archivo para agregar (**"a"**). Para abrir a un archivo de tal forma que pueda ser escrito y leído, abra el archivo para actualizar en uno de los tres modos de actualización —**"r+"**, **"w+"** o **"a+"**. El modo **"r+"** abre un archivo para lectura y escritura. El modo **"w+"** genera un

archivo para lectura y escritura. Si el archivo ya existe, el archivo es abierto y el contenido actual de dicho archivo se descarta. El modo **"a+"** abre un archivo para lectura y escritura —toda escritura se efectuará al final del archivo. Si el archivo no existe será creado.

Si al abrir un archivo en cualquiera de los modos anteriores ocurre un error, **fopen** regresará **NULL**. Algunos errores posibles son:

Error común de programación 11.4

Abrir para lectura un archivo no existente.

Error común de programación 11.5

Abrir un archivo para lectura o escritura sin haber obtenido los derechos de acceso al archivo apropiados (esto depende del sistema operativo).

Error común de programación 11.6

Abrir un archivo para escritura cuando no hay espacio disponible en disco. En la figura 11.6 se enlistan los modos de apertura de archivos.

Error común de programación 11.7

Puede llevar a errores devastadores abrir un archivo utilizando el modo de archivo incorrecto. Por ejemplo, abrir un archivo en el modo de escribir ("w") cuando debería haberse abierto en modo de actualizar ("r+") hace que sea descartado todo el contenido del archivo.

Práctica sana de programación 11.3

Abra un archivo únicamente para lectura (y no para actualizar), si el contenido del archivo no debe modificarse. Esto evitará cambios no intencionales en el contenido del archivo. Este es otro ejemplo del principio del mínimo privilegio.

11.5 Cómo leer datos de un archivo de acceso secuencial

Los datos se almacenan en archivos, de tal forma que cuando sea necesario puedan ser recuperados para su proceso. La sección anterior demostró cómo crear un archivo para acceso secuencial. En esta sección, analizamos como leer secuencialmente los datos de un archivo.

Modo	Descripcion
r	Abrir una archivo para lectura.
w	Crear un archivo para escritura. Si el archivo ya existe, se descarta el contenido actual.
a	Agregar; abrir o crear un archivo para escribir al final del mismo.
r+	Abrir un archivo para actualizar (leer y escribir).
w+	Crear un archivo para actualizar. Si el archivo ya existe, se descarta el contenido actual.
a+	Agregar; abrir o crear un archivo para actualizar; la escritura se efectuará al final del archivo.

Fig. 11.6 Modos de apertura de archivo.

El programa de la figura 11.7 lee registros del archivo **"clients.dat"** creados por el programa de la figura 11.3 e imprime el contenido de los registros. El enunciado

```
FILE *cfPtr;
```

indica que **cfPtr** es un apuntador a un **FILE**. La línea

```
if ((cfPtr = fopen("clients.dat", "r")) ==NULL)
```

intenta abrir el archivo **"clients.dat"** para lectura (**'r'**) y determina si el archivo se ha abierto con éxito (es decir, que **fopen** no regresa **NULL**). El enunciado

```
fscanf(cPtr, "%d%s%f, &account, name, &balance);
```

lee un "registro" del archivo. La función **fscanf** es equivalente a la función **scanf**, salvo que **fscanf** recibe como argumento un apuntador a un archivo para el archivo del cual se van a leer datos. Después de que el enunciado anterior es ejecutado por primera vez, **account** tendrá el valor **100**, **name** tendrá el valor **"Jones"**, y **balance** tendrá el valor **24.98**. Cada vez que se ejecute el segundo enunciado **fscanf**, se leerá otro registro del archivo y **account**, **name** y **balance** tomarán nuevos valores. Cuando se llegue al final del archivo, éste se cerrará y el programa terminará.

```
/* Reading and printing a sequential file */
#include <stdio.h>

main()
{
   int account;
   char name[30];
   float balance;
   FILE *cfPtr;    /* cfPtr = clients.dat file pointer */

   if ((cfPtr = fopen("clients.dat", "r")) == NULL)
      printf("File could not be opened\n");
   else {
      printf("%-10s%-13s%s\n", "Account", "Name", "Balance");
      fscanf(cfPtr, "%d%s%f", &account, name, &balance);

      while (!feof(cfPtr)) {
         printf("%-10d%-13s%7.2f\n", account, name, balance);
         fscanf(cfPtr, "%d%s%f", &account, name, &balance);
      }

      fclose(cfPtr);
   }

   return 0;
}
```

Fig. 11.7 Cómo leer e imprimir un archivo secuencial (parte 1 de 2).

```
Account  Name      Balance
100      Jones       24.98
200      Doe        345.67
300      White        0.00
400      Stone      -42.16
500      Rich       224.62
```

Fig. 11.7 Cómo leer e imprimir un archivo secuencial (parte 2 de 2).

Para recuperar secuencialmente datos de un archivo, un programa normalmente empieza a leer a partir del principio del archivo, y lee todos los datos en forma consecutiva, hasta que encuentra los datos deseados. Durante la ejecución de un programa pudiera ser deseable procesar los datos secuencialmente en un archivo varias veces (a partir del principio del archivo). Un enunciado como

```
rewind(cfPtr);
```

genera un *apuntador de posición de archivo* del programa —que indica el número del siguiente byte del archivo a leerse o a escribirse— a que se recoloque al principio del archivo (es decir en el byte 0) al cual apunta **cfPtr**. El apuntador de posición de archivo no es realmente un apuntador. Más bien es un valor entero que especifica la posición de byte en el archivo, en el cual ocurrirá la siguiente lectura o escritura. Esto a veces se denomina el *desplazamiento de archivo*. El apuntador de posición de archivo es un miembro de la estructura **FILE** asociado con cada archivo.

Ahora introducimos un programa (figura 11.8) que le permite a un gerente de crédito obtener listas de clientes con saldo 0 (es decir, clientes que no deben ningún dinero), clientes con saldos acreedores (es decir, clientes a los cuales la empresa les debe dinero), clientes con saldos deudores (es decir, clientes que le deben dinero a la empresa por bienes y servicios recibidos). Un saldo acreedor es una cantidad negativa; un saldo deudor es una cantidad positiva.

El programa despliega un menú y le permite al gerente de crédito introducir una de tres opciones para obtener información de crédito. La opción 1 produce una lista de cuentas con saldos en cero. La opción 2 produce una lista de cuentas con saldos acreedores. La opción 3 produce una lista de cuentas con saldos deudores. La opción 4 termina la ejecución del programa. Una salida de muestra se despliega en la figura 11.9.

Note que los datos en este tipo de archivo secuencial no pueden ser modificados sin riesgo de destruir otros datos dentro del archivo. Por ejemplo, si el nombre "**White**" necesitara ser modificado a "**Worthington**", el nombre antiguo simplemente no puede ser sobrescrito. El registro para **White** fue escrito al archivo como

```
300 White 0.00
```

Si el registro se reescribe utilizando el nuevo nombre empezando en la misma posición en el archivo, el registro sería

```
300 Worthington 0.00
```

El nuevo registro es más largo que el original. Los caracteres más allá de la segunda "**o**" de "**Worthington**" sobreescribirían el principio del siguiente registro secuencial en el archivo. El


```
/* Credit inquiry program */
#include <stdio.h>

main()
{
   int request, account;
   float balance;
   char name[30];
   FILE *cfPtr;

   if ((cfPtr = fopen("clients.dat", "r")) == NULL)
      printf("File could not be opened\n");
   else {
      printf("Enter request\n"
             " 1 - List accounts with zero balances\n"
             " 2 - List accounts with credit balances\n"
             " 3 - List accounts with debit balances\n"
             " 4 - End of run\n? ");
      scanf("%d", &request);

      while (request != 4) {
         fscanf(cfPtr, "%d%s%f", &account, name, &balance);

         switch (request) {
            case 1:
               printf("\nAccounts with zero balances:\n");

               while (!feof(cfPtr)) {

                  if (balance == 0)
                     printf("%-10d%-13s%7.2f\n",
                            account, name, balance);

                  fscanf(cfPtr, "%d%s%f",
                         &account, name, &balance);
               }

               break;
            case 2:
               printf("\nAccounts with credit balances:\n");

               while (!feof(cfPtr)) {

                  if (balance < 0)
                     printf("%-10d%-13s%7.2f\n",
                            account, name, balance);

                  fscanf(cfPtr, "%d%s%f",
                         &account, name, &balance);
               }

               break;
```

Fig. 11.8 Programa de consulta de crédito (parte 1 de 2).

```
            case 3:
               printf("\nAccounts with debit balances:\n");

               while (!feof(cfPtr)) {

                  if (balance > 0)
                     printf("%-10d%-13s%7.2f\n",
                            account, name, balance);

                  fscanf(cfPtr, "%d%s%f",
                         &account, name, &balance);
               }

               break;
         }

         rewind(cfPtr);
         printf("\n? ");
         scanf("%d", &request);
      }

      printf("End of run.\n");
      fclose(cfPtr);
   }

   return 0;
}
```

Fig. 11.8 Programa de consulta de crédito (parte 2 de 2).

```
Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run
? 1

Accounts with zero balances:
300        White                0.00

? 2

Accounts with credit balances:
400        Stone              -42.16

? 3

Accounts with debit balances:
100        Jones               24.98
200        Doe                345.67
500        Rich               224.62

? 4
End of run.
```

Fig. 11.9 Salida de muestra del programa de consulta de crédito de la figura 11.8.

problema aquí es que en el modelo de entrada/salida con formato utilizando **`fprintf`** y **`fscanf`**, los campos —y por lo tanto los registros— pueden variar de tamaño. Por ejemplo, 7, 14, -117, 2074, y 27383 son todos **`int`** almacenados internamente en el mismo número de bytes, pero imprimen en la pantalla o **`fprintf`** en el disco como campos de tamaño diferente.

Por lo tanto, el acceso secuencial mediante **`fprint`** y **`fscanf`** normalmente no se utiliza para actualizar registros en su sitio. En vez de ello, usualmente la totalidad del archivo se vuelve a escribir. Para llevar a cabo la modificación de nombre anteriormente citada, los registros anteriores a **`300 White 0.00`** en un archivo como éste, de acceso secuencial, serían copiados a un nuevo archivo, sería escrito el nuevo registro, y los registros existentes después de **`300 White 0.00`** serían vueltos a copiar al nuevo archivo. Esto significa el procesamiento de todos los registros en el archivo para simplemente actualizar un registro.

11.6 Archivos de acceso directo

Como hemos indicado anteriormente, los registros en un archivo creados con la función de salida **`fprintf`** con formato, no necesariamente son de la misma longitud. Sin embargo, los registros individuales de un *archivo de acceso directo* normalmente son de longitud fija y se puede tener acceso a ellos directamente (y por lo tanto rápidamente) sin tener que buscar a través de otros registros. Esto hace que los archivos de acceso directo sean apropiados para sistemas de reservación de aerolíneas, sistemas bancarios, sistemas de punto de venta y otros tipos de *sistemas de procesamiento de transacciones,* que requieren de acceso rápido a datos específicos. Existen otras formas para poner en funcionamiento archivos de acceso directo, pero limitaremos nuestro análisis a este enfoque sencillo de utilización de registros de longitud fija.

Dado que en un archivo de acceso directo todos los registros normalmente tienen la misma longitud, la posición exacta de un registro en relación con el principio del archivo puede ser calculada como una función del registro clave. Pronto veremos como esto facilita el acceso inmediato a registros específicos, inclusive en archivos grandes.

En la figura 11.10 se ilustra una forma para poner en marcha un archivo de acceso directo. Un archivo como éste es similar a un tren de carga con muchos vagones —algunos vacíos y algunos con carga. Cada vagón en el tren tiene la misma longitud.

Los datos pueden ser insertos en un archivo de acceso directo sin destruir otros datos en el archivo. Los datos almacenados anteriormente también pueden ser actualizados o borrados, sin tener que reescribir todo el archivo. En las secciones que siguen explicaremos como se crea un archivo de acceso directo, como se introduce información, como se leen los datos tanto secuencial como directamente, como se actualizan los datos y como se borran aquellos datos que ya no se requieran.

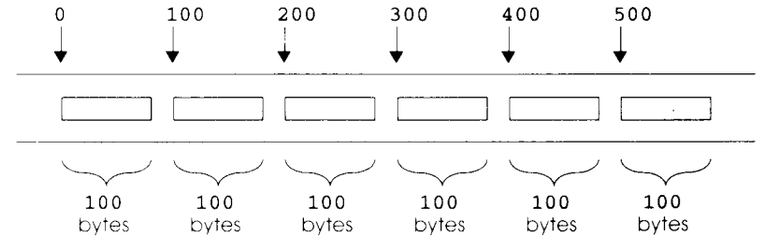


Fig. 11.10 Vista de un archivo de acceso directo con registros de longitud fija.

11.7 Cómo crear un archivo de acceso directo

La función **fwrite** transfiere a un archivo un número especificado de bytes empezando en una posición especificada de memoria. Los datos se escriben al principio de la posición en el archivo indicada mediante el apuntador de posición de archivo. La función **fread** transfiere un número especificado de bytes de la posición en el archivo, especificado por el apuntador de posición de archivo, a un área en memoria empezando a partir de una dirección especificada. Ahora, al escribir un entero, en vez de utilizar

```
fprintf(fPtr, "%d", number);
```

podría imprimir desde 1 dígito hasta un máximo de 11 dígitos (10 dígitos más un signo, cada uno de los cuales requiere 1 byte de almacenamiento) para un entero de 4 bytes, podemos utilizar

```
fwrite(&number, sizeof(int), 1, fPtr);
```

que siempre escribirá 4 bytes (o 2 bytes en un sistema con enteros de 2 bytes) de la variable **number** al archivo representado por **fPtr** (explicaremos en breve el argumento **1**). Más adelante, **fread** podrá ser utilizado para leer 4 de estos bytes a la variable entera **number**. Aunque **fread** y **fwrite** leen y escriben datos, como son enteros, en tamaño fijo en vez de en formato de tamaño variable, los datos que manejan se procesan en formato "en bruto" de computadora (es decir, bytes de datos), en vez de en el formato legible para los seres humanos **printf** y **scanf**.

Las funciones **fwrite** y **fread** son capaces de leer y de escribir arreglos de datos hacia y desde el disco. El tercer argumento, tanto de **fread** como **fwrite** es el número de elementos en el arreglo que deberá ser leído del disco, o escrito al disco. La llamada anterior de función **fwrite** escribe un solo entero al disco, por lo que el tercer argumento es **1** (como si un elemento de un arreglo fuera escrito).

Los programas de procesamiento de archivos rara vez escriben un solo campo a un archivo. Normalmente, escriben un **struct** a la vez, como veremos en los ejemplos siguientes.

Considere el siguiente enunciado de problema:

Crear un sistema de procesamiento de crédito capaz de almacenar hasta 100 registros de longitud fija. Cada registro deberá estar formado de un número de cuenta, que será utilizado como registro clave, un apellido, un nombre y un saldo. El programa resultante deberá ser capaz de actualizar una cuenta, insertar un nuevo registro de cuenta, borrar una cuenta y enlistar todos los registros de cuenta en un archivo de texto con formato para su impresión. Utilice un archivo de acceso directo.

En las siguientes varias secciones se introducen las técnicas necesarias para crear el programa de procesamiento de crédito. El programa de la figura 11.11 muestra como abrir un archivo de acceso directo, definir un formato de registro utilizando un **struct**, escribir datos al disco, y cerrar el archivo. Este programa inicializa los 100 registros del archivo **"credit.dat"**, con **structs** vacíos utilizando la función **fwrite**. Cada **struct** vacío contiene **0** para el número de cuenta, **NULL** (representado por comillas vacías) para el apellido, **NULL** para el nombre y **0.0** para el saldo. El archivo se inicializa de esta forma para crear espacio en el disco en el cual se almacenará el archivo, y para que sea posible determinar si un registro contiene datos.

La función **fwrite** escribe un bloque (un número específico de bytes) de datos a un archivo. En nuestro programa, el enunciado

```
fwrite(&blankClient, sizeof(struct clientData), 1, cfPtr);
```



```
/* Creating a randomly accessed file sequentially */
#include <stdio.h>

struct clientData {
   int acctNum;
   char lastName[15];
   char firstName[10];
   float balance;
};

main()
{
   int i;
   struct clientData blankClient = {0, "", "", 0.0};
   FILE *cfPtr;

   if ((cfPtr = fopen("credit.dat", "w")) == NULL)
      printf("File could not be opened.\n");
   else {

      for (i = 1; i <= 100; i++)
         fwrite(&blankClient,
                sizeof(struct clientData), 1, cfPtr);

      fclose (cfPtr);
   }

   return 0;
}
```

Fig. 11.11 Cómo crear un archivo de acceso directo en forma secuencial.

hace que la estructura **blankClient** de tamaño **sizeof(struct clientData)** se escriba al archivo al cual apunta **cfPtr**. El operador **sizeof** regresa el tamaño en bytes del objeto contenido en los paréntesis (en este caso **struct clientData**). El operador **sizeof** es un operador unario en tiempo de compilación que regresa un entero no signado. El operador **sizeof** puede ser utilizado para determinar el tamaño en bytes de cualquier tipo o expresión de datos. Por ejemplo, **sizeof(int)** se utiliza para determinar si en una computadora en particular un entero está almacenado en 2 o en 4 bytes.

Sugerencia de rendimiento 11.2

*Muchos programadores piensan erróneamente que **sizeof** es una función, y que utilizándolo se genera una sobrecarga en tiempo de ejecución de una llamada de función. No existe tal sobrecarga, porque **sizeof** es un operador en tiempo de compilación.*

La función **fwrite** puede de hecho ser utilizada para escribir varios elementos de un arreglo de objetos. Para escribir varios elementos de arreglo, el programador proporciona un apuntador a un arreglo como primer argumento en la llamada **fwrite**, y especifica el número de elementos a escribirse como el tercer argumento en la llamada **fwrite**. En el enunciado anterior, **fwrite** fue utilizado para escribir un solo objeto que no era un elemento de arreglo. Escribir un solo objeto es el equivalente a escribir un elemento de un arreglo, de ahí el **1** en la llamada **fwrite**.

11.8 Cómo escribir datos directamente a un archivo de acceso directo

El programa de la figura 11.2 escribe datos al archivo **"credit.dat"**. Utiliza la combinación de **fseek** y **fwrite** para almacenar datos en posiciones específicas dentro del archivo. La función **fseek** define el apuntador de posición de archivo a una posición específica dentro del archivo, y a continuación **fwrite** escribe los datos. Una ejecución de muestra aparece en la figura 11.13.

El enunciado

```
fseek(cfPtr, (accountNum - 1) * sizeof(struct clientData);
      SEEK_SET);
```

```
/* Writing to a random access file */
#include <stdio.h>

struct clientData {
   int acctNum;
   char lastName[15];
   char firstName[10];
   float balance;
};

main()
{
   FILE *cfPtr;
   struct clientData client;

   if ((cfPtr = fopen("credit.dat", "r+")) == NULL)
      printf("File could not be opened.\n");
   else {
      printf("Enter account number"
             " (1 to 100, 0 to end input)\n? ");
      scanf("%d", &client.acctNum);

      while (client.acctNum != 0) {
         printf("Enter lastname, firstname, balance\n? ");
         scanf("%s%s%f", &client.lastName,
               &client.firstName, &client.balance);
         fseek(cfPtr, (client.acctNum - 1) *
               sizeof(struct clientData), SEEK_SET);
         fwrite(&client, sizeof(struct clientData), 1, cfPtr);
         printf("Enter account number\n? ");
         scanf("%d", &client.acctNum);
      }
   }

   fclose(cfPtr);

   return 0;
}
```

Fig. 11.12 Cómo escribir datos directamente a un archivo de acceso directo.


```
Enter account number (1 to 100, 0 to end input)
? 37
Enter lastname, firstname, balance
? Barker Doug 0.00
Enter account number
? 29
Enter lastname, firstname, balance
? Brown Nancy -24.54
Enter account number
? 96
Enter lastname, firstname, balance
? Stone Sam 34.98
Enter account number
? 88
Enter lastname, firstname, balance
? Smith Dave 258.34
Enter account number
? 33
Enter lastname, firstname, balance
? Dunn Stacey 314.33
Enter account number
? 0
```

Fig. 11.13 Ejecución de muestra del programa de la figura 11.12.

posiciona el apuntador de posición de archivo para el archivo referenciado por **cfPtr**, a la posición de bytes calculada por **(accountNum - 1) * sizeof(struct clientData)**; el valor de esta expresión se conoce como el *desplazamiento*. Dado que el número de cuenta está entre 1 y 100, pero las posiciones de bytes en el archivo empiezan con **0**, al calcular la posición de bytes dentro del registro se resta 1 del número de cuenta. Entonces, para el registro 1, el apuntador de posición de archivo se define al byte 0 del archivo. La constante simbólica **SEEK_SET** indica que el apuntador de posición de archivo está colocado en relación con el principio del archivo, en la cantidad del desplazamiento. Como indica el enunciado anterior, una búsqueda para el número de cuenta 1 en el archivo define el apuntador de posición de archivo al principio del archivo, porque la posición de bytes calculada es 0. La figura 11.14 ilustra el apuntador de archivo que hace referencia a una estructura **FILE** en memoria. El apuntador de posición de archivo indica que el siguiente byte a leerse o escribirse está a 5 bytes del principio del archivo.

El estándar ANSI muestra la función prototipo para **fseek** como

```
int fseek(FILE *stream, long int offset, int whence);
```

donde **offset** es el número de bytes a partir de la posición **whence** en el archivo al cual apunta **stream**. El argumento **whence** puede tener uno de tres valores —**SEEK_SET**, **SEEK_CUR** o **SEEK_END**— indicando la posición en el archivo a partir del cual se inicia la búsqueda. **SEEK_SET** indica que la búsqueda se inicia al principio del archivo; **SEEK_CUR** indica que la búsqueda se inicia en la posición actual en el archivo; y **SEEK_END** indica que la búsqueda se inicia en el final del archivo. Estas tres constantes simbólicas se definen en el archivo de cabecera **stdio.h**.

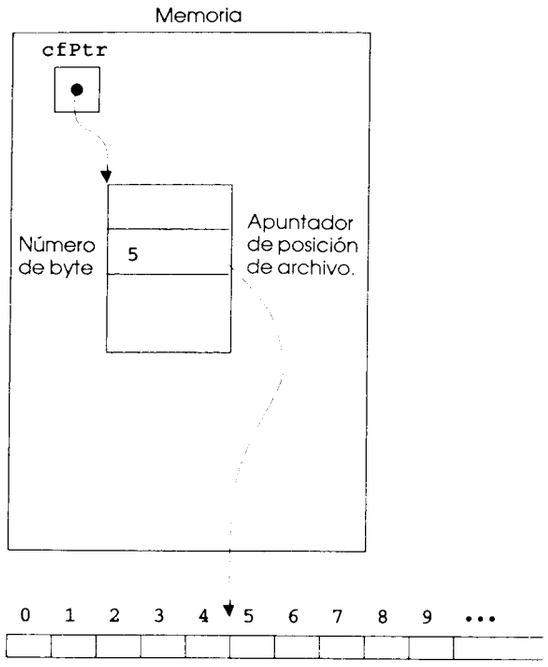


Fig. 11.14 Apuntador de posición de archivo, indicando un desplazamiento de 5 bytes a partir del principio del archivo.

11.9 Cómo leer datos directamente de un archivo de acceso directo

La función **fread** lee un número especificado de bytes de un archivo a la memoria. Por ejemplo, el enunciado

```
fread(&client, sizeof(struct clientData), 1, cfPtr);
```

lee el número de bytes determinado por **sizeof(struct clientData)** correspondiente al archivo referenciado por **cfPtr** y almacena el dato en la estructura **client**. Los bytes son leídos de la posición en el archivo especificado por el apuntador de posición de archivo. La función **fread** puede ser utilizada para leer varios elementos de arreglo de tamaño fijo, proporcionando un apuntador al arreglo en el cual los elementos se almacenarán, e indicando el número de elementos a leerse. El enunciado anterior especifica que un elemento deberá ser leído. Para poder leer más de un elemento, especifique el número de elementos en el tercer argumento del enunciado **fread**.

El programa de la figura 11.15 lee en forma secuencial todos los registros en el archivo **"credit.dat"**, determina si cada uno de dichos registros contiene datos, e imprime los datos con formato correspondientes a los registros que contengan datos. La función **feof** determina cuando se alcanza el fin de archivo, y la función **fread** transfiere los datos del disco a la estructura **client** de **clientData**.


```
/* Reading a random access file sequentially */
#include <stdio.h>

struct clientData {
   int acctNum;
   char lastName[15];
   char firstName[10];
   float balance;
};

main()
{
   FILE *cfPtr;
   struct clientData client;

   if ((cfPtr = fopen("credit.dat", "r")) == NULL)
      printf("File could not be opened.\n");
   else {
      printf("%-6s%-16s%-11s%10s\n", "Acct", "Last Name",
             "First Name", "Balance");

      while (!feof(cfPtr)) {
         fread(&client, sizeof(struct clientData), 1, cfPtr);

         if (client.acctNum != 0)
            printf("%-6d%-16s%-11s%10.2f\n",
                   client.acctNum, client.lastName,
                   client.firstName, client.balance);
      }
   }

   fclose(cfPtr);

   return 0;
}
```

```
Acct  Last Name      First Name  Balance
29    Brown          Nancy        -24.54
33    Dunn           Stacey       314.33
37    Barker         Doug           0.00
88    Smith          Dave         258.34
96    Stone          Sam           34.98
```

Fig. 11.5 Cómo leer secuencialmente un archivo de acceso directo.

11.10 Estudio de caso: un programa de procesamiento de transacciones

Ahora presentamos un programa sustancial de procesamiento de transacciones, que utiliza archivos de acceso directo. El programa mantiene información de cuentas de un banco. El programa actualiza las cuentas existentes, añade cuentas nuevas, borra cuentas y almacena un enlistado de todas las cuentas actuales, en un archivo de texto para su impresión. Suponemos que el programa de la figura 11.11 ha sido ejecutado para crear el archivo **credit.dat**.

El programa tiene cinco opciones. La opción 1 llama a la función **textFile** para almacenar una lista con formato de todas las cuentas en un archivo de texto llamado **accounts.txt**, que pudiera ser impreso más adelante. La función utiliza **fread** y las técnicas de acceso secuencial a archivo utilizadas en el programa de la figura 11.15. Después de seleccionar la opción 1 el archivo **accounts.txt** contiene:

```
Acct  Last Name      First Nam      Balance
29    Brown          Nancy           -24.54
33    Dunn           Stacey          314.33
37    Barker         Doug              0.00
88    Smith          Dave            258.34
96    Stone          Sam              34.98
```

La opción 2 llama a la función **updateRecord** para actualizar una cuenta. La función únicamente actualizará un registro ya existente, por lo que la función primero averiguará si el registro especificado por el usuario está vacío. El registro se lee a la estructura **client** utilizando a **fread**, y a continuación se utiliza **strcmp** para determinar si es **NULL** el miembro **lastName** de la estructura **client**. De ser así, el registro no contiene información, y se imprime un mensaje indicando que el registro está vacío. A continuación se despliegan las selecciones de menú. Si el registro contiene información, la función **updateRecord** introduce la cantidad de transacción, calcula el nuevo saldo, y vuelve a escribir el registro al archivo. Una salida típica correspondiente a la opción 2 es:

```
Enter account to update (1 - 100): 37
37  Barker          Doug            0.00

Enter charge (+) or payment (-): +87.99
37  Barker          Doug           87.99
```

La opción 3 llama a la función **newRecord** a fin de añadir una nueva cuenta al archivo. Si el usuario introduce un número de cuenta correspondiente a una cuenta existente, **newRecord** despliega un mensaje de error, indicando que el registro ya contiene información, y las selecciones de menú se vuelven a imprimir. Esta función utiliza para añadir una nueva cuenta el mismo proceso que en el programa de la figura 11.12. Una salida típica para la opción 3 es

```
Enter new account number (1 - 100): 22
Enter lastname, firstname, balance
? Johnston Sarah 247.45
```

La opción 4 llama a la función **deleteRecord** para borrar un registro del archivo. El borrado se lleva a cabo solicitándole al usuario el número de cuenta y volviendo a inicializar el registro. Si la cuenta no contiene información, **deleteRecord** despliega un mensaje de error, indicando que la cuenta no existe. La opción 5 termina la ejecución del programa. El programa se muestra en la figura 11.16. Advierta que el archivo **"credit.dat"** se abre para actualizar (lectura y escritura) mediante el modo **"r+"**.


```
/* This program reads a random access file sequentially, *
 * updates data already written to the file, creates new *
 * data to be placed in the file, and deletes data       *
 * already in the file.                                  */

#include <stdio.h>

struct clientData {
   int acctNum;
   char lastName[15];
   char firstName[10];
   float balance;
};

int enterChoice(void);
void textFile(FILE *);
void updateRecord(FILE *);
void newRecord(FILE *);
void deleteRecord(FILE *);

main()
{
   FILE *cfPtr;
   int choice;

   if ((cfPtr = fopen("credit.dat", "r+")) == NULL)
      printf("File could not be opened.\n");
   else {

      while ((choice = enterChoice()) != 5) {

         switch (choice) {
            case 1:
               textFile(cfPtr);
               break;
            case 2:
               updateRecord(cfPtr);
               break;
            case 3:
               newRecord(cfPtr);
               break;
            case 4:
               deleteRecord(cfPtr);
               break;
         }

      }
   }

   fclose(cfPtr);
   return 0;
}
```

Fig. 11.16 Programa de cuentas de banco (parte 1 de 4).

```
void textFile(FILE *readPtr)
{
   FILE *writePtr;
   struct clientData client;

   if ((writePtr = fopen("accounts.txt", "w")) == NULL)
      printf("File could not be opened.\n");
   else {
      rewind(readPtr);
      fprintf(writePtr, "%-6s%-16s%-11s%10s\n",
              "Acct", "Last Name", "First Name","Balance");

      while (!feof(readPtr)) {
         fread(&client, sizeof(struct clientData), 1, readPtr);

         if (client.acctNum != 0)
            fprintf(writePtr, "%-6d%-16s%-11s%10.2f\n",
                    client.acctNum, client.lastName,
                    client.firstName, client.balance);
      }
   }

   fclose(writePtr);
}

void updateRecord(FILE *fPtr)
{
   int account;
   float transaction;
   struct clientData client;

   printf("Enter account to update (1 - 100): ");
   scanf("%d", &account);
   fseek(fPtr, (account - 1) * sizeof(struct clientData),
         SEEK_SET);
   fread(&client, sizeof(struct clientData), 1, fPtr);

   if (client.acctNum == 0)
      printf("Acount #%d has no information.\n", account);
   else {
      printf("%-6d%-16s%-11s%10.2f\n\n",
             client.acctNum, client.lastName,
             client.firstName, client.balance);
      printf("Enter charge (+) or payment (-): ");
      scanf("%f", &transaction);
      client.balance += transaction;
      printf("%-6d%-16s%-11s%10.2f\n",
             client.acctNum, client.lastName,
             client.firstName, client.balance);
      fseek(fPtr, (account - 1) * sizeof(struct clientData),
            SEEK_SET);
      fwrite(&client, sizeof(struct clientData), 1, fPtr);
   }
}
```

Fig. 11.16 Programa de cuentas de banco (parte 2 de 4).


```
void deleteRecord(FILE *fPtr)
{
   struct clientData client, blankClient = {0, "", "", 0};
   int accountNum;

   printf("Enter account number to delete (1 - 100): ");
   scanf("%d", &accountNum);
   fseek(fPtr, (accountNum - 1) * sizeof(struct clientData),
         SEEK_SET);
   fread(&client, sizeof(struct clientData), 1, fPtr);

   if (client.acctNum == 0)
      printf("Account %d does not exist.\n", accountNum);
   else {
      fseek(fPtr, (accountNum - 1) * sizeof(struct clientData),
            SEEK_SET);
      fwrite(&blankClient, sizeof(struct clientData), 1, fPtr);
   }
}

void newRecord(FILE *fPtr)
{
   struct clientData client;
   int accountNum;
   printf("Enter new account number (1 - 100): ");
   scanf("%d", &accountNum);
   fseek(fPtr, (accountNum - 1) * sizeof(struct clientData),
         SEEK_SET);
   fread(&client, sizeof(struct clientData), 1, fPtr);

   if (client.acctNum != 0)
      printf("Account #%d already contains information.\n",
             client.acctNum);
   else {
      printf("Enter lastname, firstname, balance\n? ");
      scanf("%s%s%f", &client.lastName, &client.firstName,
            &client.balance);
      client.acctNum = accountNum;
      fseek(fPtr, (client.acctNum - 1) *
            sizeof(struct clientData), SEEK_SET);
      fwrite(&client, sizeof(struct clientData), 1, fPtr);
   }
}
```

Fig. 11.16 Programa de cuentas de banco (parte 3 de 4).

Resumen

- Todos los elementos de datos procesados por una computadora se reducen a combinaciones de ceros y de unos.
- En una computadora el elemento más pequeño de datos puede asumir el valor `0` o el valor `1`. Este elemento de datos se conoce como un bit (abreviatura de "binary digit" —un dígito que puede asumir uno de dos valores).

```
int enterChoice(void)
{
   int menuChoice;

   printf("\nEnter your choice\n"
          "1 - store a formatted text file of acounts called\n"
          "    \"accounts.txt\" for printing\n"
          "2 - update an account\n"
          "3 - add a new account\n"
          "4 - delete an account\n"
          "5 - end program\n? ");
   scanf("%d", &menuChoice);
   return menuChoice;
}
```

Fig. 11.16 Programa de cuentas de banco (parte 4 de 4).

- Los dígitos, las letras y los símbolos especiales se conocen como caracteres. El conjunto de todos los caracteres que pueden ser utilizados para escribir programas y representar elementos de datos en una computadora en particular se conoce como el conjunto de caracteres de la computadora. Cada uno de los caracteres en el conjunto de caracteres de la computadora se representa como un patrón de ocho `1`s y `0`s (conocido como un byte).
- Un campo es un grupo de caracteres que contiene significado.
- Un registro es un grupo de campos relacionados.
- Por lo menos un campo es seleccionado normalmente en cada registro como registro clave. El registro clave identifica un registro como perteneciente a una persona o entidad particular.
- El tipo de organización más popular para registros en un archivo, se conoce como archivo de acceso secuencial, en el cual se tiene acceso a registros en forma consecutiva hasta que son localizados los datos deseados.
- Un grupo de archivos relacionados a veces se llama una base de datos. Un conjunto o colección de programas diseñados para crear y administrar bases de datos se conoce como un sistema de administración de bases de datos (DBMS, por database management system).
- C considera cada archivo como simplemente un flujo secuencial de bytes.
- Al empezar la ejecución de un programa C abre automáticamente tres archivos y sus flujos asociados —entrada estándar, salida estándar y error estándar.
- Los apuntadores de archivo asignados a la entrada estándar, a la salida estándar y al error estándar son **`stdin`**, **`stdout`** y **`stderr`**, respectivamente.
- La función **`fgetc`** lee un carácter a partir de un archivo especificado.
- La función **`fputc`** escribe un carácter a un archivo especificado.
- La función **`fgets`** lee una línea de un archivo especificado.
- La función **`fputs`** escribe una línea en un archivo especificado.
- **`FILE`** es un tipo de estructura definida en el archivo de cabecera **`stdio.h`**. Para poder utilizar archivo el programador no necesita saber los detalles específicos de esta estructura. Conforme se abre un archivo, se regresa un apuntador a la estructura **`FILE`** del archivo.

- La función **`fopen`** toma dos argumentos— un nombre de archivo y un modo del archivo abierto— y abre el archivo. Si el archivo existe, el contenido del archivo se descarta sin advertencia. Si el archivo no existe y el archivo está siendo abierto para escribir, **`fopen`** crea el archivo.
- La función **`feof`** determina si ha sido definido el indicador de fin de archivo para el mismo.
- La función **`fprintf`** es equivalente a **`printf`**, excepto que **`printf`** recibe como argumento un apuntador al archivo hacia el cual se escribirán los datos.
- La función **`fclose`** cierra el archivo al cual apunta su argumento.
- Para crear un archivo, o para descartar el contenido de un archivo antes de escribir datos, abra el archivo para escritura (**`"w"`**). Para leer un archivo existente, ábralo para lectura (**`"r"`**). Para añadir registros al final de un archivo existente, abra el archivo para agregar (**`"a"`**). Para abrir un archivo de tal forma que pueda ser escrito y leído, abra el archivo para actualizar en alguno de los tres modos de actualización —**`"r+"`**, **`"w+"`** o **`"a+"`**. El modo **`"r+"`** simplemente abre el archivo para lectura y escritura. El modo **`"w+"`** crea el archivo si no existe, y si existe descarta el contenido actual del archivo. En el modo **`"a+"`** se crea el archivo si no existe, y la escritura se efectúa al final del archivo.
- La función **`fscanf`** es equivalente a **`scanf`**, excepto que **`fscanf`** recibe como argumento un apuntador al archivo (normalmente distinto a **`stdin`**) a partir del cual se leerán los datos.
- La función **`rewind`** hace que el programa recoloque el apuntador de posición del archivo correspondiente al archivo especificado al principio del mismo.
- Se utiliza el procesamiento de acceso directo de archivos para tener acceso.directo a registros.
- Para facilitar el acceso directo, los datos se almacenan en registros de longitud fija. Dado que todos los registros son de la misma longitud, la computadora puede rápidamente calcular (como una función del registro clave) la posición exacta de un registro en relación con el principio del archivo.
- Los datos pueden ser añadidos fácilmente a un archivo de acceso directo sin destruir otros datos en el archivo. Los datos previamente almacenados en un archivo con registros de longitud fija, también pueden ser modificados y borrados sin tener que reescribir todo el archivo.
- La función **`fwrite`** escribe un bloque (número específico de bytes) de datos a un archivo.
- El operador en tiempo de compilación **`sizeof`** regresa el tamaño en bytes de su operando.
- La función **`fseek`** establece el apuntador de posición de archivo a una posición específica, en un archivo basado en la posición inicial del punto de búsqueda en el archivo. La búsqueda puede iniciarse a partir de una de tres posiciones —**`SEEK_SET`** inicia a partir del principio del archivo, **`SEEK_CUR`** inicia a partir de la posición actual en el archivo y **`SEEK_END`** inicia a partir del fin del archivo.
- La función **`fread`** lee un bloque (número específico de bytes) de datos de un archivo.

Terminología

modo de archivo abierto **`a`**
modo de archivo abierto **`a+`**
campo alfabético
campo alfanumérico
orden alfa
dígito binario
bit
byte

carácter
campo de carácter
conjunto de caracteres
cerrar un archivo
jerarquía de datos
base de datos
sistema de administración de base de datos
dígito decimal
desplazamiento
número de doble precisión
fin de archivo
indicador de fin de archivo
`fclose`
`feof`
`fgetc`
`fgets`
campo
archivo
almacenamiento temporal de archivo
nombre de archivo
modo de archivo abierto
apuntador de archivo
apuntador de posición de archivo
estructura `FILE`
`fopen`
entrada/salida con formato
`fprintf`
`fputc`
`fputs`
`fread`
`fscanf`
`fseek`
`fwrite`
número entero
espacios a la izquierda
letra
campo numérico
desplazamiento
abrir un archivo
modo de archivo abierto `r`
acceso directo
archivo de acceso directo
registro
registro clave
parámetro del número de registro
`rewind`
modo de archivo abierto `r+`
`SEEK_CUR`
`SEEK_END`
`SEEK_SET`
archivo de acceso secuencial
número de una sola precisión
`stderr` (error estándar)
`stdin` (entrada estándar)
`stdout` (salida estándar)
flujo
espacios a la derecha
modo de archivo abierto `w`
modo de archivo abierto `w+`
ceros y unos

Errores comunes de programación

11.1 Abrir un archivo existente para escritura (`"w"`) cuando, de hecho, el usuario desea conservar el archivo; el contenido del archivo se descartará sin advertencia.
11.2 Olvidar abrir un archivo antes de intentar hacer referencia a él en un programa.
11.3 Usar el apuntador de archivo incorrecto para referirse a un archivo.
11.4 Abrir para lectura un archivo no existente.
11.5 Abrir un archivo para lectura o escritura sin haber obtenido los derechos de acceso al archivo apropiados (esto depende del sistema operativo).
11.6 Abrir un archivo para escritura cuando no hay espacio disponible en disco. En la figura 11.6 se listan los modos de apertura de archivos.
11.7 Puede llevar a errores devastadores abrir un archivo utilizando el modo de archivo incorrecto. Por ejemplo, abrir un archivo en el modo de escribir (`"w"`) cuando debería haberse abierto en modo de actualizar (`"r+"`) hace que sea descartado todo el contenido del archivo.

Prácticas sanas de programación

11.1 Asegúrese que en un programa las llamadas a las funciones de procesamiento de archivos contienen los apuntadores de archivo correctos.

11.2 Cierre cada archivo en forma explícita, tan pronto sepa que el programa ya no hará otra vez referencia al archivo.
11.3 Abra un archivo únicamente para lectura (y no para actualizar), si el contenido del archivo no debe modificarse. Esto evitará cambios no intencionales en el contenido del archivo. Este es otro ejemplo del principio del mínimo privilegio.

Sugerencia de rendimiento

11.1 Cerrar un archivo puede liberar recursos que están siendo esperados por otros usuarios o programas.
11.2 Muchos programadores piensan erróneamente que `sizeof` es una función, y que utilizándolo se genera una sobrecarga en tiempo de ejecución de una llamada de función. No existe tal sobrecarga, porque `sizeof` es un operador en tiempo de compilación.

Sugerencia de portabilidad

11.1 La estructura `FILE` depende del sistema operativo (es decir, los miembros de la estructura varían de un sistema a otro, según la forma en que cada sistema maneja sus archivos).

Ejercicios de autoevaluación

11.1 Llene los espacios vacíos en cada uno de los siguientes:
a) En última instancia, todos los elementos de datos procesados en una computadora se reducen a combinaciones de _____ y _____.
b) El elemento de datos más pequeño que puede procesar una computadora se conoce como un _____.
c) Un _____ es un grupo de registros relacionados.
d) Dígitos, letras y símbolos especiales se conocen como _____.
e) Un grupo de archivos relacionados se llama una _____.
f) La función _____ cierra un archivo.
g) El enunciado _____ lee datos de un archivo en una forma similar a la forma en que `scanf` lee a partir de `stdin`.
h) La función _____ lee un carácter de un archivo especificado.
i) La función _____ lee una línea de un archivo especificado.
j) La función _____ abre un archivo.
k) La función _____ se utiliza normalmente para leer datos de un archivo en aplicaciones de acceso directo.
l) La función _____ recoloca el apuntador de posición de archivo a una posición específica dentro del archivo.

11.2 Indique cuáles de los siguientes son verdaderos y cuáles son falsos (para aquellos que son falsos, explique por qué).
a) La función `fscanf` no puede ser utilizada para leer datos de la entrada estándar.
b) El programador debe utilizar `fopen` explícitamente, para abrir los flujos de entrada estándar, salida estándar y error estándar.
c) Para cerrar un archivo un programa debe llamar en forma explícita a la función `fclose`.
d) Si el apuntador de posición de archivo apunta a una posición en un archivo secuencial distinto al principio del mismo, el archivo debe de ser cerrado y vuelto a abrir para leer a partir del principio del mismo.
e) La función `fprintf` puede escribir a la salida estándar.
f) Los datos en los archivos de acceso secuencial se actualizan siempre sin sobreescribir otros datos.

g) No es necesario buscar en todos los registros de un archivo de acceso directo para encontrar un registro específico.
h) Los registros en archivos de acceso directo no son de longitud uniforme.
i) La función **fseek** puede buscar únicamente en relación con el principio de un archivo.

11.3 Escriba un solo enunciado para que se ejecuten cada uno de los siguientes. Suponga que cada uno de estos enunciados se aplica al mismo programa.

a) Escriba un enunciado que abra el archivo `"oldmast.dat"` para lectura y asigne el apuntador de archivo regresado a `ofPtr`.
b) Escriba un enunciado que abra el archivo `"trans.dat"` para lectura y asigne el apuntador de archivo regresado a `tfPtr`.
c) Escriba un enunciado que abra el archivo `"newmast.dat"` para escritura (y creación) y asigne el apuntador de archivo regresado a `nfPtr`.
d) Escriba un enunciado que lea un registro del archivo `"oldmast.dat"`. El registro está formado del entero `accountNum`, de la cadena `Name`, y del punto flotante `currentBalance`.
e) Escriba un enunciado que lea un registro del archivo `"trans.dat"`. El registro está formado del entero `accountNum` y del punto flotante `dollarAmount`.
f) Escriba un enunciado que escriba un registro al archivo `"newmast.dat"`. El registro está formado del entero `accountNum`, de la cadena `Name`, y del punto flotante `currentBalance`.

11.4 Encuentre el error en cada uno de los siguientes segmentos de programa. Explique cómo se puede corregir dicho error.

a) El archivo referido por `fPtr` (`"payables.dat"`) no ha sido abierto.
```
fprintf(fPtr, "%d%s%d\n", account, company, amount);
```
b)
```
open("receive.dat", "r+");
```
c) El siguiente enunciado debería leer un registro del archivo `"payables.dat"`. El apuntador de archivo `payPtr` se refiere a este archivo, y el apuntador de archivo `recPtr` se refiere al archivo `"receive.dat"`
```
fscanf(recPtr, "%d%s%d\n", &account, company, &amount);
```
d) El archivo `"tools.dat"` debería ser abierto para añadir datos al archivo, sin descartar los datos actuales.
```
if ((tfPtr = fopen("tools.dat", "w")) != NULL)
```
e) El archivo `"courses.dat"` debería ser abierto para agregar sin modificar el contenido actual del archivo.
```
if ((cfPtr = fopen("courses.dat", "w+")) != NULL)
```

Respuestas a los ejercicios de autoevaluación

11.1 a) 1s, 0s. b) Bit, c) Archivo. d) Caracteres. e) Bases de datos f) `fclose`. g) `fscanf`. h) `getc` o bien `fgetc`. i) `fgets`. j) `fopen`. k) `fread`. l)`fseek`.

11.2 a) Falso. La función `fscanf` sólo puede ser utilizada para leer a partir de la entrada estándar incluyendo `stdin`, el apuntador al flujo estándar de entrada, en la llamada a `fscanf`.
b) Falso. Estos tres flujos son automáticamente abiertos por C cuando se inicia la ejecución del programa.
c) Falso. Los archivos serán terminados cuando se termine la ejecución del programa, pero todos los archivos deberían ser cerrados en forma explícita utilizando `fclose`.
d) Falso. La función `rewind` puede ser utilizada para volver a colocar el apuntador de posición de archivo al principio del archivo.
e) Verdadero.

f) Falso. En la mayor parte de los casos, los registros de archivos secuenciales no son de longitud uniforme. Por lo tanto, es posible que al actualizar un registro se cause la sobreescritura de otros datos.
g) Verdadero.
h) Falso. Los registros en un archivo de acceso directo normalmente son de longitud uniforme.
i) Falso. Es posible buscar a partir del principio del archivo, a partir del fin del archivo y a partir de la posición actual en el archivo, de acuerdo con el apuntador de posición de archivo.

11.3
a)
```
ofPtr = fopen("oldmast.dat", "r");
```
b)
```
tfPtr = fopen("trans.dat", "r");
```
c)
```
nfPtr = fopen("newmast.dat", "w");
```
d)
```
fscanf(ofPtr, "%d%s%f", &accountNum, name, &currentBalance);
```
e)
```
fscanf(tfPtr, "%d%f", &accountNum, &dollarAmount);
```
f)
```
fprintf(nfPtr, "%d%s%.2f", accountNum, name, currentBalance);
```

11.4 a) Error: el archivo `"payables.dat"` no ha sido abierto antes de hacer referencia a su apuntador de archivo.
Corrección: utilice `fopen` para abrir `"payables.dat"` para escritura, agregar o actualizar.
b) Error: la función `open` no es una función de ANSI C.
Corrección: utilice la función `fopen`.
c) Error: El enunciado `fscanf` utiliza el apuntador de archivo incorrecto para referirse al archivo `"payables.dat"`.
Corrección: utilice el apuntador de archivo `payPtr` para referirse a `"payables.dat"`.
d) Error: el contenido del archivo será descartado, porque el archivo ha sido abierto para escritura (`"w"`).
Corrección: para añadir datos al archivo, abra el archivo ya sea para actualizar (`"r+"`) o abra el archivo para agregar (`"a"`).
e) Error: el archivo `"courses.dat"` está abierto para actualizar en modo `"w+"`, lo que hace que se descarte el contenido actual del archivo.
Corrección: Abra el archivo en modo `"a"`.

Ejercicios

11.5 Llene los espacios vacíos en cada uno de los siguientes:

a) Las computadoras almacenan grandes cantidades de datos en dispositivos de almacenamiento secundarios como son _____.
b) Un _____ está compuesto de varios campos.
c) Un campo que pudiera contener dígitos, letras y espacios se llama un campo de _____.
d) Para facilitar la recuperación de registros específicos a partir de un archivo, se selecciona un campo de cada registro como _____.
e) La gran mayoría de la información almacenada en sistemas de computación se archiva en archivos _____.
f) Un grupo de caracteres relacionados que contienen significados se conocen como un _____.
g) Los apuntadores de archivo para los tres archivos que se abren automáticamente por C al iniciarse la ejecución de un programa se llaman _____, _____, y _____.
h) La función _____ escribe un carácter a un archivo especificado.
i) La función _____ escribe una línea a un archivo especificado.
j) La función _____ se usa generalmente para escribir datos a un archivo de acceso directo.
k) La función _____ recoloca el apuntador de posición de archivo al principio del mismo.

11.6 Indique cuáles de los siguientes son verdaderos y cuáles son falsos (en el caso de los falsos, explique porqué):

a) Las funciones impresionantes ejecutadas por las computadoras involucran esencialmente el manejo de ceros y de unos.
b) Las personas prefieren manipular bits en vez de caracteres y campos, porque los bits son más compactos.
c) Las personas especifican elementos de programas y de datos como caracteres; las computadoras a continuación manipulan y procesan dichos caracteres como grupos de ceros y de unos.
d) El código postal de una persona es un ejemplo de un campo numérico.
e) En aplicaciones de computadora, la calle de la dirección de una persona se considera generalmente un campo alfabético.
f) Los elementos de datos procesados por una computadora forman una jerarquía de datos, en la cual los elementos de datos se hacen más grandes y más complejos conforme se avanza desde los campos a los caracteres, a los bits, etcétera.
g) Un registro clave identifica un registro como perteneciente a un campo particular.
h) La mayor parte de las organizaciones almacenan toda su información en un solo archivo a fin de facilitar el procesamiento en computadora.
i) Los archivos son siempre referidos por nombre en los programas de C.
j) Cuando un programa genera un archivo, el archivo es automáticamente conservado por la computadora para referencia futura.

11.7 El ejercicio 11.3 le solicitó al lector que escribiera una serie de enunciados sencillos. De hecho, estos enunciados forman el núcleo de un tipo importante de programas de procesamiento de archivos, es decir, un programa de cotejo de archivos. En el procesamiento comercial de datos, es común tener varios archivos en cada sistema. Por ejemplo, en un sistema de cuentas por cobrar, generalmente existe un archivo maestro que contiene información detallada relativa a cada cliente como es nombre, dirección, número telefónico, saldo actual, límite de crédito, descuentos, arreglos contractuales, y posiblemente una historia condensada de las compras más recientes y de sus pagos.

Conforme ocurren transacciones (es decir, se efectúan ventas y llegan pagos por el correo), se introducen en un archivo. Al final de cada periodo de negocios (es decir, en algunas empresas un mes, una semana en otras y en algunos casos un día) el archivo de transacciones (llamado `"trans.dat"` en el ejercicio 11.3) se aplica al archivo maestro (llamado `"oldmast.dat"` del ejercicio 11.3), actualizando así el registro de cada cuenta en compras y pagos. Después de cada una de estas corridas de actualización, el archivo maestro se vuelve a escribir como un nuevo archivo (`"newmast.dat"`), que entonces se utiliza al fin del siguiente periodo de negocios, para volver a empezar el siguiente proceso de actualización.

Los programas de cotejo de archivos deben enfrentarse a ciertos problemas que no existen en programas de un solo archivo. Por ejemplo, no siempre ocurrirá una coincidencia. Un cliente existente en el archivo maestro pudiera, en el periodo de negocios actual, no haber efectuado ninguna compra o ningún pago y, por lo tanto, en el archivo de transacción no aparecerá en ningún registro para este cliente. Similarmente, un cliente que sí hizo algunas compras o pagos, quizás acaba de trasladarse a esta comunidad, y la compañía todavía no habrá tenido la oportunidad de crear un registro maestro para dicho cliente.

Utilice los enunciados escritos en el ejercicio 11.3 como base para escribir un programa completo para cotejar cuentas por cobrar. Utilice el número de cuenta en cada archivo como registro clave para cotejo. Suponga que cada archivo es un archivo secuencial con registros almacenados en orden creciente de número de cuenta.

Cuando ocurra una coincidencia (es decir, cuando registros con el mismo número de cuenta aparezcan tanto en el archivo maestro como en el archivo de transacción) añada la cantidad en dólares existente en el archivo de transacción al saldo actual existente en el archivo maestro, y escriba el registro `"newmast.dat"`. (Suponga que las adquisiciones o compras están indicadas por cantidades positivas en el archivo de transacción, y que los pagos están indicados por cantidades negativas.) Cuando exista un registro maestro relativo a una cuenta particular sin registro de transacción correspondiente, simplemente

escriba el registro maestro a `"newmast.dat"`. Cuando exista un registro de transacción sin el correspondiente registro maestro, imprima el mensaje `"Unmatched transaction record for account number ..."` (incluya el número de cuenta proveniente del registro de transacción).

11.8 Después de escribir el programa del ejercicio 11.7 escriba un programa sencillo para crear alguna información de prueba para verificar el programa del ejercicio 11.7. Utilice los datos de cuentas de muestras siguientes:

Archivo maestro Número de cuenta	Nombre	Balance
100	Alan Jones	348.17
300	Mary Smith	27.19
500	Sam Sharp	0.00
700	Suzy Green	-14.22

Archivo de transacción: Número de cuenta	Cantidad en dólares
100	27.14
300	62.11
400	100.56
900	82.17

11.9 Ejecute el programa del ejercicio 11.7 utilizando los archivos de los datos de prueba creados en el ejercicio 11.8 Utilice el programa enlistado de la sección 11.7 para imprimir un nuevo archivo maestro. Verifique cuidadosamente los resultados.

11.10 Es posible (y de hecho común) tener varios registros de transacciones para un mismo registro clave. Esto ocurre porque un cliente en particular durante un periodo de negocios pudiera efectuar varias compras y varios pagos. Vuelva a escribir su programa de cotejo de cuentas por cobrar del ejercicio 11.7 para que incluya la posibilidad de manejar varios registros de transacción con el mismo registro clave. Modifique los datos de prueba del ejercicio 11.8 para incluir los siguientes registros de transacción adicionales:

Número de cuenta	Cantidad en dólares
300	83.89
700	80.78
700	1.53

11.11 Escriba enunciados que lleven a cabo cada uno de los siguientes. Suponga que la estructura

```
struct person {
   char lastName[15];
   char firstName[15];
   char age[2];
}
```

ha sido definida, y que el archivo ya está abierto para escritura.

a) Inicialice el archivo `"nameage.dat"` de tal forma que existan 100 registros con `lastName = "unassigned"`, `firstname = ""`, y `age = "0"`
b) Introduzca `10` apellidos, nombres y edades y escríbalos al archivo.
c) Actualice un registro; si no existe información en el registro, indique al usuario `"No info"`.
d) Borre un registro que tenga información mediante la reinicialización de dicho registro en particular.

11.12 Usted es el propietario de una ferretería y necesita mantener un inventario que le pueda indicar cuáles son las herramientas que tiene, cuántas tiene y el costo de cada una. Escriba un programa que inicialice el archivo `"hardware.dat"` a 100 registros vacíos, que le permita introducir los datos correspondientes a cada herramienta, enliste todas sus herramientas, borrar un registro correspondiente a una herramienta que ya no posea, y le deje actualizar *cualquier* información dentro del archivo. El número de identificación de la herramienta deberá ser el número de registro. Utilice la siguiente información para iniciar su archivo:

Registro #	Nombre de la herramienta	Cantidad	Costo
3	Electric sander	7	57.98
17	Hammer	76	11.99
24	Jig saw	21	11.00
39	Lawn mower	3	79.50
56	Power saw	18	99.99
68	Screwdriver	106	6.99
77	Sledge hammer	11	21.50
83	Wrench	34	7.50

11.13 ***Generador de palabras de números telefónicos.*** Los marcadores telefónicos estándar contienen los dígitos 0 al 9. Los números 2 hasta el 9 cada uno de ellos tiene tres letras asociadas, tal y como se indican en la tabla siguiente:

Dígito	Letras
2	A B C
3	D E F
4	G H I
5	J K L
6	M N O
7	P R S
8	T U V
9	W X Y

Muchas personas encuentran difícil memorizar los números telefónicos, por lo que utilizan la correspondencia existente entre dígitos y letras para desarrollar palabras de siete letras, que correspondan a sus números telefónicos. Por ejemplo, una persona cuyo número telefónico es 686-2377 pudiera utilizar la correspondencia indicada en la tabla precedente, para desarrollar la palabra de siete letras "NUMBERS".

Los negocios frecuentemente intentan obtener números telefónicos que sean fáciles de recordar para sus clientes. Si un negocio puede anunciar una sola palabra para que la marquen sus clientes, entonces sin duda dicho negocio recibirá unas cuantas llamadas más.

Cada palabra de siete letras corresponde a exactamente un número telefónico de siete dígitos. El restaurante que desee aumentar su negocio de venta de comidas para llevar, podría seguramente hacerlo con el número 825-3688 (es decir "TAKEOUT").

Cada número telefónico de 7 dígitos corresponde a muchas palabras distintas de 7 letras. Desafortunadamente, la mayor parte de ellas representan yuxtaposiciones irreconocibles de letras. Es posible, sin embargo, que el propietario de una peluquería se sintiera complacido al saber que el número telefónico de su tienda, 424-7288 corresponde a "HAIRCUT". El propietario de una tienda de licores estaría encantado, sin duda, al averiguar que el número telefónico de la tienda 233-7226, corresponde a "BEERCAN". Un médico veterinario con el número telefónico 738-2273 estaría complacido al aprender que el número corresponde a las letras "PETCARE".

Escriba un programa en C que, dado un número de 7 dígitos, escriba a un archivo todas las palabras posibles de 7 letras que correspondan a dicho número. Existen 2187 (3 a la séptima potencia) palabras posibles. Evite números telefónicos que contengan los dígitos 0 y 1.

11.14 Si tiene disponible un diccionario computarizado, modifique el programa que escribió en el Ejercicio 11.13 para buscar las palabras en el diccionario. Algunas combinaciones de siete letras creadas por este programa consisten de dos o más palabras (el número telefónico 843-2677 produce "THEBOSS").

11.15 Modifique el ejemplo de la figura 8.14 para utilizar las funciones `fgetc` y `fputs`, en vez de `getchar` y `puts`. El programa deberá darle al usuario la opción de leer desde la entrada estándar y escribir a la salida estándar, o de leer de un archivo especificado y escribir a un archivo especificado. Si el usuario se decide por la segunda opción, haga que el usuario introduzca los nombres de archivo correspondientes a los archivos de entrada y de salida.

11.16 Escriba un programa que utilice el operador `sizeof` para determinar los tamaños en bytes de varios tipos de datos en su sistema de computación. Escriba los resultados al archivo `"datasize.dat"` de tal forma que más tarde se puedan imprimir los resultados. El formato para los resultados del archivo deberá ser:

```
Data type             Size
char                     1
unsigned char            1
short int                2
unsigned short int       2
int                      4
unsigned int             4
long int                 4
unsigned long int        4
float                    4
double                   8
long double             16
```

Nota: los tamaños de tipo en su computadora pudieran no ser iguales a los arriba listados.

11.17 En el ejercicio 7.19, usted escribió una simulación en software de una computadora, que utilizaba un lenguaje máquina especial llamado lenguaje de máquina Simpletron (LMS). En la simulación, cada vez que usted deseaba ejecutar un programa LMS, usted introducía el programa en el simulador a partir del teclado. Si al capturar el programa LMS cometía algún error, el simulador se volvía a iniciar y el código LMS se volvía a introducir. Sería agradable, en vez de tener que escribirlo cada vez, poder leer el programa LMS a partir de un archivo. Esto reduciría tiempo y errores en la preparación de la ejecución de programas LMS.

a) Modifique el simulador que escribió en el ejercicio 7.19 para leer programas LMS a partir de un archivo especificado por el usuario desde el teclado.
b) Una vez que Simpletron se ejecuta, saca el contenido de sus registros y de memoria a la pantalla. Sería también deseable capturar la salida en un archivo, por lo que modifique el simulador para escribir su salida a un archivo, además de que despliegue dicha salida en la pantalla.

12

Estructuras de datos

Objetivos

- Ser capaz de asignar y liberar memoria dinámicamente para objetos de datos.
- Ser capaz de formar estructuras de datos enlazadas mediante el uso de apuntadores, estructuras autorreferenciadas y recursión.
- Ser capaz de crear y manipular listas enlazadas, colas, pilas y árboles binarios.
- Comprender varias aplicaciones importantes de las estructuras de datos enlazadas.

Mucho de lo que sujeto, no puedo liberar;
Mucho que puedo liberar regresó a mí.
Lee Wilson Dodd

'¿Quieres andar un poco más rápido?' dijo una merluza a un caracol. 'Hay una tortuga detrás de nosotros, y me está pisando la cola'.
Lewis Carroll

Siempre hay lugar en la cima.
Daniel Webster

Empuja —mantente en movimiento.
Thomas Morton

Pienso que no veré jamás
Tan bello poema como es un árbol.
Joyce Kilmer

Sinopsis

Resumen • Terminología • Errores comunes de programación • Prácticas sanas de programación • Sugerencias de rendimiento • Sugerencias de portabilidad • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios.

12.1 Introducción

Hemos estudiado *estructuras de datos* de tamaño fijo, como arreglos de un solo subíndice, de doble subíndice y **structs**. Este capítulo presenta *estructuras dinámicas de datos,* cuyo tamaño crece y se encoge en tiempo de ejecución. Las *listas enlazadas* son colecciones de elementos de datos "alineados en una fila" —en una lista enlazada las inserciones y las eliminaciones se efectúan en cualquier parte. Las *pilas* son importantes en compiladores y sistemas operativos —en una pila las inserciones y las eliminaciones se efectúan únicamente en un extremo— en su *parte superior.* Las *colas de espera* representan líneas de espera; las inserciones se efectúan en la parte trasera (también conocida como la *cola*) de la misma, y las eliminaciones se hacen de la parte delantera (también conocida como cabeza de la cola). Los *árboles binarios* facilitan la búsqueda y clasificación de los datos a alta velocidad, la eliminación eficiente de elementos duplicados de datos, la representación de sistemas de directorios de archivo y las expresiones de compilación en lenguaje de máquina. Cada una de esas estructuras de datos tiene muchas otras interesantes aplicaciones.

Analizaremos cada uno de los tipos principales de estructuras de datos y pondremos en operación programas que crean y manipulan estas estructuras de datos. En la siguiente parte del libro —la introducción a **C++** y a la programación orientada a objetos, en los capítulos del 15 hasta el 21 —estudiaremos la abstracción de datos. Esta técnica nos permitirá construir estas estructuras de datos de una forma dramáticamente diferente, diseñada para producir software mucho más fácil de mantener y especialmente mucho más fácil de volver a utilizar.

Este es un capítulo retador. Los programas son sustanciales e incorporan la mayor parte de lo que ha aprendido en los capítulos anteriores. Estos programas están llenos de manipulaciones de apuntadores, un tema que muchas personas consideran de entre los temas más difíciles de C. El capítulo está lleno de programas altamente prácticos, que podrá utilizar en cursos más avanzados; el capítulo incluye una valiosa colección de ejercicios que enfatizan aplicaciones prácticas de las estructuras de datos.

Sinceramente esperamos que usted intentará el importante proyecto descrito en la sección especial titulada "Cómo construir su propio compilador". Usted ha estado utilizando un compilador para traducir sus programas C a lenguaje de máquina, a fin de poder ejecutar sus programas en su computadora. En este proyecto, realmente construirá su propio compilador. Este leerá un archivo de enunciados, escritos en un simple, aunque poderoso lenguaje de alto nivel, similar a las versiones primeras del popular lenguaje BASIC. Su compilador convertirá estos enunciados en un archivo de instrucciones de lenguaje de máquina Simpletron. LMS es el lenguaje que aprendió en la sección especial del capítulo 7, "Cómo construir su propia computadora". ¡A continuación su programa simulador Simpletron ejecutará el programa LMS producido por su compilador! Este proyecto le dará la maravillosa oportunidad de practicar la mayor parte de lo que en este curso ha aprendido. La sección especial lo lleva cuidadosamente a través de las especificaciones del lenguaje de alto nivel, y describe los algoritmos que necesitará para convertir cada tipo de enunciado del lenguaje de alto nivel a instrucciones en lenguaje máquina. Si usted es afecto a aceptar retos, pudiera intentar llevar a cabo las muchas mejoras, tanto al compilador como al simulador Simpletron que se sugieren en los ejercicios.

12.2 Estructuras autorreferenciadas

Una *estructura autorreferenciada* contiene un miembro de apuntador que apunta a una estructura del mismo tipo de estructura. Por ejemplo, la definición

```
struct node {
   int data;
   struct node *nextPtr;
};
```

define un tipo, **struct node**. Una estructura del tipo **struct node** tiene dos miembros —el miembro entero **data** y el miembro de apuntador **nextPtr**. El miembro **nextPtr** apunta a una estructura de tipo **struct node** —una estructura del mismo tipo que la que se está declarando aquí, de ahí el "término estructura autorreferenciada". El miembro **nextPtr** se conoce como un *enlace o vínculo* —es decir, **nextPtr** puede ser utilizada para "vincular" una estructura del tipo **struct node** con otra estructura del mismo tipo. Las estructuras autorreferenciadas pueden ser enlazadas juntas para formar útiles estructuras de datos como son las listas, las colas de espera, las pilas y los árboles. En la figura 12.1 se ilustran dos estructuras autorreferenciadas, enlazadas juntas para formar una lista. Note que se coloca una diagonal —que representa un apuntador **NULL**— en el miembro enlazado de la segunda estructura autorreferenciada, para indicar que el enlace no apunta a otra estructura. La diagonal aparece sólo para fines de ilustración; no corresponde al carácter de diagonal de C. Igual que el carácter **NULL** indica el final de una cadena, normalmente un apuntador **NULL** indica el fin de una estructura de datos.

Error común de programación 12.1

No establecer a NULL el enlace en el último nodo de una lista.

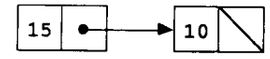


Fig. 12.1 Dos estructuras autorreferencladas enlazadas juntas.

12.3 Asignación dinámica de memoria

La creación y mantenimiento de estructuras dinámicas de datos, requiere de la *asignación dinámica de memoria* —la capacidad por parte de un programa de obtener, en tiempo de ejecución, más espacio de memoria para contener nuevos nodos, y de poder liberar espacio ya no requerido. El límite de la asignación dinámica de memoria pude ser tan grande como la totalidad de memoria física disponible en la computadora, o la cantidad de memoria virtual disponible, en un sistema de memoria virtual. A menudo, los límites son mucho menores, porque la memoria disponible deberá ser compartida entre muchos usuarios.

Las funciones **malloc** y **free** y el operador **sizeof**, son esenciales a la asignación dinámica de memoria. La función **malloc** toma como argumento el número de bytes a asignarse, y regresa un apuntador del tipo **void*** (*apuntador a void*) a la memoria asignada. Un apuntador **void*** puede asignarse a una variable de cualquier tipo de apuntador. Normalmente la función **malloc** se utiliza conjuntamente con el operador **sizeof**. Por ejemplo, el enunciado

```
newPtr = malloc(sizeof(struct node));
```

evalúa **sizeof(struct node)** para determinar el tamaño en bytes de una estructura del tipo **struct node**, asigna en memoria una nueva área de tamaño **sizeof(struct node)** bytes, y almacena en la variable **newPtr** un apuntador a la memoria asignada. Si no existe memoria disponible, **malloc** regresa un apuntador **NULL**.

La función **free** cancela la asignación de la memoria —es decir, se regresa la memoria al sistema, de tal forma que en el futuro ésta pueda ser vuelta a asignar. Para liberar memoria asignada dinámicamente mediante una llamada **malloc** previa, utilice el enunciado

```
free(newPtr);
```

En las siguientes secciones se analizan listas, pilas, colas de espera y árboles. Cada una de estas estructuras de datos se crea y se mantiene utilizando la asignación dinámica de memoria y las estructuras autorreferenciadas.

Sugerencia de portabilidad 12.1

El tamaño de una estructura no es necesariamente la suma de los tamaños de sus miembros. Esto es debido a varios requisitos de alineación de límites, que son dependientes de la máquina (vea el capítulo 10).

Error común de programación 12.2

Suponer que el tamaño de una estructura es simplemente la suma de los tamaños de sus miembros.

Práctica sana de programación 12.1

Para determinar el tamaño de una estructura utilice el operador `sizeof`.

Práctica sana de programación 12.2

Al utilizar **malloc**, *compruebe si es* **NULL** *el valor de regreso de apuntador. Si la memoria solicitada no ha sido asignada imprima un mensaje de error.*

Error común de programación 12.3

No regresar memoria dinámicamente asignada cuando ésta ya no es necesaria, puede hacer que el sistema se quede sin memoria prematuramente. Esto se conoce a veces como "fuga de memoria".

Práctica sana de programación 12.3

Cuando ya no se requiera memoria que fue dinámicamente asignada, utilice **free**, *para regresar esta memoria inmediatamente al sistema.*

Error común de programación 12.4

Utilizando **malloc**, *liberar memoria no dinámicamente asignada.*

Error común de programación 12.5

Referirse a memoria que ya ha sido liberada.

12.4 Listas enlazadas

Una *lista enlazada* es una colección lineal de estructuras autorreferenciadas llamadas *nodos*, conectadas por *enlaces* de apuntador —de ahí el término lista "enlazada". Se tiene acceso a una lista enlazada vía un apuntador al primer nodo de la lista. Se puede tener acceso a los nodos subsecuentes vía el apuntador de enlace almacenado en cada nodo. Por regla convencional, para marcar el fin de la lista, el apuntador de enlace, en el último nodo de una lista, se define a **NULL**. En una lista enlazada los datos se almacenan dinámicamente —cada nodo se crea conforme sea necesario. Un nodo puede contener datos de cualquier tipo, incluyendo otras **struct**. Las pilas y las colas de espera también son estructuras lineales de datos, y como veremos, son versiones restringidas de listas enlazadas. Los árboles son estructuras no lineales de datos.

Las listas de datos pueden ser almacenadas en arreglos, pero las listas enlazadas proporcionan varias ventajas. Una lista enlazada es apropiada cuando no es predecible de inmediato el número de elementos de datos a representarse en la estructura. Las listas enlazadas son dinámicas, por lo que conforme sea necesario la longitud de una lista puede aumentar o disminuir. Por su parte, el tamaño de un arreglo no puede ser modificado, porque la memoria del arreglo es asignada en tiempo de compilación. Los arreglos pueden llenarse. Las listas enlazadas sólo se llenan cuando el sistema no tiene suficiente memoria para satisfacer las solicitudes de asignación dinámica de almacenamiento.

Sugerencia de rendimiento 12.1

Podría declararse un arreglo que contenga más elementos que el número esperado de elementos de datos, pero esto puede desperdiciar memoria. En estas situaciones las listas enlazadas pueden obtener una mejor utilización de la memoria.

Las listas enlazadas pueden mantenerse en orden, insertando cada elemento nuevo en el punto apropiado dentro de la lista.

Sugerencia de rendimiento 12.2

Puede resultar muy tardado insertar y eliminar en un arreglo ya ordenado —deberán ser desplazados en forma apropiada. Todos los elementos que sigan al elemento insertado o borrado.

Sugerencia de rendimiento 12.3

Los elementos de un arreglo se almacenan en forma contigua en memoria. Esto permite acceso inmediato a cualquier arreglo del elemento, porque la dirección de cualquier elemento puede ser calculada directamente, basada en su posición en relación con el principio del arreglo. Las listas enlazadas no proporcionan un acceso inmediato como éste a sus elementos.

Normalmente, los nodos de las listas enlazadas no están almacenados en memoria en forma contigua. Sin embargo, lógicamente, los nodos de una lista enlazada aparecen como contiguos. En la figura 12.2 se ilustra una lista enlazada con varios nodos.

Sugerencia de rendimiento 12.4

Tratándose de estructuras de datos que crecen o se reducen en tiempo de ejecución, es posible ahorrar memoria utilizando asignación dinámica de memoria (en vez de los arreglos). Recuerde, sin embargo, que los apuntadores toman espacio, y que la asignación dinámica de memoria incurre en sobrecarga por las llamadas de función.

El programa de la figura 12.3 (cuya salida se muestra en la figura 12.4) manipula una lista de caracteres. El programa da dos opciones: 1) insertar un carácter en la lista en orden alfabético (función **insert**) y 2) borra un carácter de la lista (función **delete**). Este es un programa grande y complejo. Sigue un análisis detallado respecto al programa. El ejercicio 12.20 solicita al estudiante que ponga en operación una función recursiva, que imprima una lista al revés. En el ejercicio 12.21 se le pide al estudiante que ponga en marcha una función recursiva, que busque en una lista enlazada un elemento particular de datos.

Las dos funciones primarias de las listas enlazadas son **insert** y **delete**. La función **isEmpty** se conoce como una *función predicada* —no altera en forma alguna la lista; más bien determina si la lista está vacía (es decir, si el apuntador al primer nodo de la lista es **NULL**). Si la lista está vacía, se regresa **1**; de lo contrario se regresa **0**. La función **printList** imprime la lista.

En la lista los caracteres se insertan en orden alfabético. La función **insert** recibe la *dirección* de la lista y el carácter a insertarse. Es necesaria la dirección de la lista cuando se va a insertar un valor en el inicio de la lista. Proporcionar la dirección de la lista permite que se pueda modificar la lista (el apuntador al primer nodo de la lista) vía una llamada por referencia. Dado que la lista propiamente dicha es un apuntador (a su primer elemento), pasar la dirección de la lista crea un *apuntador a un apuntador* (es decir, una *doble indirección*). Este es un concepto complejo y requiere de cuidadosa programación. Los pasos para la inserción de un carácter en la lista son como sigue (vea la figura 12.5):

1) Crear un nodo llamando **malloc**, asignando a **newPtr** la dirección de la memoria asignada, asignando el carácter a insertarse a **newPtr->data**, y asignando **NULL** a **newPtr->nextPtr**.
2) Inicialice **previousPtr** a **NULL**, y **currentPtr** a ***sPtr** (el apuntador al inicio de la lista). Los apuntadores **previousPtr** y **currentPtr** se utilizan para almacenar las posiciones del nodo anterior al punto de inserción y del nodo posterior al punto de inserción.

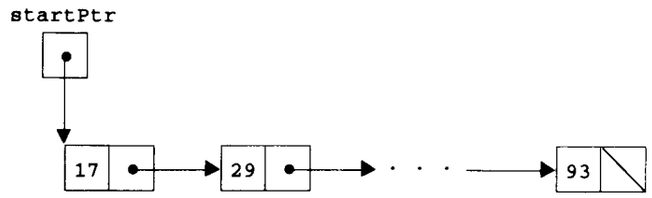


Fig. 12.2 Representación gráfica de una lista enlazada.


```
/* Operating and maintaining a list */
#include <stdio.h>
#include <stdlib.h>

struct listNode {  /* self-referential structure */
   char data;
   struct listNode *nextPtr;
};

typedef struct listNode LISTNODE;
typedef LISTNODE *LISTNODEPTR;

void insert(LISTNODEPTR *, char);
char delete(LISTNODEPTR *, char);
int isEmpty(LISTNODEPTR);
void printList(LISTNODEPTR);
void instructions(void);

main()
{
   LISTNODEPTR startPtr = NULL;
   int choice;
   char item;

   instructions();  /* display the menu */
   printf("? ");
   scanf("%d", &choice);

   while (choice != 3) {

      switch (choice) {
         case 1:
            printf("Enter a character: ");
            scanf("\n%c", &item);
            insert(&startPtr, item);
            printList(startPtr);
            break;
         case 2:
            if (!isEmpty(startPtr)) {
               printf("Enter character to be deleted: ");
               scanf("\n%c", &item);

               if (delete(&startPtr, item)) {
                  printf("%c deleted.\n", item);
                  printList(startPtr);
               }
               else
                  printf("%c not found.\n\n", item);
            }
            else
               printf("List is empty.\n\n");

            break;
```

Fig. 12.3 Cómo insertar y borrar nodos en una lista (parte 1 de 3).

```
            default:
               printf("Invalid choice.\n\n");
               instructions();
               break;
         }

         printf("? ");
         scanf("%d", &choice);
      }

      printf("End of run.\n");
      return 0;
   }

   /* Print the instructions */
   void instructions(void)
   {
      printf("Enter your choice:\n"
             "   1 to insert an element into the list.\n"
             "   2 to delete an element from the list.\n"
             "   3 to end.\n");
   }

   /* Insert a new value into the list in sorted order */
   void insert(LISTNODEPTR *sPtr, char value)
   {
      LISTNODEPTR newPtr, previousPtr, currentPtr;

      newPtr = malloc(sizeof(LISTNODE));

      if (newPtr != NULL) {    /* is space available */
         newPtr->data = value;
         newPtr->nextPtr = NULL;

         previousPtr = NULL;
         currentPtr = *sPtr;

         while (currentPtr != NULL && value > currentPtr->data) {
            previousPtr = currentPtr;          /* walk to ...   */
            currentPtr = currentPtr->nextPtr;  /* ... next node */
         }

         if (previousPtr == NULL) {
            newPtr->nextPtr = *sPtr;
            *sPtr = newPtr;
         }
         else {
            previousPtr->nextPtr = newPtr;
            newPtr->nextPtr = currentPtr;
         }
      }
      else
         printf("%c not inserted. No memory available.\n", value);
   }
```

Fig. 12.3 Cómo insertar y borrar nodos en una lista (parte 2 de 3).


```
   /* Delete a list element */
   char delete(LISTNODEPTR *sPtr, char value)
   {
      LISTNODEPTR previousPtr, currentPtr, tempPtr;

      if (value == (*sPtr)->data) {
         tempPtr = *sPtr;
         *sPtr = (*sPtr)->nextPtr;  /* de-thread the node */
         free(tempPtr);             /* free the de-threaded node */
         return value;
      }
      else {
         previousPtr = *sPtr;
         currentPtr = (*sPtr)->nextPtr;

         while (currentPtr != NULL && currentPtr->data != value) {
            previousPtr = currentPtr;          /* walk to ...   */
            currentPtr = currentPtr->nextPtr;  /* ... next node */
         }

         if (currentPtr != NULL) {
            tempPtr = currentPtr;
            previousPtr->nextPtr = currentPtr->nextPtr;
            free(tempPtr);
            return value;
         }
      }

      return '\0';
   }

   /* Return 1 if the list is empty, 0 otherwise */
   int isEmpty(LISTNODEPTR sPtr)
   {
      return sPtr == NULL;
   }

   /* Print the list */
   void printList(LISTNODEPTR currentPtr)
   {
      if (currentPtr == NULL)
         printf("List is empty.\n\n");
      else {
         printf("The list is:\n");

         while (currentPtr != NULL) {
            printf("%c --> ", currentPtr->data);
            currentPtr = currentPtr->nextPtr;
         }

         printf("NULL\n\n");
      }
   }
```

Fig. 12.3 Cómo insertar y borrar nodos en una lista (parte 3 de 3)

```
Enter your choice:
   1 to insert an element into the list.
   2 to delete an element from the list.
   3 to end.
? 1
Enter a character: B
The list is:
B -> NULL

? 1
Enter a character: A
The list is:
A -> B -> NULL

? 1
Enter a character: C
The list is:
A -> B -> C -> NULL

? 2
Enter character to be deleted: D
D not found.

? 2
Enter character to be deleted: B
B deleted.
Tne list is:
A -> C -> NULL

? 2
Enter character to be deleted: C
C deleted.
The list is:
A -> NULL

? 2
Enter character to be deleted: A
A deleted.
List is empty.

? 4
Invalid choice.

Enter your choice:
   1 to insert an element into the list.
   2 to delete an element from the list.
   3 to end.
? 3
End of run.
```

Fig. 12.4 Salida de muestra del programa de la figura 12.3.

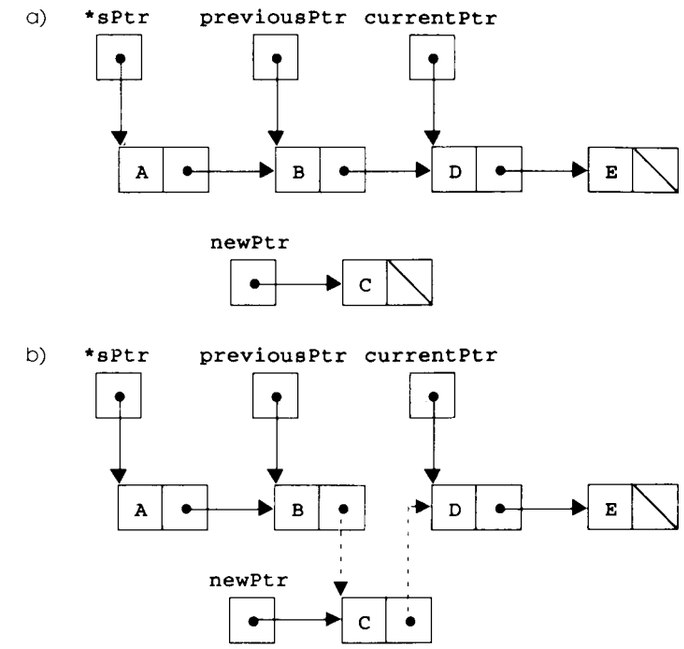


Fig. 12.5 Cómo insertar un nodo en orden dentro de una lista.

3) En tanto **currentPtr** no sea **NULL** y el valor a insertarse sea mayor que **currentPtr->data**, asigne **currentPtr** a **previousPtr** y avance **currentPtr** al siguiente nodo en la lista. Esto posiciona en la lista el punto de inserción del valor.
4) Si **previousPtr** es **NULL**, se inserta el nuevo nodo como primer nodo de la lista. Asigne ***sPtr** a **newPtr->nextPtr** (el nuevo enlace de nodo apunta al anterior primer nodo), y asigne **newPtr** a ***sPtr** (***sPtr** apunta al nuevo nodo). Si **previousPtr** no es **NULL**, el nuevo nodo se inserta en su lugar. Asigne **newPtr** a **previousPtr->nextPtr** (el nodo anterior apunta al nuevo nodo), y asigne **currentPtr** a **newPtr->nextPtr** (el enlace de nuevo nodo apunta al nodo actual).

Práctica sana de programación 12.4

Asigne **NULL** *al miembro de enlace de un nuevo nodo. Los apuntadores deben ser inicializados antes de ser utilizados.*

En la figura 12.5 se ilustra la inserción de un nodo conteniendo el carácter **'C'** en una lista ordenada. La parte a) de la figura muestra la lista y el nuevo nodo antes de la inserción. La parte b) de la figura muestra el resultado de la inserción del nuevo nodo. Los apuntadores reasignados están representados por flechas y líneas punteadas.

La función **delete** recibe la dirección del apuntador hacia el principio de la lista y un carácter a borrarse. Los pasos para borrar un carácter de la lista son como siguen:

1) Si el carácter a borrarse coincide con el primer carácter del primer nodo de la lista, asigna `*sPtr` a `tempPtr` (`tempPtr` será utilizado para liberar, usando `free`, la memoria no necesaria), asigna `(*sPtr)->nextPtr` a `*sPtr` (`*sPtr` ahora apunta al segundo nodo de la lista), `free` libera la memoria apuntada por `tempPtr`, y regresa el carácter que fue borrado.
2) De no ser así, inicializa `previousPtr` con `*sPtr` e inicializa `currentPtr` con `(*sPtr)->nextPtr`.
3) En tanto `currentPtr` no sea `NULL` y el valor a borrarse no sea igual `currentPtr->data`, asigna `currentPtr` a `previousPtr`, y asigna `currentPtr->nextPtr` a `currentPtr`. Esto localizará el carácter a borrarse, si está contenido dentro de la lista.
4) Si `currentPtr` no es `NULL`, asigna `currentPtr` a `tempPtr`, asigna `currentPtr->nextPtr` a `previousPtr->nextPtr`, libera el nodo al cual apunta `tempPtr`, y regresa el carácter que fue borrado de la lista. Si `currentPtr` es `NULL`, regresa el carácter `NULL ('\0')`, para significar que el carácter a borrarse no fue encontrado dentro de la lista.

En la figura 12.6 se ilustra el borrado de un nodo de una lista enlazada. La parte a) de la figura muestra la lista enlazada, antes de la operación de inserción anterior. La parte b) muestra la reasignación del elemento de enlace de `previousPtr` y la asignación de `currentPtr` a `tempPtr`. El apuntador `tempPtr` se utiliza para liberar la memoria asignada para almacenar `'C'`.

La función `printList` recibe un apuntador al inicio de la lista como un argumento, y se refiere al apuntador como `currentPtr`. La función primero determina si la lista está vacía. Si es así, `printList` imprime `"The list is empty"` y termina. De lo contrario, imprime

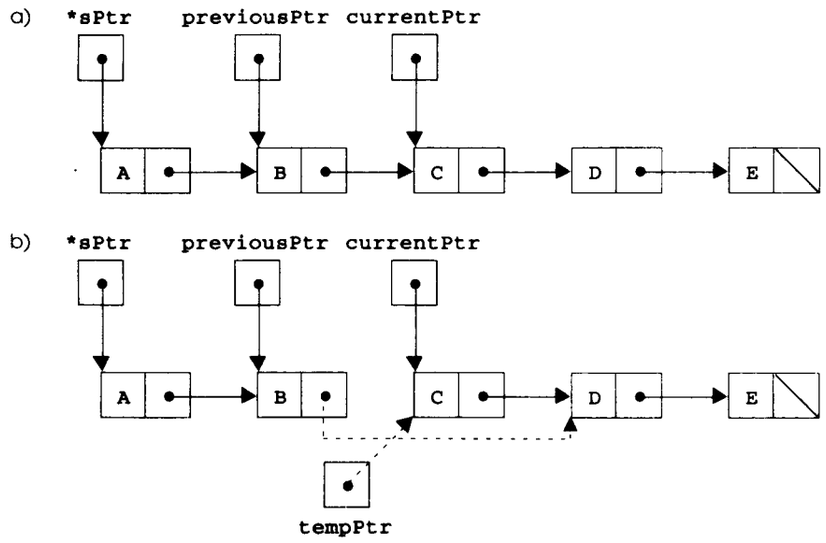


Fig. 12.6 Cómo borrar un nodo de una lista.

los datos en la lista. En tanto `currentPtr` no sea `NULL`, `currentPtr->data` será impreso por la función, y `currentPtr->nextPtr` será asignado a `currentPtr`. Note que si en el último nodo el enlace de la lista no es `NULL`, el algoritmo de impresión tratará de imprimir más allá del final de la lista, y ocurrirá un error. El algoritmo de impresión es idéntico para listas enlazadas, pilas y colas de espera.

12.5 Pilas

Una *pila* es una versión restringida de una lista enlazada. A una pila se le pueden añadir y retirar nuevos nodos únicamente de su parte superior. Por esta razón, se conoce una pila como una estructura de datos como *últimas entradas, primeras salidas* (*LIFO por last-in, first-out*). Se referencía una pila mediante un apuntador al elemento superior de la misma. El miembro de enlace en el último nodo de la pila se define a `NULL`, para indicar que se trata de la parte inferior de la pila misma.

En la figura 12.7 se ilustra una pila con varios nodos. Note que las pilas y las listas enlazadas se representan en forma idéntica. La diferencia entre las pilas y las listas enlazadas es que en una lista enlazada las inserciones y borrados pueden ocurrir en cualquier parte, pero en una pila únicamente en su parte superior.

Error común de programación 12.6

No definir a `NULL` el enlace en el nodo inferior de una pila.

Las funciones primarias utilizadas para manipular una pila son ***push*** y ***pop***. La función `push` crea un nuevo nodo y lo coloca en la parte superior de la pila. La función `pop` elimina un nodo de la parte superior de la pila, liberando la memoria que fue asignada al nodo retirado, y regresando el valor retirado.

El programa de la figura 12.8 (cuya salida se muestra en la salida 12.9) representa una pila de enteros simple. El programa presenta tres opciones: 1) incluir (push) un valor en la pila (función `push`), 2) retirar (pop) un valor de la pila (función `pop`) y 3) terminar el programa.

La función `push` coloca un nuevo nodo en la parte superior de la pila. La función está formada de tres pasos:

1) Crear un nuevo nodo llamando a `malloc`, asignando la posición de la memoria asignada a `newPtr`, asignando el valor a colocarse en la pila a `newPtr->data`, y asignar `NULL` a `newPtr->nextPtr`.
2) Asignar `*topPtr` (el apuntador a la parte superior de la pila) a `newPtr->nextPtr` —el miembro de enlace de `newPtr` ahora apunta al nodo superior anterior.
3) Asigna `newPtr` a `*topPtr` —`*topPtr` apunta ahora a la nueva parte superior de la pila.

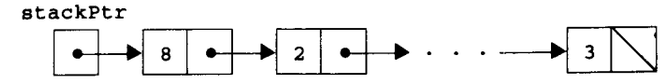


Fig. 12.7 Representación gráfica de una pila.

```
/* dynamic stack program */
#include <stdio.h>
#include <stdlib.h>

struct stackNode {  /* self-referential structure */
   int data;
   struct stackNode *nextPtr;
};

typedef struct stackNode STACKNODE;
typedef STACKNODE *STACKNODEPTR;

void push(STACKNODEPTR *, int);
int pop(STACKNODEPTR *);
int isEmpty(STACKNODEPTR);
void printStack(STACKNODEPTR);
void instructions(void);

main()
{
   STACKNODEPTR stackPtr = NULL;  /* points to the stack top */
   int choice, value;

   instructions();
   printf("? ");
   scanf("%d", &choice);

   while (choice != 3) {

      switch (choice) {
         case 1:      /* push value onto stack */
            printf("Enter an integer: ");
            scanf("%d", &value);
            push(&stackPtr, value);
            printStack(stackPtr);
            break;
         case 2:      /* pop value off stack */
            if (!isEmpty(stackPtr))
               printf("The popped value is %d.\n",
                      pop(&stackPtr));

            printStack(stackPtr);
            break;
         default:
            printf("Invalid choice.\n\n");
            instructions();
            break;
      }

      printf("? ");
      scanf("%d", &choice);
   }

   printf("End of run.\n");
   return 0;
}
```

Fig. 12.8 Un programa de pilas simple (parte 1 de 3).


```
/* Print the instructions */
void instructions(void)
{
   printf("Enter choice:\n"
          "1 to push a value on the stack\n"
          "2 to pop a value off the stack\n"
          "3 to end program\n");
}

/* Insert a node at the stack top */
void push(STACKNODEPTR *topPtr, int info)
{
   STACKNODEPTR newPtr;

   newPtr = malloc(sizeof(STACKNODE));
   if (newPtr != NULL) {
      newPtr->data = info;
      newPtr->nextPtr = *topPtr;
      *topPtr = newPtr;
   }
   else
      printf("%d not inserted. No memory available.\n", info);
}

/* Remove a node from the stack top */
int pop(STACKNODEPTR *topPtr)
{
   STACKNODEPTR tempPtr;
   int popValue;

   tempPtr = *topPtr;
   popValue = (*topPtr)->data;
   *topPtr = (*topPtr)->nextPtr;
   free(tempPtr);
   return popValue;
}

/* Print the stack */
void printStack(STACKNODEPTR currentPtr)
{
   if (currentPtr == NULL)
      printf("The stack is empty.\n\n");
   else {
      printf("The stack is:\n");

      while (currentPtr != NULL) {
         printf("%d --> ", currentPtr->data);
         currentPtr = currentPtr->nextPtr;
      }

      printf("NULL\n\n");
   }
}
```

Fig. 12.8 Un programa de pilas simple (parte 2 de 3).

```
/* Is the stack empty? */
int isEmpty(STACKNODEPTR topPtr)
{
   return topPtr == NULL;
}
```

Fig. 12.8 Un programa de pilas simple (parte 3 de 3).

```
Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 1
Enter an integer: 5
The stack is:
5 --> NULL

? 1
Enter an integer: 6
The stack is:
6 --> 5 --> NULL

? 1
Enter an integer: 4
The stack is:
4 --> 6 --> 5 --> NULL

? 2
The popped value is 4.
The stack is:
6 --> 5 --> NULL

? 2
The popped value is 6.
The stack is:
5 --> NULL

? 2
The popped value is 5.
The stack is empty.

? 2
The stack is empty.

? 4
Invalid choice.

Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 3
End of run.
```

Fig. 12.9 Salida de muestra correspondiente al programa de la figura 12.8.

Las manipulaciones que involucran a ***topPtr** modifican el valor de **stackPtr** en **main**. La figura 12.10 ilustra la función **push**. La parte a) de la figura muestra la pila y el nuevo nodo, antes de la operación **push**. En la parte b) las flechas con líneas punteadas ilustran los pasos 2 y 3 de la operación **push**, que le permiten al nodo que contiene **12** convertirse en la nueva parte superior de la pila. La función **pop** retira un nodo de la parte superior de la pila. Note que **main** determina, antes de llamar a **pop**, si la pila está vacía. La operación **pop** consiste de cinco pasos.

1) Asigna ***topPtr** a **tempPtr** (**tempPtr** se utilizará para liberar memoria no necesaria).
2) Asigna **(*topPtr)->data** a **popValue** (guarda el valor almacenado en el nodo superior).
3) Asigna **(*topPtr)->nextPtr** a ***topPtr** (asigna ***topPtr** la dirección del nuevo nodo superior).
4) Libera la memoria a la cual apunta **tempPtr**.
5) Regresa **popValue** al llamador (**main**, en el programa de la figura 12.8).

La figura 12.11 ilustra el uso de la función **pop**. La parte a) muestra la pila antes de la operación anterior **push**. La parte b) muestra **tempPtr** apuntando al primer nodo de la pila y **topPtr** apuntando al segundo nodo de la misma. La función **free** es utilizada para liberar la memoria a la cual apunta **tempPtr**.

Las pilas tienen muchas aplicaciones interesantes. Por ejemplo, siempre que se hace una llamada de función, la función llamada debe saber cómo regresar a su llamador, por lo que la dirección de regreso es introducida en una pila. Si ocurre una serie de llamadas de función, los valores de regreso sucesivos son introducidos en la pila, en orden de últimas entradas, primeras salidas, de forma tal que cada función pueda regresar a su llamador. Las pilas aceptan llamadas de función recursivas, de la misma forma que llamadas normales no recursivas.

Las pilas contienen el espacio creado para variables automáticas en cada invocación de una función. Cuando la función regresa a su llamador, el espacio para las variables automáticas de dicha función es retirado (popped off) de la pila, y dichas variables dejan de ser conocidas para el programa.

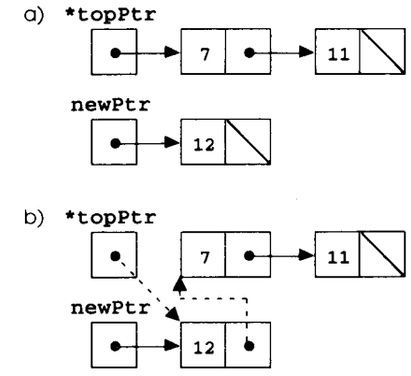


Fig. 12.10 La operación **push**.

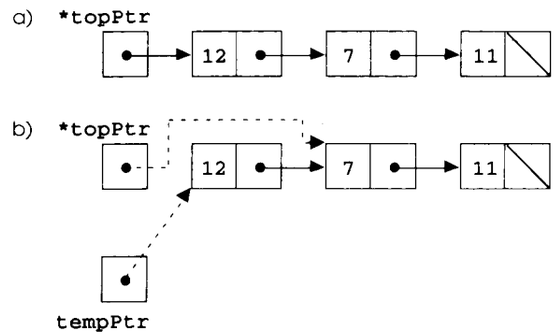


Fig. 12.11 La operación **pop**.

Las pilas son utilizadas por los compiladores en el proceso de evaluar expresiones y de generar código de lenguaje máquina. En los ejercicios se exploran varias aplicaciones de las pilas.

12.6 Colas de espera

Otra estructura de datos común es la *cola de espera*. Una cola de espera es similar a una línea de pagos en un supermercado —la primera persona en la línea es atendida primero, y los otros clientes entran en la línea únicamente por la parte final y esperan para ser atendidos. Los nodos de la cola son eliminados sólo de la parte delantera o *cabeza* de la cola, y son incluidos o insertos únicamente en la *parte trasera* de la cola. Por esta razón, una cola se conoce como una estructura de datos de *primeras entradas, primeras salidas (FIFO por first-in, first-out)*. Las operaciones de insertar y de retirar se conocen como *enqueue* y *dequeue*.

Las colas tienen muchas aplicaciones en sistemas de cómputo. Muchas computadoras tienen únicamente un solo procesador, de tal forma que sólo un usuario puede ser servido a la vez. Las entradas de los demás usuarios son colocados en una cola. Cada entrada avanza en forma gradual hacia el frente de la cola, conforme los usuarios reciben servicio. La entrada que aparece en la parte delantera de la cola es la siguiente a recibir servicio.

Las colas también se utilizan para apoyar colas de impresión. Un entorno de multiusuario pudiera tener una sola impresora. Muchos usuarios podrían estar generando salidas para impresión. Si la impresora está ocupada, aún así se pueden generar otras salidas. Estas quedan en "espera" en el disco, donde esperan en una cola hasta que la impresora quede disponible.

En las redes de computadoras los paquetes de información también esperan en colas. Cada vez que un paquete llega a un nodo de red, debe ser encaminado al siguiente nodo de red, siguiendo una trayectoria hacia el destino final del paquete. El nodo de encaminamiento envía un paquete a la vez, por lo que los paquetes adicionales quedan en cola, hasta que el encaminador pueda enviarlos. En la figura. 12.12 se ilustra una cola con varios nodos. Note los apuntadores a la cabeza y a la parte trasera de la cola.

Error común de programación 12.7

No definir a **NULL** *el enlace en el último nodo de una cola.*

El programa de la figura 12.13 (cuya salida aparece en la figura 12.14) ejecuta manipulaciones de colas. El programa presenta varias opciones: insertar un nodo en la cola (función **enqueue**), eliminar o retirar un nodo de la cola (función **dequeue**) y terminar el programa.

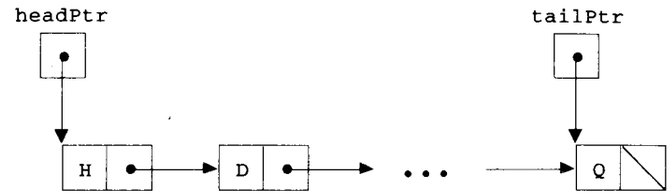


Fig. 12.12 Representación gráfica de una cola.

```
/* Operating and maintaining a queue */

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

struct queueNode {  /* self-referential structure */
   char data;
   struct queueNode *nextPtr;
};

typedef struct queueNode QUEUENODE;
typedef QUEUENODE *QUEUENODEPTR;

/* function prototypes */
void printQueue(QUEUENODEPTR);
int isEmpty(QUEUENODEPTR);
char dequeue(QUEUENODEPTR *, QUEUENODEPTR *);
void enqueue(QUEUENODEPTR *, QUEUENODEPTR *, char);
void instructions(void);

main()
{
   QUEUENODEPTR headPtr = NULL, tailPtr = NULL;
   int choice;
   char item;

   instructions();
   printf("? ");
   scanf("%d", &choice);

   while (choice != 3) {

      switch(choice) {

         case 1:
            printf("Enter a character: ");
            scanf("\n%c", &item);
            enqueue(&headPtr, &tailPtr, item);
            printQueue(headPtr);
            break;
```

Fig. 12.13 Procesamiento de una cola (parte 1 de 3).

```
            case 2:
               if (!isEmpty(headPtr)) {
                  item = dequeue(&headPtr, &tailPtr);
                  printf("%c has been dequeued.\n", item);
               }

               printQueue(headPtr);
               break;

            default:
               printf("Invalid choice.\n\n");
               instructions();
               break;
         }

         printf("? ");
         scanf("%d", &choice);
      }

      printf("End of run.\n");
      return 0;
   }

   void instructions(void)
   {
      printf ("Enter your choice:"
              "   1 to add an item to the queue"
              "   2 to remove an item from the queue"
              "   3 to end");
   }

   void enqueue(QUEUENODEPTR *headPtr, QUEUENODEPTR *tailPtr,
                char value)
   {
      QUEUENODEPTR newPtr;

      newPtr = malloc(sizeof(QUEUENODE));

      if (newPtr != NULL) {
         newPtr->data = value;
         newPtr->nextPtr = NULL;

         if (isEmpty(*headPtr))
            *headPtr = newPtr;
         else
            (*tailPtr)->nextPtr = newPtr;

         *tailPtr = newPtr;
      }
      else
         printf("%c not inserted. No memory available.\n", value);
   }
```

Fig. 12.13 Procesamiento de una cola (parte 2 de 3).


```
char dequeue(QUEUENODEPTR *headPtr, QUEUENODEPTR *tailPtr)
{
   char value;
   QUEUENODEPTR tempPtr;

   value = (*headPtr)->data;
   tempPtr = *headPtr;
   *headPtr = (*headPtr)->nextPtr;

   if (*headPtr == NULL)
      *tailPtr = NULL;

   free(tempPtr);
   return value;
}

int isEmpty(QUEUENODEPTR headPtr)
{
   return headPtr == NULL;
}

void printQueue(QUEUENODEPTR currentPtr)
{
   if (currentPtr == NULL)
      printf("Queue is empty.\n\n");
   else {
      printf("The queue is:\n");

      while (currentPtr != NULL) {
         printf("%c --> ", currentPtr->data);
         currentPtr = currentPtr->nextPtr;
      }

      printf("NULL\n\n");
   }
}
```

Fig. 12.13 Procesamiento de una cola (parte 3 de 3).

La función **enqueue** recibe desde **main** tres argumentos: la dirección del apuntador a la cabeza de la cola, la dirección del apuntador a la parte trasera de la cola, y el valor a ser inserto en la cola. La función está formada de tres pasos:

1) Para crear un nuevo nodo: llama **malloc**, asigna a **newPtr** la posición asignada de memoria, asigna el valor que se va a insertar en la cola a **newPtr->data** y asigna **NULL** a **newPtr->nextPtr**.
2) Si la cola está vacía, asigna **newPtr** a ***headPtr**; de no ser así, asigna el apuntador **newPtr** a **(*tailPtr)->nextPtr**.
3) Asigna **newPtr** a ***tailPtr**.

La figura 12.15 ilustra una operación **enqueue**. La parte a) de la figura muestra la cola y el nuevo nodo antes de la operación. Las flechas con líneas punteadas de la parte b) ilustran los pasos 2 y 3 de la función **enqueue** que permiten añadir un nuevo nodo al final de una cola que no esté vacía.

```
Enter your choice:
   1 to add an item to the queue
   2 to remove an item from the queue
   3 to end
? 1
Enter a character: A
The queue is:
A --> NULL

? 1
Enter a character: B
The queue is:
A --> B --> NULL

? 1
Enter a character: C
The queue is:
A --> B --> C --> NULL

? 2
A has been dequeued.
The queue is:
B --> C --> NULL

? 2
B has been dequeued.
The queue is:
C --> NULL

? 2
C has been dequeued.
Queue is empty.

? 2
Queue is empty.

? 4
Invalid choice.

Enter your choice:
   1 to add an item to the queue
   2 to remove an item from the queue
   3 to end
? 3
End of run.
```

Fig. 12.14 Salida de muestra del programa 12.13.

La función **dequeue** recibe como argumentos la dirección del apuntador a la cabeza de la cola y la dirección del apuntador a la parte trasera de la cola, y retira el primer nodo de la cola. La operación **dequeue** consiste de seis pasos:

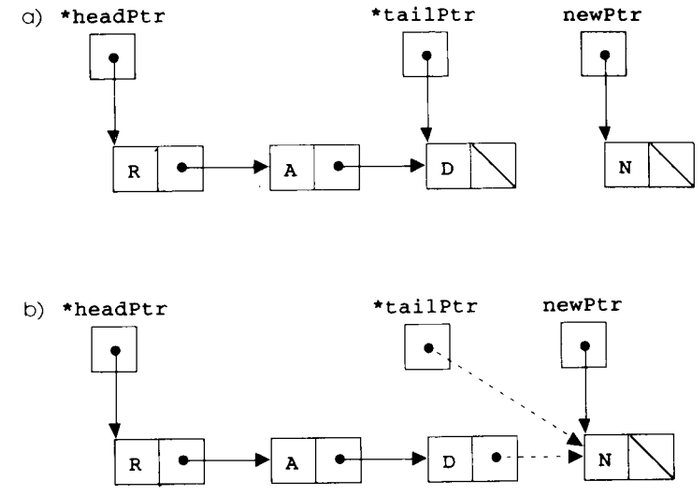


Fig. 12.15 Representación gráfica de la operación **enqueue**.

1) Asigna **(*headPtr)->data** a **value** (guarda los datos)
2) Asigna ***headPtr** a **tempPtr** (**tempPtr** es utilizado para liberar, mediante **free**, la memoria no necesaria).
3) Asigna **(*headPtr)->nextPtr** a ***headPtr** (***headPtr** apunta ahora al primer nodo en la cola).
4) Si ***headPtr** es **NULL**, asigna **NULL** a ***tailPtr**
5) Libera la memoria a la cual apunta **tempPtr**.
6) Regresa **value** al llamador (la función de **dequeue** es llamada desde **main** en el programa de la figura 12.13).

La figura 12.16 ilustra la función **dequeue**. La parte a) muestra la cola antes de la operación **enqueue** anterior. La parte b) muestra **tempPtr** apuntando al nodo a retirarse (dequeue), y **headPtr** apuntando al nuevo primer nodo de la cola. La función **free** es utilizada para recuperar la memoria a la cual **tempPtr** apunta.

12.7 Arboles

Las listas enlazadas, las pilas y las colas son *estructuras lineales de datos*. Un árbol es una estructura no lineal y de dos dimensiones de datos, con propiedades especiales. Los nodos de los árboles contienen dos o más enlaces. Esta sección analiza los *árboles binarios* (figura 12.17) —árboles cuyos nodos todos ellos contienen dos enlaces (ninguno, uno, o ambos de los cuales pudieran ser **NULL**). El *nodo raíz* es el primer nodo de un árbol. Cada enlace en el nodo raíz se refiere a un *hijo*. El *hijo izquierdo* es el primer nodo en el *subárbol izquierdo* y el *hijo derecho* es el primer nodo en el *subárbol izquierdo*. Los hijos de un nodo se conocen como *descendientes*. Un nodo sin hijos se conoce como *nodo de hoja*. Los científicos de la computación normalmente dibujan los árboles partiendo del nodo raíz hacia abajo —en forma exactamente opuesta a los árboles en la naturaleza.

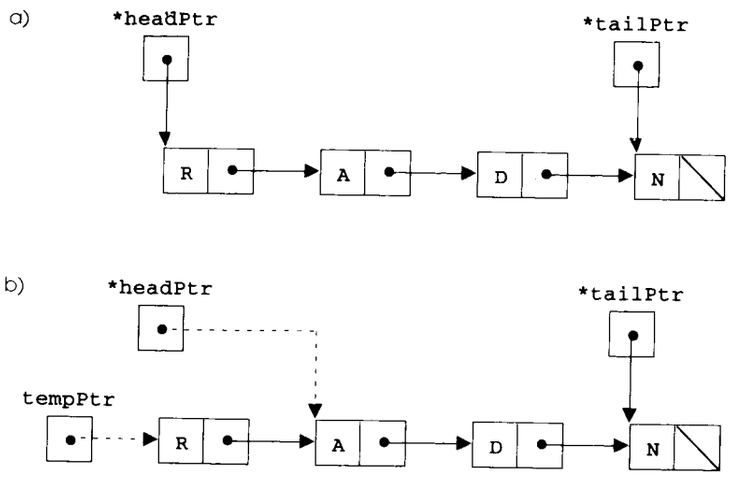


Fig. 12.16 Representación gráfica de la operación **dequeue**.

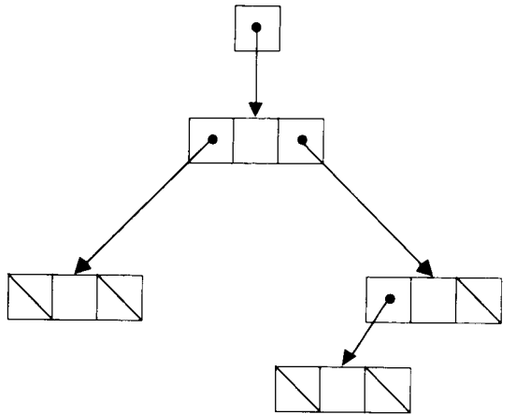


Fig. 12.17 Representación gráfica de un árbol binario.

En esta sección, se creará un árbol binario especial conocido como un *árbol de búsqueda binario*. Un árbol de búsqueda binario (que no tiene valores duplicados de nodos) tienen la característica que los valores en cualquier subárbol izquierdo son menores que el valor en sus nodos padre, y los valores en cualquier subárbol derecho son mayores que el valor en sus nodos padre. En la figura 12.18 se ilustra un árbol de búsqueda binario con 12 valores. Note que la forma del árbol de búsqueda binario que corresponde a un conjunto de datos puede variar, dependiendo del orden en el cual los valores están insertos dentro del árbol.

Error común de programación 12.8

No definir a `NULL` *los enlaces en los nodos de hoja de un árbol.*

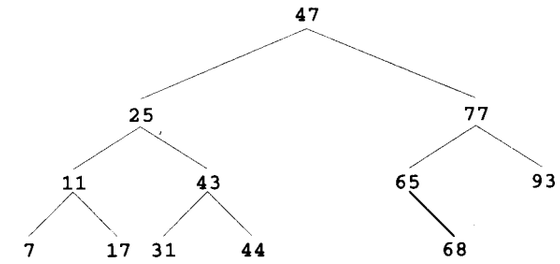


Fig. 12.18 Un árbol de búsqueda binario.

El programa de la figura 12.19 (cuya salida se muestra en la figura 12.20) crea un árbol de búsqueda binario y lo recorre de tres formas —*enorden, preorden* y *postorden*. El programa genera 10 números aleatorios e inserta cada uno de ellos en el árbol, a excepción de los valores duplicados, que son descartados.

Las funciones utilizadas en la figura 12.19, para crear un árbol de búsqueda binario y recorrer el árbol, son recursivas. La función **`insertNode`** recibe como argumentos la dirección del árbol y un entero para almacenarse en el árbol. *En un árbol de búsqueda binario un nodo puede ser únicamente inserto como nodo de hoja.* Los pasos para insertar un nodo en un árbol de búsqueda binario, son como sigue:

1) Si **`*treePtr`** es **`NULL`**, crea un nuevo nodo. Llame **`malloc`**, asigna la memoria asignada a **`*treePtr`**, asigna a **`(*treePtr)->data`** el entero a almacenarse, asigna a **`(*treePtr)->leftPtr`** y **`(*treePtr)->rightPtr`** el valor **`NULL`** y devuelve o regresa el control al llamador (ya sea a **`main`** o a una llamada anterior a **`insertNode`**).
2) Si el valor de **`*treePtr`** no es **`NULL`**, y el valor a insertarse es menor que **`(*treePtr)->data`**, se llama a la función **`insertNode`** con la dirección de **`(*treePtr)->leftPtr`**. De no ser así, se llama a la función **`insertNode`** con la dirección de **`(*treePtr)->rightPtr`**. Se continúan los pasos recursivos hasta que se encuentre un apuntador **`NULL`**, entonces se ejecutará el paso 1) para insertar el nuevo nodo.

Las funciones **`inOrder`**, **`preOrder`** y **`postOrder`** cada una de ellas recibe un árbol (es decir, el apuntador al nodo raíz del árbol) y recorren el árbol.

Los pasos para un recorrido **`inorder`** son:

1) Recorrer el subárbol izquierdo **`inOrder`**.
2) Procesar el valor en el nodo.
3) Recorrer el subárbol derecho **`inOrder`**.

El valor en un nodo no es procesado en tanto no sean procesados los valores de su subárbol izquierdo. El recorrido **`inOrder`** del árbol en la figura 12.21 es

```
6 13 17 27 33 42 48
```

Note que el recorrido **`inOrder`** de un árbol de búsqueda binario imprime los valores de nodo en valor ascendente. El proceso de crear un árbol de búsqueda binario, de hecho ordena los datos —y por lo tanto este proceso se llama la *clasificación de árbol binario*.

```
/* Create a binary tree and traverse it
   preorder, inorder, and postorder */

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

struct treeNode {
   struct treeNode *leftPtr;
   int data;
   struct treeNode *rightPtr;
};

typedef struct treeNode TREENODE;
typedef TREENODE *TREENODEPTR;

void insertNode(TREENODEPTR *, int);
void inOrder(TREENODEPTR);
void preOrder(TREENODEPTR);
void postOrder(TREENODEPTR);

main()
{
   int i, item;
   TREENODEPTR rootPtr = NULL;

   srand(time(NULL));

   /* attempt to insert 10 random values betwenn 0 and 14 in the tree */
   printf("The numbers being placed in the tree are:\n");

   for (i = 1; i <= 10; i++) {
      item = rand() % 15;
      printf("%3d", item);
      insertNode(&rootPtr, item);
   }

   /* traverse the tree preOrder */
   printf("\n\nThe preOrder traversal is:\n");
   preOrder(rootPtr);

   /* traverse the tree inOrder */
   printf("\n\nThe inOrder traversal is:\n");
   inOrder(rootPtr);

   /* traverse the tree postOrder */
   printf("\n\nThe postOrder traversal is:\n");
   postOrder(rootPtr);

   return 0;
}
```

Fig. 12.19 Cómo crear y recorrer un árbol binario (parte 1 de 2).


```
void insertNode(TREENODEPTR *treePtr, int value)
{
   if (*treePtr == NULL) {   /* *treePtr is NULL */
      *treePtr = malloc(sizeof(TREENODE));

      if (*treePtr != NULL) {
         (*treePtr)->data = value;
         (*treePtr)->leftPtr = NULL;
         (*treePtr)->rightPtr = NULL;
      }
      else
         printf("%d not inserted. No memory available.\n",
                value);

   }
   else
      if (value < (*treePtr)->data)
         insertNode(&((*treePtr)->leftPtr), value);
      else
         if (value > (*treePtr)->data)
            insertNode(&((*treePtr)->rightPtr), value);
         else
            printf("dup");
}

void inOrder(TREENODEPTR treePtr)
{
   if (treePtr != NULL) {
      inOrder(treePtr->leftPtr);
      printf("%3d", treePtr->data);
      inOrder(treePtr->rightPtr);
   }
}

void preOrder(TREENODEPTR treePtr)
{
   if (treePtr != NULL) {
      printf("%3d", treePtr->data);
      preOrder(treePtr->leftPtr);
      preOrder(treePtr->rightPtr);
   }
}

void postOrder(TREENODEPTR treePtr)
{
   if (treePtr != NULL) {
      postOrder(treePtr->leftPtr);
      postOrder(treePtr->rightPtr);
      printf("%3d", treePtr->data);
   }
}
```

Fig. 12.19 Cómo crear y recorrer un árbol binario (parte 2 de 2).

```
The numbers being placed in the tree are:
  7  8  0  6 14  1  0dup 13  0dup  7dup

The preOrder traversal is:
  7  0  6  1  8 14 13

The inOrder traversal is:
  0  1  6  7  8 13 14

The postOrder traversal is:
  1  6  0 13 14  8  7
```

Fig. 12.20 Salida de muestra correspondiente al programa de la figura 12.19.

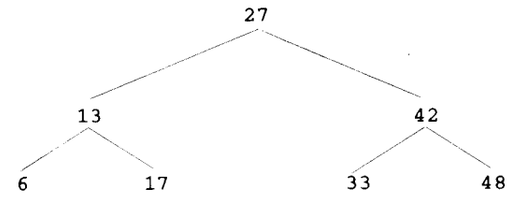


Fig. 12.21 Un árbol de búsqueda binario.

Los pasos para un recorrido **preOrder** son:

1) Procesar el valor en el nodo.
2) Recorrer el subárbol izquierdo **preOrder**.
3) Recorrer el subárbol derecho **preOrder.**

El valor en cada nodo es procesado conforme se pasa por cada nodo. Después de que se procese el valor en un nodo dado, son procesados los valores del subárbol izquierdo, y a continuación los valores en el subárbol derecho. El recorrido **preOrder** del árbol en la figura 12.21 es:

```
27 13 6 17 42 33 48
```

Los pasos para un recorrido **postOrder** son:

1) Recorrer el subárbol izquierdo **postOrder**.
2) Recorrer el subárbol derecho **postOrder**.
3) Procesar el valor en el nodo.

El valor en cada nodo no se imprime hasta que sean impresos los valores de sus hijos. El recorrido **postOrder** del árbol de la figura 12.21 es:

```
6 17 13 33 48 42 27
```


El árbol de búsqueda binario facilita la eliminación de duplicados. Conforme se crea el árbol, cualquier intento para insertar un valor duplicado será detectado porque en cada una de las comparaciones un duplicado seguirá las mismas decisiones "ir a la izquierda" o "ir a la derecha" que utlizó el valor original. Entonces, el duplicado eventualmente será comparado con un nodo que contenga el mismo valor. Llegado a este punto el valor duplicado pudiera simplemente ser descartado.

También es rápido buscar en un árbol binario un valor que coincida con un valor clave. Si el árbol es denso, entonces cada nivel contendrá aproximadamente dos veces tantos elementos como el nivel anterior. Por lo tanto un árbol de búsqueda binario con n elementos tendría un máximo de $\log_2 n$ (logaritmo de base 2 de n niveles) y, por lo tanto, tendrían que efectuarse un máximo $\log_2 n$ de comparaciones, ya sea para encontrar una coincidencia, o para determinar que no existe ninguna. Esto significa, por ejemplo, que al buscar en un árbol de búsqueda binario de 1000 elementos (densamente empacados), no es necesario llevar a cabo más de 10 comparaciones, porque $2^{10} > 1000$. Al buscar en un árbol de búsqueda binario de 1,000,000 (densamente empacados), no es necesario llevar a cabo más de 20 comparaciones, porque $2^{20} > 1{,}000{,}000$.

En los ejercicios se presentan algoritmos para otras operaciones de árboles de búsqueda binarios, como es borrar un elemento de un árbol binario, imprimir un árbol binario en un formato de árbol de dos dimensiones, y ejecutar un recorrido en orden de niveles de un árbol binario. El recorrido de orden de niveles de un árbol binario pasa por los nodos de un árbol, renglón por renglón, empezando por el nivel del nodo raíz. En cada uno de los niveles de árbol, se pasa por los nodos de izquierda a derecha. Otros ejercicios de árboles binarios incluyen permitir que un árbol de búsqueda binario contenga valores duplicados, insertar valores de cadenas en un árbol binario, y determinar cuántos niveles están incluidos en un árbol binario.

Resumen

- Las estructuras autorreferenciadas contienen miembros conocidos como enlaces, que apuntan a estructuras del mismo tipo de estructura.
- Las estructuras autorreferenciadas permiten que se enlacen muchas estructuras juntas en pilas, colas, listas y árboles.
- La asignación dinámica de memoria reserva un bloque de bytes en la memoria para almacenar un objeto de datos durante la ejecución del programa.
- La función **malloc** toma como argumento un número de bytes a asignarse, y regresa un apuntador **void** a la memoria asignada. La función **malloc** se utiliza normalmente junto con el operador **sizeof**. El operador **sizeof** determina el tamaño en bytes de la estructura para la cual se está asignando memoria.
- La función **free** cancela la asignación de memoria.
- Una lista enlazada es una colección de datos almacenados en un grupo de estructuras autorreferenciadas conectadas.
- Una lista enlazada es una estructura dinámica de datos —la longitud de la lista puede aumentarse o reducirse conforme sea necesario.
- Las listas enlazadas pueden continuar creciendo en tanto exista memoria disponible.
- Las listas enlazadas proporcionan un mecanismo para insertar y borrar datos en forma simple mediante la reasignación de apuntadores.

- Las pilas y las colas son versiones especializadas de una lista enlazada.
- Se añaden nuevos nodos a una pila y son retirados nodos de una pila únicamente de su parte superior. Por esta razón, se conoce una pila como una estructura de datos de últimas entradas, primeras salidas (LIFO).
- El miembro de enlace en el último nodo de la pila se define a **NULL** para indicar la parte inferior de la pila.
- Las dos operaciones primarias utilizadas para manipular una pila son **push** y **pop**. La operación **push** crea un nuevo nodo y lo coloca en parte superior de la pila. La operación **pop** retira un nodo de la parte superior de la pila, libera la memoria que estaba asignada al nodo retirado, y regresa el valor retirado.
- En una estructura de datos de cola, los nodos son retirados de la cabeza y añadidos a la parte trasera. Por esta razón, una cola se conoce como una estructura de datos de primeras entradas, primeras salidas (FIFO). Las operaciones de añadir y de retirar se conocen como **enqueue** y **dequeue**.
- Los árboles son estructuras de datos más complejas que las listas enlazadas, las colas y las pilas. Los árboles son estructuras de datos de dos dimensiones, que requieren de dos o más enlaces por nodo.
- Los árboles binarios contienen dos enlaces por nodo.
- El nodo raíz es el primer nodo en el árbol.
- Cada uno de los apuntadores en el nodo raíz se refiere a un hijo. El hijo izquierdo es el primer nodo en el subárbol izquierdo, y el hijo derecho es el primer nodo en el subárbol derecho. Los hijos de un nodo se conocen como descendientes. Si un nodo no tiene ningún descendiente, se le conoce como un nodo de hoja.
- Un árbol de búsqueda binario tiene la característica que tiene el valor en el hijo izquierdo del nodo menor que el valor del nodo padre, y el valor en el hijo derecho de un nodo es mayor o igual que el valor del nodo padre. Si se puede determinar que no existen valores de datos duplicados, simplemente el valor en el hijo derecho es mayor que el valor en el nodo padre.
- Un recorrido enorden de un árbol binario recorre enorden el subárbol izquierdo, procesa el valor en el nodo, y recorre enorden el subárbol derecho. El valor en un nodo no será procesado hasta que hayan sido procesados los valores en su subárbol izquierdo.
- Un recorrido en preorden procesa el valor en el nodo, recorre en preorden el subárbol izquierdo, y recorre en preorden el subárbol derecho. El valor en cada nodo se procesa conforme se encuentra cada nodo.
- Un recorrido en postorden recorre en postorden el subárbol izquierdo, recorre en postorden el subárbol derecho, y procesa el valor en el nodo. El valor en cada nodo no es procesado hasta que sean procesados los valores en ambos de sus subárboles.

Terminología

árbol de búsqueda binaria
árbol binario
clasificación de árbol binario
nodo hijo
descendientes
borrar un nodo
dequeue
doble indirección
estructuras dinámicas de datos
asignación dinámica de memoria

enqueue
FIFO (primeras entradas, primeras salidas)
free
cabeza de una cola
recorrido enorden
inserción de un nodo
nodo de hoja
hijo izquierdo
subárbol izquierdo
LIFO (últimas entradas, primeras salidas)
estructura lineal de datos
lista enlazada
malloc (asignar memoria)
nodo
estructura de datos no lineal
apuntador **NULL**
nodo padre
apuntador a un apuntador
pop
recorrido postorden
función predicada
recorrido preorden
push
cola
hijo derecho
subárbol derecho
nodo raíz
estructura autorreferenciada
descendencias
sizeof
pila
subárbol
parte trasera de una cola
parte superior
recorrido
árbol
pasar por un nodo

Errores comunes de programación

12.1 No establecer a **NULL** el enlace en el último nodo de una lista.
12.2 Suponer que el tamaño de una estructura es simplemente la suma de los tamaños de sus miembros.
12.3 No regresar memoria dinámicamente asignada cuando ésta ya no es necesaria, puede hacer que el sistema se quede sin memoria prematuramente. Esto se conoce a veces como "fuga de memoria".
12.4 Utilizando **malloc**, liberar memoria no dinámicamente asignada.
12.5 Referirse a memoria que ya ha sido liberada.
12.6 No definir a **NULL** el enlace en el nodo inferior de una pila.
12.7 No definir a **NULL** el enlace en el último nodo de una cola.
12.8 No definir a **NULL** los enlaces en los nodos de hoja de un árbol.

Prácticas sanas de programación

12.1 Para determinar el tamaño de una estructura utilice el operador **sizeof**.
12.2 Al utilizar **malloc**, compruebe si es **NULL** el valor de regreso de apuntador. Si la memoria solicitada no ha sido asignada imprima un mensaje de error.
12.3 Cuando ya no se requiera memoria que fue dinámicamente asignada, utilice **free**, para regresar esta memoria inmediatamente al sistema.
12.4 Asigne **NULL** al miembro de enlace de un nuevo nodo. Los apuntadores deben ser inicializados antes de ser utilizados.

Sugerencias de rendimiento

12.1 Podría declararse un arreglo que contenga más elementos que el número esperado de elementos de datos, pero esto puede desperdiciar memoria. En estas situaciones las listas enlazadas pueden obtener una mejor utilización de la memoria.
12.2 Puede resultar muy tardado insertar y eliminar en un arreglo ya ordenado —deberán ser desplazados en forma apropiada. Todos los elementos que sigan al elemento insertado o borrado.

12.3 Los elementos de un arreglo se almacenan en forma contigua en memoria. Esto permite acceso inmediato a cualquier arreglo del elemento, porque la dirección de cualquier elemento puede ser calculada directamente, basada en su posición en relación con el principio del arreglo. Las listas enlazadas no proporcionan un acceso inmediato como éste a sus elementos.

12.4 Tratándose de estructuras de datos que crecen o se reducen en tiempo de ejecución, es posible ahorrar memoria utilizando asignación dinámica de memoria (en vez de los arreglos). Recuerde, sin embargo, que los apuntadores toman espacio, y que la asignación dinámica de memoria incurre en sobrecarga por las llamadas de función.

Sugerencia de portabilidad

12.1 El tamaño de una estructura no es necesariamente la suma de los tamaños de sus miembros. Esto es debido a varios requisitos de alineación de límites, que son dependientes de la máquina (vea el capítulo 10).

Ejercicios de autoevaluación

12.1 Llene cada uno de los siguientes espacios vacíos:

a) Una estructura auto_____ se utiliza para formar estructuras dinámicas de datos.
b) La función _____ se utiliza para asignar dinámicamente la memoria.
c) Una _____ es una versión especializada de una lista enlazada, en la cual los nodos pueden ser insertos y borrados únicamente a partir del principio de la lista.
d) Las funciones que no modifican una lista enlazada, pero que simplemente analizan la lista se conocen como _____.
e) Una cola de espera se conoce como una estructura de datos _____ porque los primeros nodos insertos son los primeros nodos retirados.
f) El apuntador al siguiente nodo en una lista enlazada se conoce como un _____.
g) La función _____ se utiliza para recuperar memoria dinámicamente asignada.
h) Una _____ es una versión especializada de una lista enlazada en la cual los nodos pueden ser insertos únicamente al principio de la lista y retirados o borrados únicamente de la parte final de la lista.
i) Una _____ es una estructura de datos de dos dimensiones no lineal, que contiene nodos con dos o más enlaces.
j) Una pila se conoce como una estructura de datos _____ porque el último nodo insertado es el primer nodo retirado.
k) Los nodos de un árbol _____ contienen dos miembros enlazados.
l) El primer nodo de un árbol es el nodo _____.
m) Cada enlace en un nodo de árbol apunta a un _____ o a un _____ de dicho nodo.
n) Un nodo de árbol que no tiene hijos se conoce como un nodo _____.
o) Los tres algoritmos de recorrido para un árbol binario son _____, _____ y _____.

12.2 ¿Cuáles son las diferencias entre una lista enlazada y una pila?

12.3 ¿Cuáles son las diferencias entre una pila y una cola de espera?

12.4 Escriba un enunciado, o un conjunto de enunciados para llevar a cabo cada uno de los siguientes. Suponga que todas las manipulaciones ocurren en `main` (por lo tanto, no son necesarias direcciones a variables de apuntador) y suponga las definiciones siguientes:

```
struct gradeNode {
   char lastName[20];
   float grade;
   struct gradeNode *nextPtr;
};
```



```
typedef struct gradeNode GRADENODE;
typedef GRADENODE *GRADENODEPTR;
```

a) Crea un apuntador al principio de la lista llamado `startPtr`. La lista está vacía.
b) Crea un nuevo nodo del tipo `GRADENODE` al cual apunta el apuntador `newPtr` del tipo `GRADENODEPTR`. Asigne la cadena `"Jones"` al miembro `LastName` y el valor `91.5` al miembro `grade` (utilice `strcpy`). Incluya todas las declaraciones y enunciados necesarios.
c) Suponga que la lista a la cual apunta `startPtr` está formada actualmente de dos nodos —uno que contiene `"Jones"` y uno que contiene `"Smith"`. Los nodos están en orden alfabético. Proporcione los enunciados necesarios para insertar otros nodos que contengan los siguientes datos para `lastName` y `grade`:

```
"Adams"      85.0
"Thompson"   73.5
"Pritchard"  66.5
```

Utilice los apuntadores `previousPtr`, `currentPtr` y `newPtr` para llevar a cabo las inserciones. Declare hacia donde apuntan `previousPtr`, y `currentPtr` antes de cada inserción. Suponga que `newPtr` siempre apunta al nuevo nodo, y que el nuevo nodo ya ha sido asignado con los datos.
d) Escriba un ciclo `while` que imprima los datos de cada nodo de la lista. Utilice el apuntador `currentPtr` para pasar a lo largo de la misma.
e) Escriba un ciclo `while` que borre todos los nodos en la lista y que libere la memoria asociada con cada uno de los nodos. Utilice el apuntador `currentPtr` y el apuntador `tempPtr` para caminar a lo largo de la lista y liberar memoria respectivamente.

12.5 Proporcione manualmente los recorridos enorden, preorden y postorden del árbol de búsqueda binario de la Figura 12.22.

Respuestas a los ejercicios de autoevaluación

12.1 a) referenciados. b) `malloc`. c) pila. d) predicados. e) FIFO. f) enlace. g) `free`. h)cola de espera. i) árbol. j) LIFO. k) binario. l) raíz. m) hijo o subárbol. n) hoja. o) enorden, preorden y postorden.

12.2 Es posible insertar un nodo en cualquier parte de una lista enlazada, y eliminar un nodo de cualquier parte de una lista enlazada. Sin embargo, los nodos en una pila pueden únicamente ser insertos en la parte superior de la misma y eliminados también de la parte superior de la misma.

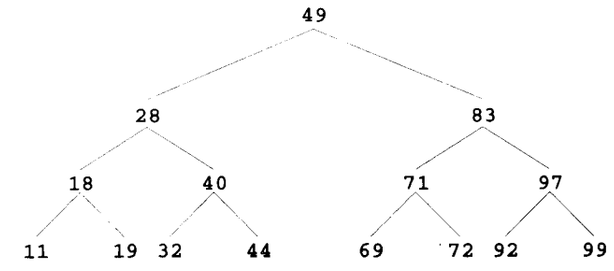


Fig. 12.22 Un árbol de búsqueda binario de 15 nodos.

12.3 Una cola tiene apuntadores tanto a su cabeza como a su parte trasera, de tal forma que los nodos pueden ser insertos en la parte trasera y borrados de la cabeza. Una pila tiene un solo apuntador a la parte superior de la misma donde se ejecuta tanto la inserción como el borrado de los nodos.

12.4 a)
```
GRADENODEPTR startPtr = NULL;
```
b)
```
GRADENODEPTR newPtr;
newPtr = malloc(sizeof(GRADENODE));
strcpy(newPtr->lastName, "Jones");
newPtr->grade = 91.5;
newPtr->nextPtr = NULL
```
c) Para insertar `"Adams"`;
`previousPtr` es `NULL`, `current Ptr` apunta al primer elemento en la lista.
```
newPtr->nextPtr = currentPtr;
startPtr = newPtr;
```
Para insertar `"Thompson"`:
`previousPtr` apunta al último elemento en la lista (que contiene `"Smith"`)
`current Ptr` es `NULL`
```
newPtr->nextPtr = currentPtr;
previousPtr->nextPtr = newPtr;
```
Para insertar `"Pritchard"`
`previousPtr` apunta al nodo que contiene `"Jones"`
`currentPtr` apunta al nodo que contiene `"Smith"`.
```
newPtr->nextPtr = currentPtr;
previousPtr->nextPtr = newPtr;
```
d)
```
current Ptr = startPtr;
while (current Ptr != NULL) {
   printf("Lastname = %s\nGrade = %6.2f\n",
          currentPtr->lastName, currentPtr->grade);
   currentPtr = current Ptr->nextPtr;
}
```
e)
```
currentPtr = startPtr;
while (current Ptr != NULL) {
   tempPtr = currentPtr;
   currentPtr = currentPtr->nextPtr;
   free(tempPtr);
}
startPtr = NULL;
```

12.5 El recorrido enorden es:

```
11 18 19 28 32 40 44 49 69 71 72 83 92 97 99
```

El recorrido preorden es:

```
49 28 18 11 19 40 32 44 83 71 69 72 97 92 99
```

El recorrido postorden es:

```
11 19 18 32 44 40 28 69 72 71 92 99 97 83 49
```

Ejercicios

12..6 Escriba un programa que concatene dos listas enlazadas de caracteres. El programa deberá incluir la función **concatenate** que toma como argumentos apuntadores a ambas listas y que concatena la segunda lista a la primera.

12.7 Escriba un programa que combine dos listas ordenadas de enteros en una sola lista ordenada de enteros. La función **merge** deberá recibir apuntadores al primer nodo de cada una de las listas a combinarse, y deberá regresar un apuntador al primer nodo de la lista combinada.

12.8 Escriba un programa que inserte 25 enteros al azar desde 0 hasta 100 en orden en una lista enlazada. El programa deberá calcular la suma de los elementos, y el promedio en punto flotante de los mismos.

12.9 Escriba un programa que origine una lista enlazada de 10 caracteres, y a continuación origine una copia de la lista de orden inverso.

12.10 Escriba un programa que introduzca una línea de texto y utilice una pila para imprimir la línea invertida.

12.11 Escriba un programa que utilice una pila para determinar si una cadena es un palíndromo (es decir, si la cadena se deletrea en forma idéntica hacia adelante y hacia atrás). El programa deberá ignorar espacios y puntuaciones.

12.12 Las pilas son utilizadas por los compiladores para auxiliarse en el proceso de evaluar expresiones y para generar código en lenguaje máquina. En este ejercicio y en el siguiente, investigamos cómo los compiladores evalúan expresiones aritméticas formadas únicamente por constantes, operadores y paréntesis.

Los seres humanos normalmente escriben las expresiones como `3 + 4` y `7 /9`, en el cual el operador (aquí `+` o `/`) se escribe entre sus operandos —esto se conoce como *notación infija*. Las computadoras "prefieren" la *notación postfija*, en la cual el operador se escribe a la derecha de sus dos operandos. Las expresiones infijas anteriores aparecerían en notación postfija como `3 4 +` y `7 9 /`, respectivamente.

Para evaluar una expresión infija compleja, un compilador primero convertiría la expresión a notación postfija, y a continuación evaluaría la versión postfija de la expresión. Cada uno de estos algoritmos requiere únicamente de una pasada de izquierda a derecha en la expresión. Cada algoritmo utiliza una pila para apoyarse en esta operación, y en cada uno la pila se utiliza para fines distintos.

En este ejercicio, usted escribirá una versión en C del algoritmo de conversión de infijos a postfijos. En el siguiente ejercicio, escribirá una versión en C del algoritmo de evaluación de la expresión postfija.

Escriba un programa que convierta una expresión infija aritmética ordinaria (suponga que se ha introducido una expresión válida) con enteros de un solo dígito, como

```
(6 + 2) * 5 - 8 / 4
```

a una expresión postfija. La expresión postfija de la expresión infija anterior es

```
6 2 + 5 * 8 4 / -
```

El programa debería leer la expresión al arreglo de caracteres **infix**, y utilizar las versiones modificadas de las funciones de pila puestas en práctica en este capítulo para crear la expresión postfija en el arreglo de caracteres **postfix**. El algoritmo para crear una expresión postfija es como sigue:

1) Inserte (push) un paréntesis izquierdo `'('` en la parte superior de la pila.
2) Agregue un paréntesis derecho `')'` al final de infix.
3) En tanto la pila no esté vacía, lea infix de izquierda a derecha y haga lo siguiente:
   Si el carácter actual en **infix** es un dígito, cópielo al siguiente elemento de **postfix**.
   Si el carácter actual en **infix** es un paréntesis izquierdo, insértelo (push) sobre la pila.
   Si el carácter actual en **infix** es un operador,
      Retire (pop) los operadores (si es que hay alguno) en la parte superior de la pila en tanto tengan precedencia igual o mayor que el operador actual, e inserte los operadores retirados en **postfix**.
   Inserte (push) el carácter actual en **infix** sobre la pila.

Si el carácter actual en `infix` es un paréntesis derecho
Retire (pop) los operadores de la parte superior de la pila e insértelos en `postfix` hasta que en la parte superior de la pila quede un paréntesis izquierdo.
Retire (pop) y descarte el paréntesis izquierdo de la pila.

Las siguientes operaciones aritméticas se permiten en una expresión:

+	adición
-	substracción
*	multiplicación
/	división
^	exponenciación
%	módulo

La pila deberá ser mantenida utilizando las siguientes declaraciones:

```
struct stackNode {
   char data;
   struct stackNode *nextPtr;
};
typedef struct stackNode STACKNODE;
typedef STACKNODE *STACKNODEPTR;
```

El programa deberá estar formado por `main` y 8 otras funciones, con los siguientes encabezados de función:

`void convertToPostfix(char infix[], char postfix[])`
Convierta la expresión infija a notación postfija.
`int isOperator(char c)`
Determinar si `c` es un operador.
`int precedence(char operator1, char operator2)`
Determinar si la precedencia de `operador1` es menor que, igual a, o mayor que la precedencia de `operador2`. La función regresa -1, 0 y 1, respectivamente.
`void push(STACKNODEPTR *topPtr, char value)`
Insertar (push) un valor en la pila.
`char pop(STACKNODEPTR *topPtr)`
Retirar (pop) un valor de la pila.
`char stackTop(STACKNODEPTR topPtr)`
Regresar el valor superior de la pila sin retirar nada de la pila.
`int isEmpty(STACKNODEPTR topPtr)`
Determinar si la pila está vacía.
`void printStack(STACKNODEPTR topPtr)`
Imprimir la pila.

12.13 Escriba un programa que evalúe una expresión postfija (suponga que es válida) como

```
6 2 + 5 * 8 4 / -
```

El programa deberá leer una expresión postfija formada de dígitos y de operadores a un arreglo de caracteres. Utilizando versiones modificadas de las funciones de pila puestas anteriormente en práctica en este capítulo, el programa deberá rastrear la expresión y evaluarla. El algoritmo es como sigue:

1) Agregue el carácter `NULL` (`'\0'`) al final de la expresión postfija. Cuando se encuentre el carácter `NULL`, ya no es necesario procesamiento adicional.
2) En tanto no se encuentre `'\0'`, lea la expresión de izquierda a derecha.
   Si el carácter actual es un dígito,
   Inserte (push) su valor entero en la pila (el valor entero de un carácter de dígito es su valor en el conjunto de caracteres de la computadora, menos el valor de `'0'` en el conjunto de caracteres de la computadora).

De lo contrario, si el carácter actual es un operador,
Retire (pop) los dos elementos superiores de la pila a las variables `x` e `y`.
Calcule `y operator x`
Inserte (push) el resultado del cálculo en la pila.
3) Cuando en la expresión se encuentre el carácter `NULL`, retire (pop) el valor superior de la pila. Ese es el resultado de la expresión postfija.

Nota: en 2) arriba, si el operador es `'/'`, la parte superior de la pila es `2`, y el siguiente elemento en la pila es `8`, entonces retira (pop) `2` a `x`, retira (pop) `8` a `y`, evalúa `8/2` e inserta (push) el resultado con `4`, de regreso a la pila. Esta nota también se aplica al operador `'-'`. Las operaciones aritméticas permitidas en una expresión son:

+	adición
-	substracción
*	multiplicación
/	división
^	exponenciación
%	módulo

La pila deberá ser mantenida utilizando las declaraciones siguientes:

```
struct stackNode {
   int data;
   struct stackNode *nextPtr;
};
typedef struct stackNode STACKNODE;
typedef STACKNODE *STACKNODEPTR;
```

El programa deberá consistir de `main` y de 8 otras funciones, con los siguientes encabezados de función:

`int evaluatePostfixExpression(char *expr)`
Evalúa la expresión postfija.
`int calculate(int op1, int op2, char operator)`
Evalúa la expresión `op1 operator op2`.
`void push(STACKNODEPTR *topPtr, int value)`
Insertar (push) un valor en la pila.
`int pop(STACKNODEPTR *topPtr)`
Retirar (pop) un valor de la pila.
`int isEmpty(STACKNODEPTR topPtr)`
Determinar si la pila está vacía.
`void printStack(STACKNODEPTR topPtr)`
Imprimir la pila.

12.14 Modifique el programa evaluador postfijo del ejercicio 12.13, de tal forma que pueda procesar operandos de enteros mayores que 9.

12.15 *(Simulación de supermercado).* Escriba un programa que simule una línea de caja en el supermercado. La línea es una cola de espera. Los clientes llegan a intervalos enteros al azar, de 1 a 4 minutos. También, cada cliente es atendido en intervalos enteros al azar, de entre 1 y 4 minutos. Obviamente, las velocidades necesitan equilibrarse. Si la velocidad de llegada promedio es mayor que la velocidad promedio de servicio, la cola crecerá en forma indefinida. Inclusive tratándose de velocidades equilibradas, el azar todavía puede causar líneas largas. Ejecute la simulación de supermercado para un día de 12 horas (720 minutos) utilizando el algoritmo siguiente:

1) Escoja un entero al azar entre 1 y 4 para determinar el minuto en el cual el primer cliente llega.
2) En el momento de llegada del primer cliente:
   Determine el tiempo de servicio para ese cliente (entero al azar de 1 a 4);
   Empiece a darle servicio al cliente;

programe el tiempo de llegada del siguiente cliente (un entero al azar de 1 a 4, añadido al tiempo actual).
3) Para cada minuto del día:
Si llega el siguiente cliente,
Póngalo en la cola;
Programe el tiempo de llegada del siguiente cliente;
Si para el último cliente el servicio ya fue terminado;
Dígalo así
Retire de la cola para darle servicio al cliente siguiente
Determine el tiempo de terminación del servicio del cliente (entero al azar de 1 a 4, añadido al tiempo actual).

Ahora ejecute su simulación durante 720 minutos y conteste cada una de las siguientes preguntas:

a) ¿Cuál es en cualquier momento el número máximo de clientes en la cola?
b) ¿Cuál es la espera más larga experimentada por un cliente?
c) ¿Qué pasa si el intervalo de llegadas es modificado de 1 a 4 minutos, a 1 a 3 minutos?

12.16 Modifique el programa de la figura 12.19 para permitir que el árbol binario contenga valores duplicados.

12.17 Escriba un programa basado en el programa de la figura 12.19 que introduzca una línea de texto, divida léxicamente la oración en palabras por separado, inserte las palabras en un árbol de búsqueda binaria, e imprima los recorridos enorden, preorden y postorden del árbol.

Sugerencia: lea la línea de texto a un arreglo. Utilice `strtok` para dividir léxicamente el texto. Una vez encontrada una división, genere un nuevo nodo en el árbol, asigne el apuntador que regresa `strtok` al miembro `string` del nuevo nodo, e inserte el nodo en el árbol.

12.18 En este capítulo, vimos al crear un árbol de búsqueda binario, que la eliminación de duplicados es sencilla. Describa cómo ejecutaría usted la eliminación de duplicados utilizando únicamente un arreglo de un solo subíndice. Compare el rendimiento de la eliminación de duplicados basado en arreglos, con el rendimiento de la eliminación de duplicados basado en el árbol de búsqueda binario.

12.19 Escriba una función `depth` que recibe un árbol binario y que determine cuántos niveles tiene.

12.20 *(Impresión recursiva de una lista en orden inverso).* Escriba una función `printListBackwards` que en forma recursiva extraiga los elementos en una lista en orden inverso. Utilice su función en un programa de prueba que origine una lista ordenada de enteros e imprima la lista en orden inverso.

12.21 *(Búsqueda de una lista en forma recursiva).* Escriba una función `searchList` que busque en una lista enlazada en forma recursiva, buscando un valor específico. Si éste es hallado la función deberá regresar un apuntador al valor; de lo contrario, `NULL` deberá ser regresado. Utilice su función en un programa de prueba que origine una lista de enteros. El programa deberá solicitar al usuario un valor a localizar dentro de la lista.

12.22 *(Borrado de árbol binario).* En este ejercicio, analizamos el borrado de elementos de árboles de búsqueda binarios. El algoritmo de borrado no es tan sencillo como el algoritmo de inserción. Al borrar un elemento se presentan tres casos distintos —el elemento está contenido en un nodo de hoja (es decir, no tiene hijo), el elemento está contenido en un nodo que tiene un hijo, o el elemento está contenido en un nodo que tiene dos hijos.

Si el elemento a borrarse está contenido en un nodo de hoja, el nodo se borra y el apuntador en el nodo padre se define a `NULL`.

Si el elemento a borrarse está contenido en un nodo con un hijo, el apuntador en el nodo padre se define para apuntar al nodo hijo y el nodo conteniendo el elemento de dato se borra. Esto hace que el nodo hijo tome el lugar del nodo borrado dentro del árbol.

El último caso es el más difícil. Cuando es borrado un nodo que tiene dos hijos, otro nodo dentro del árbol debe tomar su lugar. Sin embargo, el apuntador en el nodo padre no puede simplemente asignarse para apuntar a uno de los hijos del nodo a borrarse. En la mayor parte de los casos, el árbol de búsqueda binario no cumpliría con la característica siguiente de los árboles de búsqueda binarios: *los valores en cualquier subárbol izquierdo son menores que el valor en el nodo padre, y los valores en cualquier subárbol derecho son mayores que el valor en el nodo padre.*

¿Qué nodo se utilizará como *nodo de remplazo* para poder mantener esta característica? Ya sea el nodo que contenga el valor más grande en el árbol menor que el valor del nodo a borrarse, o el nodo que contenga el valor más pequeño en el árbol mayor que el valor del nodo a borrarse. Consideremos el nodo con el valor menor. En un árbol de búsqueda binario, el valor menor que el valor del padre se localiza en el subárbol izquierdo del nodo padre y se garantiza que está contenido en el nodo más a la derecha del subárbol. Este nodo se localiza dirigiéndose hacia abajo por el subárbol izquierdo, hacia la derecha, hasta que resulte `NULL` el apuntador hacia el hijo derecho del nodo actual. Estamos ahora apuntando al nodo de remplazo, que es o un nodo de hoja o un nodo con un hijo a su izquierda. Si el nodo de remplazo es un nodo de hoja, los pasos para llevar a cabo el borrado son como sigue:

1) Almacene el apuntador al nodo a borrarse en una variable de apuntador temporal (este apuntador se utiliza para liberar memoria dinámicamente asignada).
2) Defina el apuntador en el padre del nodo a borrarse, para que apunte al nodo de remplazo.
3) Defina a `NULL` el apuntador en el padre del nodo de remplazo.
4) Defina el apuntador al subárbol derecho, en el nodo de remplazo, para que apunte al subárbol derecho del nodo a borrarse.
5) Borre el nodo al cual apunta la variable de apuntador temporal.

Los pasos de borrado para un nodo de remplazo con un hijo izquierdo son similares a los de un nodo de remplazo sin hijos, pero el algoritmo también debe mover el hijo a la posición del nodo de remplazo en el árbol. Si el nodo de remplazo es un nodo con un hijo izquierdo, los pasos a llevarse a cabo para el borrado son como sigue:

1) Almacene en una variable de apuntador temporal el apuntador al nodo a borrarse.
2) Defina el apuntador en el padre del nodo a borrarse, para apuntar al nodo de remplazo.
3) Defina el apuntador en el padre del nodo de remplazo, para apuntar al hijo izquierdo del nodo de remplazo.
4) Defina el apuntador al subárbol derecho en el nodo de remplazo, para apuntar al subárbol derecho del nodo a borrarse.
5) Borre el nodo hacia el cual apunta la variable de apuntador temporal.

Escriba la función `deleteNode`, que toma como sus argumentos un apuntador al nodo raíz del árbol y el valor a borrarse. La función deberá localizar el nodo que contenga el valor a borrarse en el árbol y utilizar los algoritmos analizados aquí para borrarlo. Si no se encuentra el valor en el árbol, la función deberá imprimir un mensaje, que indique si se borra o no el valor. Modifique el programa de la figura 12.19 para utilizar esta función. Después de borrar un elemento, llame a las funciones de recorrido `inOrder`, `preOrder` y `postOrder` para confirmar que la operación de borrado fue correctamente ejecutada.

12.23 *(Búsqueda de árbol binario).* Escriba la función `binaryTreeSearch`, que intenta localizar un valor especificado en un árbol de búsqueda binario. La función deberá tomar como argumentos un apuntador al nodo raíz del árbol binario y una clave de búsqueda a localizar. Si se encuentra el nodo que contenga la clave de búsqueda, la función deberá regresar un apuntador a dicho nodo; de lo contrario, la función deberá regresar un apuntador `NULL`.

12.24 *(Recorrido de árbol binario en orden de nivel).* El programa de la figura 12.19 ilustró tres métodos recursivos de recorrer un árbol binario —recorrido enorden, recorrido preorden y recorrido postorden. En este ejercicio se presenta el *recorrido en orden de nivel* de un árbol binario, en el cual los valores de nodo se imprimen nivel por nivel, iniciándose en el nivel del nodo raíz. En cada nivel los nodos se imprimen de

izquierda a derecha. El recorrido en orden de nivel no es un algoritmo recursivo. Utiliza la estructura de datos de cola de espera para controlar la salida de los nodos. El algoritmo es como sigue:

1) Inserte en la cola de espera el nodo raíz.
2) En tanto existan nodos en la cola,
   Obtenga el nodo siguiente en la cola
   Imprima el valor del nodo
   Si no es `NULL` el apuntador al hijo izquierdo del nodo
      Inserte en la cola el nodo del hijo izquierdo
   Si no es `NULL` el apuntador al hijo derecho del nodo
      Inserte en la cola el nodo del hijo derecho.

Escriba la función `levelOrder` para ejecutar un recorrido en orden de nivel de un árbol binario. La función deberá tomar como argumento un apuntador al nodo raíz del árbol binario. Modifique el programa de la figura 12.19 para utilizar esa función. Compare la salida correspondiente a esa función con las salidas de otros algoritmos de recorrido, para ver si funcionó correctamente. (Nota: también será necesario modificar e incorporar dentro de este programa las funciones de procesamiento de colas de la figura 12.13.)

12.25 *(Impresión de árboles).* Escriba una función recursiva `outputTree` para desplegar en pantalla un árbol binario. La función deberá de extraer el árbol, renglón por renglón, con la parte superior del árbol en la parte izquierda de la pantalla, y la parte inferior del árbol hacia la derecha de la pantalla. Cada renglón es extraído en forma vertical. Por ejemplo, el árbol binario ilustrado en la figura 12.22 es extraído como sigue:

```
            99
      97
            92
   83
            72
      71
            69
49
            44
      40
            32
   28
            19
      18
            11
```

Note que el nodo de hoja más a la derecha aparece en la parte superior de la salida, en la columna más a la derecha y el nodo de raíz aparece a la izquierda de la salida. Cada columna extraída se inicia cinco espacios a la derecha de la columna anterior. La función `outputTree` debe recibir como argumentos un apuntador al nodo raíz del árbol y un entero `totalSpaces`, representando el número de espacios que anteceden al valor a ser extraído (esta variable deberá iniciarse en cero, de tal forma que el nodo raíz sea extraído en la parte izquierda de la pantalla). La función utiliza para sacar el árbol un recorrido enorden modificado —se inicia en el nodo más a la derecha en el árbol y funciona de regreso hacia el izquierdo. El algoritmo es como sigue:

En tanto no sea `NULL` el apuntador al nodo actual
   Llame recursivamente `outputTree` con el subárbol derecho del nodo actual y `totalSpace + 5`
   Utilice una estructura `for` para contar de 1 hasta `totalSpace` y extraiga los espacios.

   Extraiga el valor del nodo actual
   Defina el apuntador al nodo actual para que apunte al subárbol izquierdo del nodo actual.
   Incremente `totalSpaces` en 5.

Sección especial: Cómo construir su propio compilador

En los ejercicios 7.18 y 7.19 presentamos el lenguaje de máquina Simpletron (LMS) y creamos el simulador de computadora Simpletron para ejecutar programas escritos en LMS. En esta sección, construiremos un compilador que convierte a LMS programas escritos en un lenguaje de programación de alto nivel. Esta sección se "encadena" con todo el proceso de programación. Escribiremos programas en este nuevo lenguaje de alto nivel, compilaremos los programas en el compilador que construiremos, y ejecutaremos los programas en el simulador que construimos en el ejercicio 7.19.

12.26 *(El Lenguaje Simple).* Antes de que empecemos a construir el compilador, analizamos un lenguaje de alto nivel simple, aunque poderoso, similar a las versiones primeras del popular lenguaje BASIC. Llamamos el lenguaje *Simple*. Cada *enunciado* Simple consiste de un *número de línea* y de una *instrucción* Simple. Los números de línea deben aparecer en orden ascendente. Cada instrucción inicia con uno de los siguientes *comandos* Simple: `rem`, `input`, `let`, `print`, `goto`, `if/goto` o `end` (vea la figura 12.23). Todos los comandos, a excepción de `end`, pueden ser utilizados en forma repetida. Simple sólo evalúa expresiones enteras utilizando los operadores +, -, *, y /. Estos operadores tienen la misma precedencia que en C. Pueden utilizarse paréntesis para cambiar el orden de evaluación de una expresión.

Nuestro compilador Simple reconoce únicamente letras minúsculas. Todos los caracteres en un archivo Simple deberán ser minúsculas (las letras mayúsculas resultarán en un error de sintaxis, a menos de que aparezcan en un enunciado `rem`, en cuyo caso serán ignoradas). Un *nombre variable* es una sola letra. Simple no permite nombres descriptivos de variables, por lo que las variables deberán ser explicadas en comentarios, para documentar su uso dentro del programa. Simple utiliza únicamente variables enteras. Simple no tiene declaraciones de variables —el mero mencionar de un nombre de variable en un programa hace que la variable sea declarada y automáticamente inicializada a cero. La sintaxis de Simple no permite

Comando	Enunciado de muestra	Descripción
`rem`	`50 rem this is a remark`	Cualquier texto que siga al comando `rem` es sólo para fines de documentación y será ignorado por el compilador
`input`	`30 input x`	Despliega un signo de interrogación, para solicitar al usuario que introduzca un entero. Lee dicho entero del teclado y almacena el entero en `x`.
`let`	`80 let u = 4 * (j - 56)`	Asigna `u` el valor de `4* (j - 56)`. Note que a la derecha del signo igual puede aparecer una expresión arbitrariamente compleja.
`print`	`10 print w`	Despliega el valor de `w`.
`goto`	`70 goto 45`	Transfiere el control del programa a la línea `45`.
`if/goto`	`35 if i == z goto 80`	Compara `i` con `z` buscando igualdad y transfiere el control del programa a la línea `80`, si la condición es verdadera; de lo contrario, continúa la ejecución en el enunciado siguiente.
`end`	`99 end`	Termina la ejecución del programa.

Fig. 12.23 Comandos Simple.

la manipulación de cadenas (lectura de una cadena, escritura de una cadena, comparación de cadenas, etcétera).

Si en un programa Simple se encuentra una cadena (después de un comando distinto que `rem`), el compilador generará un error de sintaxis. Nuestro compilador supondrá que los programas Simple se escriben en forma correcta. En el ejercicio 12.19 se le solicita al estudiante que modifique el compilador para que lleve a cabo verificación de errores de sintaxis.

Simple utiliza durante la ejecución del programa el enunciado condicional `if/goto` y el enunciado incondicional `goto` para alterar el flujo de control. Si la condición en el enunciado `if/goto` es verdadera, el control se transfiere a una línea específica dentro del programa. Los siguientes operadores de igualdad relacionales son válidos en un enunciado `if/goto`: `<`, `>`, `<=`, `>=`, `==`, o `!=`. La precedencia de estos operadores es la misma que en C.

Consideremos ahora varios programas Simple que demuestran las características de Simple. El primer programa (la figura 12.24) lee dos enteros del teclado, almacena los valores en las variables `a` y `b` y calcula e imprime su suma (almacenada en la variable `c`).

El programa de la figura 12.25 determina e imprime cuál es el mayor de dos enteros. Los enteros son introducidos desde el teclado y almacenados en `s` y en `t`. El enunciado `if/goto` prueba la condición `s=> t`. Si la condición es verdadera, el control se transfiere a la línea `90` y se extrae `s`; de lo contrario, se extrae `t` y el control es transferido al enunciado `end` en la línea `99`, donde el programa termina.

```
10 rem   determine and print the sum of two integers
15 rem
20 rem   input the two integers
30 input a
40 input b
45 rem
50 rem   add integers and store result in c
60 let c = a + b
65 rem
70 rem   print the result
80 print c
90 rem   terminate program execution
99 end
```

Fig. 12.24 Cómo determinar la suma de dos enteros.

```
10 rem   determine the larger of two integers
20 input s
30 input t
32 rem
35 rem   test if s >= t
40 if s >= t goto 90
45 rem
50 rem   t is greater than s, so print t
60 print t
70 goto 99
75 rem
80 rem   s is greater than or equal to t, so print s
90 print s
99 end
```

Fig. 12.25 Cómo encontrar el mayor de dos enteros.

Simple no posee una estructura de repetición (como `for`, `while` o `do/while` de C). Sin embargo, Simple puede simular cada una de las estructuras de repetición de C, mediante el uso de los enunciados `if/goto` y `goto`. La figura 12.26 utiliza un ciclo controlado por centinela para calcular los cuadrados de varios enteros. Cada entero es introducido desde el teclado y almacenado en la variable `j`. Si el valor introducido es el centinela `-9999`, el control se transfiere a la línea `99`, donde el programa termina. De lo contrario, `k` es asignado el valor del cuadrado de `j`, `k` es extraído a la pantalla y el control se pasa a la línea `20`, donde se introduce el siguiente entero.

Utilizando los programas de muestra de la figura 12.24, figura 12.25 y figura 12.26 como guía, escriba un programa Simple para llevar a cabo cada uno de los siguientes:

a) Introducir tres enteros, determinar su promedio e imprimir el resultado.
b) Utilizar un ciclo controlado por centinela para introducir 10 enteros y calcular e imprimir su suma.
c) Utilizar un ciclo controlado por contador para introducir 7 enteros, algunos positivos y algunos negativos, y calcular e imprimir su promedio.
d) Introducir una serie de enteros y determinar e imprimir cuál es el mayor. El primer entero introducido indicará cuántos números deberán de ser procesados.
e) Introducir 10 enteros e imprimir el más pequeño.
f) Calcular e imprimir la suma de enteros pares, desde 2 hasta 30.
g) Calcular e imprimir el producto de los enteros impares desde 1 hasta 9.

12.27 *(Cómo construir un compilador; prerrequisito: haber terminado los ejercicios 7.18, 7.19. 12.12, 12.13 y 12.26).* Ahora que se ha presentado el lenguaje Simple (ejercicio 12.26), analizamos cómo construir nuestro compilador Simple. Primero, consideraremos el proceso mediante el cual un programa Simple se convierte a LMS y es ejecutado por el simulador Simpletron (vea la figura 12.27). Un archivo que contiene un programa Simple es leído por el compilador y convertido a código LMS. El código LMS es extraído a un archivo en disco, en el cual las instrucciones LMS aparecen, una por línea. A contiuación el archivo LMS es cargado en el simulador Simpletron, y los resultados se envían a un archivo en disco y a pantalla. Note que el programa Simpletron desarrollado en el ejercicio 7.19, tomó su entrada desde el teclado. Deberá ser modificado para poder leerse desde un archivo, de tal forma que pueda ejecutar los programas producidos por nuestro compilador.

Para convertirlo a LMS, el compilador ejecuta dos *pasadas* del programa Simple. La primera pasada construye una *tabla simbólica*, en la cual se almacenan todos *los números de línea, nombres de variable* y *constantes* del programa Simple, con su tipo y posición correspondiente en el código final LMS (la tabla simbólica se analiza en detalle más adelante). Esta primera pasada también produce las instrucciones correspondientes LMS para cada enunciado Simple. Como veremos, si el programa Simple contiene

```
10 rem   calculate the squares of several integers
20 input j
23 rem
25 rem   test for sentinel value
30 if j == -9999 goto 99
33 rem
35 rem   calculate square of j and assign result to k
40 let k = j * j
50 print k
53 rem
55 rem   loop to get next j
60 goto 20
99 end
```

Fig. 12.26 Cómo calcular los cuadrados de varios enteros.

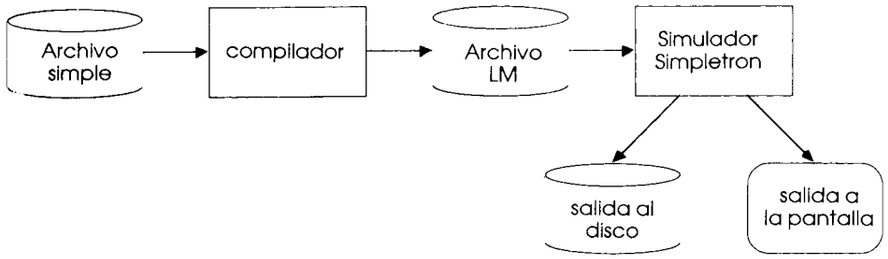


Fig. 12.27 Cómo escribir, compilar y ejecutar un programa de lenguaje Simple.

enunciados en el programa que transfieren el control a una línea más adelante, la primera pasada resulta en un programa LMS que contiene algunas instrucciones incompletas. La segunda pasada del compilador localiza y completa las instrucciones incompletas, y extrae el programa LMS a un archivo.

`Primera pasada`

El compilador empieza leyendo a la memoria un enunciado del programa Simple. La línea deberá ser dividida en sus *componentes léxicos* individuales (es decir, las "piezas" de un enunciado) para proceso y compilación (para facilitar esta tarea se puede utilizar la función estándar de biblioteca **`strtok`**). Recuerde que cada enunciado se inicia con un número de línea seguido por un comando. Conforme el compilador divide un enunciado en *componentes léxicos*, si el componente es un número de línea, una variable o una constante, es colocado en la tabla simbólica. Un número de línea se coloca en la tabla simbólica sólo si se trata del primer componente de un enunciado. El **`symbolTable`** es un arreglo de estructuras **`tableEntry`** que representan cada uno de los símbolos en el programa. No existen restricciones en el número de símbolos que pueden aparecer en el programa. Por lo tanto, el **`symbolTable`** para un programa en particular pudiera resultar estenso. Por ahora, haga **`symbolTable`** un arreglo de 100 elementos. Una vez que el programa esté trabajando podrá aumentar o disminuir su tamaño.

La definición de estructura **`tableEntry`** es como sigue:

```
struct tableEntry {
   int symbol;
   char type;          /* 'C', 'L', or 'V' */
   int location; /* 00 to 99 */
}
```

Cada estructura **`tableEntry`** contiene tres miembros. El miembro **`symbol`** es un entero que contiene la representación ASCII de una variable (recuerde que los nombres de variable son un carácter solo), un número de línea o una constante. El miembro **`type`** es uno de los siguientes caracteres, indicando el tipo del símbolo: **`'C'`** para constante, **`'L'`** para número de línea, o **`'V'`** para variable. El miembro **`location`** contiene la posición en memoria Simpletron (**`00`** a **`99`**) al cual se refiere el símbolo. La memoria Simpletron es un arreglo de 100 enteros, en el cual se almacenan las instrucciones LMS y los datos. Para un número de línea, la posición es el elemento en el arreglo de memoria Simpletron en el cual las instrucciones LMS empiezan para el enunciado Simple. Para una variable o una constante, la posición es el elemento en el arreglo de memoria Simpletron en el cual está almacenada la variable o la constante. Las variables y las constantes están asignadas hacia atrás a partir del final de la memoria Simpletron. La primera variable o constante se almacena en la posición en **`99`**, la siguiente en la posición en **`98`**, etcétera.

La tabla simbólica juega una parte integral en la conversión de programa Simple a LMS. Aprendimos en el capítulo 7 que una instrucción LMS es un entero de cuatro dígitos, formados de dos partes —el *código de operación* y el *operando*. El código de operación está determinado en Simple mediante comandos. Por ejemplo, el comando sencillo **`input`** corresponde al código de operación LMS **`10`** (leer), y el comando Simple **`print`** corresponde al código de operación LMS **`11`** (escribir). El operando es una posición de

memoria, que contiene los datos sobre los cuales el código de operación ejecutará su tarea (por ejemplo, el código de operación **`10`** lee un valor del teclado y lo almacena en la posición de memoria especificado por el operando). El compilador busca en **`symbolTable`** para determinar la posición en la memoria Simpletron de cada símbolo, a fin de que la posición correspondiente pueda ser utilizada para completar las instrucciones LMS.

La compilación de cada enunciado Simple se basa en su comando. Por ejemplo, una vez inserto en la tabla simbólica el número de línea de un enunciado **`rem`**, el resto del enunciado es ignorado por el compilador, dado que el comentario es únicamente para fines de documentación. Los enunciados **`input`**, **`print`**, **`goto`** y **`end`** corresponden a las instrucciones *read, write, branch* (a una posición específica) y *halt* de LMS. Los enunciados que contengan estos comandos Simple se convierten directamente a LMS (note que un enunciado **`goto`** pudiera contener una referencia sin resolver, si la línea especificada se refiere a un enunciado posterior del archivo de programa Simple; esto a veces se conoce como una referencia hacia adelante).

Cuando se compila un enunciado **`goto`** con una referencia sin resolver, la instrucción LMS deberá ser marcada con una *bandera* para indicar que la instrucción será terminada en la segunda pasada del compilador. Las banderas se almacenan en un arreglo de 100 elementos **`flags`** del tipo **`int`**, en el cual cada elemento se inicializa a **`-1`**. Si la posición de memoria a la cual se refiere el número de línea en el programa Simple aún no es conocido (es decir, aún no aparece en la tabla simbólica), el número de línea se almacena en el arreglo **`flags`** en aquel elemento que tenga el mismo subíndice que la instrucción incompleta. El operando de la instrucción incompleta se establece temporalmente a **`00`**. Por ejemplo, una instrucción incondicional de bifurcación (haciendo una referencia hacia adelante) se deja como **`+4000`** hasta la siguiente pasada del compilador. La segunda pasada del compilador será descrita en breve.

La compilación de los enunciados **`if/goto`** y **`let`** es más complicada que la de otros enunciados —son los únicos enunciados que producen más de una instrucción LMS. Para un enunciado **`if/goto`**, el compilador produce código para probar la condición y si es necesario, para bifurcarse a otra línea. El resultado de la bifurcación podría ser una referencia sin resolver. Cada uno de los operadores relacionales y de igualdad pueden ser simulados utilizando las instrucciones *branch zero* y *branch negative* de LMS (o posiblemente una combinación de ambas).

En el caso de un enunciado **`let`**, el compilador produce un código para evaluar una expresión arbitrariamente compleja, compuesta de variables y/o constantes enteras. Las expresiones deberán de separar cada operando y operador con espacios. Los ejercicios 12.12 y 12.13 presentaron el algoritmo de conversión infijo a postfijo y el algoritmo de evaluación postfijo, utilizado por compiladores para evaluar expresiones. Antes de seguir adelante con su compilador, deberá de haber completado cada uno de esos ejercicios. Cuando un compilador se encuentra con una expresión, convierte la expresión de notación infija a notación postfija, y a continuación evalúa la expresión postfija.

¿Cómo es que el compilador produce el lenguaje máquina para valuar una expresión que contenga variables? El algoritmo de evaluación postfijo contiene un "gancho" que permite que nuestro compilador genere instrucciones LMS, en vez de que proceda a evaluar la expresión. Para habilitar este "gancho" dentro del compilador, deberá ser modificado el algoritmo de evaluación postfijo, para que cada símbolo que encuentre (y posiblemente inserte) sea buscado en la tabla simbólica, determine la posición de memoria correspondiente a dicho símbolo, e *inserte (push) la posición de memoria sobre la pila en lugar del símbolo.* Cuando en la expresión postfija se encuentra un operador, son extraídas (pop) las dos posiciones de memoria de la parte superior de la pila, y se produce lenguaje máquina para llevar a cabo la operación, utilizando como operandos las posiciones de memoria. El resultado de cada una de estas subexpresiones se almacena en una posición de memoria temporal y se inserta (push) de vuelta a la pila, de tal forma que se pueda continuar con la evaluación de la expresión postfija. Cuando quede terminada la evaluación postfija, la posición de memoria que contiene el resultado es la única posición remanente sobre la pila. Esta es extraída (popped) y se generan instrucciones LMS para asignar el resultado a la variable en la parte izquierda del enunciado **`let`**.

Segunda pasada

La segunda pasada del compilador lleva a cabo dos tareas; resolver cualquier referencia pendiente de resolver y extraer el código LMS a un archivo. La resolución de las referencias se ejecuta como sigue:

1) Busca en el arreglo `flags` alguna referencia sin resolver (es decir, un elemento que tenga un valor distinto a `-1`).
2) Localiza la estructura en el arreglo `symbolTable`, que contiene el símbolo almacenado en el arreglo `flags` (asegúrese que el tipo del símbolo es `'L'`, correspondiente a número de línea).
3) Inserte la posición de memoria del miembro de estructura `location` dentro de la instrucción con la referencia pendiente de resolver (recuerde que una instrucción que contenga una referencia pendiente de resolver tiene un operando `00`).
4) Repita los pasos 1, 2 y 3, hasta que se llegue al final del arreglo `flags`.

Una vez terminado el proceso de resolución, todo el arreglo que contiene el código LMS es extraído a un archivo de disco, con una instrucción LMS por línea. Este archivo puede ser leído por el Simpletron para su ejecución (después de que el simulador haya sido modificado para que pueda leer su entrada a partir de un archivo).

Un ejemplo completo

El ejemplo siguiente ilustra una conversión completa de un programa Simple a LMS, como sería ejecutado por el compilador Simple. Considere un programa Simple, que introduce un entero y suma los valores desde 1 hasta dicho entero. El programa y las instrucciones LMS producidas por la primera pasada se ilustran en la figura 12.28. La tabla simbólica construida en la primera pasada se muestra en la figura 12.29.

La mayor parte de los enunciados Simple se convierten directamente a instrucciones sencillas LMS. Las excepciones a este programa son los comentarios, el enunciado `if/goto` de la línea `20`, y los enunciados `let`. Los comentarios no se traducen a lenguaje máquina. Sin embargo, el número de línea correspondiente a un comentario es colocado en la tabla simbólica, por si en un enunciado `goto` o en un enunciado `if/goto` el número de línea aparece referenciado. La línea `20` del programa especifica que si la condición `y == x` es verdadera, el control del programa se transfiere a la línea `60`. Dado que la línea `60` aparece más adelante en el programa, en la primera pasada del compilador no ha colocado aún `60` en la tabla simbólica (los números de línea se colocan en la tabla simbólica únicamente cuando aparecen como primera división léxica de un enunciado). Por lo tanto, en este momento no es posible determinar el operando de la instrucción *branch zero* LMS en la posición `03` en el arreglo de instrucciones LMS. El compilador colocará `60` en la posición `03` del arreglo `flags`. para indicar que esta instrucción se completará en la segunda pasada.

En el arreglo LMS debemos mantener registro de la posición de la siguiente instrucción, porque no existe una correspondencia uno a uno, entre los enunciados Simple y las instrucciones LMS. Por ejemplo, el enunciado `if/goto` de la línea `20` se compila en tres instrucciones LMS. Cada vez que se produce una instrucción, dentro del arreglo LMS debemos incrementar el *contador de instrucciones* a la siguiente posición. Note que el tamaño de la memoria Simpletron podría presentar un problema para los programas Simple que contengan muchos enunciados, variables y constantes. Es concebible que el compilador se quede sin memoria disponible. Para comprobar si éste es el caso, su programa deberá contener un *contador de datos,* a fin de llevar registro de la posición dentro del arreglo LMS en la cual se almacenará la siguiente variable o constante. Si el valor del contador de instrucciones es mayor que el valor que el del contador de datos, el arreglo LMS está lleno. En este caso, deberá darse por terminado el proceso de compilación y el compilador deberá imprimir un mensaje de error, indicando que durante la compilación se le terminó la memoria.

Revisión paso a paso del proceso de compilación

Recorramos el proceso de compilación del programa Simple de la figura 12.28. El compilador lee a la memoria la primera línea del programa

```
5 rem sum 1 to x
```


Programa simple	Localización e instrucción LMS		Descripción
`5 rem sum 1 to x`		*ninguna*	`rem` es ignorado
`10 input x`	`00`	`+1099`	leer `x` a la posición `99`
`15 rem check y == x`		*ninguna*	`rem` es ignorado
`20 if y == x goto 60`	`01`	`+2098`	cargar `y` (`98`) a acumulador
	`02`	`+3199`	substraer `x` (`99`) del acumulador
	`03`	`+4200`	*bifurcar cero a posición sin resolver*
`25 rem increment y`		*ninguna*	`rem` es ignorado
`30 let y = y + 1`	`04`	`+2098`	cargar `y` al acumulador
	`05`	`+3097`	añadir `1` (`97`) al acumulador
	`06`	`+2196`	almacenar en posición temporal `96`
	`07`	`+2096`	cargar de la posición temporal `96`
	`08`	`+2198`	almacenar acumulador en `y`
`35 rem add y to total`		*ninguna*	`rem` es ignorado
`40 let t = t + y`	`09`	`+2095`	cargar `t` (`95`) al acumulador
	`10`	`+3098`	añadir `y` al acumulador
	`11`	`+2194`	almacenar en posición temporal `94`.
	`12`	`+2094`	cargar de la posición temporal `94`
	`13`	`+2195`	almacenar acumulador en `t`
`45 rem loop y`		*ninguna*	`rem` es ignorado
`50 goto 20`	`14`	`+4001`	bifurcarse a la posición `01`
`55 rem output result`		*ninguna*	`rem` es ignorado
`60 print t`	`15`	`+1195`	extraer `t` a pantalla
`99 end`	`16`	`+4300`	terminar la ejecución.

Fig. 12.28 Instrucciones LMS, producidas después de la primera pasada del compilador.

La primera división léxica del enunciado (el número de línea) se determina utilizando `strtok` (vea el capítulo 8 en el que se analizan las funciones de manipulación de las cadenas en C). El componente léxico regresado por `strtok` se convierte a un entero, mediante el uso de `atoi`, de tal forma que el símbolo `5` puede ser localizado en la tabla simbólica. Si no se encuentra el símbolo, se insertará en la tabla simbólica. Dado que estamos al principio del programa y ésta es la primera línea, en la tabla aún no existen símbolos. Por lo tanto, en la tabla simbólica se inserta `5` como del tipo `L` (número de línea) y se le asigna la primera posición en el arreglo LMS (`00`): aunque esta línea se trata de un comentario, aún así en la tabla simbólica se le asigna un espacio para el número de línea (por si es referenciado por un `goto` o por un `if/goto`). Como en relación con un enunciado `rem` no se genera ninguna instrucción LMS, no se incrementa el contador de instrucciones.

El enunciado

```
10 input x
```

es analizado léxicamente después. El número de línea `10` se coloca en la tabla simbólica como del tipo `L` y se le asigna la primera posición en el arreglo LMS (`00` debido a que un comentario fue el que inició el

Símbolo	Tipo	Posición
5	L	00
10	L	00
'x'	V	99
15	L	01
20	L	01
'y'	V	98
25	L	04
30	L	04
1	C	97
35	L	09
40	L	09
't'	V	95
45	L	14
50	L	14
55	L	15
60	L	15
99	L	16

Fig. 12.29 Tabla simbólica correspondiente al programa 12.28.

programa, por lo que el contador de instrucciones es todavía `00`). El comando `input` indica que la siguiente división léxica es una variable (únicamente una variable puede aparecer en un enunciado `input`). Dado que `input` corresponde directamente a un código de operación LMS, el compilador simplemente tiene que determinar la posición de `x` en el arreglo LMS. El símbolo `x` no se encuentra en la tabla simbólica.

Por lo tanto, en la tabla simbólica se le inserta como la representación ASCII de `x`, se le da el tipo `V`, y en el arreglo LMS se le asigna a la posición `99` (el almacenamiento de datos se inicia en `99` y se asigna hacia atrás). El código LMS puede ser generado a partir de este enunciado. El código de operación `10` (el código de operación de lectura LMS) se multiplica por 100, y se le añade la posición de `x` (como quedó determinada en la tabla simbólica), para completar la instrucción. La instrucción es entonces almacenada en el arreglo LMS en la posición `00`. El contador de instrucciones se aumenta en 1, porque se produjo una sola instrucción LMS.

El enunciado

```
15 rem check y == x
```

se divide léxicamente después. El número de línea `15` es buscado en la tabla simbólica (y no se encuentra). El número de línea se inserta como del tipo `L` y se le asigna la siguiente posición en el arreglo, `01` (recuerde que los enunciados `rem` no producen código, por lo que no se aumenta el contador de instrucciones).

El enunciado

```
20 if y == x goto 60
```

se divide a continuación. El número de línea `20` se inserta en la tabla simbólica, se le da el tipo `L` con la siguiente posición `01` en el arreglo LMS. El comando `if` indica que se tendrá que hacer una evaluación de condición. La variable `y` no se encuentra en la tabla simbólica, por lo que se inserta y dándosele el tipo `V` y la posición LMS `98`. A continuación, se generan instrucciones LMS para evaluar la condición. Dado que no existe un equivalente directo en LMS para `if/goto`, debe de ser simulado, ejecutando un cálculo utilizando `x` y `y` y bifurcación, basándose en el resultado. Si `y` es igual a `x`, el resultado de restar `x` de `y` debe ser cero, por lo que para simular el enunciado `if/goto` la instrucción *branch zero* puede ser utilizada

con el resultado del cálculo. El primer paso requiere que se cargue `y` (de la posición `98` LMS) al acumulador. Esto produce la instrucción `01 +2098`. A continuación se resta `x` del acumulador. Esto produce la instrucción `02 +3199`. El valor que aparece en el acumulador puede ser cero, positivo o negativo. Dado que el operador es `==`, deseamos hacer una *bifurcación cero*. Primero, se busca la tabla simbólica por la posición de bifurcación (en este caso `60`), misma que no se encuentra. Por lo tanto, `60` se coloca en el arreglo `flags` en la posición `03`, y se genera la instrucción `03 +4200` (no podemos añadir la posición de bifurcación, porque todavía no hemos asignado en el arreglo LMS una posición a la línea `60`). El contador de instrucciones se incrementa a `04`.

El compilador sigue con el enunciado

```
25 rem increment y
```

El número de línea 25 se inserta en la tabla simbólica como de tipo `L` y se le asigna la posición LMS `04`. El contador de instrucciones no se incrementa.

Cuando se divide léxicamente el enunciado

```
30 let y = y + 1
```

el número de línea `30` se inserta en la tabla simbólica como del tipo `L` y se le asigna la posición LMS `04`. El comando `let` indica que la línea es un enunciado de asignación. Primero, todos los símbolos en la línea se insertan en la tabla simbólica (si todavía no están ahí). El entero `1` se añade a la tabla simbólica como del tipo `C` y se le asigna la posición LMS `97`. A continuación, el lado derecho de la asignación se convierte de notación infija a postfija. A continuación se evalúa la expresión postfija (`y 1 +`). El símbolo `y` se localiza en la tabla simbólica y se inserta en la pila su posición correspondiente en memoria. El símbolo `1` también es localizado en la tabla simbólica y se inserta dentro de la pila su posición correspondiente en memoria. Cuando se llega al operador `+`, el evaluador de postfijo extrae (pop) la pila hacia el operando derecho del operador y vuelve a extraer la pila nuevamente en el operando izquierdo del operador, y a continuación produce las instrucciones LMS

```
04 +2089 (cargar y)
05 +3097 (añadir 1)
```

El resultado de la expresión se almacena en una posición temporal en memoria (`96`) mediante la instrucción

```
06 +2196 (almacenar temporal)
```

y la posición temporal se inserta en la pila. Ahora que ha sido evaluada la expresión, el resultado debe ser almacenado en `y` (es decir la variable del lado izquierdo del signo igual). Por lo tanto, la posición temporal se carga al acumulador y el acumulador se almacena en `y`, mediante las instrucciones

```
07 +2096 (cargar temporal)
08 +2198 (almacenar y)
```

El lector notará de inmediato que las instrucciones LMS son aparentemente redundantes. Analizaremos este tema en breve.

Cuando se divide léxicamente el enunciado

```
35 rem    add y to total
```

el número de línea 35 se inserta en la tabla simbólica como del tipo `L` y se le asigna la posición `09`.

El enunciado

```
40 let t = t + y
```

es similar a la línea 30. La variable se inserta en la tabla simbólica como en el tipo `V` y se le asigna la posición LMS `95`. Las instrucciones siguen la misma lógica y formato que en la línea `30`, y son generadas las instrucciones `09 + 2095, 10 +3098, 11 +2194, 12 +2094,` y `13 +2195`. Note que el resultado

de `t + y` se asigna a la posición temporal `94` antes de asignarse a `t (95)`. Otra vez, el lector notará que las instrucciones en las posiciones de memoria `11` y `12` aparecen como redundantes. Otra vez, en breve analizaremos lo anterior.

El enunciado

```
45 rem    loop y
```

es un comentario, por lo que la línea `45` se añade a la tabla simbólica como del tipo `L` y se le asigna la posición LMS `14`.

El enunciado

```
50 goto 20
```

transfiere el control a la línea 20. El número de línea `50` se inserta en la tabla simbólica como del tipo `L` y se le asigna a la posición LMS `14`. En LMS el equivalente de `goto` es la instrucción *unconditional branch* (`40`) que transfiere control a una posición LMS específica. El compilador busca la línea 20 en la tabla simbólica y encuentra que corresponde a la posición LMS 01. El código de operación (`40`) se multiplica por 100 y la posición `01` se añade a esta cifra, para producir la instrucción `14 +4001`.

El enunciado

```
55 rem    output result
```

es un comentario, por lo que la línea `55` se inserta a la tabla simbólica como del tipo `L` y se le asigna la posición LMS `15`.

El enunciado

```
60 print t
```

es un enunciado de salida. El número de línea `60` se inserta en la tabla simbólica como del tipo `L` y se le asigna la posición LMS `15`. El equivalente de `print` en código de operación LMS es `11` (*write*). La posición de `t` se determina a partir de la tabla simbólica y se añade al resultado de multiplicar el código de operación por 100.

El enunciado

```
99 end
```

es la línea final del programa. El número de línea `99` se almacena en la tabla simbólica como del tipo `L` y se le asigna la posición LMS `16`. El comando `end` produce la instrucción LMS `+4300` (en LMS `43` es *halt*) que se escribe como instrucción final en el arreglo de memoria LMS.

Esto completa la primera pasada del compilador. Ahora veamos la segunda pasada. Valores distintos a `-1` son buscados dentro del arreglo `flags`. La posición `03` contiene `60`, por lo que el compilador sabe que la instrucción `03` está incompleta. El compilador completa la instrucción buscando en la tabla simbólica el `60`, determinando su posición y añadiendo la posición a la instrucción incompleta. En este caso, la búsqueda determina que la línea `60` corresponde a la posición LMS `15`, por lo que en remplazo de `03` `+4200` se produce la instrucción completa `03 +4215`. Ahora el programa Simple ha quedado ya compilado con éxito.

Para construir el compilador tendrá que llevar a cabo cada una de las siguientes tareas.

a) Modificar el programa simulador Simpletron, que usted escribió en el ejercicio 7.19, para que tome su entrada a partir de un archivo especificado por el usuario (vea el capítulo 11). También, el simulador deberá extraer sus resultados a un archivo de disco, con el mismo formato que la salida a pantalla.

b) Modifique el algoritmo de evaluación infijo a postfijo del ejercicio 12.12, para que pueda procesar operandos enteros multidigitales, y operandos de nombres de variables de una sola letra. *Sugerencia:* puede ser utilizada la función estándar de biblioteca `strtok` para localizar dentro de una expresión cada constante y variable, y las constantes pueden ser convertidas de

cadenas a enteros utilizando la función estándar de biblioteca `atoi`. (Nota: deberá ser modificada la representación de datos de la expresión postfija, para que acepte nombres de variables y constantes enteras).

c) Modifique el algoritmo de evaluación postfijo para que pueda procesar operandos enteros de más de un dígito y operandos de nombre de variable. También, el algoritmo deberá ahora poner en funcionamiento el "gancho" discutido anteriormente, de tal forma que sean producidas las instrucciones LMS en vez de que la expresión se evalúe directamente. Sugerencia: la función estándar de biblioteca `strtok` puede ser utilizada para localizar dentro de una expresión cada una de las constantes y variables, y las constantes pueden ser convertidas de cadenas a enteros utilizando la función estándar de biblioteca `atoi`. (Nota: deberá ser modificada la representación de datos de una expresión postfija, para que acepte nombres de variables y constantes enteras.)

d) Construya el compilador. Incorpore las partes (b) y (c) para evaluar las expresiones en los enunciados `let`. Su programa deberá contener una función que ejecute la primera pasada del compilador, y una función que ejecute la segunda pasada del compilador. Para llevar a cabo sus tareas ambas funciones pueden llamar a otras funciones.

12.28 *(Cómo optimizar el compilador Simple).* Cuando se compila un programa y se convierte a LMS, se generan un conjunto de instrucciones. A menudo ciertas combinaciones de instrucciones se repiten, normalmente en grupos de tres, llamadas *producciones*. Una producción está formada normalmente de tres instrucciones, como *load, add,* y *store*. Por ejemplo, en la figura 12.30 se ilustran cinco de las instrucciones LMS, que fueron producidas en la compilación del programa de la figura 12.28. Las primeras tres instrucciones son la producción que añade `1` a `y`. Note que las instrucciones `06` y `07` almacenan el valor en el acumulador en la posición temporal `96`. Y a continuación vuelven a cargar el valor en el acumulador, de tal forma que la instrucción `08` puede almacenar el valor en la posición `98`. A menudo una producción es seguida por una instrucción de carga para la misma posición que fue almacenada inmediatamente antes. Este código puede ser *optimizado* eliminando la instrucción de almacenar y las instrucciones de cargar subsecuentes, que operan en la misma posición de memoria. Esta optimización permitiría que el programa Simpletron se ejecutara más aprisa, porque en esta versión existirían menos instrucciones. En la figura 12.31 se ilustra el LMS optimizado, correspondiente al programa de la figura 12.28. Note que en el código optimizado hay cuatro instrucciones menos —un ahorro de espacio en memoria del 25%.

Modifique el compilador para que pueda dar la opción de optimizar el código que produce en lenguaje de máquina Simpletron. Compare manualmente el código no optimizado con el código optimizado, y calcule el porcentaje de reducción.

12.29 *(Modificaciones al compilador Simple).* Lleve a cabo las siguientes modificaciones al compilador Simple. Algunas de estas modificaciones también pudieran requerir modificaciones al programa simulador Simpletron escrito en el ejercicio 7.19.

a) Haga posible que en los enunciados `let` se pueda utilizar el operador de módulo (%). El lenguaje de máquina Simpletron deberá ser modificado para incluir la instrucción de módulo.

b) Haga posible la exponenciación en un enunciado `let` utilizando el `^` como operador de exponenciación. El lenguaje de máquina Simpletron debe ser modificado para incluir la instrucción de exponenciación.

```
04 +2098    (load)
05 +3097    (add)
06 +2196    (store)
07 +2096    (load)
08 +2198    (store)
```

Fig. 12.30 Código sin optimizar del programa de la figura 12.28.

Programa simple	Localización e instrucción LMS		Descripción
`5 rem sum 1 to x`		*ninguna*	`rem` es ignorado
`10 input x`	`00`	`+1099`	leer `x` a la posición `99`
`15 rem check y == x`		*ninguna*	`rem` es ignorado
`20 if y == x goto 60`	`01`	`+2098`	cargar `y` (`98`) a acumulador
	`02`	`+3199`	substraer `x` (`99`) del acumulador
	`03`	`+4211`	bifurcar cero a posición sin resolver
`25 rem increment y`		*ninguna*	`rem` es ignorado
`30 let y = y + 1`	`04`	`+2098`	cargar `y` al acumulador
	`05`	`+3097`	añadir `1` (`97`) al acumulador
	`06`	`+2198`	almacenar acumulador en `y` (`98`)
`35 rem add y to total`		*ninguna*	`rem` es ignorado
`40 let t = t + y`	`07`	`+2096`	cargar `t` de posición (`96`)
	`08`	`+3098`	añadir `y` (`98`) a acumulador
	`09`	`+2196`	almacenar acumulador en `t` (`96`)
`45 rem loop y`		*ninguna*	`rem` es ignordo
`50 goto 20`	`10`	`+4001`	bifurcarse a la posición `01`
`55 rem output result`		*ninguna*	`rem` es ignorado
`60 print t`	`11`	`+1195`	extraer `t` (`96`) a pantalla
`99 end`	`12`	`+4300`	terminar la ejecución.

Fig. 12.31 Código optimizado para el programa de la figura 12.28.

c) Haga posible que en los enunciados Simple el compilador reconozca letras mayúsculas y minúsculas (por ejemplo `'A'` sea equivalente a `'a'`). No son requeridas modificaciones al simulador Simpletron.

d) Haga posible que los enunciados `input` puedan leer valores para variables múltiples, como `input x, y`. No se requieren modificaciones al simulador Simpletron.

e) Haga posible que el compilador extraiga varios valores en un enunciado único `print`, como `print a, b, c`. No se requieren modificaciones al simulador Simpletron.

f) Añada capacidades de verificación de sintaxis al compilador, de tal forma que cuando en un programa Simple se encuentren errores de sintaxis se emitan mensajes de error. No se requieren modificaciones al simulador Simpletron.

g) Haga posible el uso de arreglos de enteros. No se requieren modificaciones al simulador Simpletron.

h) Permitir subrutinas especificadas por los comandos Simple `gosub` y `return`. El comando `gosub` pasa el control del programa a una subrutina y el comando `return` pasa el control de regreso al enunciado existente después del `gosub`. Esto es similar a una llamada de función en C. La misma subrutina puede ser llamada a partir de muchos `gosub` distribuidos a lo largo de un programa. No se requiere de modificaciones al simulador Simpletron.

i) Haga posible estructuras de repetición de la forma


```
for x = 2 to 10 step 2
    enunciados Simple
next
```

Este enunciado `for` cicla desde `2` hasta `10` con un incremento de `2`. La línea `next` marca el fin del cuerpo de la línea `for`. No se requieren de modificaciones al simulador Simpletron.

j) Haga posible estructuras de repetición de la forma

```
for x = 2 to 10
    enunciados Simple
next
```

Este enunciado `for` cicla desde `2` hasta `10` con un incremento preestablecido de `1`. No se requieren de modificaciones al simulador Simpletron.

k) Haga posible que el compilador procese entradas y salidas de cadenas. Esto requiere la modificación del simulador Simpletron para procesar y almacenar valores de cadenas. *Sugerencia:* cada palabra Simpletron puede ser dividida en dos grupos, cada uno de los grupos conteniendo un entero de dos dígitos. Cada entero de dos dígitos representa el equivalente decimal ASCII de un carácter. Añada una instrucción en lenguaje máquina que imprimirá una cadena empezando en una posición de memoria Simpletron. La primera mitad de la palabra en dicha posición es una cuenta del número de caracteres dentro de la cadena (es decir la longitud de la cadena). Cada media palabra sucesiva contiene un carácter ASCII expresado como dos dígitos decimales. La instrucción en lenguaje máquina verifica la longitud e imprime la cadena traduciendo cada número de dos dígitos en su carácter equivalente.

l) Haga posible que además de enteros el compilador procese valores de punto flotante. También deberá ser modificado el simulador Simpletron para procesar valores de punto flotante.

12.30 *(Un intérprete Simple).* Un intérprete es un programa que lee un enunciado de un programa en un lenguaje de alto nivel, determina la operación a ejecutarse por dicho enunciado, y de inmediato procede a ejecutar la operación. El programa no se convierte primero a lenguaje máquina. Los intérpretes ejecutan lentamente, porque cada enunciado que se encuentra en el programa deberá primero ser descifrado. Si los enunciados están incluidos en un ciclo, cada vez que se encuentren dentro del ciclo, los enunciados serán descifrados. Las versiones primeras del lenguaje de programación BASIC eran puestas en operación por medio de intérpretes.

Escriba un intérprete para el lenguaje Simple analizado en el ejercicio 12.26. El programa deberá utilizar el convertidor de infijo a postfijo desarrollado en el ejercicio 12.12 y el evaluador postfijo desarrollado en el ejercicio 12.13 para evaluar las expresiones en un enunciado `let`. En este programa deberán ser impuestas las mismas restricciones puestas en práctica en el lenguaje Simple del ejercicio 12.26. Pruebe el intérprete con los programas Simple escritos en el ejercicio 12.26. Compare los resultados de ejecutar estos programas en el intérprete con los resultados de compilar programas Simple y ejecutarlos en el simulador Simpletron que se construyó en el ejercicio 7.19.

13

El preprocesador

Objetivos

- Ser capaz de utilizar **#include** para el desarrollo de programas extensos.
- Ser capaz de utilizar **#define** para crear macros y macros con argumentos.
- Comprender la compilación condicional.
- Ser capaz de mostrar mensajes de error durante la compilación condicional.
- Ser capaz de usar afirmaciones para probar si los valores de expresiones son correctos.

Sostén lo bueno; defínelo bien.
Alfred, Lord Tennyson

Le he encontrado un argumento; pero no estoy obligado de encontrarle una comprensión.
Samuel Johnson

Un buen símbolo es el mèjor argumento, y es un misionero para persuadir a miles.
Ralph Waldo Emerson

Las condiciones son básicamente sanas.
Herbert Hoover (Diciembre 1929)

Al partidario, cuando está inmerso en una discusión, no le importan los derechos de la pregunta, sino que solo está ansioso de convencer a sus oyentes de sus propias afirmaciones.
Platón

Sinopsis

Resumen • Terminología • Errores comunes de programación • Prácticas sanas de programación • Sugerencias de rendimiento • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios.

13.1 Introducción

Este capítulo presenta el *preprocesador C*. El preprocesamiento ocurre antes de que se compile un programa. Algunas acciones posibles son: inclusión de otros archivos en el archivo bajo compilación, definición de *constantes simbólicas* y de *macros*, *compilación condicional* de código de programa, y *ejecución condicional de directivas* de preprocesador. Todas las directivas de preprocesador empiezan con el signo **#**, y antes de una directiva de preprocesador en una línea sólo pueden aparecer espacios en blanco.

13.2 La directiva de preprocesador # include

La *directiva de preprocesador* `#include` ha sido utilizada a todo lo largo de este texto. La directiva **#include** hace que en el lugar de la directiva se incluya una copia del archivo especificado. Las dos formas de la directiva **#include** son:

```
#include <filename>
#include "filename"
```

La diferencia entre estas formas estriba en la localización donde el preprocesador buscará el archivo a incluirse. Si el nombre del archivo está encerrado entre comillas, el preprocesador buscará el archivo a incluirse en el mismo directorio que el del archivo que se está compilando. Este método se utiliza normalmente para incluir archivos de cabecera definidos por el programador. Si el nombre del archivo esta encerrado en corchetes angulares (< y >) —que se utilizan para los *archivos de cabecera estándar de biblioteca*— la búsqueda se llevará a cabo de forma independiente a la puesta en práctica, por lo regular a través de directorios predesignados.

La directiva **#include** se utiliza por lo general para incluir archivos de cabecera de biblioteca estándar, como **stdio.h** y **stdlib.h** (vea la figura 5.6). La directriz **#include** también se utiliza con programas formados de varios archivos fuente, que deben ser compilados juntos. A menudo es creado e incluido en el archivo un *archivo de cabecera* conteniendo declaraciones comunes a los varios archivos de un programa. Ejemplos de declaraciones como éstas son declaraciones de estructuras y de uniones, en numeraciones y prototipos de función.

En UNIX, los archivos de programa se compilan utilizando el *comando* CC. Por ejemplo, para compilar y enlazar **main.c** con **square.c** escriba el comando.

```
cc main.c square.c
```

en el indicador de UNIX. Esto produce el archivo ejecutable **a.out**. Para mayor información sobre la compilación, el enlace y la ejecución de programas, vea los manuales de referencia correspondientes a su compilador.

13.3 La directiva de preprocesador # define: constantes simbólicas

La *directiva* **#define** crea *constantes simbólicas* —constantes representadas como símbolos— y *macros*— operaciones definidas como símbolos. El formato de la directiva **#define** es

```
#define identifier replacement-text
```

Cuando en un archivo aparece esta línea, todas las subsecuentes apariciones de *identifier* serán de forma automática remplazadas por *replacement-text*, antes de la compilación del programa. Por ejemplo,

```
#define PI 3.14159
```

remplaza todas las ocurrencias subsiguientes de la constante simbólica **PI** por la constante numérica **3.14159**. Las constantes simbólicas permiten al programador ponerle un nombre a una constante y utilizarlo a todo lo largo del programa. Si durante el programa la constante requiere de modificación, puede ser modificada una vez en la directiva **#define** y cuando sea el programa recompilado, todas las ocurrencias de la constante dentro del programa de manera automática serán modificadas. Nota: *todo lo que exista a la derecha del nombre de la constante simbólica, remplaza a la constante simbólica.* Por ejemplo, **#define PI = 3.14159** hace que el preprocesador remplace todas las ocurrencias de **PI** con **= 3.14159**. Esto es causa de muchos errores sutiles de lógica y de sintaxis. Es también un error redefinir una constante simbólica con un nuevo valor.

Práctica sana de programación 13.1

Utilizar nombres significativos para las constantes simbólicas, ayuda a que los programas resulten más autodocumentados.

13.4 La directiva de preprocesador #define: macros

Una *macro* es una operación definida en una directiva de preprocesador **#define**. Como en el caso de las constantes simbólicas, antes de compilarse el programa, el *macro identifier* se remplaza en el mismo por el *replacement-text*. Las macros pueden ser definidas con o sin *argumentos*. Una macro sin argumentos se procesa como si fuera una constante simbólica. En una macro con argumentos, los argumentos se sustituyen en el texto de remplazo, y a continuación la macro se *expande*, es decir, en el programa el texto de remplazo remplaza al identificador y a la lista de argumentos.

Veamos la siguiente definición de macro con un argumento, correspondiente al área de un círculo:

```
#define CIRCLE_AREA(x) PI * (x) * (x)
```

Siempre que en el archivo aparezca **CIRCLE_AREA(x)**, el valor de **x** será sustituido por **x** en el texto de remplazo, la constante simbólica **PI** será remplazada por su valor (previamente definido), y en el programa la macro se expandirá. Por ejemplo, el enunciado

```
area = CIRCLE_AREA(4);
```

se expande a

```
area = 3.14159 * (4) * (4);
```

Dado que la expresión está sólo formada de constantes, al momento de compilarse se evalúa el valor de la expresión y se asigna a la variable **area**. Cuando el argumento del macro es una expresión, los paréntesis alrededor de cada **x** en el texto de remplazo obligan al orden apropiado de evaluación. Por ejemplo, el enunciado

```
area = CIRCLE_AREA(c + 2);
```

se expande a

```
area = 3.14159 * (c + 2) * (c + 2);
```

que se evalúa de forma correcta, porque los paréntesis obligan al orden correcto de evaluación. Si se omiten los paréntesis, la expansión del macro es

```
area = 3.14159 * c + 2 * c + 2;
```

lo que se evalúa de forma incorrecta como

```
area = (3.14159 * c) + (2 * c) + 2;
```

en razón de las reglas de precedencia de operadores.

Error común de programación 13.1

Olvidar encerrar los argumentos de macro en paréntesis en el texto de remplazo.

La macro **CIRCLE_AREA** podría ser definida como una función. La función **circleArea**

```
double circleArea(double x)
{
   return 3.14159 * x * x;
}
```

lleva a cabo el mismo cálculo que la macro **CIRCLE_AREA**, pero con la función **circleArea** se asocia la sobrecarga de una llamada de función. Las ventajas de la macro **CIRCLE_AREA** es que las macros insertan código en el programa directamente evitando sobrecarga de función y conservándose el programa legible, porque el cálculo **CIRCLE_AREA** es definido por separado y nombrado de forma significativa. Una desventaja estriba en que su argumento debe evaluarse dos veces.

Sugerencia de rendimiento 13.1

A veces las macros pueden ser utilizadas para sustituir una llamada de función por código en línea, previo al tiempo de ejecución. Esto elimina la sobrecarga de una llamada de función.

La siguiente es una definición de macro con dos argumentos, correspondiente al área de un rectángulo:

```
#define RECTANGLE_AREA(x, y) (x) * (y)
```

Siempre que en el programa aparezca **RECTANGLE_AREA(x, y)**, los valores de **x** y **y** serán sustituidos por el texto de remplazo de la macro, y la macro será expandida en lugar del nombre de la macro. Por ejemplo, el enunciado

```
rectArea = RECTANGLE_AREA(a + 4, b + 7);
```

se expande a

```
rectArea = (a + 4) * (b + 7);
```

El valor de la expresión es evaluado y asignado a la variable **rectArea.**

El texto de remplazo para una macro o una constante simbólica es por lo regular cualquier texto sobre la línea, después del identificador en la directriz **#define**. Si el texto de remplazo de una macro o de una constante simbólica es más largo que el resto de la línea, deberá colocarse una diagonal invertida (\) al final de la misma, indicando que el texto de remplazo continúa sobre la siguiente línea.

Las constantes simbólicas y las macros pueden ser descartadas utilizando la *directiva de preprocesador* **#undef**. La directiva **#undef** "elimina" la definición de una constante simbólica o de un nombre de macro. El *alcance* de una constante simbólica o de una macro cubre desde su definición hasta que mediante **#undef** queda eliminada su definición, o si no, hasta el final del archivo. Una vez eliminada la definición, puede volverse a definir un nombre utilizando **#define**.

Las funciones en la biblioteca estándar son definidas algunas veces como macros basadas en otras funciones de biblioteca. Una macro, comúnmente definida en el archivo de cabecera **stdio.h**, es

```
#define getchar() getc(stdin)
```

La definición de macro de **getchar** utiliza la función **getc** para obtener un carácter a partir del flujo de entrada estándar. A menudo la función **putchar** en el encabezado **stdio.h**, y las funciones de manejo de caracteres del encabezado **ctype.h**, también son puestas en operación como macros. Note que las expresiones con efectos colaterales (es decir, donde los valores de las variables se modifican) no deberían ser pasadas a una macro porque los argumentos de macro pudieran ser evaluados más de una vez.

13.5 Compilación condicional

La *compilación condicional* le permite al programador controlar la ejecución de las directivas de preprocesador, y la compilación del código del programa. Cada una de las directivas de preprocesador evalúa una expresión entera constante. Las expresiones de cambio de tipo, las expresiones **sizeof**, y las constantes de enumeración no pueden ser evaluadas en directivas de preprocesador.

El constructor de preprocesador condicional es muy parecido a la estructura de selección **if**. Veamos el siguiente código de preprocesador:

```
#if !defined(NULL)
   #define NULL 0
#endif
```

Estas directivas determinan si NULL ha sido definido. La expresión `defined(NULL)` se evalúa a `1`, si NULL está definido; y a `0` de lo contrario. Si el resultado es `0`, `!defined(NULL)` se evalúa a `1`, y NULL queda definido. De lo contrario, la directiva `#define` es pasada por alto. Cada constructor `#if` termina con `#endif`. Las directivas `#ifdef` y `#ifndef` son abreviaturas correspondientes a `#ifdefined(`*nombre*`)` y `#if !defined(`*nombre*`)`. Un constructor de preprocesador condicional de varias partes puede ser probado utilizando las directivas `#elif` (el equivalente de `else if` en una estructura `if`) y `#else` (el equivalente de `else` en una estructura `if`).

Durante el desarrollo del programa, a menudo los programadores encuentran útil "anotar" grandes porciones de código para evitar que sean compilados. Si el código contiene comentarios, para esta tarea no es posible utilizar `/*` y `*/`. En vez de ello, el programador puede recurrir al siguiente constructor de preprocesador

```
#if 0
   code prevented from compiling
#endif
```

Para permitir que el código sea compilado, en el constructor anterior el `0` es remplazado por `1`.

La compilación condicional se utiliza por lo común como ayuda de depuración. Muchas instalaciones en C proporcionan *depuradores*. Sin embargo, con frecuencia los depuradores son difíciles de utilizar y de comprender, por lo que son rara vez utilizados por los estudiantes de un primer curso de programación. En vez de ello, se usan enunciados `printf` para imprimir valores de variables y para confirmar el flujo de control. Estos enunciados `printf` pueden ser encerrados en directivas de preprocesador condicionales, de tal forma que los enunciados sólo sean compilados mientras no se haya terminado el proceso de depuración. Por ejemplo

```
#ifdel DEBUG
   printf("Variable x = %d\n", x);
#endif
```

hace que se compile en el programa un enunciado `printf` si la constante simbólica DEBUG ha sido definida (`#define DEBUG`) antes de la directiva `#ifdef DEBUG`. Cuando se haya terminado la depuración, se elimina del archivo fuente la directiva `#define`, y durante la compilación los enunciados `printf`, insertos para fines de depuración, serán ignorados. En programas más extensos pudiera ser deseable definir varias constantes simbólicas distintas, que controlen la compilación condicional en secciones separadas del archivo fuente.

Error común de programación 13.2

Insertar enunciados `printf` compilados condicionalmente para fines de depuración, en posiciones donde en ese momento C espera un enunciado. En este caso, el enunciado compilado condicional debería haberse encerrado en un enunciado compuesto. Entonces, cuando el programa se compile con enunciados de depuración, el flujo del control del programa no será alterado.

13.6 Las directivas de preprocesador #error y #pragma

La *directiva* `#error`

```
#error tokens
```

imprime un mensaje, que depende de la puesta en práctica, incluyendo las *divisiones o componentes léxicas* especificadas en la directriz. Las divisiones léxicas son secuencias de caracteres separadas por espacios. Por ejemplo,


```
#error 1 - Out of range error
```

contiene 6 divisiones léxicas. En Borland C++ para PC, por ejemplo, cuando se procesa la directiva `#error`, se muestran las componentes o divisiones léxicas en la directiva como un mensaje de error, el preproceso se detiene, y el programa no se compila.

La *directiva* `#pragma`

```
#pragma tokens
```

genera una acción definida por la puesta en práctica. Un pragma no reconocido por la puesta en práctica será ignorado. Borland C++, por ejemplo, reconoce varios pragmas que le permiten al programador aprovechar completamente la puesta en marcha de Borland C++. Para mayor información sobre `#error` y `pragma`, vea la documentación en su instalación de C.

13.7 Los operadores # y

Los operadores de preprocesador # y ## están disponibles sólo en ANSI C. El operador # hace que una división léxica de texto de remplazo se convierta en una cadena encerrada entre comillas. Considere la siguiente definición de macro:

```
#define HELLO(x) printf("Hello, " #x "\n");
```

Cuando `HELLO(John)` aparece en un archivo de programa, se expande a

```
printf("Hello, " "John" "\n");
```

La cadena `"John"` remplaza `#x` en el texto de remplazo. Las cadenas separadas por un espacio en blanco son concatenadas durante el preprocesamiento, por lo que el enunciado arriba citado es equivalente a

```
printf("Hello, John\n");
```

Note que el operador # deberá ser utilizado en una macro con argumentos, porque el operando de # se refiere a un argumento de la macro.

El operador ## concatena dos componentes léxicos. Considere la siguiente definición de macro:

```
#define TOKENCONCAT(x, y) x ## y
```

Cuando en el programa aparece TOKENCONCAT, sus argumentos son concatenados y utilizados para remplazar la macro. Por ejemplo, en el programa `TOKENCONCAT(O, K)` es remplazado por `OK`. El operador ## debe tener dos operandos.

13.8 Números de línea

La *directiva de preprocesador* `#line` hace que las líneas de código fuente subsecuentes se renumeren, iniciándose con el valor entero constante especificado. La directiva

```
#line 100
```

inicia la numeración de líneas a partir de `100`, empezando con la siguiente línea de código fuente. En la directiva `#line` puede incluirse un nombre de archivo. La directiva

```
#line 100 "file1.c"
```

indica que las líneas están numeradas a partir de **100** empezando desde la siguiente línea de código fuente, y el nombre del archivo para fines de cualquier mensaje de compilador es **"file1.c"**. Esta directiva normalmente se utiliza para ayudar a preparar los mensajes producidos debido a errores de sintaxis y a advertencias más significativas del compilador. En el archivo fuente no aparecen los números de línea.

13.9 Constantes simbólicas predefinidas

Existen cinco *constantes simbólicas predefinidas* (figura 13.1). Los identificadores correspondientes a cada una de las constantes simbólicas predefinidas empiezan y terminan con *dos* subrayados. Estos identificadores y el identificador **defined** (utilizado en la sección 13.5) no pueden ser utilizados en las directivas **#define** o **#undef**.

13.10 Asertos

La *macro* `assert` definida en el archivo de cabecera **assert.h**, prueba el valor de una expresión. Si el valor de la expresión es **0** (falso), entonces **assert** imprime un mensaje de error, y llama a la función *abort* (de la biblioteca general de utilerías **stdlib.h**) para terminar la ejecución del programa. Esta es una herramienta de depuración útil, para probar si una variable tiene el valor correcto. Por ejemplo, suponga que en un programa la variable **x** jamás debería ser mayor de **10**. Se podría utilizar una afirmación para probar el valor de **x** y si el valor de **x** es incorrecto imprimir un mensaje de error. El enunciado sería

```
assert(x <= 10);
```

Si cuando se encuentre este enunciado en un programa **x** es mayor que **10**, se imprimirá un mensaje de error, conteniendo el número de línea y el nombre del archivo, y el programa terminará. Entonces el programador podrá concentrarse en esta área del código y encontrar el error. Si queda definida la constante simbólica **NDEBUG**, asertos subsecuentes serán ignoradas. Entonces, una vez que ya no sean requeridas los asertos, la línea

Constante simbólica	Explicación
__LINE__	Número de línea de la línea actual del código fuente (una constante entera).
__FILE__	El nombre supuesto del archivo fuente (una cadena).
__DATE__	La fecha de compilación del archivo fuente (una cadena de la forma **"Mmm dd yyyy** como **"Jan 19 1991"**).
__TIME__	La hora en que fue compilado el archivo fuente (una literal de cadena de la forma **"hh:mm:ss"**).
__STDC__	La constante entera 1. Se utiliza para indicar que ésta puesta en marcha cumple con ANSI.

Fig. 13.1 Las constantes simbólicas predefinidas.


```
#define NDEBUG
```

es inserta en el archivo del programa, en vez de borrar de manera manual cada una de los asertos.

Resumen

- Todas las directivas de preprocesador empiezan con **#**.
- Sólo caracteres de espacio en blanco pueden aparecer en una línea antes de una directiva de preprocesador.
- La directiva **#include** incluye una copia del archivo especificado. Si el nombre del archivo está encerrado entre comillas, el preprocesador empieza buscando el archivo a incluirse en el mismo directorio que el archivo que se está compilando. Si el nombre del archivo está encerrado en corchetes angulares (< y >), la búsqueda se ejecuta de una forma definida de acuerdo a la puesta en marcha.
- La directiva de preprocesador **#define** se utiliza para crear constantes simbólicas y macros.
- Una constante simbólica es el nombre para una constante.
- Una macro es una operación definida en una directiva de preprocesador **#define**. Las macros pueden ser definidas con o sin argumentos.
- El texto de remplazo para una macro o una constante simbólica es cualquier texto que quede en la línea después del identificador en la directiva **#define**. Si el texto de remplazo para una macro o una constante simbólica es más largo que el resto de la línea, se coloca una diagonal invertida (\) al final de la línea, lo que indica que el texto de remplazo continúa sobre la línea siguiente.
- Las constantes simbólicas y las macros pueden ser descartadas utilizando la directiva de preprocesador **#undef**. La directiva **#undef** "elimina la definición" de la constante simbólica o del nombre de la macro.
- El alcance de una constante simbólica o de una macro es desde su definición hasta que queda eliminada su definición mediante **#undef**, o si no hasta el final del archivo.
- La compilación condicional le permite al programador controlar la ejecución de las directivas de preprocesador y la compilación del código de programa.
- Las directivas de preprocesador condicionales evalúan expresiones enteras constantes. Las expresiones de cambio de tipo, las expresiones **sizeof** y las constantes de enumeración no pueden ser evaluadas en directivas de preprocesador.
- Cada constructor **#if** termina con **#endif**.
- Las directivas **#ifdef** y **#ifndef** han sido diseñadas como abreviaturas de **#if defined**(*nombre*) y **#if !defined**(*nombre*).
- Un constructor de preprocesador condicional de varias partes puede ser probado utilizando **#elif** (el equivalente de **else if** en una estructura **if**) y **#else** (el equivalente de **else** en una estructura **if**) directivas.
- La directiva **#error** imprime un mensaje, que depende de la puesta en práctica, y que incluye las componentes léxicas especificadas en la directriz.
- La directiva **#pragma** genera una acción definida por la puesta en práctica. Si el pragma no es reconocido por la puesta en marcha, será ignorado.

- El operador # hace que un componente léxico de texto de remplazo, se convierta a una cadena encerrada entre comillas. El operador # debe ser utilizado en una macro con argumentos, porque el operando # debe ser un argumento de la macro.
- El operador ## concatena dos componentes léxicos. El operador ## debe de tener dos operandos.
- La directiva de preprocesador **#line** hace que las líneas de código fuente subsecuentes vuelvan a ser numeradas, iniciándose a partir del valor entero constante especificado.
- Existen cinco constantes simbólicas predefinidas. La constante **__LINE__** es el número de línea de la línea de código fuente actual (un entero). La constante **__FILE__** es el nombre supuesto del archivo (una cadena). La constante **__DATE__** es la fecha de compilación del archivo fuente (una cadena). La constante **__TIME__** es la hora de compilación del archivo fuente (una cadena). La constante **__STDC__** es **1**; su propósito es indicar que la instalación cumple con ANSI. Note que cada una de las constantes simbólicas predefinidas empieza y termina con dos subrayados.
- La macro **assert** definida en el archivo de cabecera **assert.h**, —prueba el valor de una expresión. Si el valor de la expresión es **0** (falsa), entonces **assert** imprime un mensaje de error y llama a la función **abort** para terminar la ejecución del programa.

Terminología

#define
#elif
#else
#endif
#error
#if
#ifdef
#ifndef
#include <filename>
#include "filename"
#line
#pragma
#undef
\ (diagonal invertida) carácter de continuación
__DATE__
__FILE__
__LINE__
__STDC__
__TIME__
a. out en UNIX
abort
argumento
assert
assert.h
preprocesador C
comando **cc** en UNIX
operador **##** de concatenación de preprocesador
compilación condicional
directrices de preprocesador de ejecución condicional
operador # de preprocesador de conversión a cadena
depurador
expandir una macro
archivo de cabecera
macro
macro con argumentos
constantes simbólicas predefinidas
directiva de preprocesador
texto de remplazo
alcance de una constante simbólica o de una macro
archivos de cabecera de la biblioteca estándar
stdio.h
stdlib.h
constante simbólica

Errores comunes de programación

13.1 Olvidar encerrar los argumentos de macro en paréntesis en el texto de remplazo.

13.2 Insertar enunciados **printf** compilados condicional para fines de depuración, en posiciones donde en ese momento C espera un enunciado. En este caso, el enunciado compilado condicional debería haberse encerrado en un enunciado compuesto. Entonces, cuando el programa se compile con enunciados de depuración, el flujo del control del programa no será alterado.

Práctica sana de programación

13.1 Utilizar nombres significativos para las constantes simbólicas, ayuda a que los programas resulten más autodocumentados.

Sugerencia de rendimiento

13.1 A veces las macros pueden ser utilizadas para substituir una llamada de función por código en línea, previo al tiempo de ejecución. Esto elimina la sobrecarga de una llamada de función.

Ejercicios de autoevaluación

13.1 Llene los espacios en blanco con cada uno de los siguientes:

a) Cada directiva de preprocesador deberá empezar con _____.
b) El constructor de compilación condicional puede ser extendido para probar casos múltiples utilizando las directivas _____ y _____.
c) La directiva _____ genera macros y constantes simbólicas.
d) Sólo caracteres _____ pueden aparecer sobre una línea antes de una directiva de preprocesador.
e) La directiva _____ descarta constantes simbólicas y nombres de macros.
f) Las directivas _____ y _____ son una notación abreviada en lugar de **#if defined (*nombre*)** y **#if !defined (*nombre*)**.
g) _____ le permite al programador controlar la ejecución de las directivas de preprocesador, y la compilación del código del programa.
h) La macro _____ imprime un mensaje y termina la ejecución de un programa si el valor de la expresión que la macro evalúa es 0.
i) La directiva _____ inserta un archivo en otro archivo.
j) El operador _____ concatena sus dos argumentos.
k) El operador _____ convierte su operando a una cadena.
l) El carácter _____ indica que el texto de remplazo de una constante simbólica o de una macro continúa en la línea siguiente.
m) La directiva _____ hace que las líneas de código fuente se enumeren a partir del valor indicado, empezando con la siguiente línea de código fuente.

13.2 Escriba un programa que imprima los valores de las constantes simbólicas predefinidas, listadas en la figura 13.1.

13.3 Escriba una directiva de preprocesador para que lleve a cabo cada una de las siguientes:

a) Definir la constante simbólica **YES** para que tenga el valor **1**.
b) Definir la constante simbólica **NO** para que tenga el valor **0**.
c) Incluir el archivo de cabecera **common.h**. El archivo de cabecera se encuentra en el mismo directorio que el archivo bajo compilación.
d) Renumerar las líneas restantes del archivo, empezando con el número de línea **3000**.
e) Si está definida la constante simbólica **TRUE**, elimine su definición, y vuélvala a definir como **1**. No utilice **#ifdef**.
f) Si la constante simbólica **TRUE** está definida, elimine su definición, y vuélvala a definir como **1**. Utilice la directiva de preprocesador **#ifdef**.

g) Si la constante simbólica **TRUE** no es igual a 0, defina la constante simbólica **FALSE** como 0. De lo contrario defina **FALSE** como 1.
h) Defina la macro **SQUARE_VOLUME**, que calcula el volumen de un cuadrado. La macro toma un argumento.

Respuesta a los ejercicios de autoevaluación

13.1 a) #. b) **#elif**, **#else**. c) **#define**. d) espacio en blanco. e) **#undef**. f) **#ifdef**, **#ifndef**. g) compilación condicional. h) **assert**. i) **#include**. j) ##. k) #. l) \. m) **#line**.

13.2
```
/* Print the values of the predefined macros */
#include <stdio.h>
main()
{
   printf("__LINE__ = %d\n", __LINE__);
   printf("__FILE__ = %s\n", __FILE__);
   printf("__DATE__ = %s\n", __DATE__);
   printf("__TIME__ = %s\n", __TIME__);
   printf("__STDC__ = %d\n", __STDC__);
}
```

```
__LINE__ = 5
__FILE__ = macros.c
__DATE__ = Sep 08 1993
__TIME__ = 10:23:47
__STDC__ = 1
```

13.3 a)
```
#define YES 1
```
b)
```
#define NO 0
```
c)
```
#include "common.h"
```
d)
```
#line 3000
```
e)
```
#if defined(TRUE)
   #undef TRUE
   #define TRUE 1
#endif
```
f)
```
#ifdef TRUE
   #undef TRUE
   #define TRUE 1
#endif
```
g)
```
#if TRUE
   #define FALSE 0
#else
    #define FALSE 1
#endif
```
h)
```
#define SQUARE_VOLUME(x)  (x) * (x) * (x)
```

Ejercicios

13.4 Escriba un programa que defina una macro con un argumento para calcular el volumen de una esfera. El programa deberá calcular el volumen para esferas de radios de 1 a 10, e imprimir los resultados en formato tabular. La fórmula del volumen de una esfera es:

$$(4/3) * \pi * r^3$$

donde π es **13.14159**

13.5 Escriba un programa que produzca la siguiente salida:

```
The sum of x and y is 13
```

El programa deberá definir la macro **SUM** con dos argumentos, **x** y **y**, y utilizar **SUM** para producir la salida.

13.6 Escriba un programa que utilice la macro **MINIMUM2** para determinar el más pequeño de dos valores numéricos. Introduzca los valores desde el teclado.

13.7 Escriba un programa que utilice la macro **MINIMUM3** para determinar el más pequeño de tres valores numéricos. La macro **MINIMUM3** deberá utilizar la macro **MINIMUM2** definida en el ejercicio 13.6 para determinar el valor más pequeño. Introduzca los valores desde el teclado.

13.8 Escriba un programa que utilice la macro **PRINT** para imprimir un valor de cadena.

13.9 Escriba un programa que utilice la macro **PRINTARRAY** para imprimir un arreglo de enteros. La macro deberá recibir como argumentos el arreglo y el número de elementos en el arreglo.

13.10 Escriba un programa que utilice la macro **SUMARRAY** para sumar los valores de un arreglo numérico. La macro deberá recibir como argumentos el arreglo y el número de elementos en el arreglo.

14

Temas avanzados

Objetivos

- Ser capaz de redirigir la entrada de teclado para que provenga de un archivo.
- Ser capaz de redirigir la salida a pantalla a un archivo.
- Ser capaz de escribir funciones que utilicen listas de argumentos de longitud variable.
- Ser capaz de procesar argumentos en la línea de comandos.
- Ser capaz de asignar tipos específicos a constantes numéricas.
- Ser capaz de utilizar archivos temporales.
- Ser capaz de procesar eventos inesperados dentro de un programa.
- Ser capaz de asignar memoria dinámicamente para los arreglos.
- Ser capaz de cambiar el tamaño de la memoria anteriormente dinámica asignada.

Utilizaremos una señal que ya he probado y comprobado llega lejos y es fácil de gritar. ¡Waa-hoo!
Zane Grey

Uselo, gástelo;
aprovéchelo o descártelo.
Anónimo

Verdaderamente es un problema de tres pipas.
Sir Arthur Conan Doyle

Sinopsis

Resumen • Terminología • Errores comunes de programación • Sugerencias de portabilidad • Sugerencias de rendimiento • Observaciones de ingeniería de software • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios.

14.1 Introducción

En esta capítulo se presentan varios temas avanzados, por lo regular no se tratan en cursos introductorios. Muchas de las capacidades que aquí se analizan son específicas a sistemas operativos particulares, en especial a UNIX y/o a DOS.

14.2 Cómo redirigir entradas/salidas en sistemas UNIX y DOS

Normalmente la entrada a un programa es a partir del teclado (entrada estándar), y la salida de un programa se muestra en la pantalla (salida estándar). En la mayor parte de los sistemas de computación y en particular en los sistemas UNIX y DOS es posible *redirigir* las entradas, para que en vez del teclado provengan de un archivo, y redirigir las salidas, para que en vez de en la pantalla sean colocadas en un archivo. Ambas formas de redirección pueden ser llevadas a cabo sin tener que utilizar las capacidades de procesamiento de archivos de la biblioteca estándar.

A partir de la línea de comandos UNIX existen varias maneras para redirigir la entrada y la salida. Considere al archivo ejecutable **sum.** que introduce enteros uno por uno, y lleva un total acumulativo de los valores hasta que se define el indicador de fin de archivo, y a continuación imprime el resultado. Por lo regular el usuario introduce enteros a partir del teclado y escribe la combinación de teclas de fin de archivo, para indicar que no se introducirán más valores. Con la redirección de la entrada, la entrada puede estar almacenada en un archivo. Por ejemplo, si los datos están almacenados en el archivo **input**, la línea de comando


```
$ sum < input
```

hace que se ejecute el programa **sum**; el *símbolo de redirección de entrada (<)* indica que los datos en el archivo **input** deben ser utilizados como entrada para el programa. En un sistema DOS la redirección de la entrada se efectúa en forma idéntica.

Note que en UNIX la indicación de la línea de comandos es el signo **$** (algunos sistemas UNIX utilizan la indicación %). Los estudiantes encuentran a menudo difícil comprender que la redirección es una función del sistema operativo, y no otra característica de C.

El segundo método para redirigir la entrada es *el entubamiento*. Una *tubería (|)* hace que la salida de un programa sea redirigida como entrada de otro programa. Suponga que el programa **random** tiene como salida una serie de enteros al azar; la salida de **random** puede ser "entubada" en forma directa al programa **sum**, utilizando la línea de comandos UNIX

```
$ random | sum
```

Esto hace que se calcule la suma de los enteros producidos por **random**. El entubamiento puede ser ejecutado en UNIX y en DOS.

La salida de programa puede ser redirigida a un archivo, mediante el uso del símbolo de *redirección de salida (>)* (tanto en UNIX como en DOS se utiliza el mismo símbolo). Por ejemplo, para redirigir la salida del programa **random** al archivo **out**, utilice

```
$ random > out
```

Por último, la salida del programa puede ser agregada al final de un archivo existente utilizando el *símbolo de agregar salida (>>)* (tanto en UNIX como en DOS se utiliza el mismo símbolo). Por ejemplo, para agregar la salida del programa **random** al archivo **out**, creado en la línea de comando anterior, utilice la línea de comando

```
$ random >> out
```

14.3 Listas de argumentos de longitud variable

Es posible crear funciones que reciban un número no especificado de argumentos. La mayor parte de los programas en el texto han utilizado la función **printf** estándar de biblioteca la cual, como usted sabe, toma un número variable de argumentos. Como mínimo, **printf** debe recibir una cadena como primer argumento, pero **printf** puede recibir cualquier número de argumentos adicionales. El prototipo de función correspondiente a **printf** es

```
int printf(const char *format, ...);
```

En el prototipo de función los puntos suspensivos (**...**) indican que la función recibe un número variable de argumentos de cualquier tipo. Note que los puntos suspensivos deberán siempre estar colocados al final de la lista de parámetros.

Los macros y definiciones del *encabezado de argumentos variables* **stdarg.h** (figura 14.1) proveen las capacidades necesarias para construir funciones con listas de argumentos de longitud variable. El programa de la figura 14.2 demuestra una función **average**, que recibe un número variable de argumentos. El primer argumento de **average** es siempre el número de valores a promediarse.

Identificador	Explicación
va_list	Un tipo utilizable para conservar información necesaria por las macros **va_start**, **va_arg**, y **va_end**. Para tener acceso a los argumentos en una lista de argumentos de longitud variable, debe declararse un objeto del tipo **va_list**.
va_start	Una macro que se invoca antes de tener acceso a los argumentos de una lista de argumentos de longitud variable. La macro inicializa el objeto declarado mediante **va_list**, para que sea utilizada con las macros **va_arg**, y **va_end**.
va_arg	Una macro que se expande a una expresión del valor y del tipo del siguiente argumento, existente en la lista de argumentos de longitud variable. Cada invocación de **va_arg** modifica el objeto declarado con **va_list**, de tal forma que el objeto señale al siguiente argumento dentro de la lista.
va_end	Una macro que facilita el regreso normal de una función cuya lista de argumentos de longitud variable ha sido referenciada por la macro **va_start**.

Fig. 14.1 El tipo y las macros definidas en el encabezado **stdarg.h**.

La función **average** utiliza todas las definiciones y macros del encabezado **stdarg.h**. El objeto **ap**, del tipo **va_list**, es utilizado por las macros **va_start**, **va_arg** y **va_end** para procesar la lista de argumentos de longitud variable de la función **average**. La función empieza invocando la macro **va_start** para inicializar el objeto **ap** para su uso en **va_arg** y **va_end**. La macro recibe dos argumentos —el objeto **ap** y el identificador al argumento más a la derecha dentro de la lista de argumentos, antes de los puntos suspensivos— en este caso **i** (aquí **va_start** utilizará **i** para determinar dónde empieza la lista de argumentos de longitud variable). A continuación la función **average** añade en forma repetida los argumentos en la lista de argumentos de longitud variable de la variable **total**. El valor a añadirse a **total** se toma de la lista de argumentos invocando la macro **va_arg**. La macro **va_arg** recibe dos argumentos: el objeto **ap**, y el tipo de valor esperado en la lista de argumentos, en este caso **double**. La macro regresa el valor del argumento. La función **average** invoca la macro **va_end**, con el objeto **ap** como argumento, para facilitar un regreso normal hacia **main** a partir de **average**. Por último, se calcula el promedio y se regresa a **main**.

Error común de programación 14.1

Colocar puntos suspensivos en la mitad de una lista de parámetros de función. Los puntos suspensivos sólo pueden ser colocados al final de la lista de parámetros.

El lector puede cuestionar cómo pueden saber las funciones **printf** y **scanf** qué tipo utilizar en cada macro **va_arg**. La respuesta es que **printf** y **scanf** rastrean los especificadores de conversión de formato existentes en la cadena de control de formato, a fin de determinar el tipo del siguiente argumento a procesar.


```
/* Using variable-length argument lists */

#include <stdio.h>
#include <stdarg.h>

double average(int, ...);

main()
{
   double w = 37.5, x = 22.5, y = 1.7, z = 10.2;

   printf("%s%.1f\n%s%.1f\n%s%.1f\n%s%.1f\n\n",
          "w = ", w, "x = ", x, "y = ", y, "z = ", z);
   printf("%s%.3f\n%s%.3f\n%s%.3f\n",
          "The average of w and x is ",
          average(2, w, x),
          "The average of w, x, and y is ",
          average(3, w, x, y),
          "The average of w, x, y, and z is ",
          average(4, w, x, y, z));

   return 0;
}

double average(int i, ...)
{
   double total = 0;
   int j;
   va_list ap;

   va_start(ap, i);

   for (j = 1; j <= i; j++)
      total += va_arg(ap, double);

   va_end(ap);
   return total / i;
}
```

```
w = 37.5
x = 22.5
y = 1.7
z = 10.2

The average of w and x is 30.000
The average of w, x, and y is 20.567
The average of w, x, y, and z is 17.975
```

Fig. 14.2 Cómo utilizar listas de argumentos de longitud variable.

14.4 Cómo utilizar argumentos en la línea de comandos

En muchos sistemas y en particular en DOS y en UNIX incluyendo los parámetros **int argc** y **char *argv[]** en la lista de parámetros de **main**, es posible pasar argumentos a **main** a partir de la línea de comandos. El parámetro **argc** recibe el número de argumentos de la línea de comando. El parámetro **argv** es un arreglo de cadenas, en el cual se almacenan los argumentos de la línea de comando reales. Usos comunes de los argumentos en la línea de comandos incluyen la impresión de los mismos, pasar opciones a un programa, y pasar nombres de archivo a un programa.

El programa de la figura 14.3 copia un archivo a otro archivo, carácter por carácter. El archivo ejecutable para el programa se llama **copy**. En un sistema UNIX una línea de comando típica para el programa **copy** es

```
$ copy input output
```

Esta línea de comando indica que el archivo **input** será copiado al archivo **output**. Cuando se ejecuta el programa, si **argc** no es **3** (**copy** cuenta como uno de los argumentos), el programa imprime un mensaje de error y termina. De lo contrario, **argv** contiene las cadenas **"copy"**, **"input"**, y **"output"**. Los argumentos segundos y terceros de la línea de comandos se utilizan como nombres de archivo por el programa. Los archivos se abren utilizando la función **fopen**. Si ambos archivos se abren con éxito, los caracteres se leen del archivo **input** y se escriben al archivo **output** hasta que el indicador de fin de archivo del archivo **input** queda definido. Entonces termina el programa. El resultado es una copia exacta del archivo **input**. Note que no todos los sistemas de cómputo aceptan argumentos en la línea de comandos tan fácil como UNIX y DOS. Por ejemplo, los sistemas Macintosh y VMS, requieren de ajustes especiales para el proceso de argumentos en la línea de comandos. Vea los manuales de su sistema para más información relativa a argumentos en la línea de comandos.

14.5 Notas sobre la compilación de programas con varios archivos fuente

Como se indicó en el texto, es posible construir programas que estén formados por múltiples archivos fuente (vea el capítulo 16, "Clases"). Existen varias consideraciones a tomar en cuenta al crear programas en múltiples archivos. Por ejemplo, la definición de una función deberá estar por completo contenida en un archivo —no puede estar contenida en dos o más archivos.

En el capítulo 5, presentamos los conceptos de clase de almacenamiento y alcance. Aprendimos que las variables declaradas por afuera de cualquier definición de función son por omisión de clase de almacenamiento estática, y se conocen como variables globales. Una vez declaradas, las variables globales son accesibles a cualquier función definida en el mismo archivo. Las variables globales también son accesibles a funciones en otros archivos, pero sin embargo, estas variables globales deberán ser declaradas en cada uno de los archivos en los cuales serán utilizadas. Por ejemplo, si en un archivo definimos la variable entera global **flag**, y en un segundo archivo nos referimos a ella, el segundo archivo deberá contener la declaración.

```
extern int flag;
```

antes del uso de la variable en dicho archivo. En la declaración anterior, el especificador de clase de almacenamiento **extern** le indica al compilador que la variable **flag** está definida más adelante en el mismo archivo o en uno distinto. El compilador informa al enlazador que en dicho archivo aparecen referencias a la variable **flag** sin resolver (el compilador no sabe donde **flag**


```
/* Using command-line arguments */
#include <stdio.h>

main(int argc, char *argv[])
{
   FILE *inFilePtr, *outFilePtr;
   int c;

   if (argc != 3)
      printf("Usage: copy infile outfile\n");
   else
      if ((inFilePtr = fopen(argv[1], "r")) != NULL)

         if ((outFilePtr = fopen(argv[2], "w")) != NULL)

            while ((c = fgetc(inFilePtr)) != EOF)
               fputc(c, outFilePtr);

         else
            printf("File \"%s\" could not be opened\n", argv[2]);

      else
         printf("File \"%s\" could not be opened\n", argv[1]);

   return 0;
}
```

Fig. 14.3 Cómo utilizar argumentos en la línea de comandos.

queda definida, por lo que le deja al enlazador intentar encontrar a **flag**). Si el enlazador no puede localizar la definición de **flag**, se genera un error de enlace, y no se produce un archivo ejecutable. Si el enlazador localiza una definición global apropiada, éste resuelve las referencias indicando donde está localizada **flag**.

Sugerencia de rendimiento 14.1

Las variables globales aumentan el rendimiento debido a que son de manera directa accesibles por cualquier función —se elimina la sobrecarga de pasar datos a función.

Observación de ingeniería de software 14.1

Las globales variables deberían de evitarse, a menos de que el rendimiento de la aplicación sea crítico, porque violan el principio del mínimo privilegio.

Igual que las declaraciones **extern** pueden ser utilizadas para declarar variables globales en otros archivos de programa, los prototipos de función pueden extender el alcance de una función más allá del archivo en el cual ha sido definida (en un prototipo de función no es requerido el especificador **extern**). Esto se lleva a cabo incluyendo el prototipo de función en cada uno de los archivos en los cuales dicha función fue invocada, y compilando ambos archivos juntos (vea la sección 13.2). Los prototipos de función le indican al compilador que la función especificada se define más adelante en el mismo archivo o en un archivo distinto. De nuevo, el compilador no intenta resolver referencias a dicha función esa tarea se le deja al enlazador. Si el enlazador no puede localizar una definición de función apropiada, se genera un error.

Como ejemplo del uso de prototipos de función para extender el alcance de una función, considere cualquier programa que contenga la directiva de preprocesador **#include <stdio.h>**. Esta directiva incluye en un archivo los prototipos de función para funciones tales como **printf** y **scanf**. Otras funciones en el archivo pueden utilizar **printf** y **scanf** para llevar a cabo sus tareas. Las funciones **printf** y **scanf** se definen para nosotros en forma independiente. No necesitamos saber dónde están definidas. Estamos sólo volviendo a utilizar dicho código en nuestros programas. El enlazador resuelve nuestras referencias a dichas funciones en forma automática. Este proceso nos permite utilizar las funciones estándar de biblioteca.

Observación de ingeniería de software 14.2

La creación de programas en múltiples archivos fuente facilita la reutilización del software y buena ingeniería de software. Las funciones pueden ser comunes para muchas aplicaciones. En casos como éstos, estas funciones deberán ser almacenadas en sus propios archivos fuente, y cada archivo fuente deberá tener su correspondiente archivo de cabecera, que contenga los prototipos de función. Esto permite a los programadores de distintas aplicaciones para reutilizar el mismo código, mediante la inclusión del archivo de cabecera apropiado, y compilar su aplicación con el archivo fuente correspondiente.

Sugerencia de portabilidad 14.1

Algunos sistemas no aceptan nombres de variables globales o nombres de funciones con más de 6 caracteres. Esto deberá ser tomado en cuenta al escribir programas que se planea serán transportados a varias plataformas.

Es posible restringir el alcance de una variable global o de una función al archivo en el cual está definido. El especificador de clase de almacenamiento **static**, al ser aplicado a una variable global o a una función, impide que sea utilizada por cualquier función que no quede definida dentro del mismo archivo. Esto se conoce como *enlace interno*. Las variables globales y las funciones que en sus definiciones no estén precedidas por **static** tienen *enlace externo* —se puede tener acceso a ellas desde otros archivos, si dichos archivos contienen las declaraciones apropiadas y/o los prototipos de función.

La declaración de variable global

```
static float pi = 3.14159;
```

crea la variable **pi** del tipo **float**, la inicializa a **3.14159**, e indica que **pi** es conocida sólo por las funciones dentro del archivo en la cual está definida.

El especificador **static** se utiliza por lo general con funciones de utilería, que sólo son llamadas por funciones de un archivo en particular. Si fuera de un archivo en particular una función no es requerida, deberá cumplirse mediante el uso de **static** con el principio del mínimo privilegio. Si en un archivo una función se define antes de su uso, **static** deberá ser aplicado a la definición de función. De lo contrario, **static** deberá aplicarse a el prototipo de función.

Al construir programas extensos en múltiples archivos fuente, la compilación del programa se convierte en tediosa si se hacen pequeñas modificaciones a un archivo y todo el programa debe ser vuelto a compilar. Muchos sistemas tienen utilerías especiales, que recopilan sólo el archivo de programa modificado. En sistemas UNIX esta utilería se denomina ***make***. La utilería **make** lee un archivo llamado ***makefile***, que contiene instrucciones para compilar y enlazar el programa. Los sistemas como Borland C++ y Microsoft C/C++ 7.0 para PC también ofrecen utilerías **make**. Para mayor información sobre las utilerías **make**, vea el manual correspondiente a su sistema particular.

14.6 Terminación de programa mediante Exit y Atexit

La biblioteca general de utilerías (**stdlib.h**) contiene métodos de terminar la ejecución de programa distintos al regreso convencional a partir de la función **main**. La función ***exit*** obliga a que se termine un programa, como si se hubiera ejecutado en forma normal. Esta función se utiliza a menudo para terminar un programa cuando se detecta un error en la entrada o si un archivo a procesarse por el programa no puede ser abierto. La función ***atexit*** *registra* en el programa, una función, que a la terminación exitosa del programa será llamada, es decir, ya sea cuando termina el programa llegando al final de **main**, o cuando se invoca a **exit**.

La función **atexit** toma como argumento un apuntador a una función (es decir, el nombre de la función). Las funciones llamadas a la terminación del programa no pueden tener argumentos, y no pueden regresar un valor. Se pueden registrar hasta 32 funciones para ejecución a la terminación del programa.

La función **exit** toma un argumento. El argumento por lo regular es o la constante simbólica ***EXIT_SUCCESS*** o la constante simbólica ***EXIT_FAILURE***. Si **exit** es llamada mediante ***EXIT_SUCCESS***, el valor definido por la puesta en marcha para una terminación exitosa es regresado al entorno llamador. Si **exit** es llamado mediante **EXIT_FAILURE**, el valor definido por la puesta en marcha correspondiente a una terminación no exitosa es regresado. Cuando se invoca la función **exit**, cualesquiera funciones registradas ya con **atexit** son invocadas en orden inverso de su registro, son vaciados y cerrados todos los flujos asociados con el programa, y el control regresa al entorno huésped. El programa de la figura 14.4 prueba las funciones **exit** y **atexit**. El programa solicita al usuario que determine si el programa debe terminarse con **exit** o llegando al final de **main**. Note que en cada caso la función **print** se ejecuta a la terminación del programa.

14.7 El calificador de tipo volátil

En los capítulos 6 y 7, presentamos el calificador de tipo **const**. ANSI C también dispone del calificador de tipo **volatile**. El estándar ANSI (An90) indica que cuando se utiliza **volatile** para calificar un tipo, la naturaleza del acceso a un objeto de dicho tipo dependerá de la puesta en marcha. Kernighan y Ritchie (Ke88) indican que el calificador **volatile** se utiliza para suprimir varios tipos de optimizaciones.

14.8 Sufijos para constantes enteras y de punto flotante

C contiene sufijos enteros y de punto flotante para especificar los tipos de las constantes enteras y de punto flotante. Los sufijos enteros son: **u** o bien **U** para un entero **unsigned**, **l** o **L** para un entero **long** y **ul** o **UL** para un entero **unsigned long**. Las siguientes constantes son del tipo **unsigned, long** y **unsigned long** respectivamente:

```
174u
8358L
28373ul
```

Si una constante entera no tiene sufijo, su tipo queda determinado con el primer tipo capaz de almacenar un valor de dicho tamaño (primero **int**, después **long int**, y por último **unsigned long int**).

Los sufijos de punto flotante son: **f** o **F** para un **float**, y **l** o **L** para un **long double**. Las siguientes constantes son del tipo **long double** y **float**, respectivamente:

```
/* Using the exit and atexit functions */

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

void print(void);

main()
{
   int answer;

   atexit(print);       /* register function print */
   printf("Enter 1 to terminate program with function exit\n"
          "Enter 2 to terminate program normally\n");
   scanf("%d", &answer);

   if (answer == 1) {
      printf("\nTerminating program with function exit\n");
      exit(EXIT_SUCCESS);
   }

   printf("\nTerminating program by reaching the end of main\n");
   return 0;
}

void print(void)
{
   printf("Executing function print at program termination\n"
          "Program terminated\n");
}
```

```
Enter 1 to terminate program with function exit
Enter 2 to terminate program normally
: 1

Terminating program with function exit
Executing function print at program termination
Program terminated
```

```
Enter 1 to terminate program with function exit
Enter 2 to terminate program normally
: 2

Terminating program by reaching the end of main
Executing function print at program termination
Program terminated
```

Fig. 14.4 Cómo utilizar las funciones **exit** y **atexit**.


```
3.14159L
1.28f
```

Una constante de punto flotante sin sufijo automáticamente es del tipo **double**.

14.9 Más sobre archivos

En el capítulo 11, se presentaron las capacidades para procesar archivos de texto mediante acceso secuencial y acceso directo. C también proporciona capacidades para el proceso de archivos binarios, pero algunos sistemas de cómputo no aceptan archivos binarios. Si no son aceptados los archivos binarios, y un archivo es abierto en modo de archivo binario (figura 14.5), el archivo será procesado como un archivo de texto. Los archivos binarios deberán ser utilizados en lugar de los archivos de texto, sólo en aquellas situaciones en que las condiciones de velocidad, almacenamiento y/o compatibilidad demanden archivos binarios. De lo contrario, siempre los archivos de texto deberán preferirse, debido a su inherente portabilidad, y debido a la posibilidad de utilizar otras herramientas estándar para examinar y manipular los datos del archivo.

Sugerencia de rendimiento 14.2

Considere utilizar archivos binarios en lugar de archivos de texto en aquellas aplicaciones que exijan alto rendimiento

Modo	Descripción
rb	Abre un archivo binario para lectura
wb	Crea un archivo binario para escritura. Si el archivo ya existe, descartará el contenido actual.
ab	Agrega; abre o crea un archivo binario para escritura al final del archivo.
rb+	Abre un archivo binario para actualizar (lectura y escritura).
wb+	Crea un archivo binario para actualizar. Si el archivo ya existe, descartará el contenido actual.
ab+	Agrega; abre o crea un archivo binario para actualizar; toda la escritura se efectuará al final del archivo.

Fig. 14.5 Modos de archivo binario abierto.

Sugerencia de portabilidad 14.2

Cuando esté escribiendo programas portátiles utilice archivos de texto.

Además de las funciones de procesamiento de archivos analizadas en el capítulo 11, la biblioteca estándar también tiene la función **tmpfile**, que abre un archivo temporal en modo **"wb+"**. Aunque este es un modo de archivo binario, algunos sistemas procesan los archivos temporales como archivos de texto. Un archivo temporal existe sólo hasta que es cerrado con **fclose**, o bien hasta que el programa termina.

El programa de la figura 14.6 sustituye los tabuladores existentes en un archivo por espacios. El programa solicita al usuario que escriba el nombre del archivo a modificarse. Si tanto el archivo

escrito por el usuario como el archivo temporal son abiertos con éxito, el programa lee caracteres del archivo a modificarse, y los escribe en el archivo temporal. Si el carácter leído es un tabulador (**'\t'**), será remplazado por un espacio y será escrito al archivo temporal. Cuando se llegue al fin del archivo bajo modificación, utilizando **rewind** los apuntadores de archivo para cada archivo serán reposicionados al principio de cada uno. A continuación, el archivo temporal se copia al archivo original, carácter por carácter. Con el fin de confirmar que los caracteres escritos son correctos, el programa imprime el archivo original conforme copia caracteres al archivo temporal, así como imprime el nuevo archivo conforme copia caracteres del archivo temporal al archivo original.

```
/* Using temporary files */
#include <stdio.h>

main()
{
   FILE *filePtr, *tempFilePtr;
   int c;
   char fileName[30];

   printf("This program changes tabs to spaces.\n"
          "Enter a file to be modified: ");
   scanf("%s", fileName);

   if ( ( filePtr = fopen(fileName, "r+") ) != NULL)

      if ( ( tempFilePtr = tmpfile() ) != NULL) {
         printf("\nThe file before modification is:\n");

         while ( ( c = getc(filePtr) ) != EOF) {
            putchar(c);
            putc(c == '\t' ? ' ': c, tempFilePtr);
         }

         rewind(tempFilePtr);
         rewind(filePtr);
         printf("\n\nThe file after modification is:\n");

         while ( ( c = getc(tempFilePtr) ) != EOF) {
            putchar(c);
            putc(c, filePtr);
         }

      }
      else
         printf("Unable to open temporary file\n");

   else
      printf("Unable to open %s\n", fileName);

   return 0;
}
```

Fig. 14.6 Cómo utilizar los archivos temporales (parte 1 de 2).


```
This program changes tabs to spaces.
Enter a file to be modified: data

The file before modification is:
0       1       2       3       4
        5       6       7       8       9

The file after modification is:
0 1 2 3 4
 5 6 7 8 9
```

Fig. 14.6 Cómo utilizar los archivos temporales (parte 2 de 2).

14.10 Manejo de señales

Un evento inesperado o *señal*, puede hacer que termine un programa en forma prematura. Algunos eventos inesperados son *interrupciones* (escribir **<ctrl> c** en un sistema UNIX o en DOS), *instrucciones ilegales, violaciones de segmentación, órdenes de terminación provenientes del sistema operativo* y *excepciones de punto flotante* (división entre cero o la multiplicación de valores de punto flotante grandes). Mediante la función ***signal***, la *biblioteca de manejo de señales* da la capacidad de *atrapar* eventos inesperados. La función **signal** recibe dos argumentos un número de señal entero y un apuntador a la función de manejo de señal. Las señales pueden ser generadas por la función ***raise*** que toma como argumento un número de señal entero. En la figura 14.7 se resumen las señales estándar, definidas en el archivo de cabecera ***signal.h***. El programa de la figura 14.8 demuestra el uso de las funciones **signal** y **raise**.

El programa de la figura 14.8 utiliza la función **signal** para atrapar una señal interactiva (**SIGINT**). El programa empieza llamando **signal** mediante **SIGINT** y un apuntador a la función **signal_handler** (recuerde que el nombre de una función es un apuntador al principio de la misma). Cuando se genera una señal del tipo **SIGINT**, se pasa el control a la función **signal_handler**, se imprime un mensaje y se le da al usuario la opción de continuar con la

Señal	Explicación
SIGABRT	Terminación anormal del programa (como una llamada a la función **abort**).
SIGFPE	Una operación aritmética errónea, como sería una división entre cero o una operación que resulte en un desbordamiento.
SIGILL	Detección de una instrucción ilegal.
SIGINT	Recepción de una señal de atención interactiva.
SIGSEGV	Un acceso inválido a almacenamiento.
SIGTERM	Una solicitud de terminación definida al programa.

Fig. 14.7 Las señales definidas en el encabezado **signal.h**.

ejecución normal del programa. Si el usuario desea continuar con la ejecución, el manejador de señal es reinicializado, llamando otra vez a **signal** (algunos sistemas requieren que el manejador de señal sea reinicializado), y el control regresa al punto en el programa donde la señal fue detectada. En este programa, se utiliza la función **raise** para simular una señal interactiva. Se escoge un número al azar entre **1** y **50**. Si el número es **25**, entonces se llama a **raise** para generar la señal. Por lo regular, la señales interactivas son iniciadas desde fuera del programa. Por ejemplo, si en un sistema UNIX o DOS se escribe **<ctrl> c** durante la ejecución del programa, se genera una señal interactiva, que termina la ejecución del programa. El manejo de señales puede ser utilizado para atrapar una señal interactiva y evitar que el programa se dé por terminado.

14.11 Asignación dinámica de memoria: funciones Calloc y Realloc

En el capítulo 12, "Estructuras de datos", se presentó la noción de la asignación dinámica de la memoria, mediante el uso de la función **malloc**. Como se explicó en el capítulo 12, tratándose de ordenamiento rápido, búsqueda y acceso a los datos los arreglos resultan mejores que las listas enlazadas. Sin embargo, por lo regular los arreglos son *estructuras estáticas de datos*. Para la asignación dinámica de memoria, la biblioteca general de utilerías (**stdlib.h**) contiene otras dos funciones —**calloc** y **realloc**. Estas funciones pueden ser utilizadas para crear y modificar *arreglos dinámicos*. Como fue mostrado en el capítulo 7, "Apuntadores", un apuntador a un arreglo puede ser marcado con un subíndice, de la misma forma que un arreglo. Entonces, un apuntador a una porción contigua de memoria generada por **calloc**, puede ser manipulado como si fuera un arreglo. La función **calloc** asigna de forma dinámica la memoria para un arreglo. El prototipo para **calloc** es

```
void *calloc(size_t nmemb, size_t size);
```

Recibe dos argumentos —el número de elementos (**nmemb**) y el tamaño de cada elemento (**size**)— e inicializa los elementos del arreglo a cero. La función regresa un apuntador a la memoria asignada, o un apuntador **NULL** si no se asigna memoria.

La función **realloc** modifica el tamaño de un objeto asignado mediante una llamada a **malloc, calloc** o **realloc** anterior. Siempre y cuando la cantidad de memoria asignada sea mayor que la cantidad ya asignada, el contenido del objeto original no es modificado. De lo contrario, se conservará el contenido sin modificación hasta el tamaño del nuevo objeto. El prototipo correspondiente a **realloc** es

```
void *realloc(void *ptr, size_t size);
```

La función **realloc** toma dos argumentos un apuntador al objeto original (**ptr**) y el nuevo tamaño del objeto (**size**). Si **ptr** es **NUL**, **realloc** funciona en forma idéntica a **malloc**. Si **size** es **0** y **ptr** no es **NULL**, se libera la memoria correspondiente al objeto. De lo contrario, si **ptr** no es **NULL** y el tamaño es mayor que **0**, **realloc** intenta asignar un nuevo bloque de memoria para el objeto. Si el nuevo espacio no puede ser asignado, el objeto al cual señala **ptr** se conserva sin modificar. La función **realloc** regresa ya sea un apuntador a la memoria reasignada, o un apuntador **NULL**.

14.12 La bifurcación incondicional: Goto

A lo largo del texto hemos insistido en la importancia de utilizar técnicas de programación estructurada para construir software confiable, que resulte fácil de depurar, mantener y modificar. En algunos casos, el rendimiento es de mayor importancia que una estricta adherencia a las


```
/* Using signal handling */

#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <time.h>

void signal_handler(int);

main()
{
   int i, x;

   signal(SIGINT, signal_handler);
   srand(clock());

   for (i = 1; i <= 100; i++) {
      x = 1 + rand() % 50;

      if (x == 25)
         raise(SIGINT);

      printf("%4d", i);

      if (i % 10 == 0)
         printf("\n");
   }

   return 0;
}

void signal_handler(int signalValue)
{
   int response;

   printf("%s%d%s\n%s",
          "\nInterrupt signal (", signalValue, ") received.",
          "Do you wish to continue (1 = yes or 2 = no)? ");

   scanf("%d", &response);

   while (response != 1 && response != 2) {
      printf("(1 = yes or 2 = no)? ");
      scanf("%d", &response);
   }

   if (response == 1)
      signal(SIGINT, signal_handler);
   else
      exit(EXIT_SUCCESS);
}
```

Fig. 14.8 Cómo utilizar el manejo de señales (parte 1 de 2).

```
   1   2   3   4   5   6   7   8   9  10
  11  12  13  14  15  16  17  18  19  20
  21  22  23  24  25  26  27  28  29  30
  31  32  33  34  35  36  37  38  39  40
  41  42  43  44  45  46  47  48  49  50
  51  52  53  54  55  56  57  58  59  60
  61  62  63  64  65  66  67  68  69  70
  71  72  73  74  75  76  77  78  79  80
  81  82  83  84  85  86  87  88
Interrupt signal (4) received.
Do you wish to continue (1 = yes or 2 = no)? 1
  89  90
  91  92  93  94  95  96  97  98  99  100
```

Fig. 14.8 Cómo utilizar el manejo de señales (parte 2 de 2).

técnicas de programación estructurada. En estos casos, pueden utilizarse algunas técnicas de programación no estructuradas. Por ejemplo, podemos utilizar **break** para terminar la ejecución de una estructura de repetición, antes de que se convierta en falsa la condición de continuación de ciclo. Esto ahorrará repeticiones innecesarias del ciclo, si antes de la terminación de éste la tarea ha sido terminada.

Otro ejemplo de programación no estructurada es el *enunciado* **goto** —una bifurcación incondicional. El resultado del enunciado **goto** es un cambio en el flujo de control del programa al primer enunciado que siga a la *etiqueta* especificada en el enunciado **goto**. Una etiqueta es un identificador seguido por dos puntos. Una etiqueta deberá aparecer en la misma función que el enunciado **goto** al cual se hace referencia. El programa de la figura 14.9 utiliza enunciados **goto** para ciclar diez veces e imprimir el valor de contador cada una de las veces. Después de inicializar **count** a **1**, el programa verifica **count** para determinar si es mayor que **10** (la etiqueta **start** es pasada por alto, porque las etiquetas no ejecutan ninguna acción). De ser así, el control se transfiere desde **goto** al primer enunciado después de la etiqueta **end**. De lo contrario, se imprime **count** y se incrementa, y el control se transfiere de **goto** al primer enunciado después de la etiqueta **start**.

En el capítulo 3 indicamos que para escribir cualquier programa sólo eran requeridas 3 estructuras de control: secuencia, selección y repetición. Cuando se siguen las reglas de la programación estructurada, es posible crear estructuras de control muy anidadas, de las cuales es difícil salir con eficacia. En situaciones como esas, algunos programadores utilizan enunciados **goto**, como una salida rápida de una estructura muy anidada. Esto elimina la necesidad de probar múltiples condiciones para salir de una estructura de control.

Sugerencia de rendimiento 14.3

El enunciado **goto** *puede ser utilizado para salir con eficacia de estructuras de control muy anidadas.*

Observación de ingeniería de software 14.3

El enunciado **goto** *debe ser utilizado sólo en aplicaciones orientadas a rendimiento. El enunciado* **goto** *no es estructurado y puede llevar a programas más difíciles de depurar, mantener y modificar.*


```
/* Using goto */
#include <stdio.h>

main()
{
   int count = 1;

   start:                          /* label */
      if (count > 10)
         goto end;

      printf("%d  ", count);
      ++count;
      goto start;

   end:                            /* label */
      putchar('\n');

   return 0;
}
```

```
   1  2  3  4  5  6  7  8  9  10
```

Fig. 14.9 Cómo utilizar **goto**.

Resumen

- En muchos sistemas de cómputo y en particular, en sistemas UNIX y DOS es posible redirigir la entrada a un programa y redirigir la salida de un programa.
- La entrada se redirige desde la línea de comandos de UNIX y de DOS utilizando el símbolo de redirección de entrada (<)o mediante una tubería (|).
- La salida es redirigida desde la línea de comandos de UNIX y de DOS utilizando el símbolo de redirección de salida (>) o el símbolo de agregar salida (>>). El símbolo de redirección de salida sólo almacena la salida de programa en un archivo, y el símbolo de agregar salida, agrega la salida al final de un archivo.
- Las macros y definiciones del encabezado de argumentos variables **stdarg.h** proporcionan las capacidades necesarias para construir funciones utilizando listas de argumentos de longitud variable.
- Unos puntos suspensivos (...) en una función prototipo indican un número variable de argumentos.
- El tipo **va_list** es adecuado para contener información necesaria para las macros **va_start**, **va_arg**, y **va_end**. Para tener acceso a los argumentos en una lista de argumentos de longitud variable, deberá ser declarado un objeto del tipo **va_list**.
- Antes de poder tener acceso a los argumentos de una lista de argumentos de longitud variable, se invoca la macro **va_start**. La macro inicializa el objeto declarado mediante **va_list**, para su uso con las macros **va_arg**, y **va_end**.

- La macro **va_arg** se expande a una expresión del valor y del tipo del siguiente argumento de la lista de argumentos de longitud variable. Cada invocación de **va_arg** modifica el objeto declarado en **va_list**, de tal forma que el objeto señale al siguiente argumento dentro de la lista.
- La macro **va_end** facilita un regreso normal de una función cuya lista de argumentos variables fue referida por la macro **va_start**.
- En muchos sistemas y en particular en DOS y en UNIX incluyendo los parámetros **int argc** y **char *argv[]** en la lista de parámetros de **main**, es posible pasar argumentos a **main** desde la línea de comandos. El parámetro **argc** recibe el número de argumentos de la línea de comandos. El parámetro **argv** es un arreglo de cadenas, en el cual se almacenan los argumentos actuales de la línea de comandos.
- La definición de una función debe estar en su totalidad contenida en un archivo, no puede estar distribuida en dos o más archivos.
- Las variables globales deberán ser declaradas en cada uno de los archivos en las que serán utilizadas.
- Los prototipos de función pueden extender el alcance de una función más allá del archivo en el cual ha sido definida (en un prototipo de función no es requerido el especificador **extern**). Esto se lleva a cabo incluyendo a el prototipo de función en cada archivo en el cual sea invocada la función y compilando juntos ambos archivos.
- El especificador de clase de almacenamiento **static**, al ser aplicado a la variable global o a una función, impide que ésta sea utilizada por cualquier función que no haya sido definida en el mismo archivo. Esto se conoce como enlace interno. Las variables globales y las funciones que en sus definiciones no hayan sido precedidas por **static** tienen enlace externo —se puede tener acceso a ellas en otros archivos, si estos archivos contienen las declaraciones y/o los prototipos de función adecuadas.
- El especificador **static** se usa por lo general con funciones de utilería, que son llamadas sólo por funciones en un archivo en particular. Si una función no es requerida fuera de un archivo particular, deberá ser puesto en aplicación el principio del mínimo privilegio mediante el uso de **static**.
- Al construir programas extensos en múltiples archivos fuente, la compilación de programas se hace tediosa si al efectuar pequeñas modificaciones a un archivo se tiene que recompilar todo el programa. Muchos sistemas proporcionan utilerías especiales que recompila sólo el archivo de programa modificado. En sistemas UNIX la utilería se llama **make**. La utilería **make** lee un archivo llamado **makefile** que tiene instrucciones para compilar y enlazar el programa.
- La función **exit** obliga a la terminación de un programa como si se hubiera ejecutado normalmente.
- La función **atexit** registra una función en un programa, para que sea llamada a la terminación del programa es decir, ya sea cuando el programa termina por llegar al final de **main** o cuando se invoca a **exit**.
- La función **atexit** toma como argumento un apuntador a una función (es decir, el nombre de la función). Las funciones llamadas a la terminación del programa no pueden tener argumentos, y no pueden regresar un valor. Pueden registrarse hasta 32 funciones para su ejecución a la terminación del programa.

- La función **exit** toma un argumento. Por lo regular el argumento es la constante simbólica **EXIT_SUCCESS** o la constante simbólica **EXIT_FAILURE**. Si **exit** se llama con **EXIT_SUCCESS**, es regresado el valor definido por la puesta en práctica correspondiente a la terminación exitosa al entorno llamador. Si **exit** es llamado mediante **EXIT_FAILURE** el valor definido por la puesta en práctica correspondiente a la terminación no exitosa, es regresado.
- Cuando se invoca la función **exit**, cualesquiera funciones registradas con **atexit** son invocadas en orden inverso a su registro, todos los flujos asociados con el programa son vaciados y cerrados, y el control regresa al entorno huésped.
- El estándar ANSI (An90) indica que cuando se utiliza **volatile** para calificar un tipo, la naturaleza del acceso a un objeto de dicho tipo depende de la puesta en práctica. Kernighan y Ritchie (Ke88) indican que el calificador **volatile** se utiliza para suprimir varios tipos de optimizaciones.
- C proporciona sufijos enteros y de punto flotante para especificar los tipos de constantes enteras y de punto flotante. Los sufijos enteros son: **u** o bien **U** por un entero **unsigned**, **l** o **L** por un entero **long**, y **ul** o **UL** por un entero **unsigned long**. Si una constante entera no tiene sufijo, su tipo queda determinado por el primer tipo capaz de almacenar un valor de dicho tamaño (primero **int**, luego **long int** y por último **unsigned long int**). Los sufijos de punto flotante son: **f** o **F** para un **float**, y **l** o **L** para un **long double**. Una constante de punto flotante sin sufijo es del tipo **double**.
- C también proporciona capacidades para procesamiento de archivos binarios, pero algunos sistemas de cómputo no aceptan archivos binarios. Si los archivos binarios no son aceptados, y un archivo se abre en el modo de archivo binario, el archivo será procesado como archivo de texto.
- La función **tempfile** abre un archivo temporal en modo **"wb+"**. A pesar de que éste es un modo de archivo binario, algunos sistemas procesan los archivos temporales como archivos de texto. Un archivo temporal existe en tanto no sea cerrado mediante **fclose** o en tanto termine el programa.
- La biblioteca de manejo de señales proporciona la capacidad de atrapar eventos inesperados mediante la función **signal**. La función **signal** recibe dos argumentos un número de señal entero y un apuntador a la función de manejo de señal.
- La señales también pueden ser generadas mediante la función **raise** y un argumento entero.
- La biblioteca general de utilerías (**stdlib.h**) tiene dos funciones para la asignación dinámica de memoria **calloc** y **realloc**. Estas funciones pueden ser utilizadas para crear arreglos dinámicos.
- La función **calloc** recibe dos argumentos el número de elementos (**nmemb**) y el tamaño de cada elemento (**size**), e inicializa a cero los elementos del arreglo. La función regresa ya sea un apuntador a la memoria asignada, o un apuntador **NULL** si la memoria no ha sido asignada.
- La función **realloc** modifica el tamaño de un objeto asignado por una llamada a **malloc**, **calloc** o **realloc** previa. El contenido del objeto original no es modificado, siempre y cuando la cantidad de memoria asignada sea mayor que la cantidad ya asignada.
- La función **realloc** toma dos argumentos un apuntador al objeto original (**ptr**) y el nuevo tamaño del objeto (**size**). Si **ptr** es **NULL**, **realloc** funciona en forma idéntica a **malloc**. Si **size** es **0** y el apuntador recibido no es **NULL**, la memoria para el objeto es liberada. De lo contrario si **ptr** no es **NULL** y el tamaño es mayor que 0, **realloc** intenta asignar un nuevo

contrario si `ptr` no es `NULL` y el tamaño es mayor que 0, `realloc` intenta asignar un nuevo bloque de memoria para el objeto. Si el nuevo espacio no puede ser asignado, el objeto al cual señala `ptr`, se conserva sin modificar. La función `realloc` regresa ya sea un apuntador a la memoria reasignada, o un apuntador `NULL`.

- El resultado del enunciado `goto` es una modificación en el flujo de control del programa. La ejecución del programa continúa en el primer enunciado inmediatamente después de la etiqueta especificada en el enunciado `goto`.
- Una etiqueta es un identificador seguido por dos puntos. Debe de aparecer una etiqueta en la misma función que en el enunciado `goto` que se refiere a ella.

Terminología

símbolo de agregar salida >>
`argc`
`argv`
`atexit`
`calloc`
argumentos en la línea de comandos
`const`
arreglos dinámicos
evento
`exit`
enlace externo
especificador de clase de almacenamiento `extern`
`EXIT_FAILURE`
`EXIT_SUCCESS`
sufijo `float` (`f` o `F`)
excepción de punto flotante
enunciado `goto`
redirección de E/S
instrucción ilegal
enlace interno
interrupción
sufijo `long double` (`l` o `L`)
sufijo `long integer` (`l` o `L`)
`make`
`makefile`
tubería |
entubamiento
`raise`
`realloc`
símbolo de redirección de entrada <
símbolo de redirección de salida >
violación de segmentación
`signal`
biblioteca de manejo de señales
`signal.h`
especificador de clase de almacenamiento `static`
`stdarg.h`
`archivo temporal`
`tmpfile`
atrapar
sufijo entero `unsigned` (`u` o bien `U`)
sufijo entero `unsigned long` (`ul` o bien `UL`)
`va_arg`
`va_end`
`va_list`
`va_start`
lista de argumentos de longitud variable
`volatile`

Error común de programación

14.1 Colocar puntos suspensivos en la mitad de una lista de parámetros de función. Los puntos suspensivos sólo pueden ser colocados al final de la lista de parámetros.

Sugerencias de portabilidad

14.1 Algunos sistemas no aceptan nombres de variables globales o nombres de funciones con más de 6 caracteres. Esto deberá ser tomado en cuenta al escribir programas que se planea serán transportados a varias plataformas.

14.2 Cuando esté escribiendo programas portátiles utilice archivos de texto.

Sugerencias de rendimiento

14.1 Las variables globales aumentan el rendimiento debido a que son de forma directa accesibles por cualquier función —se elimina la sobrecarga de pasar datos a función.

14.2 Considere utilizar archivos binarios en lugar de archivos de texto en aquellas aplicaciones que exijan alto rendimiento

14.3 El enunciado `goto` puede ser utilizado para salir con eficacia de estructuras de control muy anidadas.

Observaciones de ingeniería de software

14.1 Las variables globales deberían de evitarse, a menos de que el rendimiento de la aplicación sea crítico, porque violan el principio del mínimo privilegio.

14.2 Las funciones pueden ser comunes para muchas aplicaciones. En casos como éstos, estas funciones deberán ser almacenadas en sus propios archivos fuente, y cada archivo fuente deberá tener su correspondiente archivo de cabecera, que contenga los prototipos de función. Esto permite a los programadores de distintas aplicaciones reutilizar el mismo código, mediante la inclusión del archivo de cabecera apropiado, y compilar su aplicación con el archivo fuente correspondiente.

14.3 El enunciado `goto` debe ser utilizado sólo en aplicaciones orientadas a rendimiento. El enunciado `goto` no es estructurado y puede llevar a programas más difíciles de depurar, mantener y modificar.

Ejercicios de autoevaluación

14.1 Llene los espacios vacíos de cada uno de los siguientes:

a) El símbolo _____ se utiliza para redirigir datos de entrada del teclado para que provengan de un archivo.
b) El símbolo _____ se utiliza para redirigir la salida a pantalla para que se coloque en un archivo.
c) El símbolo _____ se utiliza para agregar la salida de un programa al final de un archivo.
d) La _____ se utiliza para redirigir la salida de un programa como entrada de otro programa.
e) Una _____ en la lista de parámetros de una función indica que la función puede recibir un número variable de argumentos.
f) La macro _____ debe de ser invocada antes de que se pueda tener acceso a los argumentos de una lista de argumentos de longitud variable.
g) Se utiliza la macro _____ para tener acceso a los argumentos individuales de una lista de argumentos de longitud variable.
h) La macro _____ facilita un regreso normal de una función cuya lista de argumentos variable fue referida por la macro `va_start`.
i) El argumento _____ de `main` recibe el número de argumentos en la línea de comandos.
j) El argumento _____ de `main` almacena argumentos en la línea de comandos como cadenas de caracteres.
k) La utilería de UNIX _____ lee un archivo llamado _____ que contiene instrucciones para compilar y enlazar un programa formado de múltiples archivos fuente. La utilería también recompila el archivo si desde la última vez que fue compilado éste ha sido modificado.
l) La función _____ obliga a un programa a terminar su ejecución.
m) La función _____ registra una función que deberá ser llamada a la terminación normal del programa.
n) El calificador de tipo _____ indica que un objeto no deberá ser modificado después de haberse inicializado.
o) Se puede agregar un _____ entero o de punto flotante a una constante entera o de punto flotante para especificar el tipo exacto de la misma.

p) La función _____ abre un archivo temporal que existirá en tanto no se cierre o la ejecución del programa se termine.
q) La función _____ puede ser utilizada para atrapar eventos inesperados.
r) La función _____ genera una señal desde dentro de un programa.
s) La función _____ asigna dinámicamente la memoria para un arreglo que inicializa los elementos a cero.
t) La función _____ modifica el tamaño de un bloque de memoria ya asignada dinámicamente.

Respuestas a los ejercicios de autoevaluación

14.1 a) redirige entrada (<). b) redirige salida (>). c) agrega salida (>>). d) tubería (|). e) puntos suspensivos (...). f) `va_start`. g) `va_arg`. h) `va_end`. i) `argc`. j) `argv`. k) `make, makefile`. l) `exit`. m) `atexit`. n) `const`. o) sufijo. p) `tempfile`. q) `signal`. r) `raise`. s) `calloc`. t) `realloc`.

Ejercicios

14.2 Escriba un programa que calcule el producto de una serie de enteros que son pasados a la función `product` utilizando una lista de argumentos de longitud variable. Pruebe su función con varias llamadas, cada una con un número diferente de argumentos.

14.3 Escriba un programa que imprima los argumentos en la línea de comandos del programa.

14.4 Escriba un programa que ordene un arreglo de enteros en orden ascendente o en orden descendente. El programa deberá utilizar argumentos en la línea de comandos, para pasar o el argumento `-a` para orden ascendente o el argumento `-d` para orden descendente. (Nota: este es el formato estándar en UNIX para pasar opciones a un programa.)

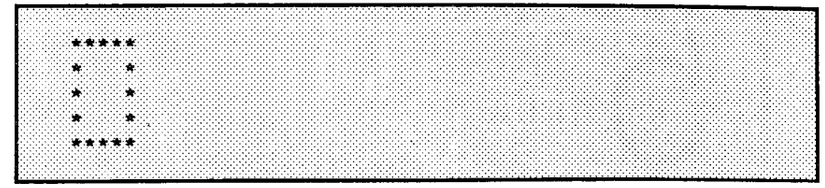
14.5 Escriba un programa que coloque un espacio entre cada carácter dentro de un archivo. El programa deberá primero escribir el contenido de un archivo a modificarse a un archivo temporal con espacios entre cada carácter, y a continuación copiar el archivo de regreso al archivo original. Esta operación deberá sobreescribir el contenido original del archivo.

14.6 Lea los manuales de su sistema para determinar qué señales contienen la biblioteca de manejo de señales (`signal.h`). Escriba un programa que contenga los manejadores de señales correspondientes a las señales estándar `SIGABRT` y `SIGINT`. El programa deberá probar el método de atrapar de estas señales mediante el llamado a las funciones `abort` para generar una señal del tipo `SIGABRT`, y mediante la escritura de `<ctrl> c` para generar una señal del tipo `SIGINT`.

14.7 Escriba un programa que asigne de forma dinámica un arreglo de enteros. El tamaño del arreglo deberá ser introducido desde el teclado. Los elementos del arreglo deberán ser valores asignados introducidos desde el teclado. Imprima los valores del arreglo. A continuación, reasigne la memoria para el arreglo a la mitad del número actual de elementos. Imprima los valores restantes en el arreglo para confirmar que corresponden a la primera mitad de los valores del arreglo original.

14.8 Escriba un programa que tome dos argumentos de la línea de comandos que son nombres de archivo, lea los caracteres del primer archivo un carácter a la vez, y escriba los caracteres en orden inverso en el segundo archivo.

14.9 Escriba un programa que utilice enunciados `goto` para simular una estructura de ciclado anidada, que imprima un cuadro de asteriscos como sigue:



El programa deberá utilizar sólo los siguientes tres enunciados `printf`:

```
printf("*");
printf(" ");
printf("\n");
```

15

C++ como un "C mejorado"

Objetivos

- Familiarizarse con las mejoras de C++ a C.
- Reconocer la razón porqué C es una base para subsecuentes estudios de la programación en general y de C ++ en particular.

Lo mejor es verte
con mi estimado.
El gran lobo feroz a la pequeña caperucita roja

Sinopsis

Resumen • Terminología • Errores comunes de programación • Prácticas sanas de programación • Sugerencia de portabilidad • Sugerencias de rendimiento • Observación de ingeniería de software • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios • Lectura recomendada • Apéndice: recursos de C++.

15.1 Introducción

¡Bienvenido a C++! C++ es una mejoría sobre muchas de las características de C, y proporciona capacidades de programación orientada a objetos *(OOP,* por *object-oriented programming)* que promete mucho para incrementar la productividad, calidad y reutilización del software. El resto de este capítulo analiza muchas de las mejoras que C++ tiene sobre C.

Los diseñadores de C y los responsables de sus primeras puestas en práctica jamás anticiparon que este lenguaje resultaría en un fenómeno como éste (lo mismo es cierto para el sistema operativo UNIX). Cuando un lenguaje de programación se torna tan arraigado como C, nuevas necesidades demandan que el lenguaje evolucione, en lugar de que sólo sea remplazado por un nuevo lenguaje.

C++ fue desarrollado por Bjarne Stroustrup en los Laboratorios Bell (St86) y originalmente fue llamado "C con clases". El nombre C++ incluye el operador de incremento (++) de C, para indicar que C++ es una versión mejorada de C.

C++ es un super conjunto de C, por lo que, para compilar los programas existentes de C, los programadores pueden utilizar un compilador C++ y posteriormente modificar de forma gradual estos programas a C++. En este momento, ya algunos proveedores importantes de software ofrecen compiladores C++ y no ofrecen productos C por separado.

Aún no existe un estándar ANSI, a pesar de que un comité ANSI está desarrollando una proposición. Muchas personas creen que para mediados de los años noventa, la mayor parte de los entornos de programación C se convertirán a C++.

15.2 Comentarios de una sola línea de C++

Con frecuencia los programadores insertan un pequeño comentario al final de una línea de código. C requiere que un comentario sea delimitado mediante `/*` y `*/`. C++ le permitirá a que empiece un comentario con `//` y que utilice el resto de la línea para texto del comentario; el fin de la línea da de manera automática por terminado el comentario. Esta forma de comentario ahorra algunos golpes de tecla, pero sólo es aplicable a comentarios que no continúen a la línea siguiente.

Por ejemplo, el comentario C

```
/* This is a single-line comment. */
```

requiere de ambos delimitadores `/*` y `*/`, aun para un comentario de una línea. La versión en una línea de C++ de lo anterior es

```
// This is a single-line comment.
```

Para comentarios de más de una línea, como

```
/* This is one way to   */
/* Write neat multiple- */
/* line comments.       */
```

La notación C++ pudiera aparecer más concisa como en

```
// This is one way to
// Write neat multiple-
// line comments.
```

Pero recuerde que los comentarios C pueden extenderse a varias líneas, por lo que en realidad, sólo se necesita un par (en vez de que de los tres pares que hemos utilizado) de delimitadores de comentario, como en

```
/* This is one way to
   Write neat multiple-
   line comments.       */
```

Dado que C++ es un super conjunto de C, en un programa C++ ambas formas de comentario son aceptables.

Error común de programación 15.1

*Olvidar cerrar un comentario de estilo C mediante */.*

Práctica sana de programación 15.1

*Usar los comentarios // de estilo C++, evita errores de comentarios sin terminar, que ocurren al olvidarse de cerrar los comentarios de estilo C con */.*

Este error en apariencia sencillo puede llevar a errores muy sutiles, algunos de los cuales pueden deslizarse sin detección por parte del compilador. El efecto de omitir el `*/` es que el compilador supone que el comentario aún no ha terminado, y que el comentario continúa a lo largo de las líneas subsiguientes hasta que alcanza otro `*/`, correspondiente a otro comentario, o el final del archivo del programa. Esto podría dar como resultado que el compilador ignore alguna parte clave del programa.

15.3 Flujo de entrada/salida de C++

C++ ofrece una alternativa a las llamadas de función **`printf`** y **`scanf`** para manejar la entrada/salida de los tipos y cadenas de datos estándar. Por ejemplo, el diálogo simple

```
printf("Enter new tag: ");
scanf("%d", &tag);
printf("The new tag is: %d\n", tag);
```

se escribe en C++ como

```
cout << "Enter new tag: ";
cin >> Tag;
cout << "The new tag is: " << tag << '\n';
```

El primer enunciado utiliza el *flujo estándar de salida* **`cout`** y el operador << (el *operador de inserción de flujo* que se pronuncia *"colocar en"*). El enunciado se lee

La cadena **`"Enter new tag"`** *es colocada en el flujo de salida* **`cout`**.

Note que el operador de inserción de flujo es también el operador de desplazamiento a la izquierda a nivel de bits. El segundo enunciado utiliza el *flujo de entrada estándar* **`cin`** y el operador >> (el *operador de extracción de flujo,* que se pronuncia *"obtener de"*). El enunciado se lee

Obtener un valor para tag del flujo de entrada **`cin`**.

Note que el operador de extracción de flujo también es un operador de desplazamiento a la derecha a nivel de bits cuando el argumento izquierdo es un tipo entero. Los operadores de inserción y de extracción de flujo, a diferencia de **`printf`** y de **`scanf`**, no requieren de cadenas de formato y de especificadores de conversión para indicar los tipos de datos que son extraídos o introducidos. C++ tiene muchos ejemplos como éste, en los cuales de forma automática "sabe" qué tipos utilizar. También note que, cuando se utiliza con el operador de extracción de flujo, la variable **`tag`** no es precedida por el operador de dirección **`&`**, como es requerido en el caso de **`scanf`**.

Para utilizar entradas/salidas de flujo, los programas C++ deben incluir el archivo de cabecera ***`iostream.h`***. El programa de la figura 15.1 solicita las variables **`myAge`** y **`friendsAge`** y determina si usted es de más edad, menos edad o de la misma que su amigo. Note la colocación de las declaraciones de edad, justo antes de la referenciación de las edades en los enunciados de entrada. En el capítulo 19, "Flujo de entrada/salida" se da una descripción detallada de las características del flujo de entrada/salida de C++.

Práctica sana de programación 15.2

Utilizar entrada/salida orientada a flujo de tipo C++ hace los programas más legibles (y menos sujetos a errores), que sus contrapartidas escritas en C mediante las llamadas de función **`printf`** *y* **`scanf`**.


```
// Simple stream input/output
#include <iostream.h>

main()
{
   cout << "Enter your age: ";
   int myAge;
   cin >> myAge;

   cout << "Enter your friend's age: ";
   int friendsAge;
   cin >> friendsAge;

   if (myAge > friendsAge)
      cout << "You are older.\n";
   else
      if (myAge < friendsAge)
         cout << "You are younger.\n";
      else
         cout << "You and your friend are the same age.\n";

   return 0;
}
```

```
Enter your age: 23
Enter your friend's age: 20
You are older.
```

```
Enter your age: 20
Enter your friend's age: 23
You are younger.
```

```
Enter your age: 20
Enter your friend's age: 20
You and your friend are the same age.
```

Fig. 15.1 Flujo de E/S y los operadores de inserción y extracción de flujo.

15.4 Declaraciones en C++

En un bloque en C, todas las declaraciones deben aparecer antes de cualquier enunciado ejecutable. En C++, las declaraciones pueden ser colocadas en cualquier parte de un enunciado ejecutable, siempre y cuando las declaraciones antecedan el uso de lo que se está declarando. Por ejemplo

```
cout << "Enter two integers: ";
int x, y;
cin >> x >> y;
cout << "The sum of " << x << " and " << y
     << " is " << x + y << '\n';
```

declara las variables **x** y **y** después del enunciado ejecutable **cout**, pero antes que sean utilizadas en el enunciado subsecuente **cin**. También, las variables pueden ser declaradas en la sección de inicialización de una estructura **for** dichas variables se mantienen en alcance hasta el final del bloque en el cual la estructura **for** está definida. Por ejemplo,

```
for (int i = 0; i <= 5; i ++)
   count << i <<  '\n';
```

declara la variable **i** ser un entero y la inicializa a **0** en la estructura **for**.

El alcance de una variable local C++ empieza en su declaración y se extiende hasta la llave derecha de cierre (**}**). Por lo tanto, los enunciados anteriores a la declaración variable no pueden referirse a la variable, aun si éstos están en el mismo bloque. Las declaraciones de variable no pueden ser colocadas en la condición de una estructura **while, do/while, for**, o **if.**

Práctica sana de programación 15.3

Colocar la declaración de una variable cerca de su primer uso puede hacer los programas más legibles.

Error común de programación 15.2

Declarar una variable después de que haya sido referenciada en un enunciado.

15.5 Cómo crear nuevos tipos de datos en C++

C++ proporciona la capacidad de crear tipos definidos por el usuario mediante el uso de la palabra reservada **enum**, la palabra reservada **struct**, la palabra reservada **union** y la nueva palabra reservada **class**. Al igual que en C, las enumeraciones C++ son declaradas mediante **enum**. Sin embargo; a diferencia de C, una enumeración en C++, cuando se declara, se convierte en un tipo nuevo. Para declarar de variable del nuevo tipo la palabra reservada **enum** no es requerida. Lo mismo es cierto en el caso de **struct, union** y **class**. Por ejemplo, las declaraciones

```
enum Boolean {FALSE, TRUE};

struct Name {
   char first[10];
   char last[10];
};
union Number {
   int i;
   float f;
};
```

crean tres tipos de datos, definidos por usuario, con los nombres de etiqueta **Boolean, Name** y **Number**. Los nombres de etiqueta pueden ser utilizados para declarar variables, como sigue:

```
Boolean done = FALSE;
Name student;
Number x;
```


Estas declaraciones crean la variable **Boolean done** (inicializada a **FALSE**), la variable **Name student** y la variable **Number x.**

Como en C, las enumeraciones por omisión son valuadas iniciándose en cero y cada elemento subsecuente se incrementa en uno. Por lo tanto, la enumeración **Boolean** asigna el valor **0** a **FALSE** y **1** a **TRUE**. Cualquier elemento en una enumeración puede ser asignado un valor entero. Los elementos subsecuentes, que no sean asignados a un valor, recibirán de manera automática el valor del elemento anterior, incrementado en uno.

15.6 Prototipos de función y verificación de tipo

El prototipo de función le permiten al compilador de C verificar por tipo la exactitud de las llamadas de función. En ANSI C, los prototipos de función son opcionales. En C++ los prototipos de función son requeridas para todas las funciones. Una función definida en un archivo, antes de cualquier llamada a la misma, no requiere de un prototipo de función por separado. En este caso, el encabezado de función actúa como prototipo de función. C++ también requiere que se declaren todos los parámetros de función en los paréntesis de la definición de función y del prototipo. Por ejemplo, una función **square**, que toma un entero de argumento y regresa un entero tiene el prototipo:

```
int square(int);
```

Las funciones que no regresan un valor se declaran con el tipo de regreso **void.**

Error común de programación 15.3

Un intento de regresar un valor de una función **void** *o de utilizar el resultado de una llamada a una función* **void.**

Para especificar en C una lista vacía de parámetros, entre paréntesis se coloca la palabra reservada **void**. Si no se coloca nada en los paréntesis de un prototipo de función de C, para dicha función se desactiva toda verificación de argumentos y nada se supone en relación con el número de argumentos y los tipos de argumentos a la función. Cualquier llamada de función correspondiente a esta función puede pasar cualesquiera argumentos que desee, sin que el compilador reporte error alguno.

En C++, una lista vacía de parámetros se especifica escribiendo **void** o absolutamente nada en los paréntesis. La declaración

```
void print();
```

especifica que la función **print** no toma argumentos y no regresa un valor. En la figura 15.2 se muestran ambas formas de declarar en C++ y de usar las funciones que no toman argumentos.

Sugerencia de portabilidad 15.1

El significado de una lista de función de parámetros vacía es muy distinto en C++ que en C. En C, significa que se deshabilita toda verificación de argumentos. En C++, significa que la función no toma argumentos. Por lo tanto, si se compilan en C++, los programas en C que utilizan esta característica pudieran ejecutarse en forma diferente.

Error común de programación 15.4

Los programas en C++ no se compilarán a menos de que sean proporcionadas los prototipos de función para cada una de las funciones o que cada función quede definida antes de ser utilizada.

```
// Functions that take no arguments
#include <iostream.h>

void f1();
void f2(void);

main()
{
   f1();
   f2();

   return 0;
}

void f1()
{
   cout << "Function f1 takes no arguments\n";
}

void f2(void)
{
   cout << "Function f2 also takes no arguments\n";
}
```

```
Function f1 takes no arguments
Function f2 also takes no arguments
```

Fig. 15.2 Dos maneras de declarar y utilizar funciones que no toman argumentos.

15.7 Funciones en línea

Desde el punto de vista de la ingeniería del software es una buena idea poner en práctica un programa como un conjunto de funciones, pero las llamadas de función involucran sobrecarga en tiempo de ejecución. C++ tiene *funciones en línea* que ayudan a reducir la sobrecarga por llamadas de función especial para pequeñas funciones. El calificador **inline** colocado en la definición de función antes del tipo de regreso de una función "aconseja" al compilador que genere una copia del código de la función "in situ" (cuando sea apropiado), a fin de evitar una llamada de función. La contrapartida es que en el programa se insertan muchas copias de código de la función, en vez de, cada vez que se llama a la función, tener una copia de la función a la cual pasarle el control. El compilador puede ignorar el calificador **inline** y típicamente así lo hará para todas, a excepción de las funciones más pequeñas.

Observación de ingeniería del software 15.1

Cualquier modificación a una función `inline` *requiere que sean recompilados todos los clientes de dicha función. Esto pudiera resultar de importancia en algunas situaciones de desarrollo y mantenimiento de programas.*

Las funciones en línea ofrecen ventajas en comparación con las macros de preprocesador (vea el capítulo 13) que expanden código en línea. Una ventaja es que las funciones **inline** son iguales a cualquier otra función de C++. Por lo tanto, en llamadas a las funciones **inline** se ejecutará

una adecuada verificación de tipo; las macros de preprocesador no aceptan verificación de tipo. Otra ventaja es que las funciones **inline** eliminan los efectos colaterales inesperados, asociados con un uso inapropiado de las macros de preprocesador. Por último, las funciones **inline** pueden ser depuradas mediante un programa depurador. El depurador no reconoce a las macros de preprocesador como unidades especiales, porque sólo son sustituciones de texto, efectuadas por el preprocesador antes de la compilación del programa. Un depurador puede ayudar a localizar errores lógicos resultado de sustituciones de macros, pero no puede atribuir dichos errores a macros específicas.

Práctica sana de programación 15.4

El calificador `inline` *deberá ser utilizado sólo tratándose de funciones pequeñas, de uso frecuente.*

Sugerencia de rendimiento 15.1

Usar funciones `inline` *puede reducir tiempo de ejecución, pero puede aumentar el tamaño del programa.*

El programa de la figura 15.3 utiliza la función **inline** llamada **cube** para calcular el volumen de un cubo del lado **s**. La palabra reservada **const** en la lista de parámetros de la función cube, le indica al compilador que la función no modifica a la variable **s**. En la sección 15.9 se analiza con mayor detalle la palabra reservada **const**.

Las macros de preprocesador son operaciones definidas en una directiva de preprocesador **#define**. Una macro está compuesta de un nombre (identificador de la macro) y texto de remplazo. Las macros pueden ser definidas sin o con lista de argumentos. Las macros sin argumentos son procesadas como si fueran constantes simbólicas dentro del programa —el texto de remplazo sustituye el identificador de la macro. Las macros con argumentos sustituyen los

```
// Using an inline function to calculate
// the volume of a cube
#include <iostream.h>

inline float cube(const float s) { return s * s * s; }

main()
{
   cout << "Enter the side length of your cube:  ";

   float side;

   cin >> side;
   cout << "Volume of cube with side "
        << side << " is " << cube(side) << '\n';

   return 0;
}
```

```
Enter the side length of your cube: 3.5
Volume of cube with side 3.5 is 42.875
```

Fig. 15.3 Cómo utilizar una función en línea para calcular el volumen de un cubo.

argumentos dentro del texto de remplazo y a continuación, expanden la macro dentro del programa.

Considere la siguiente macro de un argumento, para calcular la superficie de un cuadrado:

```
#define VALIDSQUARE(x)     (x) * (x)
```

El preprocesador localiza dentro del programa cada ocurrencia de **VALIDSQUARE(x)**, sustituye **x** (ya sea un valor o una expresión) en el texto de remplazo, y expande la macro dentro del programa. Por ejemplo,

```
cout < VALIDSQUARE(7);
```

se expande a

```
cout < (7) * (7);
```

Los paréntesis en el texto de remplazo obligan a un orden correcto de evaluación de la operación de la macro. Por ejemplo,

```
cout << VALIDSQUARE(2 + 3);
```

se expande a

```
cout << (2 + 3) * (2 + 3);
```

que se evalúa correctamente como `5 * 5 = 25`. El preprocesador no evalúa expresiones incluidas como argumentos de una macro; sólo remplaza cada instancia del nombre del argumento en el texto de remplazo, con una copia de la totalidad de la expresión. Por lo tanto, los paréntesis son necesarios, para asegurar que los argumentos son correctamente evaluados.

Considere una macro correspondiente, pero sin paréntesis en el texto de remplazo:

```
#define INVALIDSQUARE(x)   x * x
```

Si damos 2 + 3 como el argumento, la macro se expande como sigue:

```
2 + 3 * 2 +3
```

Esta expresión queda evaluada de forma incorrecta como `2 + 6 + 3 = 11` porque la multiplicación tiene una precedencia más alta que la suma.

C++ tiene las funciones en línea para eliminar los problemas asociados con las macros, para eliminar la sobrecarga asociada con las llamadas de función, y para asegurar la verificación de argumentos para las funciones. Los argumentos de las funciones en línea se evalúan antes de ser "pasados" a la función. Por lo tanto los paréntesis encerrando cada instancia de cada argumento en el cuerpo de una función en línea no son necesarios.

La función en línea

```
inline int square(int x) { return x * x; }
```

calcula el cuadrado de su argumento entero **x**. La llamada **square(2 + 3)** evalúa el argumento `2 + 3 = 5` y substituye este valor dentro del cuerpo de la función. El programa de la figura 15.4 demuestra las macros **VALIDSQUARE** y **INVALIDSQUARE**, así como la función en línea **square**.


```
// Examples of valid macros, invalid macros
// and inline functions
#include <iostream.h>

#define VALIDSQUARE(x)     (x) * (x)
#define INVALIDSQUARE(x)   x * x

inline int square(int x) { return x * x; }

main()
{
   cout << "  VALIDSQUARE(2 + 3) = "
        << VALIDSQUARE(2 + 3)
        << "\nINVALIDSQUARE(2 + 3) = "
        << INVALIDSQUARE(2 + 3)
        << "\n        square(2 + 3) = "
        << square(2 + 3) << '\n';

   return 0;
}
```

```
  VALIDSQUARE(2 + 3) = 25
INVALIDSQUARE(2 + 3) = 11
        square(2 + 3) = 25
```

Fig. 15.4 Macros de preprocesador y funciones en línea.

La figura 15.5 contiene una lista completa de las palabras reservadas de C++. La figura muestra las palabras reservadas comunes a C y a C++, y a continuación resalta las palabras reservadas, únicas en C++. Cada una de las palabras reservadas nuevas de C++ es explicada con detalle más adelante en el libro.

Error común de programación 15.5

C++ es un lenguaje en evolución y algunas de sus características pudieran no estar disponibles en su computadora. Usar características no puestas en práctica causará errores de sintaxis.

15.8 Parámetros por referencia

En C, todas las llamadas de función son llamadas por valor. En C las llamadas por referencia son simuladas pasando un apuntador a un objeto y obteniendo a continuación acceso al objeto desreferenciando el apuntador en la función llamada. Recuerde que en C los nombres de arreglos son ya apuntadores (constantes), por lo que los arreglos son automáticamente pasados en llamada simulada por referencia. Otros lenguaje de programación ofrecen formas directas de llamada por referencia como los parámetros **var** en Pascal. C++ corrige esta debilidad de C al ofrecer *parámetros de referencia.*

Un parámetro de referencia es un seudónimo de su argumento correspondiente. Para indicar que un parámetro de función es pasado por referencia, sólo coloque ampersand (**&**) después del tipo del parámetro en el prototipo de función (exactamente de la misma forma en que pondría un * después del tipo parámetro, para indicar que un parámetro es el apuntador a una variable). Utilice

Palabras reservadas en C++

C y C++

`auto`	`break`	`case`	`char`	`const`
`continue`	`default`	`do`	`double`	`else`
`enum`	`extern`	`float`	`for`	`goto`
`if`	`int`	`long`	`register`	`return`
`short`	`signed`	`sizeof`	`static`	`struct`
`switch`	`typedef`	`union`	`unsigned`	`void`
`volatile`	`while`			

C++ únicamente

`asm`	Medio definido por la puesta en práctica de utilización de lenguaje de ensamble a lo largo de C++ (vea los manuales correspondientes a su sistema).
`catch`	Maneja una excepción generada por un `throw`.
`class`	Define una nueva clase. Pueden crearse objetos de esta clase.
`delete`	Destruye un objeto de memoria creado con `new`.
`friend`	Declara una función o una clase que sea un "friend (amigo)" de otra clase. Los amigos pueden tener acceso a todos los miembros de datos y a todas las funciones miembro de una clase.
`inline`	Avisa al compilador que una función particular deberá ser generada en línea, en vez de requerir de una llamada de función.
`new`	Asigna dinámicamente un objeto de memoria "en la tienda libre" —memoria adicional disponible para el programa en tiempo de ejecución. Determina automáticamente el tamaño del objeto.
`operator`	Declara un operador " homónimo".
`private`	Un miembro de clase accesible a funciones miembro y a funciones `friend` de la clase de miembros `private`.
`protected`	Una forma extendida de acceso `private`; también se puede tener acceso a los miembros `protected` por funciones miembro de clases derivadas y amigos de clases derivadas.
`public`	Un miembro de clase accesible a cualquier función.
`template`	Declare como construir una clase o una función, usando una variedad de tipos.
`this`	Un apuntador declarado en forma implícita en toda función de miembro no `static` de una clase. Señala al objeto al cual esta función miembro ha sido invocada.
`throw`	Transfiere control a un manejador de excepción o termina la ejecución del programa si no puede ser localizado un manejador apropiado.
`try`	Crea un bloque que contiene un conjunto de números que pudieran generar excepciones, y habilita el manejo de excepciones para cualquier excepción generada.
`virtual`	Declara una función virtual.

Fig. 15.5 Las palabras reservadas en C++.

la misma regla convencional en el encabezado de función al enlistar el tipo de parámetros. Por ejemplo, la declaración

```
int &count
```

en un encabezado de función puede ser leído "`count` es una referencia a `int`". En la llamada de función, sólo mencione la variable por su nombre y automáticamente será pasada por referencia. Entonces, el mencionar en el cuerpo de la función la variable por su nombre local, de hecho la refiere a la variable original en la función llamadora, y la variable original puede ser modificada de manera directa por la función llamada.

La figura 15.6 compara la llamada por valor, la llamada por referencia mediante apuntadores, y la llamada por referencia mediante parámetros de referencia. Los argumentos en las llamadas a `squareByValue` y `squareByReference` son idénticos. Es imposible saber si cualquiera de estas funciones modifica sus argumentos sin verificar los prototipos de función o las definiciones de función. Por esta razón, algunos programadores de C++ prefieren que los argumentos modificables sean pasados a las funciones utilizando apuntadores, y los argumentos no modificables sean pasados a las funciones utilizando referencias a constantes. Las referencias a constantes proporcionan la eficiencia del paso utilizando apuntadores y evitan la modificación de los argumentos del programa llamador. Para especificar una referencia a una constante, coloque el calificador `const` en la declaración de parámetros antes del especificador de tipo (en la sección 15.9 se analiza `const` en detalle). Note la colocación de `*` y de `&` en las listas de parámetros de ambas funciones. Algunos programadores de C++ prefieren escribir `int *bPtr` en vez de `int* bPtr`, e `int& cRef` en vez de `int &cRef`.

Práctica sana de programación 15.5

Utilice apuntadores para pasar argumentos que pudieran ser modificados por la función llamada, y utilice referencias a constantes para pasar argumentos extensos, que no serán modificados.

Error común de programación 15.6

Dado que en el cuerpo de la función llamada los parámetros de referencia se mencionan sólo por nombre, el programador pudiera de forma inadvertida tratar a los parámetros de referencia como parámetros en llamada por valor. Esto puede causar efectos colaterales inesperados, si las copias originales de las variables son modificadas por la función llamadora.

Sugerencia de rendimiento 15.2

En el caso de objetos extensos, utilice un parámetro de referencia `const` para simular la apariencia y la seguridad de una llamada por valor, pero evitando la sobrecarga de pasar una copia de dicho objeto extenso.

La referencia también puede ser utilizada como un seudónimo para otras variables dentro de una función. Por ejemplo, el código

```
int count = 1;      // declare integer variable count
int &c = count;     // create c as an alias for count

++c;                // increment count (using its alias)
```

incrementa la variable `count` utilizando su seudónimo `c`. Las variables de referencia deben ser inicializadas en sus declaraciones (véase las figuras 15.7 y 15.8), y no pueden ser reasignadas como seudónimos a otras variables. Una vez declarada una referencia como un seudónimo para otra variable, todas las operaciones que supuestamente se ejecuten sobre el seudónimo (es decir, sobre la referencia) de hecho se ejecutan sobre la variable original misma. El seudónimo es sólo

```
// Comparing call by value, call by reference with
// pointers, and call by reference with references
#include <iostream.h>

int squareByValue(int);
void squareByPointer(int *);
void squareByReference(int &);

main()
{
   int x = 2, y = 3, z = 4;

   cout << "x = " << x << " before squareByValue\n"
        << "Value returned by squareByValue: "
        << squareByValue(x)
        << "\nx = " << x << " after squareByValue\n\n";

   cout << "y = " << y << " before squareByPointer\n";
   squareByPointer(&y);
   cout << "y = " << y << " after squareByPointer\n\n";

   cout << "z = " << z << " before squareByReference\n";
   squareByReference(z);
   cout << "z = " << z << " after squareByReference\n";

   return 0;
}

int squareByValue(int a)
{
   return a *= a;   // caller's argument not modified
}

void squareByPointer(int *bPtr)
{
   *bPtr *= *bPtr;  // caller's argument modified
}

void squareByReference(int &cRef)
{
   cRef *= cRef;    // caller's argument modified
}
```

```
x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

y = 3 before squareByPointer
y = 9 after squareByPointer

z = 4 before squareByReference
z = 16 after squareByReference
```

Fig. 15.6 Un ejemplo de llamada por referencia.

otro nombre de la variable original para el seudónimo —no se reserva espacio. Las referencias no pueden ser desreferenciadas mediante el apuntador de operador de indirección (*). Las referencias no pueden ser utilizadas para ejecutar "aritmética de referencia" (como cuando utilizamos apuntadores en aritmética de apuntadores para señalar a otros elementos de un arreglo). Otras operaciones inválidas sobre referencias incluyen señalar a referencias, tomar las direcciones de referencias, y comparar referencias (de hecho cada una de estas operaciones no generará un error de sintaxis; en vez de ello cada una de estas operaciones se efectuará en la variable para la cual la referencia es un seudónimo).

Las funciones pueden regresar apuntadores o referencias, pero esto puede resultar peligroso. Cuando se regresa un apuntador o una referencia a una variable declarada en la función llamada, la variable deberá ser declarada **`static`** dentro de dicha función. De lo contrario, el apuntador o referencia se refiere a una variable automática que al terminar la función se descarta; dicha variable se dice que está "indefinida" y el comportamiento de programa sería impredecible.

Error común de programación 15.7

No inicializar una variable de referencia al declararse.

Error común de programación 15.8

Intentar reasignar una referencia previa declarada como seudónimo a otra variable.

Error común de programación 15.9

Intentar desreferenciar una variable de referencia mediante un operador de indirección de apuntadores (recuerde que una referencia es un seudónimo correspondiente a una variable, y no un apuntador a una variable).

```
// References must be initialized
#include <iostream.h>

main()
{
   int x = 3, &y;   // Error: y must be initialized

   cout << "x = " << x << '\n'
        << "y = " << y << '\n';
   y = 7;
   cout << "x = " << x << '\n'
        << "y = " << y << '\n';

   return 0;
}
```

```
Compiling FIG15_7.CPP:
Error FIG15_7.CPP 6: Reference variable 'y' must be
   initialized
```

Fig. 15.7 Intento de uso de una referencia no inicializada.

```
// References must be initialized
#include <iostream.h>

main()
{
   int x = 3, &y = x;  // y is now an alias for x

   cout << "x = " << x << '\n'
        << "y = " << y << '\n';
   y = 7;
   cout << "x = " << x << '\n'
        << "y = " << y << '\n';

   return 0;
}
```

```
x = 3
y = 3
x = 7
y = 7
```

Fig. 15.8 Cómo utilizar una referencia inicializada.

Error común de programación 15.10

Regresar un apuntador o una referencia a una variable automática en una función llamada.

15.9 El calificador Const

En la sección 15.7 se ilustró la utilización del calificador **const** en la lista de parámetros de una función, para especificar que un argumento pasado a la función no es modificable en dicha función. El calificador **const** también puede ser utilizado para declarar las supuesta llamadas *"variables constantes"* (en vez de declarar constantes simbólicas en el preprocesador mediante **#define**) como en

```
const float PI = 3.14159;
```

Esta declaración genera la "variable constante" **PI** y la inicializa a **3.14159**. La variable al declararse debe ser inicializada con una expresión constante, y después no puede ser modificada (figura 15.9 y figura 15.10). Las "variables constantes" también se conocen como *constantes nombradas* o *variables de sólo lectura*. Note que el término "variable constante" es un oxímoron (contradicción), es decir, una contradicción en términos similares a "camarón jumbo" o "quemadura de congelador". (¡Por favor envíenos sus oximorones favoritos vía nuestra dirección de correo electrónico incluida en el prefacio. Gracias!).

Las variables constantes pueden ser colocadas en cualquier parte en que se espere una expresión constante. Por ejemplo, la siguiente declaración de arreglo utiliza la variable **const** para especificar el tamaño del arreglo:

```
const int arraySize = 100;
int array[arraySize];
```



```
// A const object must be initialized
main()
{
   const int x;  // Error: x must be initialized

   x = 7;        // Error: cannot modify a const variable

   return 0;
}
```

```
Compiling FIG15_9.CPP:
Error FIG15_9.CPP 4: Constant variable 'x' must be
   initialized
Error FIG15_9.CPP 6: Cannot modify a const object
```

Fig. 15.9 Un objeto **const** debe ser inicializado.

```
// Using a properly initialized constant variable
#include <iostream.h>

main()
{
   const int x = 7;  // initialized constant variable

   cout << "The value of constant variable x is: "
        << x << '\n';

   return 0;
}
```

```
The value of constant variable x is: 7
```

Fig. 15.10 Cómo inicializar correctamente y cómo utilizar una variable constante.

Las variables constantes también pueden ser colocadas en archivos de cabecera.

Error común de programación 15.11

*Usar variables **const** en declaraciones de arreglos y colocar variables **const** en archivos de cabecera (que están incluidos en varios archivos de fuente del mismo programa), ambos son ilegales en C pero legales en C++.*

Práctica sana de programación 15.6

*Una ventaja del uso de variables **const** en lugar de constantes simbólicas, es que las variables **const** son visibles con un depurador simbólico; las constantes simbólicas **#define** no lo son.*

Existen otros usos comunes para calificador **const**. Por ejemplo, puede ser declarado un apuntador constante:

```
int *const iPtr = &integer;
```

Esto declara a **iptr** como un apuntador constante a un entero. El valor al cual señala **iptr** puede ser modificado, pero **iptr** no puede ser asignado para señalar a otra posición diferente de memoria.

Un apuntador a un objeto constante puede ser declarado como sigue:

```
const int *iPtr = &integer;
```

Esto declara a **iptr** como un apuntador a una constante entera. El valor al cual **iptr** se refiere no puede ser modificado mediante **iptr**, pero **iptr** puede ser asignado para señalar a otra posición de memoria. Por lo tanto, **iptr** es un apuntador a través del cual se puede leer un valor, pero éste no puede ser modificado (en efecto, un apuntador de "sólo lectura"). El compilador protege los datos referenciados por apuntadores de sólo lectura, evitando que dichos apuntadores sean asignados a apuntadores que no sean de lectura solamente.

Error común de programación 15.12

Inicializar una variable **const** *con una expresión no* **const** *como sería una variable que no ha sido declarada* **const**.

Error común de programación 15.13

Intentar modificar una variable constante.

Error común de programación 15.14

Intentar modificar un apuntador constante.

15.10 Asignación dinámica de memoria mediante new y delete

Los operadores ***new*** y ***delete*** de C++ le permiten a los programas llevar a cabo la asignación dinámica de memoria. En ANSI C, la asignación dinámica de memoria por lo general se lleva a cabo con las funciones estándar de biblioteca **malloc** y **free**. Considere la declaración

```
typeName *ptr;
```

donde **typeName** es cualquier tipo (como **int**, **float**, **char**, etcétera). En ANSI C, el enunciado siguiente asigna en forma dinámica un objeto **typeName**, regresa un apuntador **void** al objeto, y asigna dicho apuntador a **ptr**:

```
ptr = malloc(sizeof(typeName));
```

En C la asignación dinámica de memoria requiere una llamada de función a **malloc** y una referencia explícita al operador **sizeof** (o una mención explícita al número necesario de bytes). La memoria se asigna sobre el *montón* (memoria adicional disponible para el programa en tiempo de ejecución). También, en puestas en práctica anteriores a ANSI C, el apuntador regresado por **malloc** debe ser explícitamente convertido (cast) al tipo apropiado de apuntador con el convertidor explícito (cast) (**typeName***).

En C++, el enunciado

```
ptr = new typeName
```

asigna memoria para un objeto del tipo **typeName** partiendo de la *tienda libre* del programa (término utilizado por C++ para la memoria adicional disponible para el programa en tiempo de ejecución). El operador **new** crea automáticamente un objeto del tamaño apropiado, y regresa un apuntador del tipo apropiado. Si mediante **new** no se puede asignar memoria, se regresa un apuntador nulo (para representar un apuntador nulo los programadores C++ utilizan el valor **0**, en vez de **NULL**).

Para liberar en C++ el espacio para este objeto se utiliza el siguiente enunciado:

```
delete ptr;
```

En C, se invoca la función **free** con el argumento **ptr**, a fin de desasignar memoria. El operador **delete** sólo puede ser utilizado para desasignar memoria ya asignada mediante el operador **new**. Aplicar **delete** a un apuntador previamente desasignado, puede llevar a errores inesperados durante la ejecución del programa. Aplicar **delete** a un apuntador nulo no tiene efecto en la ejecución del programa.

Error común de programación 15.15

Desasignar memoria mediante **delete** *no originalmente asignada mediante el operador* **new**.

C++ permite un *inicializador* para un objeto recién asignado. Por ejemplo,

```
float* thingPtr = new float (3.14159);
```

inicializa a **3.14159** un objeto **float** recién asignado.

También mediante **new** los arreglos pueden ser creados dinámicamente. El siguiente enunciado asigna dinámicamente un arreglo de un subíndice de 100 enteros y asigna el apuntador regresado por **new** al apuntador entero **arrayPtr**:

```
int *arrayPtr;
arrayPtr = new int [100]; // creates array dynamically
```

Para desasignar la memoria asignada dinámicamente para **arrayPtr** por **new**, utilice el enunciado

```
delete [] arrayPtr;
```

En el capítulo 16, analizaremos la asignación dinámica de memoria de objetos **class**. Veremos que los operadores **new** y **delete** ejecutan otras tareas (como llamar de forma automática a las funciones constructor y destructor de una **class**) lo que hace que el uso de **new** y de **delete** sea más poderoso y más apropiado que el uso de **malloc** y de **free**. Por ahora, asegúrese de utilizar corchetes con **delete** al desasignar arreglos de objetos.

Error común de programación 15.16

Usar **delete** *sin corchetes ([y]) al borrar arreglos de objetos dinámicamente asignados (esto resulta un problema sólo en el caso de arreglos que contengan elementos de tipos definidos por el usuario)*

15.11 Argumentos por omisión

Las llamadas de función pudieran por lo común pasar un valor particular de un argumento. El programador puede definir que dicho argumento es un *argumento por omisión*, y el programador puede introducir el valor por omisión de dicho argumento. Cuando en una llamada de función es omitido un argumento por omisión, el valor por omisión de dicho argumento es automáticamente pasado en la llamada.

Los argumentos por omisión deben ser los argumentos que aparecen más a la derecha (los últimos) en una lista de parámetros de función. Al llamar a una función con dos o más argumentos por omisión, si un argumento omitido no es el argumento más a la derecha en la lista de argumentos, todos los argumentos a la derecha de dicho argumento también deben de ser omitidos. Los argumentos por omisión deberán ser especificados junto con la primera ocurrencia del nombre de la función —típicamente en el prototipo en un archivo de cabecera. Los argumentos por omisión también pueden ser utilizados con funciones **inline**.

En la figura 15.11 se demuestra el uso de los argumentos por omisión para calcular el volumen de una caja. A los tres argumentos se les ha dado el valor por omisión de **1**. La primera llamada a la función **inline boxVolume** no especifica ningún argumento y, por lo tanto, utiliza tres valores por omisión. La segunda llamada pasa un argumento **length** y, por lo tanto, utiliza valores por omisión para los argumentos **width** y **height**. La tercera llamada pasa los argumentos para **length y width** y, por lo tanto, utiliza el valor por omisión para el argumento **width**. La última llamada pasa argumentos para **length**, **width** y **height** y, por lo tanto, no utiliza valores por omisión.

Práctica sana de programación 15.7

El uso de argumentos por omisión puede simplificar la escritura de las llamadas de función. Sin embargo, algunos programadores sienten que resulta más claro especificar explícitamente todos los argumentos.

Error común de programación 15.17

Especificar e intentar utilizar un argumento por omisión que no es el argumento más a la derecha (el último) (simultáneamente dar como por omisión todos los argumentos más a la derecha).

15.12 Operador de resolución de alcance unario

Es posible declarar variables locales y globales con un mismo nombre. En C, en tanto esté en alcance la variable local, todas las referencias a dicho nombre de variable pertenecerán a la variable local —dentro del alcance la variable local la variable global no estará visible. C++ dispone del *operador de resolución de alcance unario (::)* para tener acceso a una variable global cuando está en alcance una variable local con el mismo nombre. El operador de resolución de alcance unario no puede ser utilizado para tener acceso a una variable del mismo nombre en un bloque externo. Se puede tener acceso a una variable global directa, sin el operador de resolución de alcance unario, si el nombre de la variable global no es el mismo que el nombre de la variable local en alcance. En el capítulo 16, analizaremos el uso del *operador de resolución de alcance binario* con clases.

En la figura 15.12 se demuestra el operador de resolución de alcance unario con variables locales y globales del mismo nombre. A fin de enfatizar que las versiones local y global de la variable **value** son distintas, el programa declara una de las variables como **float** y la otra como **int.**


```
// Using default arguments
#include <iostream.h>

// Calculate the volume of a box
inline int boxVolume(int length = 1, int width = 1,
                     int height = 1)
   { return length * width * height; }

main()
{
   cout << "The default box volume is: "
        << boxVolume()
        << "\n\nThe volume of a box with length 10,\n"
        << "width 1 and height 1 is: "
        << boxVolume(10)
        << "\n\nThe volume of a box with length 10,\n"
        << "width 5 and height 1 is: "
        << boxVolume(10, 5)
        << "\n\nThe volume of a box with length 10,\n"
        << "width 5 and height 2 is: "
        << boxVolume(10, 5, 2)
        << '\n';

   return 0;
}
```

```
The default box volume is: 1

The volume of a box with length 10,
width 1 and height 1 is: 10

The volume of a box with length 10,
width 5 and height 1 is: 50

The volume of a box with length 10,
width 5 and height 2 is: 100
```

Fig. 15.11 Cómo utilizar los argumentos por omisión.

Error común de programación 15.18

Intentar tener acceso a una variable no global en un bloque externo, utilizando un operador de resolución de alcance unario.

Práctica sana de programación 15.8

Evite usar dentro de un programa variables del mismo nombre para distintos fines. Aunque en varias circunstancias esto es permitido, puede resultar confuso.

15.13 Homonimia de funciones

En C, declarar dos funciones del mismo nombre en el mismo programa es un error de sintaxis. C++ permite que sean definidas varias funciones del mismo nombre, siempre que estos nombres

```
// Using the unary scope resolution operator
#include <iostream.h>

float value = 1.2345;

main()
{
   int value = 7;

   cout << "Local value = " << value
        << "\nGlobal value = " << ::value << '\n';

   return 0;
}
```

```
Local value = 7
Global value = 1.2345
```

Fig. 15.12 Cómo utilizar el operador de resolución de alcance unario.

de funciones indiquen diferentes conjuntos de parámetros (por lo menos en lo que se refiere a sus tipos). Esta capacidad se llama *homonimia de funciones*. Cuando se llama a una función homónima, el compilador C++ selecciona de forma automática la función correcta examinando el número, tipos y orden de los argumentos de la llamada. La homonimia de la función se utiliza por lo común para crear varias funciones del mismo nombre, que ejecutan tareas similares, sobre tipos de datos diferentes.

Práctica sana de programación 15.9

La homonimia de funciones que ejecutan tareas muy relacionadas, puede hacer que los programas sean más legibles y comprensibles.

La figura 15.13 utiliza la función homónima **square** para calcular el cuadrado de un **int** y el cuadrado de un **double**. En el capítulo 17 analizaremos cómo hacer la homonimia de operadores para definir cómo deberán de operar sobre objetos de tipos de datos definidos por el usuario. En la sección 15.15 se presentan las funciones plantilla para la ejecución de tareas idénticas sobre muchos distintos tipos de datos. En el capítulo 20 se analizan las funciones plantilla y las clases plantilla con detalle.

Se pueden distinguir las funciones homónimas mediante su *firma* —una combinación del nombre de la función y de sus tipos de parámetros. El compilador codifica cada identificador de función en forma especial (conocido a veces como *mutilación de nombre o decoración de nombre*), utilizando el número y el tipo de sus parámetros, a fin de habilitar un *enlace a prueba de tipo*. Un enlace a prueba de tipo asegura que se llama a la función homónima correcta, y que los argumentos concuerdan con los parámetros. El compilador detecta y reporta los errores de enlace. El programa de la figura 15.14 fue compilado por el compilador Borland C++. Los nombres de función cifrados, producidos en lenguaje de ensamble por Borland C++, aparecen en la ventana de salida. Cada nombre "decorado" empieza con un @, seguido por el nombre de la función. La lista de parámetros codificados empieza con **$q**. En la lista de parámetros correspondiente a la función **nothing2**, **z** representa a un **char**, **i** representa a un **int**, **pf**


```
// Using overloaded functions
#include <iostream.h>

int square(int x) { return x * x; }

double square(double y) { return y * y; }

main()
{
   cout << "The square of integer 7 is "
        << square(7)
        << "\nThe square of double 7.5 is "
        << square(7.5) << '\n';

   return 0;
}
```

```
The square of integer 7 is 49
The square of double 7.5 is 56.25
```

Fig. 15.13 Cómo utilizar funciones homónimas.

representa a un **float ***, y **pd** representa a un **double ***. En la lista de parámetros correspondiente a la función **nothing1**, **i** representa a un **int**, **f** representa a un **float**, **zc** representa a un **char**, y **pi** representa a un **int***. Las dos funciones **square** se reconocen por medio de sus listas de parámetros; una especifica **d** para **double** y la otra especifica **i** para **int**. Los tipos de regreso de las cuatro funciones no se especifican en los nombres codificados. Note que el codificado de los nombres de funciones es específico a cada compilador. Por lo tanto, una función compilada con Borland C++ pudiera no tener el mismo nombre "decorado" que en otros compiladores.

Las funciones homónimas pueden tener diferentes tipos de regreso, pero deben tener diferentes listas de parámetros.

Error común de programación 15. 19

Generará un error de sintaxis crear funciones homónimas con listas de parámetros idénticas pero distintos tipos de regreso.

Para distinguir entre funciones con un mismo nombre el compilador sólo utiliza las listas de parámetros. Las funciones homónimas no necesariamente tienen el mismo número de parámetros. Al hacer la homonimia de funciones utilizando parámetros por omisión, los programadores deberán tener cuidado, ya que ello podría causar ambigüedad.

Error común de programación 15. 20

Una función con argumentos por omisión omitidos pudiera ser llamada en forma idéntica a otra función homónima; esto resultará en un error de sintaxis.

Error común de programación 15. 21

*La homonimia ayuda a eliminar el uso de las macros **#define** de C, que ejecutan tareas sobre múltiples tipos de datos, y utiliza las características poderosas de verificación de tipo de C++, que al utilizar macros no están disponibles.*

```
// Name mangling
int square(int x) { return x * x; }

double square(double y) { return y * y; }

void nothing1(int a, float b, char c, int *d) {}

char *nothing2(char a, int b, float *c, double *d) { return 0; }

main()
{
   return 0;
}
```

```
public _main
public @nothing2$qzcipfpd
public @nothing1$qifzcpi
public @square$qd
public @square$qi
```

Fig. 15.14 Decoración de nombres para habilitar enlaces a prueba de tipo.

15.14 Especificaciones de enlace

Es posible, desde un programa C++, llamar funciones escritas y compiladas con un compilador C. Como se indicó en la sección 15.13, C++ cifra en especial los nombres de la función para enlaces a prueba de tipo. C, por su parte, no codifica sus nombres de función. Por lo tanto, cuando se haga algún intento para enlazar el código C con código C++, una función compilada en C no será reconocida, porque el código C++ espera un nombre de función especialmente codificado. C++ le permite al programador dar *especificaciones de enlace* para informar al compilador que una función fue compilada en un compilador C, y evitar que el nombre de la función sea codificado por dicho compilador C++. Las especificaciones de enlace son útiles cuando se han desarrollado extensas bibliotecas de funciones especializadas, y el usuario, o no tiene acceso al código fuente para recompilarlas en C++, o no tiene el tiempo para convertir las funciones de bibliotecas de C a C++.

Para informar al compilador que una o varias funciones han sido compiladas en C, escriba los prototipos de función como sigue:

```
extern "C" function prototype    // single function

extern "C"              // multiple functions
{
   function prototypes
}
```

Estas declaraciones le informan al compilador que las funciones especificadas no están compiladas en C++, por lo que no deberá hacerse sobre las funciones enlistadas en la especificación de enlace el codificado de nombres. Estas funciones entonces podrán ser correctamente enlazadas

con el programa. Los entornos C++ incluyen normalmente las bibliotecas estándar de C y para dichas funciones no se requiere que el programador utilice especificaciones de enlace.

15.15 Plantillas de función

Las funciones homónimas se utilizan por lo regular para ejecutar operaciones similares sobre distintos tipos de datos. Si para cada tipo las operaciones son idénticas, esto se puede llevar a cabo de manera más compacta mediante *plantillas de función*, una capacidad introducida en versiones recientes de C++. El programador escribe una definición de plantilla de función. Basado en los tipos de los argumentos proporcionados en las llamadas a esta función, C++ genera de forma automática funciones de código objeto por separado para manejar cada tipo de llamada en forma correcta. En C, esta tarea puede ser llevada a cabo mediante macros creados con **`#define`**. Sin embargo, las macros presentan la posibilidad de serios efectos colaterales y no permiten que el compilador ejecute verificación de tipo. Las plantillas de función proporcionan, como las macros, una solución compacta, pero permiten verificación completa de tipo.

Todas las definiciones de plantillas de función empiezan con la palabra reservada **`template`**, seguida por una lista de parámetros formales a la plantilla de función encerrados en corchetes angulares (< y >). Cada parámetro formal es precedido por la palabra reservada **`class`**. Los parámetros formales se utilizan como tipos incorporados, o como tipos definidos por el usuario, para definir los tipos de los argumentos a la función, para definir el tipo de regreso de la función, y para declarar variables dentro de la función. A continuación se coloca la definición de función y se define como cualquier otra función.

La siguiente plantilla de función también es utilizada en el programa completo de la figura 15.15.

```
template <CLASS T>
void printArray(T *array, const int count)
{
   for (int i = 0; i < count; i++)
      cout << array[i] << " ";
   cout << '\n';
}
```

Esta plantilla de función declara un parámetro formal único **`T`** como el tipo del arreglo a imprimirse mediante la función **`printArray`**. Cuando el compilador detecta la llamada a **`printArray`** en el código fuente del programa, se substituye el tipo del primer argumento de **`printArray`** por la **`T`** en toda la definición de la plantilla, y C++ genera una plantilla de función completa, para la impresión de un arreglo de un tipo de datos especificado. A continuación se compila la nueva función creada. En la figura 15.15 se ejemplifican tres funciones una espera un arreglo **`int`**, otra espera un arreglo **`float`** y otra un arreglo **`char`**. La ejemplificación del tipo **`int`** es:

```
void printArray(int *array, const int count)
{
   for (int i = 0; i  count; i++)
      cout << array[i] << " ";
   cout << '\n';
}
```

Cada parámetro formal en la definición de plantilla debe aparecer en la lista de parámetros de la función por lo menos una vez. El nombre de un parámetro formal sólo puede ser utilizado

una vez en la lista de parámetros formales de la definición de plantilla. Los nombres de los parámetros formales entre funciones de plantilla no es necesario que deban ser únicos.

En la figura 15.15 se ilustra el uso de la función de plantilla **printArray**.

Error común de programación 15.22

No colocar la palabra reservada **class** *antes de cualquier parámetro formal de una plantilla de función.*

Error común de programación 15.23

No usar en la firma de función todos los parámetros formales de una plantilla de función.

```
// Using template functions
#include <iostream.h>

template <class T>
void printArray(T *array, const int count)
{
   for (int i = 0; i < count; i++)
      cout << array[i] << " ";

   cout << '\n';
}

main()
{
   const int aCount = 5, bCount = 7, cCount = 6;
   int a[aCount] = {1, 2, 3, 4, 5};
   float b[bCount] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
   char c[cCount] = "HELLO";  // 6th position for null

   cout << "Array a contains:\n";
   printArray(a, aCount);  // integer version

   cout << "Array b contains:\n";
   printArray(b, bCount);  // float version

   cout << "Array c contains:\n";
   printArray(c, cCount);  // character version

   return 0;
}
```

```
Array a contains:
1 2 3 4 5
Array b contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array c contains:
H E L L O
```

Fig. 15.15 Cómo utilizar funciones plantilla.

Resumen

- C++ es un superconjunto de C, por lo que los programadores pueden utilizar un compilador C++ para compilar sus programas existentes en C, y de ahí modificar de forma gradual estos programas a C++.
- C requiere que un comentario quede delimitado por **/*** y ***/**. C++ permite que un comentario empiece con **//** y que termine al final de la línea.
- C++ permite que las declaraciones aparezcan en cualquier parte en que un enunciado ejecutable pueda aparecer. Las declaraciones variables pueden también aparecer en la sección de inicialización de una estructura **for**.
- C++ proporciona una alternativa a las funciones **printf** y **scanf** para manejar la entrada/salida de los tipos de datos estándar y de las cadenas. El flujo de salida **cout**, y el operador **<<** (operador de inserción de flujo, que se lee "colocar en") le permite la salida de los datos. El flujo de entradas **cin** y el operador **>>** (el operador de extracción de flujo, que se lee "obtener de") permite que se introduzcan datos.
- C++ le permite al programador que cree tipos de datos definidos por el usuario, mediante las palabras reservadas **enum**, **struct**, **union** y **class**. El nombre de etiqueta de la enumeración, estructura, unión o clase puede entonces ser utilizada para definir variables del nuevo tipo.
- Los programas no se compilan a menos de que un prototipo de función esté incluida para cada función o una función esté definida antes de su uso (en cuyo caso no es requerida un prototipo de función por separado).
- Una función que no regrese un valor se declara con el tipo de regreso **void**. Si se hace algún intento ya sea de que la función regrese un valor o de utilizar en la función llamadora el resultado de la invocación de la función, el compilador generará un error.
- En C++, una lista de parámetros vacía se especifica con paréntesis vacíos, o con **void** entre paréntesis. En C, una lista de parámetros vacía desactiva la verificación de argumentos para la función.
- Las funciones en línea eliminan la sobrecarga de llamadas de función. El programador utiliza la palabra reservada **inline** para avisarle al compilador que genere código de función en línea (siempre que sea posible) a fin de minimizar llamadas de función. El compilador puede decidir ignorar el consejo **inline**.
- C++ ofrece una forma directa de llamada por referencia mediante el uso de parámetros de referencia. Para indicar que un parámetro de función es pasado por referencia, haga seguir al tipo de parámetro en el prototipo de función por un **&**. En la llamada de función, mencione la variable por nombre y será pasada en llamada por referencia. En la función llamada, el mencionar la variable por su nombre local, de hecho se refiere a la variable original en la función llamadora. Por lo tanto, la variable original puede ser modificada directamente por la función llamada.
- Los parámetros de referencia también pueden ser creados para uso local, como seudónimo de otras variables dentro de una función. Las variables de referencia deben ser inicializadas en sus declaraciones, y no pueden ser reasignadas como seudónimos de otras variables. Una vez declarada una variable de referencia como un seudónimo de otra variable, todas las operaciones supuestamente ejecutadas sobre el seudónimo de hecho se ejecutan sobre la variable.

- El calificador **const** también puede crear "variables constantes". Una variable constante debe ser inicializada al declarar la variable con una expresión constante, y después no puede ser modificada. Las variables constantes a menudo se llaman constantes o variables de sólo lectura. Las variables constantes pueden ser colocadas en cualquier lugar en donde se espere una expresión constante. Otros usos comunes del calificador **const** incluye apuntadores constantes, apuntadores a constantes y referencias constantes.
- Los operadores **new** y **delete** en C++ ofrecen una mejor forma de llevar a cabo la asignación dinámica de memoria que con las llamadas de función **malloc** y **free** en C. El operador **new** toma como operando un tipo, crea automáticamente un objeto del tamaño correcto, y regresa un apuntador del tipo correcto. El operador **delete** toma un apuntador a un objeto asignado con **new** y libera la memoria.
- C++ permite al programador especificar los argumentos por omisión y sus valores por omisión. Si en una llamada a una función se omite un argumento por omisión, se utiliza el valor por omisión de dicho argumento. Los argumentos por omisión deben ser los argumentos más a la derecha (los últimos) en la lista de parámetros de una función. Los argumentos por omisión deberán ser especificados en la primera ocurrencia del nombre de función.
- El *operador de resolución de alcance unario (::)* le permite a un programa tener acceso a una variable global, cuando está en alcance una variable local con el mismo nombre.
- Es posible definir varias funciones con el mismo nombre, pero con distintos tipos de parámetros. Esto se llama homonimia de función. Cuando es llamada una función homónima, el compilador selecciona automáticamente la función correcta, mediante el examen del número y tipo de los argumentos en la llamada.
- Las funciones homónimas pueden tener diferentes valores de regreso, y deben de tener diferentes listas de parámetros. Dos funciones que sólo difieran en el tipo de regreso darán como resultado un error de compilación.
- En algunos casos, es necesario llamar funciones escritas y compiladas con un compilador de C (por ejemplo, las funciones de biblioteca especializadas de una empresa que no hayan sido convertidas a C++ y recompiladas). C++ procesa los nombres de sus funciones en forma distinta a como lo hace C. Por lo tanto cuando se intente enlazar código C con código C++, una función compilada en C no será reconocida. C++ dispone de especificaciones de enlace, para informar al compilador que una función fue compilada sobre un compilador C, y evitar que se procese el nombre de la función como si fuera una función C++.
- Las plantillas de función permiten la creación de funciones que ejecutan las mismas operaciones sobre distintos tipos de datos, pero la plantilla de función se define sólo una vez.

Terminología

sufijo ampersand (**&**)
asm
catch
cin (flujo de entrada)
const
variable constante
cout (flujo de salida)
class
argumentos de función por omisión
operador **delete**
objetos dinámicos
tienda libre
friend
homonimia de funciones
operador "obtener de" (**>>**)
inicializador
función miembro **inline**
<iostream.h>
biblioteca iostream
especificaciones de enlace
constante nombrada
operador **new**
palabra reservada **operator**
homonimia
operador "colocar en" (**<<**)
variable de lectura solamente
parámetro de referencia
tipos de referencia
signatura
comentario en una sola línea (**//**)
operador de extracción de flujo (**>>**)
operador de inserción de flujo (**<<**)
template
función plantilla
this
throw
try
enlace a prueba de tipo
operador de resolución de alcance unario (**::**)
virtual

Errores comunes de programación

15.1 Olvidar cerrar un comentario de estilo C mediante ***/**.
15.2 Declarar una variable después de que haya sido referenciada en un enunciado.
15.3 Un intento de regresar un valor de una función **void** o de utilizar el resultado de una llamada a una función **void**.
15.4 Los programas en C++ no se compilarán a menos de que sean proporcionadas los prototipos de función para cada una de las funciones, o que cada función quede definida antes de ser utilizada.
15.5 C++ es un lenguaje en evolución y algunas de sus características pudieran no estar disponibles en su computadora. Usar características no puestas en práctica causará errores de sintaxis.
15.6 Dado que en el cuerpo de la función llamada los parámetros de referencia se mencionan sólo por nombre, el programador pudiera pasar por inadvertido y tratar a los parámetros de referencia como parámetros en llamada por valor. Esto puede causar efectos colaterales inesperados, si las copias originales de las variables son modificadas por la función llamadora.
15.7 No inicializar una variable de referencia al declararse.
15.8 Intentar reasignar una referencia ya declarada como seudónimo a otra variable.
15.9 Intentar desreferenciar una variable de referencia mediante un operador de indirección de apuntadores (recuerde que una referencia es un seudónimo correspondiente a una variable, y no un apuntador a una variable).
15.10 Regresar un apuntador o una referencia a una variable automática en una función llamada.
15.11 Usar variables **const** en declaraciones de arreglos y colocar variables **const** en archivos de cabecera (que están incluidos en varios archivos de fuente del mismo programa), ambos son ilegales en C pero legales en C++.
15.12 Inicializar una variable **const** con una expresión no **const** como sería una variable que no ha sido declarada **const**.
15.13 Intentar modificar una variable constante.
15.14 Intentar modificar un apuntador constante.
15.15 Desasignar memoria mediante **delete** no originalmente asignada mediante el operador **new**.
15.16 Usar **delete** sin corchetes (**[** y **]**) al borrar arreglos de objetos dinámicamente asignados (esto resulta un problema sólo en el caso de arreglos que contengan elementos de tipos definidos por el usuario)
15.17 Especificar e intentar utilizar un argumento por omisión que no es el argumento más a la derecha (el último) (simultáneamente dar como por omisión todos los argumentos más a la derecha).
15.18 Intentar tener acceso a una variable no global en un bloque externo, utilizando un operador de resolución de alcance unario.
15.19 Generará un error de sintaxis crear funciones homónimas con listas de parámetros idénticas pero distintos tipos de regreso.
15.20 Una función con argumentos por omisión omitidos pudiera ser llamada en forma idéntica a otra función homónima; esto resultará en un error de sintaxis.

15.21 La homonimia ayuda a eliminar el uso de las macros `#define` de C, que ejecutan tareas sobre múltiples tipos de datos, y utiliza las características poderosas de verificación de tipo de C++, que al utilizar macros no están disponibles.

15.22 No colocar la palabra reservada `class` antes de cualquier parámetro formal de una plantilla de función.

15.23 No usar en la firma de función todos los parámetros formales de una plantilla de función.

Prácticas sanas de programación

15.1 Usar los comentarios `//` de estilo C++, evita errores de comentarios sin terminar, que ocurren al olvidarse de cerrar los comentarios de estilo C con `*/`.

15.2 Utilizar entrada/salida orientada a flujo de tipo C++ hace los programas más legibles (y menos sujetos a errores), que sus contrapartidas escritas en C mediante las llamadas de función `printf` y `scanf`.

15.3 Colocar la declaración de una variable cerca de su primer uso puede hacer los programas más legibles.

15.4 El calificador `inline` deberá ser utilizado únicamente tratándose de funciones pequeñas, de uso frecuente.

15.5 Utilice apuntadores para pasar argumentos que pudieran ser modificados por la función llamada, y utilice referencias a constantes para pasar argumentos extensos, que no serán modificados.

15.6 Una ventaja del uso de variables `const` en lugar de constantes simbólicas, es que las variables `const` son visibles con un depurador simbólico; las constantes simbólicas `#define` no lo son.

15.7 El uso de argumentos por omisión puede simplificar la escritura de las llamadas de función. Sin embargo, algunos programadores sienten que resulta más claro especificar de manera explícita todos los argumentos.

15.8 Evite usar dentro de un programa variables del mismo nombre para distintos fines. Aunque en varias circunstancias esto es permitido, puede resultar confuso.

15.9 La homonimia de funciones que ejecutan tareas muy relacionadas, puede hacer que los programas sean más legibles y comprensibles.

Sugerencias de rendimiento

15.1 Usar funciones `inline` puede reducir tiempo de ejecución, pero puede aumentar el tamaño del programa.

15.2 En el caso de objetos extensos, utilice un parámetro de referencia `const` para simular la apariencia y la seguridad de una llamada por valor, pero evitando la sobrecarga de pasar una copia de dicho objeto extenso.

Sugerencia de portabilidad

15.1 El significado de una lista de función de parámetros vacía es dramáticamente distinto en C++ que en C. En C, significa que se deshabilita toda verificación de argumentos. En C++, significa que la función no toma argumentos. Por lo tanto, si se compilan en C++, los programas en C que utilizan esta característica pudieran ejecutarse en forma diferente.

Observación de ingeniería de software

15.1 Cualquier modificación a una función `inline` requiere que sean recompilados todos los clientes de dicha función. Esto pudiera resultar de importancia en algunas situaciones de desarrollo y mantenimiento de programas.

Ejercicios de autoevaluación

15.1 Llene cada uno de los siguientes espacios vacíos :

a) En C++, es posible tener varias funciones con el mismo nombre, cada una de ellas operando sobre diferentes tipos o números de argumentos. Esto se llama _____ de funciones.

b) La _____ permite el acceso a una global variable con el mismo nombre que una variable en el alcance actual.

c) El operador _____ asigna dinámicamente un nuevo objeto.

d) En C, suponga que `a` y `b` son variables enteras y que formamos la suma `a + b`. Ahora suponga que `c` y `d` son variables de punto flotante y que formamos la suma `c + d`. Los dos operadores `+` se están utilizando aquí para fines claramente distintos. Esto es un ejemplo de una propiedad que tiene C++, y que también tiene C, llamada _____ de función.

e) Dos objetos de flujo que hemos analizado y que están predefinidos en C++ son _____ y _____.

f) Las palabras reservadas _____, _____, _____ y _____ son utilizadas para crear nuevos tipos de datos en C++.

g) El calificador _____ sólo se utiliza para declarar variables de lectura.

h) C++ ofrece _____ para permitir a las funciones compiladas en un compilador de C que se enlacen correctamente con un programa de C++.

i) Las funciones _____ habilitan a un sola función para que pueda ser definida para ejecutar una tarea sobre muchos tipos distintos de datos.

15.2 (Verdadero/falso). Al escribir un comentario en forma de bloque que requiera de muchas líneas de texto, es más conciso el utilizar el delimitador de comentarios de C++ `//` que los delimitadores normales de C `/*` y `*/`.

15.3 En C++ , ¿por qué debería un prototipo de función contener una declaración de tipo de parámetro como `float&`?

15.4 (Verdadero/falso). Todas las llamadas en C++ se efectúan en llamada por valor.

15.5 Explique por qué en C++ los operadores `<<` y `>>` son de homonimia.

15.6 Escriba segmentos de programa en C++ para que lleven a cabo cada uno de los siguientes:

a) Escriba la cadena `"Welcome to C!"` al flujo de salida estándar `cout`.

b) Lea el valor de la variable `age` a partir del flujo de entrada estándar `cin`.

15.7 Escriba un programa en C++ completo que utilice entrada/salida de flujo y una función `inline` llamada `sphereVolume` para solicitarle al usuario el radio de una esfera, y calcular e imprimir el volumen de dicha esfera, utilizando la asignación `volume = (4/3) PI * pow(radius, 3)`.

15.8 ¿Qué pasaría en C si declarase la misma función dos veces con distintos tipos de argumentos? ¿Qué pasaría en C++?

Respuestas a los ejercicios de autoevaluación

15.1 a) homonimia. b) operador de resolución de alcance unario (`::`). c) `new`. d) homonimia. e) `cin`, `cout`. f) `enum`, `struct`, `union`, `class`. g) `const`. h) especificaciones de enlace. i) plantilla.

15.2 Falso. Los delimitadores de tipo C normales, necesita cada uno sólo ser escrito una vez, en tanto que el delimitador C++ `//` necesita aparecer al principio de cada nueva línea.

15.3 Dado que el programador está declarando un parámetro de referencia del tipo "referencia a" `float` para tener acceso mediante llamada por referencia a la variable de argumento original.

15.4 Falso. C++ permite llamada por referencia directa mediante el uso de parámetros de referencia en adición al uso de apuntadores.

15.5 El operador >> es a la vez el operador de desplazamiento a la derecha y el operador de extracción de flujo, dependiendo de dónde está utilizado. El operador << es a la vez el operador de desplazamiento de la izquierda y el operador de inserción de flujo dependiendo de dónde se utilice.

15.6 a) `cout << "Welcome to C!";`
b) `cin >> age;`

15.7
```
// Inline function that calculates the volume of a sphere
#include <iostream.h>

const float PI = 3.24159;

inline float sphereVolume(const float r) {return
                          4.0 / 3.0 * PI * r * r * r;)
main()
{
   float radius;

   cout << "Enter the length of the radius of your sphere: ";
   cin >> radius;
   cout << "Volume of sphere with radius " << radius <<
      "is " << sphereVolume(radius) << '\n';
   return 0;
}
```

15.8 En C obtendría un error de compilación indicando que las dos funciones tienen el mismo nombre. En C++ esto es un ejemplo de homonimia de funciones lo que está permitido, y no habría error.

Ejercicios

15.9 Suponga una organización que actualmente enfatiza la programación en C, y desea convertir a un entorno de programación orientado a objetos en C++. ¿Cuál sería la estrategia apropiada para pasar de entornos de C a entornos de C++?

15.10 Escriba enunciados orientados a flujo de tipo C++ para llevar a cabo cada uno de los siguientes:
a) Mostrar **"HELLO"** en la pantalla.
b) Introducir un valor para la variable `float` llamada **temperature** desde el teclado.

15.11 Compare la entrada/salida de tipo **printf/scanf** con la entrada/salida orientada a flujos de tipo C++.

15.12 En este capítulo, mencionamos que existen muchas áreas en la cual C++ "sabe" automáticamente qué tipos utilizar. Enliste tantas de éstas como usted pueda.

15.13 Escriba un programa C++ que utilice entradas/salidas orientadas a flujos, que solicite siete enteros y determine e imprima su máximo.

15.14 Escriba un programa C++ completo que utilice entrada/salida de flujos y una función `inline` de nombre **circleArea**, para solicitarle al usuario el radio de un círculo, y que calcule e imprima el área de dicho círculo.

15.15 Escriba un programa C++ completo con las tres funciones alternas especificadas más abajo, que cada una de ellas sólo añada 1 a la variable **count** definida en **main**. A continuación compare los tres métodos alternos. Las tres funciones son
a) Función **add1CallByValue**, que pasa una copia de **count** en llamada por valor, añade 1 a la copia y regresa el nuevo valor.
b) Función **add1ByPointer**, que pasa el acceso a la variable **count** vía una llamada por referencia simulada mediante apuntadores, y utiliza el operador de desreferenciación * para añadir uno a la copia original de **count** en **main**.

c) La función **add1ByReference**, que pasa **count** con una llamada por referencia verdadera vía un parámetro de referencia, y añade 1 a la copia original de **count** mediante su seudónimo (es decir el parámetro de referencia).

15.16 ¿Cuál es el propósito del operador de resolución de alcance unario?

15.17 Compare la asignación dinámica de memoria mediante los operadores de C++ **new** y **delete**, con asignación de memoria dinámica mediante las funciones de la biblioteca estándar de C **malloc** y **free**.

15.18 Enliste las varias características de C++ que fueron presentadas en este capítulo, y que representan el conjunto de mejorías no orientadas a objetos a C.

15.19 Escriba un programa que utilice una plantilla de función llamada **min**, para determinar el menor de dos argumentos. Pruebe el programa utilizando pares de enteros de caracteres y de números de punto flotante.

15.20 Escriba un programa que utilice una plantilla de función llamada **max**, para determinar el mayor de tres argumentos. Pruebe el programa utilizando pares de enteros, caracteres y números de punto flotante.

15.21 Determine si los siguientes segmentos de programa contienen errores. Para cada uno de los errores, explique cómo deben ser corregidos. Nota: para un segmento particular del programa, es posible que dentro de dicho segmento no existan errores.

a)
```
main()
{
   cout << x;
   int x = 7;
   return 0;
}
```
b)
```
template <class A>
int sum(int num1, int num2, int num3)
{
   return num1 + num2 + num3;
}
```
c)
```
/* This is a comment
```
d)
```
void printResults(int x, int y)
{
   cout << "The sum is " << x + y << '\n';
   return x + y;
}
```
e)
```
char *s = "character string";
delete s;
```
f)
```
float &ref;
*ref = 7;
```
g)
```
int x;
const int y = x;
```
h)
```
template <A>
A product(A num1, A num2, A num3)
{
   return num1 * num2 * num3;
}
```
i)
```
const double pi = 3.14159;
pi = 3;
```
j)
```
// This is a comment
```
k)
```
char *s = new char[10];
delete s;
```
l)
```
double cube(int);
int cube(int);
```

16

Clases y abstracción de datos

Objetivos

- Comprender los conceptos de ingeniería de software correspondientes a encapsulado y ocultamiento de datos.
- Comprender las nociones de abstracción de datos y de tipos de datos abstractos (ADT).
- Ser capaz de crear ADT, es decir clases, en C++.
- Comprender como crear, utilizar y destruir objetos de clase.
- Ser capaz de controlar el acceso a miembros de objetos de datos y a funciones miembro.
- Empezar a valorizar las ventajas de la orientación a objetos.

Mi objeto todo sublime, terminaré a tiempo.
W. S. Gilbert

¿Es éste un mundo para ocultar virtudes?
William Shakespeare
Décimo segunda noche

Tienes bien merecidos a tus sirvientes públicos.
Adlai Stevenson

Caras íntimas en lugares públicos
son más sabias y más agradables
Que caras públicas en lugares íntimos.
W. H. Auden

Sinopsis

16.1 Introducción

Ahora empezamos nuestra introducción a la orientación a objetos. Veremos que la orientación a objetos es una forma natural de pensar en relación con el mundo y de escribir programas de computación. ¿Por qué, entonces, desde la primera página de este libro no empezamos con la orientación a objetos? ¿Por qué estamos posponiendo hasta el capítulo 16 la programación orientada a objetos en C++? La respuesta es que los objetos que construiremos estarán compuestos en parte por piezas de programas estructurados, por lo que necesitábamos establecer primero una base de programación estructurada.

En los primeros capítulos, nos concentramos en la metodología "convencional" de la programación estructurada. En este capítulo, presentamos conceptos básicos (es decir, "objeto pensamiento") y terminología (es decir " objeto lenguaje ") de la programación orientada a objetos. En capítulos subsecuentes, atacaremos temas más sustanciales y problemas retadores utilizando las técnicas del *diseño orientado a objetos* (*OOD, por object-oriented design*), analizamos enunciados de problemas típicos que requieren la construcción de sistemas, determinamos qué

objetos necesitamos para poner en práctica los sistemas, determinamos qué atributos tendrán que tener los objetos, determinamos qué comportamientos necesitarán mostrar dichos objetos y especificamos cómo deberán interactuar los objetos uno con el otro para cumplir con las metas generales del sistema.

Empezamos presentando parte de la terminología clave de la orientación a objetos. Mire a su alrededor en el mundo real. Por todas partes donde mire los verá *¡objetos!* Personas, animales, plantas, automóviles, aviones, edificios, cortadoras de pastos, computadoras y demás. Los seres humanos pensamos en términos de objetos. Tenemos la capacidad maravillosa de la *abstracción,* que nos permite ver una imagen en pantalla como personas, aviones, árboles y montañas, en vez de puntos individuales de color. Podemos, si así lo deseamos, pensar en términos de playas, en vez de en granos de arena, bosques en lugar de árboles, y casas en lugar de tabiques.

Pudiéramos estar inclinados a dividir los objetos en dos categorías: objetos animados y objetos inanimados. De alguna forma, los objetos animados están "vivos". Se mueven y hacen cosas. Los objetos inanimados, como las toallas, no parecen hacer mucho. Simplemente "están ahí". Todos estos objetos, sin embargo, sí tienen algunas cosas en común. Todos tienen *atributos,* como tamaño, forma, color, peso y demás. Todos ellos exhiben varios *comportamientos*, por ejemplo, un balón rueda, rebota, se infla y se desinfla; un bebé llora, duerme, gatea, camina y parpadea; un automóvil se puede acelerar, frenar, girar, etcétera.

Los seres humanos aprenden lo relacionado con los objetos estudiando sus atributos y observando su comportamiento. Objetos diferentes pueden tener muchos atributos iguales y mostrar comportamientos similares. Se pueden hacer comparaciones, por ejemplo, entre bebés y adultos, entre seres humanos y chimpancés. Automóviles, camiones, pequeños carros rojos y patines tienen mucho en común.

La programación orientada a objetos (OOP) hace modelos de los objetos del mundo real mediante sus contrapartes en software. Aprovecha las relaciones de clase, donde objetos de una cierta clase, como la clase de vehículos, tienen las mismas características. Aprovecha las relaciones de *herencia*, e inclusive las relaciones de *herencia múltiple,* donde clases recién creadas de objetos se derivan heredando características de clases existentes, pero poseyendo características únicas, propias de ellos mismos. Los bebés tienen muchas características de sus padres, pero ocasionalmente padres de baja estatura tienen hijos altos.

La programación orientada a objetos nos proporciona una forma más natural e intuitiva de observar el proceso de programación, es decir *haciendo modelos* de objetos del mundo real, de sus atributos y de sus comportamientos. OOP también hace modelos de la comunicación entre los objetos. De la misma forma que las personas se envían *mensajes* uno al otro (por ejemplo, un sargento al comando de tropas para ponerlas en posición de firmes), los objetos también se comunican mediante mensajes.

OOP *encapsula* datos (atributos) y funciones (comportamiento) en paquetes llamados *objetos*; los datos y las funciones de un objeto están muy unidos. Los objetos tienen la propiedad de *ocultar la información*. Esto significa que aunque los objetos puedan saber cómo comunicarse unos con otros mediante *interfaces* bien definidas, a los objetos por lo regular no se les está permitido saber cómo funcionan otros objetos los detalles de puesta en práctica quedan ocultos dentro de los objetos mismos. Seguramente es posible conducir un automóvil eficaz sin saber los detalles de cómo es el funcionamiento interno de motores, transmisiones y sistemas de escape. Veremos por qué este ocultamiento de la información es tan crucial para una buena ingeniería de software.

En C y en otros *lenguajes de programación procedurales*, la programación tiende a ser *orientada a la acción*, en tanto que en la programación C++ tiende a ser *orientada al objeto*. En

C, la unidad de programación es la *función*. En C++, la unidad de programación es la *clase* a partir de la cual eventualmente los objetos son *producidos* (es decir son creados).

Los programadores de C se concentran en escribir funciones. Grupos de acciones que ejecutan alguna tarea común se agrupan en funciones, y a su vez las funciones se agrupan para formar programas. Es cierto que en C los datos son importantes, pero la óptica es que los datos existen de forma primordial para soportar las acciones que las funciones ejecutan. Los *verbos* en una especificación de sistema ayudan al programador de C a determinar el conjunto de funciones que juntas funcionarán para poner en práctica el sistema.

Los programadores de C++ se concentran en crear sus propios *tipos definidos por el usuario* conocidos como *clases*. Cada clase contiene datos junto con un conjunto de funciones que manipula dichos datos. Los componentes de datos de una clase se llaman *miembros de datos*. Los componentes de función de una clase se llaman *funciones miembros*. Al igual que un ejemplo de un tipo incorporado como **int** se conoce como una *variable*, un ejemplo de un tipo definido por usuario (es decir, una clase) se conoce como un *objeto*. El foco de atención en C++ está sobre los objetos, en vez de sobre las funciones. Los *nombres* en una especificación de sistema ayudan al programador de C++ a determinar el conjunto de clases, a partir de las cuales serán creados los objetos que funcionarán conjuntamente para poner en práctica el sistema.

Las clases en C++ son un desarrollo o evolución natural de la idea de **struct** de C. Antes de continuar con los detalles del desarrollo de las clases en C++, repasaremos las estructuras y construiremos un tipo definido por el usuario basado en una estructura. La debilidad que expondremos mediante este enfoque, nos ayudará a racionalizar y motivar el concepto de clase.

16.2 Definiciones de estructura

Considere la siguiente definición de estructura:

```
struct Time {
   int hour;      // 0-23
   int minute;    // 0-59
   int second;    // 0-59
};
```

Demos un repaso a la terminología de las estructuras (vea los capítulos 10 y 12). La palabra reservada **struct** presenta la definición de estructura. El identificador **Time** es la *etiqueta de estructura*. La etiqueta de estructura le da nombre a la definición de la estructura, y se utiliza para declarar variables del tipo estructura. En este ejemplo, el nombre del tipo estructura es **Time** (a diferencia del nombre de tipo más largo **struct Time**, que se requiere en C). Los nombres declarados en las llaves de la definición de estructura, son los *miembros* de la estructura. Los miembros de una misma estructura deben tener nombres únicos, pero dos estructuras distintas pueden contener sin conflicto miembros con un mismo nombre. Cada definición de estructura debe terminar con un punto y coma. Como veremos pronto, la explicación anterior es también válida para las clases.

La definición de **Time** contiene tres miembros del tipo **int**: **hour**, **minute**, y **second**. Los miembros de estructura pueden ser de cualquier tipo. Una estructura no puede, sin embargo, contener una ocurrencia de sí misma. Por ejemplo, en la definición de estructura correspondiente a **Time** no puede ser declarado un miembro del tipo **Time** Sin embargo, podría ser incluido un apuntador a una estructura **Time**. Una estructura que contenga un miembro, que es un apuntador al mismo tipo de estructura, se conoce como una *estructura autorreferenciada*. Las estructuras autorreferenciadas son útiles para formar estructuras de datos enlazados (vea el capítulo 12).

La definición anterior de estructura no reserva ningún espacio en memoria; más bien, la definición crea un nuevo tipo de datos, que es utilizado para declarar variables. Las variables de estructura se declaran, al igual que las variables de otros tipos. La declaración

```
Time timeObject, timeArray[10], *timePtr;
```

declara **TimeObject** como una variable del tipo **Time**, **TimeArray**, como un arreglo de 10 elementos del tipo **Time**, y **TimePtr** como un apuntador a un objeto **Time**.

16.3 Cómo tener acceso a miembros de estructuras

Se tiene acceso a miembros de una estructura (o de una clase) utilizando los *operadores de acceso a miembros* el *operador punto* (**.**) y el *operador flecha* (**->**). El operador punto da acceso a un miembro de estructura (o de clase) vía el nombre de variable del objeto o vía una referencia al objeto. Por ejemplo, para imprimir el miembro **hour** de la estructura **timeObject**, utilice el enunciado

```
cout << timeObject.hour;
```

El operador de flecha que consiste de un signo (**-**) y de un signo mayor que (**>**) sin espacios intermedios da acceso a un miembro de estructura (o de clase) vía un apuntador al objeto. Suponga que el apuntador **timePtr** ha sido declarado para señalar a un objeto **Time**, y que la dirección de la estructura **timeObject** ha sido asignada a **timePtr**. Para imprimir el miembro **hour** de la estructura **timeObject** mediante el apuntador **timePtr**, utilice el enunciado

```
cout << timePtr->hour;
```

La expresión **timePtr->hour** es equivalente a **(*timePtr).hour**, que desreferencia el apuntador y da acceso al miembro **hour** mediante el uso del operador de punto. Aquí se necesitan los paréntesis, porque el operador de punto (**.**) tiene una precedencia más alta que el operador de desreferenciación de apuntadores (*). El operador de flecha y el operador de miembro de estructura, junto con los paréntesis y los corchetes (**[]**) tienen la precedencia de operadores más alta (de todos los operadores hasta ahora analizados) y se asocian de izquierda a derecha.

16.4 Cómo poner en práctica mediante un struct un tipo Time definido por el usuario

En la figura 16.1 se crea un tipo de estructura definida por el usuario **Time** con tres miembros enteros: **hour**, **minute** y **second**. El programa define una estructura **Time** llamada **dinnerTime**, y utiliza el operador de punto para inicializar los miembros de la estructura con los valores **18** para **hour**, **30** para **minute** y **0** para **second**. A continuación, el programa imprime la hora en formato militar y en formato estándar. Note que las funciones de impresión reciben referencias a las estructuras constantes **Time**. Esto hace que las estructuras **Time** sean pasadas por referencia a las funciones de impresión eliminando por lo tanto, la sobrecarga de copia asociada al pasar por valor estructuras a las funciones y también evita que las funciones de impresión modifiquen la estructura **Time**. En el capítulo 17 analizaremos los objetos **const** y las funciones miembro **const**.

Sugerencia de rendimiento 16.1

Las estructuras por lo regular pasan en llamada por valor. Para evitar la sobrecarga de copiar una estructura, pase la estructura en llamada por referencia.

```
// FIG16_1.CPP
// Create a structure, set its members, and print it.
#include <iostream.h>

struct Time {     // structure definition
   int hour;      // 0-23
   int minute;    // 0-59
   int second;    // 0-59
};

void printMilitary(const Time &);  // prototype
void printStandard(const Time &);  // prototype

main()
{
   Time dinnerTime;     // variable of new type Time

   // set members to valid values
   dinnerTime.hour = 18;
   dinnerTime.minute = 30;
   dinnerTime.second = 0;

   cout << "Dinner will be held at ";
   printMilitary(dinnerTime);
   cout << " military time,\nwhich is ";
   printStandard(dinnerTime);
   cout << " standard time." << endl;

   // set members to invalid values
   dinnerTime.hour = 29;
   dinnerTime.minute = 73;
   dinnerTime.second = 103;

   cout << "\nTime with invalid values: ";
   printMilitary(dinnerTime);
   cout << endl;
   return 0;
}

// Print the time in military format
void printMilitary(const Time &t)
{
   cout << (t.hour < 10 ? "0" : "") << t.hour << ":"
        << (t.minute < 10 ? "0" : "") << t.minute << ":"
        << (t.second < 10 ? "0" : "") << t.second;
}

// Print the time in standard format
void printStandard(const Time &t)
{
   cout << ((t.hour == 0 || t.hour == 12) ? 12 : t.hour % 12)
        << ":" << (t.minute < 10 ? "0" : "") << t.minute
        << ":" << (t.second < 10 ? "0" : "") << t.second
        << (t.hour < 12 ? " AM" : " PM");
}
```

Fig. 16.1 Cómo crear una estructura, definir sus miembros e imprimirla (parte 1 de 2).


```
Dinner will be held at 18:30:00 military time,
which is 6:30:00 PM standard time.

Time with invalid values: 29:73:103
```

Fig. 16.1 Cómo crear una estructura, definir sus miembros e imprimirla (parte 2 de 2).

Sugerencia de rendimiento 16.2

A fin de evitar la sobrecarga de la llamada por valor y aún así obtener el beneficio de que la información original del llamador quede protegida contra modificaciones, pase argumentos de tamaño extenso como referencias `const`.

Existen inconvenientes para la creación de nuevos tipos de datos utilizando estructuras de esta forma. Dado que la inicialización no es específicamente requerida, es posible tener datos sin inicializar, con los problemas consiguientes. Aún si los datos están inicializados, pudieran no estar de forma correcta inicializados. A los miembros de una estructura se les pueden asignar valores inválidos (como lo hicimos en la figura 16.1) porque el programa tiene acceso directo a los datos. El programa asignó valores incorrectos a los tres miembros del objeto **`time dinnerTime`**. Si es modificada la puesta en marcha del **`struct`**, deberán de ser modificados todos los programas que utilizan el **`struct`**. Esto es debido a que el programador está manipulando de manera directa el tipo de datos. No existe "interfaz" con el programa para asegurarse que el programador utiliza correctamente el tipo de datos, y para asegurar que los datos se conservan en un estado consistente.

Observación de ingeniería de software 16.1

Es importante escribir programas que sean comprensibles y fáciles de mantener. Las modificaciones son la regla más que la excepción. Los programadores deberán prever que su código será modificado. Como veremos, las clases facilitan la capacidad de modificación de los programas.

Existen otros problemas asociados con las estructuras de tipo C. No pueden ser impresas como unidad, más bien sus miembros deben ser impresos, y se les debe dar formato uno por uno. Se podría escribir una función para imprimir los miembros de una estructura en algún formato apropiado. En el capítulo 18, "Homonimia de operadores", se ilustra cómo hacer la homonimia del operador << para habilitar los objetos de un tipo de estructura o de clase para que sean impresos con facilidad. Las estructuras no pueden ser comparadas en su totalidad; deben ser comparadas miembro por miembro. En el capítulo 18 también se ilustra cómo hacer la homonimia de los operadores de igualdad y relacionales, a fin de comparar objetos de tipos de estructura y de clase.

En la sección siguiente se vuelve a poner en práctica nuestra estructura **`Time`** como una clase y se demuestran algunas de las ventajas de la creación de tipos de datos abstractos con las clases. Veremos que en C++ las clases y las estructuras pueden ser utilizadas prácticamente en forma idéntica. La diferencia entre ambas es la accesibilidad por omisión asociada con los miembros de cada una. Esto será explicado en breve.

16.5 Cómo poner en práctica un tipo de dato abstracto Time con una clase

Las clases permiten que el programador modele objetos que tienen *atributos* (representados como *miembros de datos*) y *comportamientos* u *operaciones* (representados como *funciones*

miembro). Los tipos contienen miembros de datos y funciones miembro y en C++ son por lo regular definidos mediante la palabra reservada **class**.

Las funciones miembro en otros lenguajes de programación orientados a objetos a veces se llaman *métodos*, y son invocados en respuesta a *mensajes* enviados a un objeto. Un mensaje corresponde a una llamada de función miembro.

Una vez que se haya definido una clase, el nombre de la clase puede ser utilizado para declarar objetos de dicha clase. En la figura 16.2 se muestra una definición simple para la clase **Time**.

Nuestra definición de la clase **Time** empieza con la palabra reservada **class**. El *cuerpo* de la definición de clase se delimita mediante llaves izquierdas y derechas ({ y }). La definición de clase termina con un punto y coma. Nuestra definición de clase **Time** y nuestra definición de estructura **Time** cada una de ellas contiene tres miembros enteros **hour**, **minute** y **second**.

Error común de programación 16.1

Olvidar el punto y coma al final de una definición de clase.

Las partes restantes de la definición de clase son nuevas. La etiqueta **public:** y **private:** se conocen como especificadores de acceso de miembro. Cualquier miembro de datos o función miembro declarado después del especificador de acceso de miembro **public:** (y antes del siguiente especificador de acceso de miembro) es accesible, siempre que el programa tenga acceso a un objeto de la clase **Time**. Cualquier miembro de datos o función miembro declarada después del especificador de acceso de miembro **private:** (y hasta el siguiente especificador de acceso de miembro) sólo es accesible a las funciones miembros de la clase. Los especificadores de acceso de miembro terminan siempre con dos puntos (:) y pueden aparecer varias veces en una definición de clase.

Práctica sana de programación 16.1

Para mayor claridad y legibilidad, utilice cada especificador de acceso de miembro sólo una vez en una definición de clase. Coloque primero los miembros públicos, donde sean fácilmente localizables.

La definición de clase contiene prototipos para las cuatro siguientes funciones miembros después del especificador de acceso de miembros **public: Time, setTime, printMilitary** y **printStandard**. Estas son las *funciones miembro públicas*, o *servicios públicos* o *interfaz de la clase*. Estas funciones serán usadas por los clientes (usuarios) de la clase para manipular los datos de la clase.

```
class Time {
public:
   Time();
   void setTime(int, int, int);
   void printMilitary();
   void printStandard();
private:
   int hour;     // 0 - 23
   int minute;   // 0 - 59
   int second;   // 0 - 59
};
```

Fig. 16.2 Definición simple de la **class Time**.

Una función miembro con el mismo nombre que la clase se llama una función *constructor* de dicha clase. Un constructor es una función miembro especial, que inicializa los miembros de datos de un objeto de clase. Cuando se crea un objeto de una clase se llama a la función constructor de dicha clase.

Los tres miembros enteros aparecen después del especificador de acceso de miembro **private:**. Esto indica que estos miembros de datos de la clase son accesibles sólo a las funciones miembro y como veremos en el siguiente capítulo, a los amigos de la clase. Entonces, los miembros de datos solamente son accesibles por las cuatro funciones cuyos prototipos aparecen en la definición de clase (o por amigos de la clase). Por lo regular los miembros de datos aparecen listados en la porción **private:** de una clase y normalmente las funciones miembro aparecen listados en la porción **public:**. Como veremos más adelante es posible que existan funciones miembro **private:** y datos **public:**.

Una vez definida la clase, puede ser utilizada como un tipo en declaraciones como sigue:

```
Time sunset,                  // object of type Time
     arrayOfTimes[5],         // array of Time objects
     *pointerToTime,          // pointer to a Time object
     &dinnerTime = sunset;    // reference to a Time object
```

El nombre de clase se convierte en un nuevo especificador de tipo. Pueden existir muchos objetos de una clase, igual que pueden existir muchas variables del tipo **int**. El programador puede crear nuevos tipos de clase según requiera. Esta es una de las muchas razones por las que C++ es un *lenguaje extensible*.

En la figura 16.3 se utiliza la clase **Time**. El programa produce un objeto de la clase **Time** llamado **t**. Cuando se produce el objeto, en forma automática es llamado el constructor **Time** e inicializa a **0** cada miembro de dato privado. La hora es entonces impresa en formatos militar y estándar, a fin de confirmar que los miembros han sido inicializados de forma correcta. A continuación, se ajusta la hora utilizando la función miembro **setTime** y se vuelve a imprimir la hora en ambos formatos. A continuación la función miembro **setTime** intenta establecer los miembros de datos a valores inválidos, y otra vez se imprime la hora en ambos formatos.

```
// FIG16_3.CPP
// Time class.
#include <iostream.h>

// Time abstract data type (ADT) definition
class Time {
public:
   Time();                          // default constructor
   void setTime(int, int, int);     // set hour, minute and second
   void printMilitary();            // print military time format
   void printStandard();            // print standard time format

private:
   int hour;     // 0 - 23
   int minute;   // 0 - 59
   int second;   // 0 - 59
};
```

Fig. 16.3 Puesta en práctica de tipos de datos abstractos Time como una clase (parte 1 de 3).

```
// Time constructor initializes each data member to zero.
// Ensures all Time objects start in a consistent state.
Time::Time() { hour = minute = second = 0; }

// Set a new Time value using military time. Perform validity
// checks on the data values. Set invalid values to zero.
void Time::setTime(int h, int m, int s)
{
   hour = (h >= 0 && h < 24) ? h : 0;
   minute = (m >= 0 && m < 60) ? m : 0;
   second = (s >= 0 && s < 60) ? s : 0;
}

// Print Time in military format
void Time::printMilitary()
{
   cout << (hour < 10 ? "0" : "") << hour << ":"
        << (minute < 10 ? "0" : "") << minute << ":"
        << (second < 10 ? "0" : "") << second;
}

// Print time in standard format
void Time::printStandard()
{
   cout << ((hour == 0 || hour == 12) ? 12 : hour % 12)
        << ":" << (minute < 10 ? "0" : "") << minute
        << ":" << (second < 10 ? "0" : "") << second
        << (hour < 12 ? " AM" : " PM");
}

// Driver to test simple class Time
main()
{
   Time t;  // instantiate object t of class Time

   cout << "The initial military time is ";
   t.printMilitary();
   cout << "\nThe initial standard time is ";
   t.printStandard();

   t.setTime(13, 27, 6);
   cout << "\n\nMilitary time after setTime is ";
   t.printMilitary();
   cout << "\nStandard time after setTime is ";
   t.printStandard();

   t.setTime(99, 99, 99);  // attempt invalid settings
   cout << "\n\nAfter attempting invalid settings:\n"
        << "Military time: ";
   t.printMilitary();
   cout << "\nStandard time: ";
   t.printStandard();
   cout << endl;
   return 0;
}
```

Fig. 16.3 Puesta en práctica de tipos de datos abstractos Time como una clase (parte 2 de 3).


```
The initial military time is 00:00:00
The initial standard time is 12:00:00 AM

Military time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Military time: 00:00:00
Standard time: 12:00:00 AM
```

Fig. 16.3 Puesta en práctica de tipos de datos abstractos Time como una clase (parte 3 de 3).

Otra vez, note que los miembros de datos **hour**, **minute** y **second** están antecedidos por la etiqueta **private:**. Los miembros de datos privados de una clase Por lo regular no son accesibles fuera de la misma. (Otra vez, veremos en el capítulo 17, que los amigos de una clase pueden tener acceso a los miembros privados de dicha clase.) La filosofía aquí es que la representación de datos reales utilizados dentro de una clase no debe importarle a los clientes de dicha clase. En este sentido, la puesta en práctica de una clase se dice que está *oculta* a sus clientes. Este *ocultamiento de la información* promueve la capacidad de modificación del programa y simplifica la percepción del cliente de una clase.

Observación de ingeniería de software 16.2

Los clientes de una clase utilizan la clase sin conocer los detalles internos de cómo está puesta en práctica dicha clase. Si se modifica la puesta en práctica de la clase (para mejorar el rendimiento, por ejemplo) no es necesario modificar los clientes de la clase. Esto facilita mucho la modificación de sistemas.

En este programa, el constructor **Time** sólo inicializa a 0 los miembros de datos (es decir, el equivalente en hora militar de 12 AM). Esto asegura que cuando es creado el objeto está en un estado consistente. Los valores inválidos no pueden ser almacenados en los miembros de datos, porque al crearse el objeto **Time** el constructor es automáticamente llamado y todos los intentos subsiguientes para modificar los miembros de datos son analizados por la función **setTime**.

Observación de ingeniería de software 16.3

Las funciones miembro por lo regular están formadas por unas pocas líneas de código, porque ninguna lógica es requerida para determinar si son válidos los miembros de datos.

Note que los miembros de datos de una clase no pueden ser inicializados donde son declarados en el cuerpo de la clase. Estos miembros de datos deberán ser inicializados por el constructor de la clase, o las funciones "set" les podrán asignar valores.

Error común de programación 16.2

Intentar inicializar de forma explícita un miembro de datos de una clase.

La función con el mismo nombre que la clase, pero precedido por un *carácter tilde (~)* se llama el *destructor* de dicha clase. El destructor se ocupa del trabajo de terminación en cada objeto de clase, antes que la memoria correspondiente al objeto sea reclamada por el sistema. En este ejemplo no se incluye un destructor. Analizaremos con mayor detalle constructores y destructores, más adelante en este capítulo, y en el capítulo 17.

Note que las funciones que la clase proporciona al mundo exterior están precedidas por la etiqueta **public:**. Las funciones públicas ponen en práctica las capacidades que la clase proporciona a sus clientes. Las funciones públicas de una clase se conocen como *interfaz* o *interfaz pública* de la clase.

Observación de ingeniería de software 16.4

Los clientes tienen acceso a la interfaz de una clase, pero no deberán tener acceso a la puesta en práctica de la clase.

La definición de clase contiene declaraciones de los miembros de datos de la clase y de las funciones miembro de la clase. Las declaraciones de funciones miembro son los prototipos de función, que fueron analizados en capítulos anteriores. Las funciones miembro pueden ser definidas dentro de una clase, pero es una práctica sana de programación definir las funciones fuera de la definición de clase.

Práctica sana de programación 16.2

Defina todas, a excepción de las más pequeñas funciones miembro, por afuera de la definición de clase. Esto ayuda a separar la interfaz de la clase de su puesta en práctica.

Note el uso del operador de *resolución de alcance binario (::)* en cada definición de función miembro, que en la figura 16.3 sigue a la definición de clase. Una vez definida una clase y sus funciones miembro declaradas, estas funciones deben ser definidas. Cada función miembro de clase puede ser definida en directo al cuerpo de la clase (en vez de incluir el prototipo de función de la clase) o la función puede ser definida después del cuerpo de la clase. Cuando una función miembro se define después de su correspondiente definición de clase, el nombre de función es antecedido por el nombre de clase y por el operador de resolución de alcance binario (**::**). Dado que diferentes clases pueden tener los mismos nombres de miembros, el operador de resolución de alcance "une" el nombre del miembro con el nombre de la clase, para fijar en forma única las funciones miembro de una clase en particular.

Aún cuando una función miembro declarada en una definición de clase pudiera definirse fuera de esa definición de clase, esa función miembro aún queda dentro del *alcance de la clase*, es decir su nombre es conocido sólo por otros miembros de la clase, a menos que se haga referencia a él vía un objeto de la clase, vía una referencia a un objeto de la clase, o vía un apuntador a un objeto de la clase. Diremos en breve más en relación con el alcance de la clase.

Es posible definir las funciones miembro en el cuerpo de la definición de clase. Las funciones que tengan más de una o dos líneas, se definen normalmente por afuera de la definición de clase, esto ayuda a separar la interfaz de clase de la puesta en práctica de la clase. Si se define una función miembro en una definición de clase, la función miembro quedará automáticamente en línea. Las funciones miembro definidas fuera de la definición de clase pudieran ser puestas en línea, mediante el uso explícito de la palabra reservada **inline**. Recuerde que el compilador se reserva el derecho de poner o no en línea cualquier función.

Observación de ingeniería de software 16.5

Declarar funciones miembro dentro de una definición de clase y definir dichas funciones miembro por fuera de dicha definición de clase separa la interfaz de una clase de su puesta en práctica. Esto promueve buena ingeniería de software.

Sugerencia de rendimiento 16.3

Definir una función miembro pequeña dentro de una definición de clase automática coloca la función miembro en línea (si el compilador así lo decide). Esto puede mejorar el rendimiento, pero no promueve una mejor ingeniería de software.

Resulta interesante que las funciones miembro **printMilitary** y **printStandard** no toman argumentos. Esto es debido a que las funciones miembro saben que deben imprimir los miembros de datos del objeto particular **Time** para los cuales han sido invocadas. Esto hace que las llamadas de funciones miembro sean más concisas que las llamadas de función convencionales de la programación procedural.

Observación de ingeniería de software 16.6

Usar un enfoque de programación orientado a objetos a menudo puede simplificar las llamadas de función al reducir el número de parámetros a pasarse. Este beneficio de la programación orientada a objetos se deriva del hecho que el encapsulado de los miembros de datos y las funciones miembros dentro de un objeto le da a las funciones miembro el derecho de acceso a los miembros de datos.

Las clases simplifican la programación porque el cliente (o el usuario del objeto de clase) sólo necesita preocuparse de las operaciones encapsuladas o incrustadas dentro del objeto. Dichas operaciones por lo regular están diseñadas para ser orientadas al cliente, más bien que orientadas a la puesta en práctica. Los clientes no necesitan preocuparse con la puesta en práctica de la clase. De hecho las puestas en práctica cambian, pero con menos frecuencia que las interfaces. Cuando se modifique una puesta en práctica, aquel código que sea dependiente de la puesta en práctica, deberá ser modificado. Mediante la ocultación de la puesta en práctica, eliminamos la posibilidad de que otras partes del programa se hagan dependientes de los detalles de la puesta en práctica de la clase.

A menudo, no es necesario crear clases "desde cero". Más bien, pueden ser *derivadas*, partiendo de otras clases que contienen operaciones que las nuevas clases pueden utilizar. O las clases pueden incluir como miembros objetos de otras clases. Esta *reutilización del software* puede mejorar el rendimiento del programador de forma significativa. Se llama *herencia* derivar nuevas clases partiendo de clases existentes, y en el capítulo 19 se analizará en detalle. Incluir clases como miembros de otras clases se llama *composición*, y se analiza en el capítulo 17.

16.6 Alcance de clase y acceso a miembros de clase.

Los nombres de variables y los nombres de función declarados en una definición de clase, y los nombres de datos y funciones miembro de una clase, pertenecen al *alcance de dicha clase*. Las funciones no miembros se definen en *alcance de archivo*.

Dentro del alcance de clase, los miembros de clase son accesibles de inmediato por todas las funciones miembro de dicha clase y pueden ser referenciados sólo por su nombre. Fuera del alcance de una clase, los miembros de clase se referencían, ya sea a través del nombre del objeto, una referencia a un objeto, o un apuntador a un objeto.

Las funciones miembros de una clase pueden tener homónimas, pero sólo por funciones dentro del alcance de dicha clase. Para hacer la homonimia de una función miembro, sólo incluya en la definición de clase un prototipo para cada versión de la función homónima, y proporcione una definición de función por separado para cada una de las versiones de la función.

Error común de programación 16.3

Intentar hacer la homonimia de una función miembro mediante una función no incluida en el alcance de dicha clase.

Las funciones miembro tienen *alcance de función* dentro de una clase. Si la función miembro define una variable con el mismo nombre que una variable con alcance de clase, la variable con alcance de clase queda oculta por la variable con alcance de función, dentro del alcance de función. Se puede tener acceso a esta variable oculta mediante el operador de resolución de alcance precediendo el operador con el nombre de la clase. Se puede tener acceso a las variables globales ocultas mediante el operador de resolución de alcance unario (vea el capítulo 15).

Los operadores utilizados para tener acceso a los miembros de clase son idénticos a los operadores para tener acceso a los miembros de estructura. El *operador de selección de miembro de punto (.)* se combina con el nombre de una objeto o con una referencia a un objeto para tener acceso a los miembros del objeto. El *operador de selección de miembro de flecha (->)* se combina con un apuntador a un objeto para tener acceso a los miembros de dicho objeto.

El programa de la figura 16.4 utiliza una clase simple llamada **Count** con el miembro de datos público **x** del tipo **int**, y la función miembro pública **print** para ilustrar el acceso a los miembros de una clase mediante los operadores de selección de miembros. El programa produce tres variables del tipo **Count counter, counterRef** (una referencia a un objeto **Count**), y **counterPtr** (un apuntador a un objeto **Count**). La variable **counterRef** se declara para hacer referencia a **counter**, y la variable **counterPtr** se declara para señalar a **counter**. Es importante hacer notar que aquí el miembro de datos **x** se ha hecho público simplemente para demostrar cómo se tiene acceso a los miembros públicos. Como hemos indicado, típicamente los datos se hacen privados como lo haremos en todos los ejemplos subsiguientes.

16.7 Cómo separar la interfaz de la puesta en práctica

Uno de los principios fundamentales de buena ingeniería de software es la separación de la interfaz de la puesta en práctica. Esto facilita la modificación de los programas. Por lo que se refiere a los clientes de una clase, los cambios en la puesta en práctica de una clase no afectan al cliente, siempre y cuando no sea modificada la interfaz de la clase.

Observación de ingeniería de software 16.7

Coloque la declaración de la clase en un archivo de cabecera a incluirse por cualquier cliente que desee utilizar dicha clase. Esto forma la interfaz pública de la clase. Coloque las definiciones de las funciones miembro de la clase en un archivo fuente. Esto conforma la puesta en práctica de la clase.

Observación de ingeniería de software 16.8

Los clientes de una clase no necesitan ver el código fuente de la clase a fin de utilizarla. Los clientes deben, sin embargo, poder tener la capacidad de enlazarse con el código objeto de la clase.

Esto alienta a los fabricantes independientes de software (ISV, por *independent software vendors*) a proporcionar bibliotecas de clase, a la venta o para su licencia. En sus productos los ISV ofrecen sólo los archivos de cabecera y los módulos de objeto. Ninguna información propietaria se da a conocer como sería el caso si se proporcionara código fuente. La comunidad de usuarios de C++ se beneficia al tener disponibles mas bibliotecas de clase, producidas por los ISV.

De hecho, las cosas no son tan color de rosa. Los archivos de cabecera sí contienen alguna porción de la puesta en práctica, así como sugerencias en relación con ella. Las funciones miembro en línea, por ejemplo, necesitan estar en un archivo de cabecera, por lo que cuando el compilador compila a un cliente, el cliente pueda incluir en su lugar la definición de función en línea. Los miembros privados son listados en la declaración de clase dentro del archivo de cabecera, por lo


```
// FIG16_4.CPP
// Demonstrating the class member access operators . and ->
//
// Caution: In future examples we will use private data.
#include <iostream.h>

// Simple class Count
class Count {
public:
   int x;
   void print() { cout << x << '\n'; }
};

main()
{
   Count counter,                  // create counter object
         *counterPtr = &counter,   // pointer to counter
         &counterRef = counter;    // reference to counter

   cout << "Assign 7 to x and print using the object's name: ";
   counter.x = 7;          // assign 7 to data member x
   counter.print();        // call member function print

   cout << "Assign 8 to x and print using a reference: ";
   counterRef.x = 8;       // assign 8 to data member x
   counterRef.print();     // call member function print

   cout << "Assign 10 to x and print using a pointer: ";
   counterPtr->x = 10;     // assign 10 to data member x
   counterPtr->print();    // call member function print
   return 0;
}
```

```
Assign 7 to x and print using the object's name: 7
Assign 8 to x and print using a reference: 8
Assign 10 to x and print using a pointer: 10
```

Fig. 16.4 Cómo tener acceso a los miembros de datos de un objeto y a las funciones miembro a través del nombre del objeto, a través de una referencia o a través de un apuntador al objeto.

que estos miembros quedan visibles a los clientes, aún cuando los clientes no puedan tener acceso a ellos.

Observación de ingeniería de software 16.9

Aquella información de importancia a la interfaz con una clase debería estar incluida en el archivo de cabecera. Aquella información que sólo será utilizada internamente en la clase y que no será necesaria para los clientes de la clase, debería ser incluida en el archivo fuente no publicado. Este es otra vez otro ejemplo del principio del mínimo privilegio.

En la figura 16.5 el programa de la figura 16.3 se divide en varios archivos. Al construir un programa C++, cada definición de clase por lo regular es colocado en un *archivo de cabecera*, y

las definiciones de funciones miembro de dicha clase se colocan en los *archivos de código fuente* con el mismo nombre base. Los archivos de cabecera se incluyen (mediante **#include**) en cada uno de los archivos en los cuales se utiliza la clase, y se compila el archivo fuente de las definiciones de funciones miembro y se enlaza con el archivo que contiene el programa principal. Vea la documentación correspondiente a su compilador, a fin de determinar cómo compilar y enlazar programas formados de varios archivos fuente.

La figura 16.5 está formada del archivo de cabecera **time1.h** en el cual se declara la clase **Time**, el archivo **time1.cpp** en el cual se definen las funciones miembro de la clase **Time**, y del archivo **fig16_5.cpp** en el cual se define la función **main**. La salida para este programa es idéntica a la salida de la figura 16.3.

Note que la declaración de clase se encierra en el siguiente código de preprocesador (vea el capítulo 13):

```
// prevent multiple inclusions of header file
#ifndef TIME1_H
#define TIME1_H
   ...
#endif
```

Cuando elaboremos programas más extensos, se colocarán también otras definiciones y declaraciones en los archivos de cabecera. Las directivas anteriores de preprocesador evitan que se incluya el código entre **#ifndef** y **#endif**, si ya ha sido definido el nombre **TIME1_H**. Si el encabezado no ha sido incluido previamente en un archivo, el nombre **TIME1_H** queda definido mediante la directiva **#define** y los enunciados del archivo de cabecera serán incluidos. Si el encabezado ha sido ya incluido, **TIME1_H** ya está definido y el archivo de cabecera no se vuelve a incluir. Nota: la regla convencional que utilizamos para el nombre de la constante simbólica en las directrices de preprocesador es sólo el nombre del archivo de cabecera, con el carácter de subrayado substituyendo el punto.

Práctica sana de programación 16.3

*Utilice en un programa las directivas de preprocesador **#ifndef**, **#define** y **#endif** para evitar que los archivos de cabecera sean incluidos más de una vez.*

Práctica sana de programación 16.4

*Utilice el nombre del archivo de cabecera, con el punto reemplazado por un subrayado, en las directivas de preprocesador **#ifndef** y **#define** correspondientes a un archivo de cabecera.*

16.8 Cómo controlar el acceso a miembros

Las etiquetas **public:** y **private:** (así como **protected:**, como veremos en el capítulo 19, "Herencia") se utilizan para controlar el acceso a los miembros de datos y a las funciones miembro de una clase. El modo de acceso para las clases es **private:** por omisión, de tal forma que todos los miembros que aparezcan después del encabezado de clase y antes de la primera etiqueta son privados. Después de cada una de las etiquetas, el modo que fue invocado por dicha etiqueta será aplicado hasta la siguiente etiqueta, o hasta la llave derecha de terminación (}) de la definición de clase. Pueden repetirse las etiquetas **public:**, **private:** y **protected:**, pero este tipo de utilización es raro y puede ser confuso.


```
// TIME1.H
// Declaration of the Time class.
// Member functions are defined in TIME1.CPP

// prevent multiple inclusions of header file
#ifndef TIME1_H
#define TIME1_H

// Time abstract data type definition
class Time {
public:
   Time();                          // default constructor
   void setTime(int, int, int);     // set hour, minute and second
   void printMilitary();            // print military time format
   void printStandard();            // print standard time format
private:
   int hour;      // 0 - 23
   int minute;    // 0 - 59
   int second;    // 0 - 59
};

#endif
```

Fig. 16.5 Archivo de cabecera de la clase **Time** (parte 1 de 4).

```
// TIME1.CPP
// Member function definitions for Time class.
#include <iostream.h>
#include "time1.h"

// Time constructor initializes each data member to zero.
// Ensures all Time objects start in a consistent state.
Time::Time() { hour = minute = second = 0; }

// Set a new Time value using military time.
// Perform validity checks on the data values.
// Set invalid values to zero (consistent state).
void Time::setTime(int h, int m, int s)
{
   hour = (h >= 0 && h < 24) ? h : 0;
   minute = (m >= 0 && m < 60) ? m : 0;
   second = (s >= 0 && s < 60) ? s : 0;
}

// Print Time in military format
void Time::printMilitary()
{
   cout << (hour < 10 ? "0" : "") << hour << ":"
        << (minute < 10 ? "0" : "") << minute << ":"
        << (second < 10 ? "0" : "") << second;
}
```

Fig. 16.5 Archivo fuente de las definiciones de funciones miembro de la clase **Time** (parte 2 de 4).

```
// Print time in standard format
void Time::printStandard()
{
   cout << ((hour == 0 || hour == 12) ? 12 : hour % 12)
        << ":" << (minute < 10 ? "0" : "") << minute
        << ":" << (second < 10 ? "0" : "") << second
        << (hour < 12 ? " AM" : " PM");
}
```

Fig. 16.5 Archivo fuente de las definiciones de las funciones miembro de la clase **Time** (parte 3 de 4).

```
// FIG16_5.CPP
// Driver for Time1 class
// NOTE: Compile with TIME1.CPP
#include <iostream.h>
#include "time1.h"

// Driver to test simple class Time
main()
{
   Time t;  // instantiate object t of class time

   cout << "The initial military time is ";
   t.printMilitary();
   cout << "\nThe initial standard time is ";
   t.printStandard();

   t.setTime(13, 27, 6);
   cout << "\n\nMilitary time after setTime is ";
   t.printMilitary();
   cout << "\nStandard time after setTime is ";
   t.printStandard();

   t.setTime(99, 99, 99);  // attempt invalid settings
   cout << "\n\nAfter attempting invalid settings:\n"
        << "Military time: ";
   t.printMilitary();
   cout << "\nStandard time: ";
   t.printStandard();
   cout << endl;
   return 0;
}
```

```
The initial military time is 00:00:00
The initial standard time is 12:00:00 AM

Military time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Military time: 00:00:00
Standard time: 12:00:00 AM
```

Fig. 16.5 Programa manejador de la clase **Time** (parte 4 de 4).

Se puede tener acceso a los miembros de clase privado sólo por miembros (y amigos) de dicha clase. Se puede tener acceso a los miembros públicos de una clase mediante cualquier función del programa.

El fin primordial de los miembros públicos es presentar a los clientes de la clase un panorama de los servicios que la clase proporciona. Este conjunto de servicios forma la *interfaz pública* de la clase. Los clientes de la clase no necesitan preocuparse de cómo la clase ejecuta sus tareas. Los miembros privados de una clase, así como las definiciones de sus funciones miembro públicas, no son accesibles a los clientes de una clase. Estos componentes forman la *puesta en práctica* de la clase.

Observación de ingeniería de software 16.10

C++ alienta a que los programas sean independientes de la puesta en práctica. Cuando se modifica la puesta en práctica de una clase utilizada mediante código independiente de la misma, dicho código no necesita ser cambiado, pero pudiera necesitar ser recopilado.

Error común de programación 16.4

Un intento por parte de una función, que no sea un miembro de una clase particular (o un amigo de dicha clase) de obtener acceso a un miembro privado de dicha clase.

En la figura 16.6, se demuestra que los miembros de clase privados son accesibles a través de la interfaz de clase pública mediante el uso de las funciones miembro públicos. Cuando se compila este programa, el compilador genera dos errores, indicando que no es accesible el miembro privado especificado en cada uno de los enunciados. La figura 16.6 incluye a **time1.h** y se compila con **time1.cpp** partiendo de la figura 16.5.

```
// FIG16_6.CPP
// Demonstrate errors resulting from attempts
// to access private class members.
#include <iostream.h>
#include "time1.h"

main()
{
   Time t;

   // Error: 'Time::hour' is not accessible
   t.hour = 7;

   // Error: 'Time::minute' is not accessible
   cout << "minute = " << t.minute;

   return 0;
}
```

```
Compiling FIG16_6.CPP:
Error FIG16_6.CPP 12: 'Time::hour' is not accessible
Error FIG6_6.CPP 15: 'Time::minute' is not accessible
```

Fig. 16.6 Intento erróneo de acceso a los miembros privados de una clase.

Práctica sana de programación 16.5

Si en una declaración de clase decide listar primero los miembros privados, utilice en forma explícita la etiqueta `private:` *a pesar de que por omisión se supone que* `private:` *es asumido. Esto mejora la claridad del programa. Nuestra preferencia es listar primero los miembros* `public:` *de una clase.*

Práctica sana de programación 16.6

A pesar del hecho que las etiquetas `public:` *y* `private:` *pueden ser repetidas y entremezcladas, liste primero en un grupo todos los miembros públicos de una clase y a continuación liste en otro grupo todos los miembros privados. Esto enfoca la atención del cliente en la interfaz pública de la clase, en vez de en la puesta en práctica de la misma.*

Observación de ingeniería de software 16.11

Conserve privados todos los miembros de datos de una clase. Proporcione funciones miembro públicas para definir los valores de los miembros de datos privados y para obtener los valores de los miembros de datos privados. Esta arquitectura ayuda a ocultar la puesta en práctica de una clase a la vista de sus clientes, lo que reduce errores y mejora la capacidad de modificación del programa.

El cliente de una clase puede ser una función miembro de otra clase, o puede ser una función global.

Para miembros de una clase el acceso por omisión es privado. El acceso a miembros de una clase puede ser definido en forma explícita a protegido o a público. El acceso por omisión correspondiente a miembros `struct` y a miembros de unión es `public`. El acceso a miembros de un `struct` puede ser definido en forma explícita a `public` o a `private` (o a `protected`, como veremos en el capítulo 19). Tratándose de una `unión`, los especificadores de acceso de miembros no pueden ser utilizados en forma explícita.

Observación de ingeniería de software 16.12

Los diseñadores de clase utilizan miembros privados, protegidos y públicos para obligar a la idea de ocultamiento de información y al principio del mínimo privilegio.

Note que los miembros de una clase son privados por omisión, por lo que nunca es necesario utilizar de forma explícita el especificador de acceso de miembro `private:` Muchos programadores prefieren, sin embargo, listar primero la interfaz a una clase (es decir, los miembros públicos de la clase); a continuación se listan los miembros privados, con lo que se requiere el uso explícito del especificador de acceso de miembros `private:` dentro de la definición de clase.

Práctica sana de programación 16.7

Evita confusión usar los especificadores de acceso de miembro `public:`, `protected:`, *y* `private:` *una vez en cada definición de clase.*

El simple hecho que los datos de clase sean `private` no necesariamente significa que los clientes no puedan efectuar modificaciones a dichos datos. Los datos pueden ser modificados por funciones miembro o amigos de dicha clase. Como veremos, dichas funciones deberán ser diseñadas para asegurar la integridad de los datos.

El acceso a los datos privados de una clase puede ser controlado cuidadosamente mediante el uso de las funciones miembro, llamadas *funciones de acceso*. Por ejemplo, para permitir a los clientes leer el valor de datos privados, la clase puede proporcionar una *función "get". Para*

permitir que los clientes modifiquen datos privados, la clase puede proporcionar una *función "set"*. Una modificación como ésta parecería violar el concepto de datos privados. Pero una función miembro *set* puede proporcionar capacidades de validación de datos (como la verificación de rangos) para asegurarse que el valor se define de forma correcta. Una función *get* no necesita exponer los datos en formato "en bruto"; más bien la función *get* puede editar los datos y limitar la parte de dichos datos que el cliente verá.

Observación de ingeniería de software 16.13

Hacer privados los miembros de datos de una clase y públicas las funciones miembro de la clase, facilita la depuración, porque los problemas con las manipulaciones de datos se localizan ya sea en las funciones miembros de la clase o en los amigos de la misma

16.9 Funciones de acceso y funciones de utilería

No todas las funciones miembro necesitan ser públicas para servir como parte de la interfaz de una clase. Algunas funciones miembro se conservan privadas, y sirven como funciones de utilería para otras funciones de la clase.

Observación de ingeniería de software 16.14

Las funciones miembro tienen tendencia a agruparse en varias categorías diferentes: funciones que leen y regresan el valor de miembros de datos privados, funciones que definen el valor de miembros de datos privados, funciones que ponen en práctica las características de la clase, y funciones que ejecutan varias tareas mecánicas para la clase, como la inicialización de objetos de clase, la asignación de objetos de clase, la conversión entre clases y tipos incorporados o entre clases y otras clases, y el manejo de memoria para objetos de clase.

Las funciones de acceso pueden leer o desplegar datos. Otro uso común de las funciones de acceso es probar la veracidad o falsedad de condiciones —dichas funciones a menudo se conocen como *funciones predicadas*. Un ejemplo de una función predicada sería una función `isEmpty` para cualquier clase contenedor, como es una lista enlazada, una pila o una cola. Un programa probaría `isEmpty` antes de intentar leer cualquier otro elemento del objeto contenedor. Una función predicada `isFull` pudiera probar un objeto de clase contenedor para determinar si ya no tiene espacio adicional.

En la figura 16.7 se demuestra el concepto de una *función de utilería*. Una función de utilería no forma parte de la interfaz de una clase. Más bien, es una función miembro privada, que da apoyo a la operación de las funciones miembro públicas de la clase. Las funciones de utilería no están concebidas para ser utilizadas por los clientes de una clase.

La clase `SalesPerson` tiene un arreglo de 12 cifras de ventas mensuales inicializadas a cero por el constructor y definidas mediante la función `setSales` en valores proporcionados por el usuario. La función miembro pública `printAnnualSales` imprime las ventas totales correspondientes a los 12 últimos meses. La función de utilería `total AnnualSales` totaliza las 12 cifras de ventas mensuales, para uso y beneficio de `printAnnualSales`. La función miembro `printAnnualSales` edita las cifras de ventas y las coloca en formato de moneda. En el capítulo 21 se explica con detalle cada una de las capacidades de formato de C++ utilizadas aquí. Por ahora, sólo damos una breve explicación. La llamada

```
setprecision(2)
```

indica que serán impresos dos dígitos de "precisión" a la derecha del punto decimal, exactamente como necesitamos para cantidades en moneda, tales como 23.47. Esta llamada se conoce como

un "manipulador de flujo parametrizado". Un programa que utilice este tipo de llamadas, también deberá contener la directiva de preprocesador

```
#include <iomanip.h>
```

La línea

```
setiosflags(ios::fixed | ios::showpoint);
```

hace que se muestre la cifra de ventas en el formato conocido como de punto fijo (a diferencia de la notación científica). La opción **showpoint** obliga a que aparezca el punto decimal y los ceros después del punto, aún si el número es una cantidad redonda de dólares, como 47.00. En C++, de no haberse definido la opción **showpoint**, un número como éste se imprimiría solo como 47.

16.10 Cómo inicializar objetos de clase: constructores

Después de que los objetos son creados, sus miembros pueden ser inicializados mediante funciones *constructor*. Un constructor es una función miembro de clase con el mismo nombre que la clase. El programador proporciona el constructor el cual entonces, cada vez que se crea un objeto de dicha clase, es invocado automáticamente. *Los miembros de datos de una clase no pueden ser inicializados en la definición de clase.* Más bien, los miembros de datos deben ser inicializados en un constructor de la clase o sus valores definidos más adelante después de que el objeto haya sido creado. Los constructores no pueden especificar tipos de regreso y valores de regreso. Los constructores pueden ser sujetos de homonimia, para permitir una variedad de maneras de inicializar objetos de una clase.

Error común de programación 16.5

Intentar inicializar explícitamente un miembro de datos de una clase dentro de la definición de clase.

```
// SALESP.H
// SalesPerson class definition
// Member functions defined in SALESP.CPP

#ifndef SALESP_H
#define SALESP_H

class SalesPerson {
public:
   SalesPerson();              // constructor
   void setSales();            // user supplies sales figures
   void printAnnualSales();

private:
   double sales[13];           // 12 monthly sales figures
   double totalAnnualSales(); // utility function
};

#endif
```

Fig. 16.7 Cómo utilizar una función de utilería (parte 1 de 3).


```
// SALESP.CPP
// Member functions for class SalesPerson
#include <iostream.h>
#include <iomanip.h>
#include "salesp.h"

// Constructor function initializes array
SalesPerson::SalesPerson()
{
   for (int i = 0; i <= 12; i++)
      sales[i] = 0.0;
}

// Function to set the 12 monthly sales figures
void SalesPerson::setSales()
{
   for (int i = 1; i <= 12; i++) {
      cout << "Enter sales amount for month "
           << i << ": ";
      cin >> sales[i];
   }
}

// Private utility function to total annual sales
double SalesPerson::totalAnnualSales()
{
   double total = 0.0;

   for (int i = 1; i <= 12; i++)
      total += sales[i];

   return total;
}

// Print the total annual sales
void SalesPerson::printAnnualSales()
{
   cout << setprecision(2)
        << setiosflags(ios::fixed | ios::showpoint)
        << "\nThe total annual sales are: $"
        << totalAnnualSales() << endl;
}
```

Fig. 16.7 Cómo utilizar una función de utilería (parte 2 de 3).

Error común de programación 16.6

Intentar declarar un tipo de regreso para un constructor, y/o intentar regresar un valor proveniente de un constructor.

Práctica sana de programación 16.8

Cuando sea apropiado (casi siempre), proporcione un constructor para asegurarse que todo objeto es inicializado correctamente, con valores significativos.

```
// FIG16_7.CPP
// Demonstrating a utility function
// Compile with SALESP.CPP
#include "salesp.h"

main()
{
   SalesPerson s;          // create SalesPerson object s

   s.setSales();
   s.printAnnualSales();

   return 0;
}
```

```
Enter sales amount for month 1: 5314.76
Enter sales amount for month 2: 4292.38
Enter sales amount for month 3: 4589.83
Enter sales amount for month 4: 5534.03
Enter sales amount for month 5: 4376.34
Enter sales amount for month 6: 5698.45
Enter sales amount for month 7: 4439.22
Enter sales amount for month 8: 5893.57
Enter sales amount for month 9: 4909.67
Enter sales amount for month 10: 5123.45
Enter sales amount for month 11: 4024.97
Enter sales amount for month 12: 5923.92

Total annual sales: $60120.59
```

Fig. 16.7 Cómo utilizar una función de utilería (parte 3 de 3).

Práctica sana de programación 16.9

Cada función miembro (y amigo) que modifique los miembros de datos privados de un objeto deberán asegurarse que los datos se conserven en un estado consistente.

Cuando se declara un objeto de una clase, se pueden dar *inicializadores* en paréntesis a la derecha del nombre del objeto y antes del punto y coma. Estos inicializadores son pasados como argumentos al constructor de la clase. Veremos pronto varios ejemplos de estas *llamadas de constructor.*

16.11 Cómo utilizar argumentos por omisión con los constructores

El constructor correspondiente a **time1.cpp** (figura 16.5) inicializó a **0 hour, minute** y **second** (es decir, a las 12 de media noche en hora militar). Los constructores pueden contener argumentos por omisión. La figura 16.8 vuelve a definir la función constructor **Time** para incluir para cada variable argumentos por omisión iguales a cero. Al dar al constructor argumentos por omisión, aun si a una llamada de constructor no se dan valores, el objeto está garantizado de estar en un estado consistente, debido a los argumentos por omisión. Un constructor suministrado

por el programador, y que utiliza sus argumentos por omisión, también se llama un constructor por omisión (puede ser invocado sin argumentos). El constructor utiliza el mismo tipo de código de validación que **setTime**, para asegurarse que el valor suministrado correspondiente a **hour** está en el rango de 0 a 23, y que los valores correspondientes a **minute** y **second** son cada uno de ellos dentro del rango 0 a 59. Si un valor está fuera de rango, se define a cero (este es un ejemplo de cómo asegurar que un miembro de datos se conserva en un estado consistente). El constructor podría llamar directamente a **setTime**, pero esto crearía la sobrecarga de una llamada de función adicional. El programa de la figura 16.8 inicializa cinco objetos **Time** uno con los tres argumentos en sus valores por omisión en la llamada de constructor, uno con un argumento definido, uno con dos definidos, uno con tres definidos y uno con tres argumentos inválidos especificados. Aparece mostrado el contenido de cada uno de los miembros de datos del objeto, después de la ejemplificación y de la inicialización.

Si para una clase no se define ningún constructor, el compilador creará un constructor por omisión. Dicho constructor no ejecutará ninguna inicialización, por lo que no se garantizará que esté en un estado consistente.

Práctica sana de programación 16.10

Incluya siempre un constructor que ejecute la incialización adecuada para su clase.

16.12 Cómo utilizar destructores

Un *destructor* es una función miembro especial de una clase. El nombre del destructor para una clase es la *tilde (~)* seguida por el nombre de la clase. Esta regla convencional de identificación tiene un atractivo intuitivo, porque el operador de tilde es el operador de complemento a nivel de bits, y , en cierto sentido, el destructor es el complemento del constructor.

```
// TIME2.H
// Declaration of the Time class.
// Member functions defined in TIME2.CPP

// prevent multiple inclusions of header file
#ifndef TIME2_H
#define TIME2_H

class Time {
public:
   Time(int = 0, int = 0, int = 0);  // default constructor
   void setTime(int, int, int);
   void printMilitary();
   void printStandard();
private:
   int hour;
   int minute;
   int second;
};

#endif
```

Fig. 16.8 Cómo usar un constructor mediante argumentos por omisión (parte 1 de 4).

```
// TIME2.CPP
// Member function definitions for Time class.
#include "time2.h"
#include <iostream.h>

// Constructor function to initialize private data.
// Default values are 0 (see class definition).
Time::Time(int hr, int min, int sec)
{
   hour = (hr >= 0 && hr < 24) ? hr : 0;
   minute = (min >= 0 && min < 60) ? min : 0;
   second = (sec >= 0 && sec < 60) ? sec : 0;
}

// Set values of hour, minute, and second.
// Invalid values are set to 0.
void Time::setTime(int h, int m, int s)
{
   hour = (h >= 0 && h < 24) ? h : 0;
   minute = (m >= 0 && m < 60) ? m : 0;
   second = (s >= 0 && s < 60) ? s : 0;
}

// Display time in military format: HH:MM:SS
void Time::printMilitary()
{
   cout << (hour < 10 ? "0" : "") << hour << ":"
        << (minute < 10 ? "0" : "") << minute << ":"
        << (second < 10 ? "0" : "") << second;
}

// Display time in standard format: HH:MM:SS AM (or PM)
void Time::printStandard()
{
   cout << ((hour == 0 || hour == 12) ? 12 : hour % 12)
        << ":" << (minute < 10 ? "0" : "") << minute
        << ":" << (second < 10 ? "0" : "") << second
        << (hour < 12 ? " AM" : " PM");
}
```

Fig. 16.8 Cómo usar un constructor con argumentos por omisión (parte 2 de 4).

Un destructor de clase es llamado automáticamente cuando un objeto de una clase se sale de alcance. De hecho el destructor mismo en realidad no destruye el objeto, más bien ejecuta *trabajos de terminación*, antes de que el sistema recupere el espacio de memoria del objeto con el fin de que pueda ser utilizado para almacenar nuevos objetos.

Un destructor no recibe ningún parámetro ni regresa ningún valor. Una clase sólo puede tener un destructor —la homonimia de destructores no está permitida.

Error común de programación 16.7

Intentar pasar argumentos a un destructor, regresar valores de un destructor, o hacer la homonimia de un destructor.


```
// FIG16_8.CPP
// Demonstrating a default constructor
// function for class Time.
#include <iostream.h>
#include "time2.h"

main()
{
   Time t1, t2(2), t3(21, 34), t4(12, 25, 42),
        t5(27, 74, 99);

   cout << "Constructed with:\n"
        << "all arguments defaulted:\n   ";
   t1.printMilitary();
   cout << "\n   ";
   t1.printStandard();

   cout << "\nhour specified; minute and second defaulted:\n   ";
   t2.printMilitary();
   cout << "\n   ";
   t2.printStandard();

   cout << "\nhour and minute specified; second defaulted:\n   ";
   t3.printMilitary();
   cout << "\n   ";
   t3.printStandard();

   cout << "\nhour, minute, and second specified:\n   ";
   t4.printMilitary();
   cout << "\n   ";
   t4.printStandard();

   cout << "\nall invalid values specified:\n   ";
   t5.printMilitary();
   cout << "\n   ";
   t5.printStandard();
   cout << endl;

   return 0;
}
```

Fig. 16.8 Cómo usar un constructor con argumentos por omisión (parte 3 de 4).

Note que no hemos proporcionado destructores para las clases que hasta ahora hemos presentado. De hecho, con clases sencillas rara vez se utilizan destructores. En el capítulo 18, veremos que los destructores son apropiados para aquellas clases cuyos objetos contienen almacenamiento dinámicamente asignado (por ejemplo, para arreglos y cadenas).

16.13 Cuándo son llamados los destructores y los constructores

Por lo regular, los constructores y los destructores son llamados de forma automática. El orden en el cual son hechas estas llamadas de función, depende del orden en el cual los objetos entran y salen de alcance. En general, las llamadas de destructor se efectúan en orden inverso a las

```
Constructed with:
all arguments defaulted:
      00:00:00
      12:00:00 AM
hour specified; minute and second defaulted:
      02:00:00
      2:00:00 AM
hour and minute specified; second defaulted:
      21:34:00
      9:34:00 PM
hour, minute, and second specified:
      12:25:42
      12:25:42 PM
all invalid values specified:
      00:00:00
      12:00:00 AM
```

Fig. 16.8 Cómo usar un constructor con argumentos por omisión (parte 4 de 4).

llamadas de constructor. Sin embargo, la duración de almacenamiento (persistencia) de los objetos puede modificar el orden en el cual los destructores son llamados.

Se llaman a los constructores para objetos declarados en alcance global al principio de la ejecución del programa. Los destructores correspondientes son llamados a la terminación del programa.

Los constructores son llamados para objetos locales automáticos, cuando estos objetos son declarados. Los destructores correspondientes son llamados cuando los objetos salen de alcance (es decir, cuando salen del bloque en el cual están declarados). Note que tanto los constructores como los destructores, correspondientes a los objetos locales automáticos, podrían ser llamados muchas veces, conforme los objetos entren y salgan de alcance.

Para los objetos locales estáticos los constructores son llamados una vez cuando estos objetos son declarados. Los destructores correspondientes son llamados a la terminación del programa.

El programa de la figura 16.9 demuestra el orden en el que se llaman los constructores y los destructores para objetos del tipo **CreateAndDestroy** en varios alcances. El programa declara **first** en alcance global. Su constructor es llamado conforme el programa inicia ejecución y su destructor es llamado a la terminación del programa, después de que todos los demás objetos han sido destruidos.

La función **main** declara tres objetos. Los objetos **second** y **fourth** son objetos automáticos locales, y el objeto **third** es un objeto local estático. Los constructores correspondientes a cada uno de estos objetos son llamados cuando es declarado cada uno de estos objetos. Cuando se llega al final de **main** los destructores para los objetos **fourth** y **second** son llamados en ese orden. Dado que el objeto **third** es estático, existirá desde el momento en que se declara, hasta la terminación del programa. El destructor para el objeto **third** es llamado antes del destructor para **first**, pero después de que hayan sido destruidos todos los demás objetos.

La función **create** declara tres objetos. Los objetos **fifth** y **seventh** son objetos automáticos locales, y el objeto **sixth** es un objeto local estático. Cuando se llega al final de **create** los destructores correspondientes a los objetos **seventh** y **fifth** son llamados en ese orden. Debido a que el objeto **sixth** es estático, existe desde el punto en el cual es declarado,


```
// CREATE.H
// Definition of class CreateAndDestroy.
// Member functions defined in CREATE.CPP.

#ifndef CREATE_H
#define CREATE_H

class CreateAndDestroy {
public:
   CreateAndDestroy(int);  // constructor
   ~CreateAndDestroy();    // destructor
private:
   int data;
};

#endif
```

Fig. 16.9 Cómo demostrar el orden en el cual son llamados los constructores y los destructores (parte 1 de 3).

```
// CREATE.CPP
// Member function definitions for class CreateAndDestroy
#include <iostream.h>
#include "create.h"

CreateAndDestroy::CreateAndDestroy(int value)
{
   data = value;
   cout << "Object " << data << "   constructor";
}

CreateAndDestroy::~CreateAndDestroy()
   { cout << "Object " << data << "   destructor " << endl; }
```

Fig. 16.9 Cómo demostrar el orden en el cual son llamados los constructores y los destructores (parte 2 de 3).

hasta la terminación del programa. El destructor para el objeto **sixth** es llamado antes de los destructores correspondientes a **third** y a **first**, pero después de que todos los otros objetos hayan sido destruidos.

16.14 Cómo utilizar miembros de datos y funciones miembro

Los miembros de datos privados pueden ser manipulados sólo por funciones miembro y los amigos de la clase. Una manipulación típica pudiera ser el ajuste del saldo bancario de un cliente (por ejemplo, un miembro de dato privado de la clase **BankAccount**), hecho por una función miembro **computeInterest**.

A menudo las clases proporcionan funciones miembro públicas para permitir a los clientes de la clase definir (es decir, escribir) u obtener (es decir, leer) los valores de los miembros de datos privados. Estas funciones no necesitan ser en específico funciones "set" y "get", pero a menudo lo son. En específico, una función miembro, que define el miembro de datos **interestRate**

```
// FIG16_9.CPP
// Demonstrating the order in which constructors and
// destructors are called.
#include <iostream.h>
#include "create.h"

void create(void);    // prototype

CreateAndDestroy first(1);  // global object

main()
{
   cout << "   (global created before main)\n";

   CreateAndDestroy second(2);        // local object
   cout << "   (local automatic in main)\n";

   static CreateAndDestroy third(3);  // local object
   cout << "   (local static in main)\n";

   create();  // call function to create objects

   CreateAndDestroy fourth(4);        // local object
   cout << "   (local automatic in main)\n";
   return 0;
}

// Function to create objects
void create(void)
{
   CreateAndDestroy fifth(5);
   cout << "   (local automatic in create)\n";

   static CreateAndDestroy sixth(6);
   cout << "   (local static in create)\n";

   CreateAndDestroy seventh(7);
   cout << "   (local automatic in create)\n";
```

```
Object 1 constructor (global created before main)
Object 2 constructor (local automatic in main)
Object 3 constructor (local static in main)
Object 5 constructor (local automatic in create)
Object 6 constructor (local static in create)
Object 7 constructor (local automatic in create)
Object 7 destructor
Object 5 destructor
Object 4 constructor (local automatic in main)
Object 4 destructor
Object 2 destructor
Object 6 destructor
Object 3 destructor
Object 1 destructor
```

Fig. 16.9 Córno demostrar el orden en que se llaman constructores y destructores (parte 3 de 3).

típicamente se llamaría **setInterestRate**, y una función miembro que obtiene el **interestRate**, típicamente sería llamada **getInterestRate**. Las funciones get también por lo común son conocidas como funciones "de consulta".

Parecería que proporcionar capacidades tanto de definir como de obtener resultaría en esencial lo mismo que el hacer públicos los miembros de datos. Esto es otra vez otra sutileza de C++, que hace que este lenguaje sea tan atractivo para la ingeniería de software. Si un miembro de datos es público, entonces el miembro de datos puede ser leído o escrito a voluntad por cualquier función del programa. Si un miembro de datos es privado, una función "get" pública parecería que podría permitir a otras funciones leer los datos a voluntad, pero la función get controlaría el formato y la exhibición de los datos. Una función pública "set" podría —y muy probablemente haría— escrutinizar con cuidado cualquier intento de modificación del valor del miembro de dato. Esto garantizaría que el nuevo valor es apropiado para ese elemento de dato. Por ejemplo, serían rechazados intentos de definir el día del mes a 37, intentos de definir el peso de una persona a un valor negativo, intentos de definir una cantidad numérica a un valor alfabético, intentos de definir una calificación de un examen como 185 (cuando el rango correcto es de 0 a 100), y así en lo sucesivo.

Observación de ingeniería de software 16.15

Hacer los miembros de datos privados y controlando su acceso, especialmente el acceso de escritura, a aquellos miembros de datos vía funciones miembro, ayuda a asegurar la integridad de los datos.

Los beneficios de la integridad de los datos no son automáticos sólo porque son hechos privados los miembros de datos, el programador debe de proveer la verificación de validez. C++, sin embargo, sí proporciona una estructura en la cual los programadores pueden diseñar mejores programas de una forma conveniente.

Práctica sana de programación 16.11

Las funciones miembro que definen los valores de los datos privados, deberán verificar que los nuevos valores propuestos son correctos; si no lo son, las funciones set deben colocar los miembros de datos privados en un estado consistente apropiado.

En la figura 16.10 se extiende nuestra clase **Time** a fin de incluir las funciones get y set para los miembros de datos privados **hour**, **minute** y **second**. Las funciones set controlan de forma estricta la definición de los miembros de datos. Cualquier intento de definir cualquier miembro de datos a un valor incorrecto, hará que el miembro de dato se especifique a cero (y dejando por lo tanto dicho miembro de dato en un estado consistente). Cada función get sólo regresa el valor apropiado del miembro de dato. El programa primero utiliza las funciones set para definir los miembros de dato privados del objeto **t** de **Time** a valores válidos, y a continuación utiliza las funciones get para recuperar dichos valores para su extracción. A continuación las funciones set intentan definir los miembros **hour** y **second** a valores inválidos y el miembro **minute** a un valor válido, y a continuación las funciones get recuperan los valores para su extracción. La salida confirma que los valores inválidos hacen que los miembros de datos se ajusten a cero. Por último, el programa especifica la hora a **11:58:00** e incrementa el valor de minutos en 3 con una llamada a la función **incrementMinutes**. La función **incrementMinutes** es una función no miembro, que utiliza las funciones get y set para incrementar de forma correcta el miembro **minute**. Aunque esto funciona, al emitir varias llamadas de función incurre en sobrecarga de rendimiento. En el siguiente capítulo analizamos el concepto de las funciones amigas, como un medio de eliminar esta sobrecarga de rendimiento.

```
// TIME3.H
// Declaration of the Time class.
// Member functions defined in TIME3.CPP

// prevent multiple inclusions of header file
#ifndef TIME3_H
#define TIME3_H

class Time {
public:
   Time(int = 0, int = 0, int = 0);  // constructor

   // set functions
   void setTime(int, int, int);  // set hour, minute, second
   void setHour(int);      // set hour
   void setMinute(int);    // set minute
   void setSecond(int);    // set second

   // get functions
   int getHour();          // return hour
   int getMinute();        // return minute
   int getSecond();        // return second

   void printMilitary();   // output military time
   void printStandard();   // output standard time
private:
   int hour;               // 0 - 23
   int minute;             // 0 - 59
   int second;             // 0 - 59
};

#endif
```

Fig. 16.10 Declaración de la clase **Time** (parte 1 de 5).

La disponibilidad de las funciones set en una clase, proporciona cierto rango de flexibilidad en el diseño de constructores.

Error común de programación 16.8

Un constructor puede llamar a otras funciones miembro de la clase como son las funciones set o get, pero dado que el constructor está inicializando el objeto, los miembros de datos pudieran no estar aún en un estado consistente. Puede causar errores utilizar miembros de datos antes de que hayan sido inicializados de manera adecuada.

Desde un punto de vista de ingeniería de software, las funciones set son ciertamente de importancia, porque pueden llevar a cabo verificación de validez. Las funciones set y get tienen otra ventaja importante de ingeniería de software.

Observación de ingeniería de software 16.16

Tener acceso a datos privados mediante las funciones miembro set y get no sólo protege los miembros de datos contra la recepción de valores inválidos, sino también aísla a los clientes de la clase de la representación de los miembros de datos. Entonces, si por alguna razón cambia la representación de los datos (típicamente para reducir la cantidad de almacenamiento requerido o para mejorar el rendimiento), sólo las funciones miembro necesitan modificarse —no es necesario que los clientes cambien, en tanto la interfaz proporcionada por las funciones miembro se conserve igual. Los clientes pueden, sin embargo, requerir ser recompilados.


```
// TIME3.CPP
// Member function definitions for Time class.

#include "time3.h"
#include <iostream.h>

// Constructor function to initialize private data.
// Default values are 0 (see class definition).
Time::Time(int hr, int min, int sec)
{
   hour = (hr >= 0 && hr < 24) ? hr : 0;
   minute = (min >= 0 && min < 60) ? min : 0;
   second = (sec >= 0 && sec < 60) ? sec : 0;
}

// Set the values of hour, minute, and second.
void Time::setTime(int h, int m, int s)
{
   hour = (h >= 0 && h < 24) ? h : 0;
   minute = (m >= 0 && m < 60) ? m : 0;
   second = (s >= 0 && s < 60) ? s : 0;
}

// Set the hour value
void Time::setHour(int h)
   { hour = (h >= 0 && h < 24) ? h : 0; }

// Set the minute value
void Time::setMinute(int m)
   { minute = (m >= 0 && m < 60) ? m : 0; }

// Set the second value
void Time::setSecond(int s)
   { second = (s >= 0 && s < 60) ? s : 0; }

// Get the hour value
int Time::getHour() { return hour; }

// Get the minute value
int Time::getMinute() { return minute; }

// Get the second value
int Time::getSecond() { return second; }

// Display military format time: HH:MM:SS
void Time::printMilitary()
{
   cout << (hour < 10 ? "0" : "") << hour << ":"
        << (minute < 10 ? "0" : "") << minute << ":"
        << (second < 10 ? "0" : "") << second;
}
```

Fig. 16.10 Definiciones de funciones miembro para la clase **Time** (parte 2 de 5).

```
// Display standard format time: HH:MM:SS AM (or PM)
void Time::printStandard()
{
   cout << ((hour == 12) ? 12 : hour % 12) << ":"
        << (minute < 10 ? "0" : "") << minute << ":"
        << (second < 10 ? "0" : "") << second
        << (hour < 12 ? " AM" : " PM");
}
```

Fig. 16.10 Definiciones de funciones miembro para la clase **Time** (parte 3 de 5).

```
// FIG16_10.CPP
// Demonstrating the Time class set and get functions
#include <iostream.h>
#include "time3.h"

main()
{
   Time t;
   void incrementMinutes(Time &, const int);

   t.setHour(17);
   t.setMinute(34);
   t.setSecond(25);

   cout << "Result of setting all valid values:\n  Hour: "
        << t.getHour()
        << "  Minute: " << t.getMinute()
        << "  Second: " << t.getSecond() << "\n\n";

   t.setHour(234);       // invalid hour set to 0
   t.setMinute(43);
   t.setSecond(6373);    // invalid second set to 0

   cout << "Result of attempting to set invalid hour and"
        << " second:\n  Hour: " << t.getHour()
        << "  Minute: " << t.getMinute()
        << "  Second: " << t.getSecond() << "\n\n";

   t.setTime(11, 58, 0);
   incrementMinutes(t, 3);

   return 0;
}
```

Fig. 16.10 Cómo utilizar las funciones set y get (parte 4 de 5).

16.15 Una trampa sutil: cómo regresar una referencia a un miembro de datos privado

Una referencia a un objeto es un seudónimo del objeto mismo y por lo tanto puede ser utilizado del lado izquierdo de un enunciado de asignación. En este contexto, la referencia es un valor a la izquierda (*lvalue*) es muy aceptable, que puede recibir un valor. Una forma de aprovechar esta


```
void incrementMinutes(Time &tt, const int count)
{
   cout << "Incrementing minute " << count
        << " times:\nStart time: ";
   tt.printStandard();

   for (int i = 1; i <= count; i++) {
      tt.setMinute((tt.getMinute() + 1) % 60);

      if (tt.getMinute() == 0)
         tt.setHour((tt.getHour() + 1) % 24);

      cout << "\nminute + 1: ";
      tt.printStandard();
   }

   cout << endl;
}
```

```
Result of setting all valid values:
  Hour: 17  Minute: 34  Second: 25

Result of attempting to set invalid hour and second:
  Hour: 0  Minut: 43  Second: 0

Incrementing minute 3 times:
Start time: 11:58:00 AM
minute + 1: 11:59:00 AM
minute + 1: 12:00:00 PM
minute + 1: 12:01:00 PM
```

Fig. 16.10 Cómo utilizar funciones set y get (parte 5 de 5).

capacidad (¡desafortunadamente!) es hacer que una función miembro pública de una clase regrese una referencia a un miembro de datos privado de la misma clase.

En la figura 16.11 se utiliza una versión simplificada de la clase **Time**, para demostrar el regreso de una referencia a un miembro de datos privado. Tal regreso, de hecho efectúa una llamada a la función **badSetHour** ¡un seudónimo del miembro de datos privado **hour**! La llamada de función puede ser utilizada de cualquiera de las formas en que un miembro de datos privado pueda ser utilizado, ¡incluye un valor a la izquierda (*lvalue*) en un enunciado de asignación!

Práctica sana de programación 16.12

Nunca haga que una función miembro pública regrese una referencia no **const** *(o un apuntador) a un miembro de datos privado. Regresar una referencia como ésta viola el encapsulado de la clase.*

El programa empieza declarando el objeto **t** de **Time** y la referencia **hourRef** que está asignada a la referencia regresada por la llamada **t.badSetHour(20)**. El programa muestra el valor del seudónimo **hourRef**. A continuación, se utiliza el seudónimo para definir el valor de **hour** a 30 (un valor inválido) y se vuelve a mostrar el valor. Finalmente, se utiliza la llamada de función misma, como un valor a la izquierda (*lvalue*) y se le asigna el valor 74 (otro valor inválido), y este valor es mostrado.

```
// TIME4.H
// Declaration of the Time class.
// Member functions defined in TIME4.CPP

// prevent multiple inclusions of header file
#ifndef TIME4_H
#define TIME4_H

class Time {
public:
   Time(int = 0, int = 0, int = 0);
   void setTime(int, int, int);
   int getHour();
   int &badSetHour(int);  // DANGEROUS reference return
private:
   int hour;
   int minute;
   int second;
};

#endif
```

Fig. 16.11 Cómo regresar una referencia a un miembro de datos privado (parte 1 de 3).

```
// TIME4.CPP
// Member function definitions for Time class.
#include "time4.h"
#include <iostream.h>

// Constructor function to initialize private data.
// Calls member function setTime to set variables.
// Default values are 0 (see class definition).
Time::Time(int hr, int min, int sec)
   { setTime(hr, min, sec); }

// Set the values of hour, minute, and second.
void Time::setTime(int h, int m, int s)
{
   hour = (h >= 0 && h < 24) ? h : 0;
   minute = (m >= 0 && m < 60) ? m : 0;
   second = (s >= 0 && s < 60) ? s : 0;
}

// Get the hour value
int Time::getHour() { return hour; }

// BAD PROGRAMMING PRACTICE:
// Returning a reference to a private data member.
int &Time::badSetHour(int hh)
{
   hour = (hh >= 0 && hh < 24) ? hh : 0;

   return hour;  // DANGEROUS reference return
}
```

Fig. 16.11 Cómo regresar una referencia a un miembro de datos privado (parte 2 de 3).


```
// FIG16_11.CPP
// Demonstrating a public member function that
// returns a reference to a private data member.
// Time class has been trimmed for this example.

#include <iostream.h>
#include "time4.h"

main()
{
   Time t;
   int &hourRef = t.badSetHour(20);

   cout << "Hour before modification: "
        << hourRef << '\n';
   hourRef = 30;  // modification with invalid value
   cout << "Hour after modification: "
        << t.getHour() << '\n';

   // Dangerous: Function call that returns
   // a reference can be used as an lvalue.
   t.badSetHour(12) = 74;
   cout << "\n*********************************\n"
        << "BAD PROGRAMMING PRACTICE!!!!!!!!!\n"
        << "badSetHour as an lvalue, Hour: "
        << t.getHour()
        << "\n*********************************\n";

   return 0;
}
```

```
Hour before modification: 20
Hour after modification: 30

*********************************
BAD PROGRAMMING PRACTICE!!!!!!!!!
badSetHour as an lvalue, Hour: 74
*********************************
```

Fig. 16.11 Cómo regresar una referencia a un miembro de datos privado (parte 3 de 3).

16.16 Asignación por omisión en copia a nivel de miembro

El operador de asignación (=) es utilizado para asignar un objeto a otro objeto del mismo tipo. Dicha asignación se lleva normalmente a cabo mediante la *copia a nivel de miembro* —cada miembro de un objeto es copiado en forma individual al mismo miembro dentro de otro objeto (vea la figura 16.12). (Nota: la copia a nivel de miembro puede causar problemas serios, cuando sea utilizada con una clase cuyos miembros de datos contengan almacenamiento dinámicamente asignado; en el capítulo 18, "Homonimia de operadores", analizaremos estos problemas y mostraremos cómo resolverlos.)

```
// FIG16_12.CPP
// Demonstrating that class objects can be assigned
// to each other using default memberwise copy
#include <iostream.h>

// Simple Date class
class Date {
public:
   Date(int = 1, int = 1, int = 1990);  // default constructor
   void print();
private:
   int month;
   int day;
   int year;
};

// Simple Date constructor with no range checking
Date::Date(int m, int d, int y)
{
   month = m;
   day = d;
   year = y;
}

// Print the Date in the form mm-dd-yyyy
void Date::print()
{ cout << month << '-' << day << '-' << year; }

main()
{
   Date d1(7, 4, 1993), d2;  // d2 defaults to 1/1/90

   cout << "d1 = ";
   d1.print();
   cout << "\nd2 = ";
   d2.print();

   d2 = d1;     // assignment by default memberwise copy
   cout << "\n\nAfter default memberwise copy, d2 = ";
   d2.print();
   cout << endl;

   return 0;
}
```

```
d1 = 7-4-1993
d2 = 1-1-1990

After default memberwise copy, d2 = 7-4-1993
```

Fig. 16.12 Cómo asignar un objeto a otro mediante la copia a nivel de miembro por omisión.

Los objetos pueden ser pasados como argumentos de función y pueden ser regresados de las funciones. Este pasar y regresar se ejecuta por omisión —en llamada por valor se pasa o se regresa una copia del objeto (presentamos varios ejemplos en el capítulo 18, "Homonimia de operadores").

Sugerencia de rendimiento 16.4

Pasar un objeto en llamada por valor es bueno desde un punto de vista de seguridad, porque la función llamada no tiene acceso al objeto original, pero cuando se tiene que hacer una copia de un objeto grande, la llamada por valor puede degradar el rendimiento. Un objeto puede ser pasado en llamada por referencia, pasando ya sea un apuntador o una referencia al objeto. La llamada por referencia ofrece un buen rendimiento, pero es más débil desde un punto de vista de seguridad, debido a que la función llamada tiene acceso al objeto original. Una alternativa segura sería una referencia en llamada por `const`.

16.17 Reutilización del software

Las personas que escriben programas orientados a objetos se concentran en la puesta en práctica de clases útiles. Existe una enorme oportunidad de capturar y catalogar clases, de tal forma que sean accesibles a grandes segmentos de la comunidad de programadores. Existen en todo el mundo muchas *bibliotecas de clases*, y otras están siendo desarrolladas. Se hacen esfuerzos para que dichas bibliotecas sean muy accesibles. El software podrá entonces ser construido a partir de componentes existentes, bien definidos, cuidadosamente probados, bien documentados, portátiles y ampliamente disponibles. Este tipo de *reutilización de software* puede acelerar el desarrollo de software poderoso y de alta calidad. El desarrollo rápido de aplicaciones *(RAD por Rapid applications development)* se convierte en posible.

Antes de que se pueda alcanzar el potencial total de la reutilización del software, deben ser resueltos, sin embargo, problemas significativos. Necesitamos procesos para catalogar, procedimientos para licenciar, mecanismos de protección para asegurar que las copias maestras de clases no están corrompidas, procedimientos de descripción, de tal forma que los diseñadores de sistemas nuevos puedan determinar si los objetos existentes llenan sus necesidades, mecanismos de búsqueda para determinar qué clases están disponibles y que tan justamente cumplen los requisitos del diseñador de software, y así en lo sucesivo. Muchos problemas interesantes de investigación y desarrollo necesitan resolverse. Estos problemas *serán* resueltos, porque es enorme el valor potencial de sus soluciones.

Resumen

- Las estructuras son tipos de datos agregados construidos utilizando objetos de otros tipos.
- La palabra reservada `struct` introduce la definición de estructura. El cuerpo de una estructura queda definido entre llaves ({ y }). La definición de cada estructura debe terminar con un punto y coma.
- Se puede utilizar un nombre de etiqueta de estructura para declarar variables de un tipo de estructura.
- Las definiciones de estructura no reservan espacio en memoria; crean nuevos tipos de datos que son utilizados para declarar variables.
- Se tiene acceso a los miembros de una estructura o de una clase utilizando los operadores de acceso de miembro el operador punto (.) y el operador flecha (->). El operador punto da acceso a un miembro de estructura vía el nombre de variable del objeto o una referencia al objeto. El operador flecha da acceso a un miembro de estructura vía un apuntador al objeto.

- Los inconvenientes para la creación de nuevos tipos de datos utilizando los **`struct`** de tipo C son la posibilidad de tener datos sin inicializar; inicializaciones incorrectas; todos los programas que utilicen **`struct`** del estilo C deben ser transformados si se modifica la puesta en práctica de **`struct`**; y no se da protección para asegurar que los datos se conservan en un estado consistente con los valores de datos apropiados.
- Las clases le permiten al programador hacer modelos de objetos con atributos y comportamientos. Los tipos de clase pueden ser definidos en C++ utilizando las palabras reservadas **`class`** y **`struct`**, pero la palabra reservada **`class`** por lo regular es utilizada para este fin.
- El nombre de clase puede ser utilizado para declarar objetos de dicha clase.
- Las definiciones de clase empiezan con la palabra reservada **`class`**. El cuerpo de la definición de clase está delimitado por llaves (**`{`** y **`}`**). Las definiciones de clase terminan con un punto y coma.
- Cualquier miembro de datos o función miembro declarada después de **`public:`** en una clase es visible a cualquier función que tenga acceso a un objeto de dicha clase.
- Cualquier miembro de dato o función miembro declarada después de **`private:`** es sólo visible a amigos y otros miembros de dicha clase.
- Los especificadores de acceso de miembro siempre terminan con dos puntos (`:`) y pueden aparecer múltiples veces en una definición de clase.
- Los datos privados no son accesibles desde fuera de la clase.
- La puesta en práctica de una clase se dice que está oculta de sus clientes, es decir, está "encapsulada".
- Un constructor es una función miembro especial, con el mismo nombre de la clase, y se utiliza para inicializar los miembros de un objeto de clase. Los constructores son llamados cuando se produce o se ejemplifica un objeto de su clase.
- La función con el mismo nombre que la clase, pero antecedida con el carácter tilde (~), se llama un destructor.
- El conjunto de funciones miembro públicas de una clase se conoce como la interfaz de la clase o la interfaz pública.
- Cuando una función miembro se define por fuera de la definición de la clase, el nombre de función es antecedido por el nombre de la clase y por el operador de resolución de alcance binario (`::`).
- Las funciones miembro definidas utilizando el operador de resolución de alcance por fuera de una definición de clase, están dentro del alcance de la clase.
- Las funciones miembro definidas en una definición de clase están en línea en forma automática. El compilador se reserva el derecho de no considerar en línea a cualquier función.
- Llamar funciones miembro es más conciso que la llamada de funciones en la programación procedural, porque las funciones miembro asumen que los nombres no declarados son miembros de la clase en la cual la función miembro está definida.
- Dentro del alcance de una clase, los miembros de la clase pueden ser referenciados simplemente con sus nombres. Fuera del alcance de una clase, los miembros de la clase se referencían ya sea a través de un nombre de objeto, una referencia a un objeto o un apuntador a un objeto.
- Los operadores de selección de miembros `.` y `->` son utilizados para tener acceso a los miembros de clase.

- Un principio fundamental de buena ingeniería de software es separar la interfaz de la puesta en práctica para facilitar la capacidad de modificación del programa.
- Las definiciones de clase por lo regular están colocadas en los archivos de cabecera y las definiciones de función miembro están normalmente colocadas en los archivos de código fuente, con el mismo nombre base.
- El modo de acceso por omisión para las clases es **`private:`** de tal forma que todos los miembros después de un encabezado de clase y antes de la primera etiqueta se consideran como privados.
- Los miembros de clase públicos presentan una vista de los servicios que proporciona la clase.
- El acceso a los datos privados de una clase puede ser controlado con cuidado mediante el uso de las funciones miembro conocidas como funciones de acceso. Si una clase desea permitir que los clientes lean datos privados, la clase puede proporcionar una función "get". Para permitir que los clientes modifiquen datos privados, la clase puede proveer una función "set".
- Los miembros de datos de una clase normalmente se hacen privados y las funciones miembro de una clase a menudo se hacen públicos. Algunas funciones miembro se conservan privadas y sirven como funciones de utilería a otras funciones de la clase.
- Los miembros de datos de una clase no pueden ser inicializados en una definición de clase. Deben ser inicializados en un constructor, o sus valores pueden ser definidos más adelante, después de que su objeto haya sido creado.
- Los constructores pueden tener homónimos.
- Una vez que un objeto de clase esté correctamente inicializado, todas las funciones miembro que manipulan el objeto deberán asegurarse que el objeto se conserva en un estado consistente.
- Cuando se declara un objeto de una clase, pueden ser proporcionados los inicializadores. Estos inicializadores se pasan al constructor de la clase.
- Los constructores pueden especificar argumentos por omisión.
- Los constructores pudieran no especificar tipos de regreso, ni pudieran intentar regresar valores.
- Si no se define un constructor para una clase, el compilador crea un constructor por omisión. Un constructor por omisión suministrado por el compilador no lleva a cabo ninguna inicialización, por lo cual cuando se crea el objeto, no está garantizado que esté en un estado consistente.
- Un destructor de una clase es automáticamente llamado, cuando un objeto de una clase sale de alcance. El destructor mismo, de hecho no destruye el objeto, pero sí ejecuta trabajos de terminación, antes de que el sistema recupere el almacenamiento del objeto.
- Los destructores no reciben parámetros y no regresan valores. Una clase sólo puede tener un destructor.
- El operador de asignación (=) es utilizado para asignar un objeto a otro objeto del mismo tipo. Dicha asignación se ejecuta normalmente mediante copia a nivel de miembro —cada miembro de un objeto se copia de manera individual al mismo miembro en otro objeto.

Terminología

operador de dirección **`&`**
operador de referencia **`&`**
operador de resolución de alcance **`::`**
tipo de datos abstractos (ADT)

función de acceso
operador de asignación (=)
atributo
comportamiento
`class`
operador selector de miembro de clase (.)
declaración de clase
definición de clase
alcance de clase
nombre de etiqueta de clase
cliente de una clase
estado consistente para un miembro de datos
constructor
abstracción de datos
ocultamiento de datos
miembro de datos
tipo de datos
constructor por omisión
destructor
objetos dinámicos
encapsulado
alcance de entrada
extensibilidad
alcance de archivo
declaración hacia adelante de una clase
función get
objeto global
archivo de cabecera
puesta en práctica de una clase
ocultamiento de información
inicializar un objeto de clase
función miembro en línea
ejemplos de una clase
ejemplificación de un objeto de una clase
interfaz a una clase
salir de alcance
control de acceso de miembro
especificador de acceso de miembro
`función miembro`
inicializador de miembro
copia a nivel de miembro
mensaje
método
función no miembro
objeto local no estático
objeto
programación orientada a objetos (OOP)
función predicada
principio del mínimo privilegio
`private:`
programación procedural
`protected:`
interfaz pública de una clase
`public:`
función de consulta
desarrollo rápido de aplicaciones (RAD)
código reutilizable
operador de resolución de alcance (::)
servicios de una clase
función set
reutilización de software
archivo de código fuente
objeto local estático
trabajos de terminación
tilde (~) en nombre de destructor
tipo definido por usuario
función de utilería.

Errores comunes de programación

16.1 Olvidar el punto y coma al final de una definición de clase.
16.2 Intentar inicializar explícitamente un miembro de datos de una clase.
16.3 Intentar hacer la homonimia de una función miembro mediante una función no incluida en el alcance de dicha clase.
16.4 Un intento por parte de una función, que no sea un miembro de una clase particular (o un amigo de dicha clase) de obtener acceso a un miembro privado de dicha clase.
16.5 Intentar inicializar de forma explícita un miembro de datos de una clase dentro de la definición de clase.
16.6 Intentar declarar un tipo de regreso para un constructor, y/o intentar regresar un valor proveniente de un constructor.
16.7 Intentar pasar argumentos a un destructor, regresar valores de un destructor, o hacer la homonimia de un destructor.
16.8 Un constructor puede llamar a otras funciones miembro de la clase como son las funciones set o get, pero dado que el constructor está inicializando el objeto, los miembros de datos pudieran no estar aún en un estado consistente. Puede causar errores utilizar miembros de datos antes de que hayan sido adecuadamente inicializados.

Prácticas sanas de programación

16.1 Para mayor claridad y legibilidad, utilice cada especificador de acceso de miembro sólo una vez en una definición de clase. Coloque primero los miembros públicos, donde sean con fácilidad localizables.
16.2 Defina todas, a excepción de las más pequeñas funciones miembro, por afuera de la definición de clase. Esto ayuda a separar la interfaz de la clase de su puesta en práctica.
16.3 Utilice en un programa las directivas de preprocesador `#ifndef`, `#define` y `#endif` para evitar que los archivos de cabecera sean incluidos más de una vez.
16.4 Utilice el nombre del archivo de cabecera, con el punto reemplazado por un subrayado, en las directivas de preprocesador `#ifndef` y `#define` correspondientes a un archivo de cabecera.
16.5 Si en una declaración de clase decide listar primero los miembros privados, utilice en forma explícita la etiqueta `private:` a pesar de que por omisión se supone que `private:` es asumido. Esto mejora la claridad del programa. Nuestra preferencia es listar primero los miembros `public:` de una clase.
16.6 A pesar del hecho que las etiquetas `public:` y `private:` pueden ser repetidas y entremezcladas, liste primero en un grupo todos los miembros públicos de una clase y a continuación liste en otro grupo todos los miembros privados. Esto enfoca la atención del cliente en la interfaz pública de la clase, en vez de en la puesta en práctica de la misma.
16.7 Evita confusión usar los especificadores de acceso de miembro `public:`, `protected:`, y `private:` una sola vez en cada definición de clase.
16.8 Cuando sea apropiado (casi siempre), proporcione un constructor para asegurarse que todo objeto es inicializado de manera correcta, con valores significativos.
16.9 Cada función miembro (y amigo) que modifique los miembros de datos privados de un objeto deberán asegurarse que los datos se conserven en un estado consistente.
16.10 Incluya siempre un constructor que ejecute la incialización adecuada para su clase.
16.11 Las funciones miembro que definen los valores de los datos privados, deberán verificar que los nuevos valores propuestos son correctos; si no lo son, las funciones set deben colocar los miembros de datos privados en un estado consistente apropiado.
16.12 Nunca haga que una función miembro pública regrese una referencia no `const` (o un apuntador) a un miembro de datos privado. Regresar una referencia como ésta viola el encapsulado de la clase.

Sugerencias de rendimiento

16.1 Las estructuras por lo regular pasan en llamada por valor. Para evitar la sobrecarga de copiar una estructura, pase la estructura en llamada por referencia.
16.2 A fin de evitar la sobrecarga de la llamada por valor y aún así obtener el beneficio de que la información original del llamador quede protegida contra modificaciones, pase argumentos de tamaño extenso como referencias `const`.
16.3 Definir una función miembro pequeña dentro de una definición de clase automáticamente coloca la función miembro en línea (si el compilador así lo decide). Esto puede mejorar el rendimiento, pero no promueve una mejor ingeniería de software.
16.4 Pasar un objeto en llamada por valor es bueno desde un punto de vista de seguridad, porque la función llamada no tiene acceso al objeto original, pero cuando se tiene que hacer una copia de un objeto grande, la llamada por valor puede degradar el rendimiento. Un objeto puede ser pasado en llamada por referencia, pasando ya sea un apuntador o una referencia al objeto. La llamada por

referencia ofrece un buen rendimiento, pero es más débil desde un punto de vista de seguridad, debido a que la función llamada tiene acceso al objeto original. Una alternativa segura sería una referencia en llamada por `const`.

Observaciones de ingeniería de software

16.1 Es importante escribir programas que sean comprensibles y fáciles de mantener. Las modificaciones son la regla más que la excepción. Los programadores deberán prever que su código será modificado. Como veremos, las clases facilitan la capacidad de modificación de los programas.

16.2 Los clientes de una clase utilizan la clase sin conocer los detalles internos de cómo está puesta en práctica dicha clase. Si se modifica la puesta en práctica de la clase (para mejorar el rendimiento, por ejemplo) no es necesario modificar los clientes de la clase. Esto facilita mucho la modificación de sistemas.

16.3 Las funciones miembro por lo regular están formadas por unas pocas líneas de código, porque ninguna lógica es requerida para determinar si son válidos los miembros de datos.

16.4 Los clientes tienen acceso a la interfaz de una clase, pero no deberán tener acceso a la puesta en práctica de la clase.

16.5 Declarar funciones miembro dentro de una definición de clase y definir dichas funciones miembro por fuera de dicha definición de clase separa la interfaz de una clase de su puesta en práctica. Esto promueve buena ingeniería de software.

16.6 Usar un enfoque de programación orientado a objetos a menudo puede simplificar las llamadas de función al reducir el número de parámetros a pasarse. Este beneficio de la programación orientada a objetos se deriva del hecho que el encapsulado de los miembros de datos y las funciones miembros dentro de un objeto le da a las funciones miembro el derecho de acceso a los miembros de datos.

16.7 Coloque la declaración de la clase en un archivo de cabecera a incluirse por cualquier cliente que desee utilizar dicha clase. Esto forma la interfaz pública de la clase. Coloque las definiciones de las funciones miembro de la clase en un archivo fuente. Esto conforma la puesta en práctica de la clase.

16.8 Los clientes de una clase no necesitan ver el código fuente de la clase a fin de utilizarla. Los clientes deben, sin embargo, poder tener la capacidad de enlazarse con el código objeto de la clase.

16.9 Aquella información de importancia a la interfaz con una clase debería estar incluida en el archivo de cabecera. Aquella información que sólo será utilizada en forma interna en la clase y que no será necesaria para los clientes de la clase, debería ser incluida en el archivo fuente no publicado. Este es otra vez otro ejemplo del principio del mínimo privilegio.

16.10 C++ alienta a que los programas sean independientes de la puesta en práctica. Cuando se modifica la puesta en práctica de una clase utilizada mediante código independiente de la misma, dicho código no necesita ser cambiado, pero pudiera necesitar ser recopilado.

16.11 Conserve privados todos los miembros de datos de una clase. Proporcione funciones miembro públicas para definir los valores de los miembros de datos privados y para obtener los valores de los miembros de datos privados. Esta arquitectura ayuda a ocultar la puesta en práctica de una clase a la vista de sus clientes, lo que reduce errores y mejora la capacidad de modificación del programa.

16.12 Los diseñadores de clase utilizan miembros privados, protegidos y públicos para obligar a la idea de ocultamiento de información y al principio del mínimo privilegio.

16.13 Hacer privados los miembros de datos de una clase y públicas las funciones miembro de la clase, facilita la depuración, porque los problemas con las manipulaciones de datos se localizan ya sea en las funciones miembros de la clase o en los amigos de la misma

16.14 Las funciones miembro tienen tendencia a agruparse en varias categorías diferentes: funciones que leen y regresan el valor de miembros de datos privados, funciones que definen el valor de miembros de datos privados, funciones que ponen en práctica las características de la clase, y funciones que ejecutan varias tareas mecánicas para la clase, como la inicialización de objetos de clase, la asignación de objetos de clase, la conversión entre clases y tipos incorporados o entre clases y otras clases, y el manejo de memoria para objetos de clase.

16.15 Hacer los miembros de datos privados y controlando su acceso, especialmente el acceso de escritura, a aquellos miembros de datos vía funciones miembro, ayuda a asegurar la integridad de los datos.

16.16 Tener acceso a datos privados mediante las funciones miembro set y get no solamente protege los miembros de datos contra la recepción de valores inválidos, sino también aísla a los clientes de la clase de la representación de los miembros de datos. Entonces, si por alguna razón cambia la representación de los datos (típicamente para reducir la cantidad de almacenamiento requerido o para mejorar el rendimiento), sólo las funciones miembro necesitan modificarse no es necesario que los clientes cambien, en tanto la interfaz proporcionada por las funciones miembro se conserve igual. Los clientes pueden, sin embargo, requerir ser recompilados.

Ejercicios de autoevaluación

16.1 Llene cada uno de los siguientes espacios en blanco:

a) La palabra reservada _____ introduce una definición de estructura.

b) Se tiene acceso a los miembros de clase vía el operador _____ en conjunción con un objeto de clase o vía el operador _____ en conjunción con un apuntador a un objeto de clase.

c) Los miembros de una clase especificados como _____ son sólo accesibles a las funciones miembro de la clase y amigos de la clase.

d) Un _____ es una función miembro especial utilizada para inicializar los miembros de datos de una clase.

e) El acceso por omisión para los miembros de una clase es _____.

f) Una función _____ se utiliza para asignar valores a los miembros de dato privados de una clase.

g) _____ puede ser utilizada para asignar un objeto de una clase a otro objeto de la misma clase.

h) Las funciones miembro de una clase a menudo se hacen _____ y los miembros de datos de una clase se hacen por lo regular _____.

i) Una función _____ se utiliza para recuperar valores de datos privados de una clase.

j) El conjunto de funciones miembro públicas de una clase se conoce como la _____ de la clase.

k) La puesta en práctica de una clase se dice que está oculta de sus clientes o _____.

l) Las palabras reservadas _____ y _____ pueden ser utilizadas para introducir una definición de clase.

m) Los miembros de una clase especificados como _____ son accesibles en cualquier parte en que un objeto de la clase esté en alcance.

16.2 Encuentre el o los errores en cada una de las siguientes y explique cómo corregirlo.

a) Suponga que se declara el siguiente prototipo en la clase `Time`

```
void ~Time(int);
```

b) Lo siguiente es una definición parcial de la clase `Time`

```
class Time {
public:
   //function prototypes
private:
   int hour = 0;
   int minute = 0;
   int second = 0;
};
```

c) Suponga el siguiente prototipo que se declara en la clase `Employee`

```
int Employee(const char *, const char *);
```

Respuestas a los ejercicios de autoevaluación

16.1 a) `struct`. b) punto (`.`), flecha (`->`). c) `private`. d) constructor. e) `private`. f) set. g) copia a nivel de miembro por omisión. h) `private, public`. i) get. j) interfaz. k) encapsulado. l). `class, struct` m) `public`.

16.2 a) Error: no se permite que los destructores regresen valores o tomen argumentos.
Corrección: elimine el tipo de regreso `void` y el parámetro `int` de la declaración.

b) Error: los miembros no pueden ser inicializados de forma explícita en la definición de clase.
Corrección: elimine la inicialización explícita de la definición de clase e inicialice los miembros de datos en un constructor.

c) Error: los constructores no pueden regresar valores.
Corrección: elimine el tipo de regreso `int` de la declaración.

Ejercicios

16.3 ¿Cuál es el propósito del operador de resolución de alcance?

16.4 Compare los conceptos de `struct` y de `class` en C++.

16.5 Proporcione un constructor que sea capaz de utilizar la hora actual de la función `time()` —declarada en el encabezado `time.h`— de la biblioteca estándar de C para inicializar un objeto de la clase `Time`.

16.6 Cree una clase llamada `Complex` para ejecutar aritmética con números complejos. Escriba un programa manejador para probar su clase.

Los números complejos tienen la forma

```
realPart + imaginaryPart * i
```

donde i es

$$\sqrt{-1}$$

Utilice variables de punto flotante para representar los datos privados de la clase. Proporcione una función constructor que permita que se inicialice un objeto de esta clase cuando sea declarado. El constructor deberá contener valores por omisión, por si no se proporcionan inicializadores. Proporcione funciones miembros públicas para cada uno de los siguientes:

a) Suma de dos números `Complex`: las partes reales se suman juntas y las partes imaginarias se suman juntas también.
b) Resta de dos números `Complex`: la parte real del operando derecho se resta de la parte real del operador izquierdo y la parte imaginaria del operador derecho se resta de la parte imaginaria del operador izquierdo.
c) Imprimir números `Complex` en la forma `(a, b)` donde `a` sea la parte real y `b` la parte imaginaria.

16.7 Cree una clase llamada `RationalNumb er` (para ejecutar aritmética con fracciones). Escriba un programa manejador para probar su clase.

Utilice variables enteras para representar los datos privados de la clase el numerador y el denominador. Proporcione una función constructor que permita que se inicialice un objeto de esta clase cuando sea declarado. El constructor deberá contener valores por omisión, para el caso en que no se proporcionen inicializadores y deberá almacenar la fracción en forma simplificada (es decir, la fracción

$$\frac{2}{4}$$

deberá ser almacenada en el objeto como 1 en el numerador y 2 en el denominador). Proporcione funciones miembro públicas para cada una de las siguientes:

a) Suma de dos números `Rational`. El resultado deberá ser almacenado en forma simplificada.
b) Resta de dos números `Rational`. El resultado deberá ser almacenado en forma simplificada.
c) Multiplicación de dos números `Rational`. El resultado deberá ser almacenado en forma simplificada.
d) División de dos números `Rational`. El resultado deberá ser almacenado en forma simplificada.
e) Imprimir números `Rational` en la forma `a/b` donde `a` es el numerador y `b` el denominador.
f) Imprimir números `Rational` en formato de punto flotante.

16.8 Modifique la clase `Time` de la figura 16.10 para incluir una función miembro `tick` que incremente la hora almacenado en un objeto `Time` en un segundo. El objeto `Time` deberá conservarse siempre en un estado consistente. Escriba un programa manejador que pruebe la función miembro `tick` en un ciclo que imprima la hora en formato estándar durante cada iteración del ciclo para ilustrar que la función miembro `tick` funciona correctamente. Asegúrese de probar los casos siguientes:

a) Incrementar al siguiente minuto.
b) Incrementar a la hora siguiente.
c) Incrementar al día siguiente (es decir 11:59:59 PM a 12:00:00 AM).

16.9 Modifique la clase `Date` de la figura 16.12 para que lleve a cabo verificación de errores en los valores de inicializador correspondientes a los miembros de datos `month`, `day` y `year`. También proporcione una función miembro `nextDay` para incrementar los días en uno. El objeto `Date` deberá siempre conservarse en un estado consistente. Escriba un programa manejador que pruebe la función `nextDay` en un ciclo que imprima la fecha durante cada iteración del ciclo, a fin de ilustrar que la función `nextDay` funciona correctamente. Asegúrese de probar los casos siguientes:

a) Incrementar al siguiente mes.
b) Incrementar al siguiente año.

16.10 Combine la clase modificada `Time` del ejercicio 16.8 y la clase modificada `Date` del ejercicio 16.9 en una clase llamada `DateAndTime` (en el capítulo 19 analizaremos la herencia que nos permitirá llevar a cabo esta tarea rápidamente sin tener que modificar las definiciones existentes de clase). Modifique la función `tick` para llamar la función `nextDay` si la hora es incrementado al día siguiente. Modifique la función `printStandard` y `printMilitary` para extraer la fecha en adición a la hora. Escriba un programa manejador para probar la nueva clase `DateAndTime`. Pruebe específicamente incrementando la hora para pasar al día siguiente.

17

Clases: Parte II

Objetivos

- Ser capaz de crear y destruir dinámicamente objetos.
- Ser capaz de especificar objetos constantes y funciones miembro constantes.
- Comprender el objeto de las funciones amigo y de las clases amigo.
- Comprender el uso de los miembros de datos y de las funciones miembro estáticos.
- Comprender los varios tipos de clases contenedor.
- Ser capaz de desarrollar clases iteradoras que recorren los elementos de las clases contenedor.
- Comprender el uso del apuntador this.
- Ser capaz de crear y utilizar clases plantilla.

¿Pero qué, para servir a nuestros propios intereses, nos impide engañar a nuestros amigos?
Charles Churchill

En vez de esta absurda división de los sexos, deberían clasificar a las personas como estáticas y dinámicas.
Evelyn Waugh

Sobre todo lo siguiente: se veraz contigo mismo.
William Shakespeare
Hamlet

No tengas amigos que no sean tus iguales.
Confucius

Sinopsis

17.1 Introducción

En este capítulo continuamos nuestro estudio de las clases y de la abstracción de datos. Analizamos muchos temas más avanzados y colocamos los cimientos para el análisis de la homonimia de clases y de operadores del capítulo 18. El análisis de los capítulos 16 al 18 fomenta el uso de los objetos por parte de los programadores. Después, los capítulos 19 y 20 introducen herencia y polimorfismo las técnicas de la verdadera programación orientada a objetos.

17.2 Objetos constantes y funciones miembro const

Repetidamente hemos enfatizado el *principio del mínimo privilegio* como uno de los principios más fundamentales de una buena ingeniería de software. Veamos una forma en la cual este principio se aplica a los objetos.

Algunos objetos necesitan ser modificables y otros no. El programador puede utilizar la palabra reservada **const** para indicar que un objeto no es modificable, y que será un error cualquier intento de modificarlo. Por ejemplo,

```
const Time noon(12, 0, 0);
```


declara un objeto **const** de nombre **noon** de la clase **Time** y lo inicializa a las 12 de medio día.

Observación de ingeniería de software 17.1

*Declarar un objeto como **const** ayuda a que se cumpla el principio del mínimo privilegio. Cualquier intento accidental de modificar dicho objeto será detectado en tiempo de compilación, en vez de causar errores en tiempo de ejecución.*

Los compiladores C++ respetan de forma tan rígida las declaraciones **const**, que para objetos **const** no permiten de ninguna manera llamadas de función miembro (algunos compiladores sólo emiten una advertencia). Esto es duro, porque es probable que los clientes del objeto desearán utilizar varias funciones miembro "get" con dicho objeto. Para ello, el programador pudiera declarar funciones miembro **const**; sólo éstas pueden operar sobre objetos **const**. Naturalmente, las funciones miembro **const** no pueden modificar el objeto.

Error común de programación 17.1

*Definir como **const** una función miembro que modifique un miembro de datos de un objeto.*

Error común de programación 17.2

*Definir como **const** una función miembro que llama a una función miembro no **const**.*

Error común de programación 17.3

*Llamar a una función miembro no **const** en relación con un objeto **const**.*

Error común de programación 17.4

*Intentar modificar un objeto **const**.*

Una función es definida como **const** *tanto* en su declaración como en su definición, insertando la palabra reservada **const** después de la lista de parámetros de la función, y, en el caso de la definición de función, antes de la llave izquierda que inicia el cuerpo de la función. Por ejemplo, la función miembro

```
int getValue() const {return privateDataMember};
```

que solo regresa el valor de uno de los miembros de datos del objeto, es declarada como **const**. Si una función miembro **const** se define fuera de la definición de la clase, tanto la declaración como la definición de la función miembro deberán incluir **const**.

Observación de ingeniería de software 17.2

*Una función miembro **const** puede ser homónima en una versión no **const**. El compilador selecciona de forma automática la función miembro homónima basándose en si el objeto ha sido declarado **const** o no.*

Aquí se presenta un problema interesante tratándose de constructores y destructores, los cuales ciertamente modifican los objetos. Para constructores y destructores de objetos **const** no se requiere la declaración **const**. A un constructor debe permitírsele poder modificar un objeto de forma tal que el objeto pueda ser inicializado de manera correcta. Un destructor debe ser capaz de ejecutar sus trabajos de terminación, antes de que un objeto sea destruido.

El programa de la figura 17.1 ejemplifica un objeto constante de la clase **Time** e intenta modificar el objeto mediante las funciones miembro no constantes **setHour**, **setMinute** y **setSecond**. Las advertencias generadas por el compilador Borland C++ aparecen en la ventana de salida. El compilador fue ajustado de forma que no compile, si se produjese cualquier mensaje de advertencia.

Práctica sana de programación 17.1

Declare como **const** *todas las funciones miembro que se pretenda utilizar con objetos* **const**.

Un objeto **const** no puede ser modificado por asignación, por lo que deberá ser inicializado. Cuando un objeto miembro es **const**, un *inicializador de miembro* deberá ser utilizado para proporcionar al constructor con valores iniciales correspondientes al objeto. La figura 17.2 demuestra los usos de la sintaxis del inicializador de miembro para inicializar el miembro de datos **const** de nombre **increment** de la clase **Increment**. El constructor correspondiente a **Increment** se modifica como sigue:

```
// TIME5.H
// Declaration of the class Time.
// Member functions defined in TIME5.CPP

#ifndef TIME5_H
#define TIME5_H

class Time {
public:
   Time(int = 0, int = 0, int = 0);  // default constructor

   // set functions
   void setTime(int, int, int);  // set time
   void setHour(int);      // set hour
   void setMinute(int);    // set minute
   void setSecond(int);    // set second

   // get functions (normally declared const)
   int getHour() const;    // return hour
   int getMinute() const; // return minute
   int getSecond() const; // return second

   // print functions (normally declared const)
   void printMilitary() const;  // print military time
   void printStandard() const;  // print standard time
private:
   int hour;               // 0 - 23
   int minute;             // 0 - 59
   int second;             // 0 - 59
};

#endif
```

Fig. 17.1 Cómo utilizar una clase **Time** con objetos **const** y funciones miembro ***const*** (parte 1 de 4).


```
// TIME5.CPP
// Member function definitions for Time class.
#include "time5.h"
#include <iostream.h>

// Constructor function to initialize private data.
// Default values are 0 (see class definition).
Time::Time(int hr, int min, int sec)
{
   hour = (hr >= 0 && hr < 24) ? hr : 0;
   minute = (min >= 0 && min < 60) ? min : 0;
   second = (sec >= 0 && sec < 60) ? sec : 0;
}

// Set the values of hour, minute, and second.
void Time::setTime(int h, int m, int s)
{
   hour = (h >= 0 && h < 24) ? h : 0;
   minute = (m >= 0 && m < 60) ? m : 0;
   second = (s >= 0 && s < 60) ? s : 0;
}

// Set the hour value
void Time::setHour(int h)
   { hour = (h >= 0 && h < 24) ? h : 0; }

// Set the minute value
void Time::setMinute(int m)
   { minute = (m >= 0 && m < 60) ? m : 0; }

// Set the second value
void Time::setSecond(int s)
   { second = (s >= 0 && s < 60) ? s : 0; }

// Get the hour value
int Time::getHour() const { return hour; }

// Get the minute value
int Time::getMinute() const { return minute; }

// Get the second value
int Time::getSecond() const { return second; }

// Display military format time: HH:MM:SS
void Time::printMilitary() const
{
   cout << (hour < 10 ? "0" : "") << hour << ":"
        << (minute < 10 ? "0" : "") << minute << ":"
        << (second < 10 ? "0" : "") << second;
}
```

Fig. 17.1 Cómo utilizar una clase **Time** con objetos **const** y funciones miembro **const** (parte 2 de 4).

```
// Display standard format time: HH:MM:SS AM (or PM)
void Time::printStandard() const
{
   cout << ((hour == 12) ? 12 : hour % 12) << ":"
        << (minute < 10 ? "0" : "") << minute << ":"
        << (second < 10 ? "0" : "") << second
        << (hour < 12 ? " AM" : " PM");
}
```

Fig. 17.1 Cómo utilizar una clase **Time** con objetos **const** y funciones miembro **const** (parte 3 de 4).

```
// FIG17_1.CPP
// Attempting to access a const object with
// non-const member functions.
#include <iostream.h>
#include "time5.h"

main()
{
   const Time t(19, 33, 52);    // constant object

   t.setHour(12);     // ERROR: non-const member function
   t.setMinute(20);   // ERROR: non-const member function
   t.setSecond(39);   // ERROR: non-const member function

   return 0;
}
```

```
Compiling FIG17_1.CPP:
Warning FIG17_1.CPP: Non-const function
   Time::setHour(int) called for const object
Warning FIG17_1.CPP: Non-const function
   Time::setMinute(int) called for const object
Warning FIG17_1.CPP: Non-const function
   Time::setSecond(int) called for const object
```

Fig. 17.1 Cómo utilizar una clase **Time** con objetos **const** y funciones miembro **const** (parte 4 de 4).

```
Increment::Increment(int c, int i)
   : increment(i)
{ count = c; }
```

La notación **: increment(i)** hace que **increment** sea inicializado al valor i. Si se requieren de varios inicializadores de miembro, simplemente inclúyalos después de los dos puntos en una lista separada por comas.


```
// FIG17_2.CPP
// Using a member initializer to initialize a
// constant of a built-in data type.

#include <iostream.h>

class Increment {
public:
   Increment(int c = 0, int i = 1);
   void addIncrement() { count += increment; }
   void print() const;
private:
   int count;
   const int increment;          // const data member
};

// Constructor for class Increment
Increment::Increment(int c, int i)
   : increment(i)    // Member initializer
{ count = c; }

// Print the data
void Increment::print() const
{
   cout << "count = " << count
        << ", increment = " << increment << endl;
}

main()
{
   Increment value(10, 5);

   cout << "Before incrementing: ";
   value.print();

   for (int j = 1; j <= 3; j++) {
      value.addIncrement();
      cout << "After increment " << j << ": ";
      value.print();
   }

   return 0;
}
```

```
Before incrementing: count = 10, increment = 5
After increment 1: count = 15, increment = 5
After increment 2: count = 20, increment = 5
After increment 3: count = 25, increment = 5
```

Fig. 17.2 Cómo utilizar un inicializador de miembro para inicializar una constante de un tipo de datos incorporado.

En la figura 17.3 se ilustran los errores de compilador para un programa que intenta inicializar **increment** mediante un enunciado de asignación, en vez de mediante un inicializador de miembro.

Error común de programación 17.5

No proporcionar un inicializador de miembro para un objeto miembro **const**.

Observación de ingeniería de software 17.3

Tanto los objetos **const** *como las "variables"* **const** *necesitan ser inicializadas con sintaxis de inicializador miembro. Las asignaciones no son permitidas.*

17.3 Composición: clases como miembros de otras clases

Un objeto de clase **AlarmClock** necesita saber cuándo se supone debe sonar su alarma, por lo que, ¿por qué no incluir un objeto **Time** como miembro del objeto **AlarmClock**? Tal capacidad se llama *composición*. Una clase puede tener otras clases como miembros.

Observación de ingeniería de software 17.4

Una forma de reutilización del software es la composición, en la cual una clase tiene como miembros objetos de otras clases.

Cuando un objeto entra en alcance, su constructor es llamado automáticamente, por lo que es preciso especificar cómo se pasan argumentos a constructores de objetos miembro. Se construyen los objetos miembro, antes de que los objetos de clase que los incluyen sean construidos.

Observación de ingeniería de software 17.5

Si un objeto tiene varios objetos miembro, está indefinido el orden en el cual los objetos miembro serán construidos. No escriba código que dependa de que los constructores de objetos miembro ejecuten su trabajo en un orden específico.

En la figura 17.4 la clase **Employee** y la clase **Date** se utilizan para demostrar objetos como miembros de otros objetos. La clase **Employee** contiene los miembros de datos privados **lastName**, **firstName**, **birthDate**, y **hireDate**. Los miembros **birthDate** y **hireDate** son objetos de la clase **Date** que contiene los miembros de datos privados **month**, **day** y **year**. El programa produce un objeto **Employee**, e inicializa y muestra sus miembros de datos. Note la sintaxis del encabezado de función en la definición de constructor **Employee**:

```
Employee::Employee(char *fname, char *lname,
                   int bmonth, int bday, int byear,
                   int hmonth, int hday, int hyear)
   : birthDate(bmonth, bday, byear),
     hireDate(hmonth, hday, hyear)
```

El constructor toma ocho argumentos (**fname**, **lname**, **bmonth**, **bday**, **byear**, **hmonth**, **hday** y **hyear**). Los puntos en el encabezado separan los inicializadores de miembro de la lista de parámetros. Los inicializadores de miembro definen los argumentos **Employee** pasados a los constructores de los objetos miembro. Entonces, **bmonth**, **bday** y **byear** son pasados al constructor **birthDate**, y **hmonth**, **hday** y **hyear** son pasados al constructor **hireDate**. Varios inicializadores miembro son separados por comas.


```
// FIG17_3.CPP
// Attempting to initialize a constant of
// a built-in data type with an assignment.
#include <iostream.h>

class Increment {
public:
   Increment(int c = 0, int i = 1);
   void addIncrement() { count += increment; }
   void print() const;
private:
   int count;
   const int increment;
};

// Constructor for class Increment
Increment::Increment(int c, int i)
{              // Constant member 'increment' is not initialized
   count = c;
   increment = i;  // ERROR: Cannot modify a const object
}

// Print the data
void Increment::print() const
{
   cout << "count = " << count
        << ", increment = " << increment << '\n';
}

main()
{
   Increment value(10, 5);

   cout << "Before incrementing: ";
   value.print();

   for (int j = 1; j <= 3; j++) {
      value.addIncrement();
      cout << "After increment " << j << ": ";
      value.print();
   }

   return 0;
}
```

```
Compiling FIG17_3.CPP:
Warning FIG17_3.CPP 18: Constant member 'increment' is
   not initialized
Error FIG17_3.CPP 20: Cannot modify a const object
Warning FIG17_3.CPP 21: Parameter 'i' is never used
```

Fig. 17.3 Intento erróneo de inicializar una constante de un tipo de datos incorporado mediante asignación.

```
// DATE1.H
// Declaration of the Date class.
// Member functions defined in DATE1.CPP
#ifndef DATE1_H
#define DATE1_H

class Date {
public:
   Date(int = 1, int = 1, int = 1900);  // default constructor
   void print() const;  // print date in month/day/year format

private:
   int month;  // 1-12
   int day;    // 1-31 based on month
   int year;   // any year

   // utility function to test proper day for month and year
   int checkDay(int);
};

#endif
```

Fig. 17.4 Cómo utilizar los inicializadores objeto miembro (parte 1 de 5).

Un objeto miembro no necesita comenzar mediante un inicializador miembro. Si no se proporciona inicializador miembro, se llamará automáticamente al constructor por omisión del objeto miembro. Los valores, si es que hay alguno, que hayan sido definidos por el constructor por omisión, podrán ser pasados por alto mediante funciones set.

Error común de programación 17.6

No proporcionar un constructor por omisión para un objeto miembro, cuando no se proporciona inicializador miembro para dicho objeto miembro. Esto puede dar como resultado un objeto miembro no inicializado.

Sugerencia de rendimiento 17.1

Inicialice de forma explícita los objetos miembro mediante inicializadores miembro. Esto elimina la sobrecarga de una doble inicialización de objetos miembro una vez cuando se llame al constructor por omisión del objeto miembro, y una segunda vez cuando se utilicen las funciones set para inicializar dicho objeto miembro.

17.4 Funciones amigo y clases amigo

Una *función amigo* de una clase se define por fuera del alcance de dicha clase, pero aún así tiene el derecho de acceso a los miembros **private** (y como veremos en el capítulo 19, "Herencia", a los miembros **protected**) de la clase. Se puede declarar una función o toda una clase como un **friend** de otra clase.

Para declarar una función como un **friend** de una clase, en la definición de clase preceda el prototipo de función con la palabra reservada **friend**. Para declarar la clase **ClassTwo** como amigo de la clase **ClassOne**, prepare una declaración de la forma

```
friend ClassTwo;
```

como un miembro de la clase **ClassOne**.


```
// DATE1.CPP
// Member function definitions for Date class.

#include <iostream.h>
#include "date1.h"


// Constructor: Confirm proper value for month;
// call utility function checkDay to confirm proper
// value for day.
Date::Date(int mn, int dy, int yr)
{
   month = (mn > 0 && mn <= 12) ? mn : 1;  // validate
   year = yr;                              // could also check
   day = checkDay(dy);                     // validate

   cout << "Date object constructor for date ";
   print();
   cout << endl;
}

// Utility function to confirm proper day value
// based on month and year.
int Date::checkDay(int testDay)
{
   static int daysPerMonth[13] = {0, 31, 28, 31, 30,
                                  31, 30, 31, 31, 30,
                                  31, 30, 31};


   if (month != 2) {

      if (testDay > 0 && testDay <= daysPerMonth[month])
         return testDay;

   }
   else {  // February: Check for possible leap year
      int days = (year % 400 == 0 ||
                  (year % 4 == 0 && year % 100 != 0)) ? 29 : 28);

      if (testDay > 0 && testDay <= days)
         return testDay;
   }

   cout << "Day " << testDay << " invalid. Set to day 1.\n";
   return 1;  // leave object in consistent state if bad value
}


// Print Date object in form  month/day/year
void Date::print() const
   { cout << month << '/' << day << '/' << year; }
```

Fig. 17.4 Cómo utilizar inicializadores objeto miembro (parte 2 de 5).

```
// EMPLY1.H
// Declaration of the Employee class.
// Member functions defined in EMPLY1.CPP
#ifndef EMPLY1_H
#define EMPLY1_H

#include "date1.h"

class Employee {
public:
   Employee(char *, char *, int, int, int, int, int, int);
   void print() const;
private:
   char lastName[25];
   char firstName[25];
   Date birthDate;
   Date hireDate;
};

#endif
```

Fig. 17.4 Cómo utilizar inicializadores objeto miembro (parte 3 de 5).

```
// EMPLY1.CPP
// Member function definitions for Employee class.
#include <iostream.h>
#include <string.h>
#include "emply1.h"
#include "date1.h"

Employee::Employee(char *fname, char *lname,
                   int bmonth, int bday, int byear,
                   int hmonth, int hday, int hyear)
   : birthDate(bmonth, bday, byear),
     hireDate(hmonth, hday, hyear)
{
   strncpy(firstName, fname, 24);
   firstName[24] = '\0';
   strncpy(lastName, lname, 24);
   lastName[24] = '\0';
   cout << "Employee object constructor: "
        << firstName << ' ' << lastName << endl;
}

void Employee::print() const
{
   cout << lastName << ", " << firstName << "\nHired: ";
   hireDate.print();
   cout << "  Birthday: ";
   birthDate.print();
   cout << endl;
}
```

Fig. 17.4 Cómo usar inicializadores objeto miembro (parte 4 de 5).


```
// FIG17_4.CPP
// Demonstrating an object with a member object.
#include <iostream.h>
#include "emply1.h"

main()
{
   Employee e("Bob", "Jones", 7, 24, 49, 3, 12, 88);

   cout << endl;
   e.print();

   cout << "\nTest Date constructor with invalid values:\n";
   Date d(14, 35, 94);  // invalid Date values

   return 0;
}
```

```
Date object constructor for date 7/24/49
Date object constructor for date 3/12/88
Employee object constructor: Bob Jones

Jones, Bob
Hired: 3/12/88  Birthday: 7/24/49

Test Date constructor with invalid values:
Day 35 invalid. Set to day 1.
Date object constructor for date 1/1/94
```

Fig. 17.4 Cómo utilizar los inicializadores objeto miembro (parte 5 de 5).

Observación de ingeniería de software 17.6

Los conceptos de acceso de miembros correspondientes a `private`, `protected` *y* `public` *no tienen relación con las declaraciones de amistad, por lo que las declaraciones de amistad pueden ser colocadas en cualquier parte de la definición de clase.*

Práctica sana de programación 17.2

Coloque en la clase todas las definiciones de amistad en primer término, después del encabezado de clase, y no las anteceda con ningún especificador de acceso de miembros.

La amistad es concedida, y no tomada, es decir; para que la clase B sea un amigo de la clase A, la clase A debe declarar que la clase B es su amigo. También, la amistad no es ni simétrica ni transitiva, es decir, si la clase A es un amigo de la clase B, y la clase B es un amigo de la clase C, usted no puede inferir que la clase B sea un amigo de la clase A, que la clase C sea un amigo de la clase B o que la clase A sea un amigo de la clase C.

Observación de ingeniería de software 17.7

Algunas personas en la comunidad OOP (programación orientada a objetos) sienten que la "amistad" corrompe el ocultamiento de la información y debilita el valor del enfoque de diseño orientado a objetos.

El programa de la figura 17.5 demuestra la declaración y el uso de la función amigo **setX**, para definir el miembro de datos privado **x** de la clase **count**. Note que en la declaración de clase la declaración **friend** aparece primero (por regla convencional), inclusive antes que las funciones miembro públicas sean declaradas. El programa de la figura 17.6 demuestra los mensajes producidos por el compilador cuando es llamada la función no amigo **cannotSetX**, para modificar el miembro de datos privado **x**.

Es posible especificar funciones homónimas como amigos de una clase. Cada función homónima que se desea como un amigo, debe ser declarada en forma explícita en la definición de clase como amigo de la clase.

```
// FIG17_5.CPP
// Friends can access private members of a class.
#include <iostream.h>

// Modified Count class
class Count {
   friend void setX(Count &, int);  // friend declaration
public:
   Count() { x = 0; }               // constructor
   void print() const { cout << x << endl; }  // output
private:
   int x;  // data member
};

// Can modify private data of Count because
// setX is declared as a friend function of Count
void setX(Count &c, int val)
{
   c.x = val;  // legal: setX is a friend of Count
}

main()
{
   Count object;

   cout << "object.x after instantiation: ";
   object.print();
   cout << "object.x after call to setX friend function: ";
   setX(object, 8);  // set x with a friend
   object.print();

   return 0;
}
```

```
object.x after instantiation: 0
object.x after call to setX friend function: 8
```

Fig. 17.5 Los amigos pueden tener acceso a los miembros privados de una clase.


```
// FIG17_6.CPP
// Non-friend/non-member functions cannot access
// private data of a class.
#include <iostream.h>

// Modified Count class
class Count {
public:
   Count() { x = 0; }                    // constructor
   void print() const { cout << x << '\n'; }  // output
private:
   int x;  // data member
};

// Function tries to modify private data of Count,
// but cannot because it is not a friend of Count.
void cannotSetX(Count &c, int val)
{
   c.x = val;  // ERROR: 'Count::x' is not accessible
}

main()
{
   Count object;

   cannotSetX(object, 3); // cannotSetX is not a friend

   return 0;
}
```

```
Compiling FIG17_6.CPP:
Error FIG17_6.CPP 17: 'Count::x' is not accessible
Warning FIG17_6.CPP 18: Parameter 'c' is never used
Warning FIG17_6.CPP 18: Parameter 'val' is never used
```

Fig. 17.6 Funciones no amigas/no miembro no pueden tener acceso a los miembros privados de una clase.

17.5 Cómo utilizar el apuntador this

Cuando una función miembro referencia otro miembro de una clase en relación con un objeto específico de dicha clase, ¿Cómo C++ se asegura que es referenciado el objeto correcto? La respuesta es que cada objeto mantiene un apuntador a sí mismo llamado el *apuntador* ***this*** que es un argumento implícito en todas las referencias a miembros incluidos dentro de dicho objeto. El apuntador **this** puede también ser utilizado en forma explícita. Mediante el uso de la palabra reservada **this**, cada objeto puede determinar su propia dirección.

El apuntador **this** es utilizado de manera implícita para referenciar tanto los miembros de datos como las funciones miembro de un objeto. El tipo de este apuntador **this** depende del tipo del objeto y de si es declarada **const** la función miembro en la cual **this** es utilizado. En una función miembro no constante de la clase **Employee** el apuntador **this** tiene el tipo **Employee * const** (un apuntador constante a un objeto **Employee**). En una función miembro constante

de la clase **Employee** el apuntador **this** tiene el tipo **Employee const * const** (un apuntador constante a un objeto **Employee** constante).

Por ahora, presentamos un ejemplo sencillo de cómo utilizar de manera explícita el apuntador **this**; más adelante mostraremos algunos ejemplos substanciales y sutiles del uso de **this**. Cualquier función miembro tiene acceso al apuntador **this** al objeto para el cual está siendo invocado el miembro.

Sugerencia de rendimiento 17.2

Por razones de economía de almacenamiento, existe sólo una copia de cada función miembro por clase, y esta función miembro es invocada para todos los objetos de dicha clase. Por otra parte, cada objeto tiene su propia copia de los miembros de datos de la clase.

En la figura 17.7 se demuestra el uso explícito del apuntador **this** para permitir que una función miembro de la clase **Test** imprima los datos privados **x** de un objeto **Test**. El programa utiliza tanto el operador de flecha (->) sobre el apuntador **this**, como el operador de punto (.) sobre el apuntador **this** desreferenciado.

```
// FIG17_7.CPP
// Using the this pointer to refer to object members.
#include <iostream.h>

class Test {
public:
   Test(int = 0);
   void print() const;
private:
   int x;
};

Test::Test(int a) { x = a; }    // constructor

void Test::print() const
{
   cout << "        x = " << x
        << "\n  this->x = " << this->x
        << "\n(*this).x = " << (*this).x << '\n';
}

main()
{
   Test a(12);

   a.print();

   return 0;
}
```

```
        x = 12
  this-> = 12
(*this).x = 12
```

Fig. 17.7 Cómo utilizar el apuntador **this**.

Note los paréntesis que encierran a ***this** cuando se utiliza con el operador de selección de miembro punto (.). Los paréntesis son necesarios, porque el operador punto tiene una precedencia más alta que el operador *. Sin los paréntesis, la expresión

```
*this.x
```

sería evaluada como si tuviera los paréntesis como sigue:

```
*(this.x)
```

Esta expresión sería tratada por el compilador C++ como un error de sintaxis, porque el operador de selección de miembro no puede ser utilizado con un apuntador.

Error común de programación 17.7

Intentar utilizar el operador de selección de miembro (.) con un apuntador a un objeto (el operador de selección de miembro sólo puede ser utilizado con un objeto o con una referencia a un objeto).

Un uso interesante del apuntador **this**, es impedir que un objeto sea asignado a sí mismo. Como veremos en el capítulo 18, "Homonimia de operadores", la autoasignación puede causar errores serios cuando los objetos contengan apuntadores a almacenamientos asignados de forma dinámica mediante el operador **new**.

En la figura 17.8 se ilustra regresar una referencia a un objeto **Time** para permitir llamadas de función miembro de la clase **Time**, para concatenarlas. Cada una de las funciones miembro **Time** de nombre **setTime**, **setHour**, **setMinute** y **setSecond** regresan ***this** con un tipo de regreso **Time &**.

¿Porqué es válida la técnica de regresar ***this** como una referencia? El operador punto (.) se asocia de izquierda a derecha, por lo que la expresión

```
t.setHour(18).setMinute(30).setSecond(22);
```

primero evalúa **t.setHour(18)** y a continuación, regresa una referencia al objeto **t** como el valor de esta llamada de función. El resto de la expresión es entonces interpretado como

```
t.setMinute(30).setSecond(22);
```

La llamada **t.setMinute(30)** es ejecutada, regresando el equivalente de **t**. El resto de la expresión se interpreta como

```
t.setSecond(22);
```

Note que las llamadas

```
t.setTime(20, 20, 20).printStandard();
```

también utilizan la característica de concatenación. En esta expresión, estas llamadas deberán aparecer en este orden, porque **printStandard**, como está definida en la clase, no regresa una referencia a **t**. Colocar, en el enunciado anterior, la llamada a **printStandard** antes de la llamada a **setTime**, resultaría en un error de sintaxis.

```
// TIME6.H
// Declaration of class Time.
// Member functions defined in TIME6.CPP

#ifndef TIME6_H
#define TIME6_H

class Time {
public:
   Time(int = 0, int = 0, int = 0);  // default constructor

   // set functions
   Time &setTime(int, int, int);  // set hour, minute, second
   Time &setHour(int);     // set hour
   Time &setMinute(int);   // set minute
   Time &setSecond(int);   // set second

   // get functions (normally declared const)
   int getHour() const;    // return hour
   int getMinute() const; // return minute
   int getSecond() const; // return second

   // print functions (normally declared const)
   void printMilitary() const;  // print military time
   void printStandard() const;  // print standard time
private:
   int hour;               // 0 - 23
   int minute;             // 0 - 59
   int second;             // 0 - 59
};

#endif
```

Fig. 17.8 Cómo encadenar llamadas de función miembro (parte 1 de 4).

```
// TIME6.CPP
// Member function definitions for Time class.

#include "time6.h"
#include <iostream.h>

// Constructor function to initialize private data.
// Default values are 0 (see class definition).
Time::Time(int hr, int min, int sec)
{
   hour = (hr >= 0 && hr < 24) ? hr : 0;
   minute = (min >= 0 && min < 60) ? min : 0;
   second = (sec >= 0 && sec < 60) ? sec : 0;
}
```

Fig. 17.8 Cómo encadenar llamadas de función miembro (parte 2 de 4).


```
// Set the values of hour, minute, and second.
Time &Time::setTime(int h, int m, int s)
{
   hour = (h >= 0 && h < 24) ? h : 0;
   minute = (m >= 0 && m < 60) ? m : 0;
   second = (s >= 0 && s < 60) ? s : 0;
   return *this;   // enables chaining
}

// Set the hour value
Time &Time::setHour(int h)
{
   hour = (h >= 0 && h < 24) ? h : 0;
   return *this;   // enables chaining
}

// Set the minute value
Time &Time::setMinute(int m)
{
   minute = (m >= 0 && m < 60) ? m : 0;
   return *this;   // enables chaining
}

// Set the second value
Time &Time::setSecond(int s)
{
   second = (s >= 0 && s < 60) ? s : 0;
   return *this;   // enables chaining
}

// Get the hour value
int Time::getHour() const { return hour; }

// Get the minute value
int Time::getMinute() const { return minute; }

// Get the second value
int Time::getSecond() const { return second; }

// Display military format time: HH:MM:SS
void Time::printMilitary() const
{
   cout << (hour < 10 ? "0" : "") << hour << ":"
        << (minute < 10 ? "0" : "") << minute << ":"
        << (second < 10 ? "0" : "") << second;
}

// Display standard format time: HH:MM:SS AM (or PM)
void Time::printStandard() const
{
   cout << ((hour == 12) ? 12 : hour % 12) << ":"
        << (minute < 10 ? "0" : "") << minute << ":"
        << (second < 10 ? "0" : "") << second
        << (hour < 12 ? " AM" : " PM");
}
```

Fig. 17.8 Cómo encadenar llamadas de función miembro (parte 3 de 4).

```
// FIG17_8.CPP
// Chaining member function calls together
// with the this pointer
#include <iostream.h>
#include "time6.h"

main()
{
   Time t;

   t.setHour(18).setMinute(30).setSecond(22);
   cout << "Military time: ";
   t.printMilitary();
   cout << "\nStandard time: ";
   t.printStandard();

   cout << "\n\nNew standard time: ";
   t.setTime(20, 20, 20).printStandard();
   cout << endl;

   return 0;
}
```

```
Military time: 18:30:22
Standard time: 6:30:22 PM

New standard time: 8:20:20 PM
```

Fig. 17.8 Cómo encadenar llamadas de función miembro (parte 4 de 4).

17.6 Asignación dinámica de memoria mediante los operadores new y delete

Los operadores ***new*** y ***delete*** ofrecen una mejor forma de efectuar la asignación dinámica de memoria, que mediante las llamadas de función **malloc** y **free** de C. Considere el código siguiente

```
TypeName *typeNamePtr;
```

En ANSI C, para crear dinámicamente un objeto del tipo **TypeName**, usted diría

```
typeNamePtr = malloc(sizeof(TypeName));
```

Esto requiere una llamada de función a **malloc** y una referencia explícita al operador **sizeof**. En versiones de C anteriores a ANSI C, también tendría que haber hecho una conversión explícita (cast) del apuntador regresado por **malloc**, utilizando el cast (**TypeName ***). La función **malloc** no tiene ningún procedimiento para inicializar el bloque de memoria asignado. En C++, simplemente usted escribe

```
typeNamePtr = new TypeName;
```

El operador **new** crea en forma automática un objeto del tamaño apropiado, llama el constructor para el objeto (si existe uno disponible) y regresa un apuntador del tipo correcto. Si **new** no puede

encontrar espacio, regresa un apuntador **0**. Para liberar espacio para este objeto en C++, utilice el operador **delete**, como sigue:

```
delete typeNamePtr;
```

C++ le permite incluir un *inicializador* para un objeto recién creado como en:

```
float *thingPtr = new float (3.14159);
```

el cual inicializa a **3.14159** un objeto **float** recién creado.

Se puede crear un arreglo y asignarlo a **int * chessBoardPtr** como sigue:

```
chessBoardPtr = new int[8][8];
```

Este arreglo puede ser borrado mediante el enunciado

```
delete [] chessBoardPtr;
```

Como veremos, utilizar **new** y **delete**, en vez de **malloc** y **free**, también ofrece otros beneficios. En particular, **new** de manera automática invoca al constructor, y **delete** automáticamente invoca al destructor de la clase.

Error común de programación 17.8

*Mezclar asignación dinámica de memoria del tipo **new** y **delete** con asignación dinámica de memoria del tipo **malloc** y **free**: el espacio creado por **malloc** no podrá ser liberado por **delete**; los objetos creados mediante **new** no podrán ser borrados por **free**.*

Práctica sana de programación 17.3

*A pesar de que los programas C++ pueden contener almacenamiento creado por **malloc** y borrados por **free**, y objetos creados por **new** y borrados por **delete**, lo mejor es utilizar solo **new** y **delete**.*

17.7 Miembros de clase estáticos

Por lo regular, cada objeto de una clase tiene su propia copia de todos los miembros de datos de la clase. Esto a veces resulta un desperdicio. En ciertos casos todos los miembros de una clase deberían compartir una copia de un miembro de datos particular. Para esta y otras razones se utiliza un miembro de datos estático. Un miembro de datos estático representa información "aplicable a toda la clase". La declaración de un miembro estático empieza con la palabra reservada **static.**

Sugerencia de rendimiento 17.3

Cuando una copia de los datos sea suficiente, utilice miembros de datos estáticos a fin de ahorrar almacenamiento.

Aunque los miembros de datos estáticos pudieran parecer variables globales, los miembros de datos estáticos tienen alcance de clase. Los miembros estáticos pueden ser públicos, privados o protegidos. Los miembros de datos estáticos deben ser inicializados en alcance de archivo. Los miembros de clase estáticos públicos son accesibles a través de cualquier objeto de dicha clase o mediante el operador de resolución de alcance binario, se puede tener acceso a ellos a través del nombre de la clase. Para tener acceso a los miembros de clase estáticos privados y protegidos se deben utilizar las funciones miembro públicas de la clase. Los miembros de clase estáticos existen aun cuando no existan objetos de dicha clase. Para tener acceso a un miembro de clase estático público cuando no existan objetos de dicha clase, sólo coloque el nombre de clase y el operador de resolución de alcance binario como prefijo al miembro de datos. Para tener acceso a un miembro

de clase estático privado o protegido, cuando no existan objetos de dicha clase, deberá ser incluida una función miembro estática pública y la función deberá ser llamada mediante el uso de un prefijo en su nombre con el nombre de la clase junto con el operador de resolución de alcance binario.

El programa de la figura 17.9 demuestra el uso del miembro de datos **static** privado y de la función miembro **static** pública. El miembro de datos **count** es inicializado a cero en alcance de archivo, mediante el enunciado

```
int Employee::count = 0;
```

El miembro de datos **count** lleva cuenta del número de objetos de la clase **Employee** que han sido producidos. Cuando existan objetos de la clase **Employee**, el miembro **count** puede ser referenciado a través de cualquier función miembro de un objeto **Employee** —en este ejemplo, **count** es referenciado tanto por el constructor como por el destructor. Cuando no existan objetos de la clase **Employee**, aun así el miembro **count** se puede referenciar, pero sólo mediante una llamada a la función miembro estática **getCount**, como sigue:

```
Employee::getCount()
```

En este ejemplo, se utiliza la función **getCount** para determinar el número de objetos **Employee** en la actualidad producidos. Note que cuando no exista ningún objeto producido, la llamada de función ya citada será emitida. Sin embargo, se podrá llamar a la función **getCount** a través de uno de los objeto cuando existan objetos producidos, como en

```
e1Ptr->getCount()
```

```
// EMPLOY1.H
// An employee class
#ifndef EMPLOY1_H
#define EMPLOY1_H

class Employee {
public:
   Employee(const char*, const char*);  // constructor
   ~Employee();                 // destructor
   char *getFirstName() const;  // return first name
   char *getLastName() const;   // return last name

   // static member function
   static int getCount();  // return # objects instantiated
private:
   char *firstName;
   char *lastName;

   // static data member
   static int count;  // number of objects instantiated
};

#endif
```

Fig. 17.9 Cómo utilizar un miembro de datos estático para llevar cuenta del número de objetos de una clase (parte 1 de 5).


```
// EMPLOY1.CPP
// Member functions definitions for class Employee
#include <iostream.h>
#include <string.h>
#include <assert.h>
#include "employ1.h"

// Initialize the static data member
int Employee::count = 0;

// Define the static member function that
// returns the number of employee objects instantiated.
int Employee::getCount() { return count; }

// Constructor dynamically allocates space for the
// first and last name and uses strcpy to copy
// the first and last names into the object
Employee::Employee(const char *first, const char *last)
{
   firstName = new char[ strlen(first) + 1 ];
   assert(firstName != 0);   // ensure memory allocated
   strcpy(firstName, first);

   lastName = new char[ strlen(last) + 1 ];
   assert(lastName != 0);    // ensure memory allocated
   strcpy(lastName, last);

   ++count;  // increment static count of employees
   cout << "Employee constructor for " << firstName
        << ' ' << lastName << " called.\n";
}

// Destructor deallocates dynamically allocated memory
Employee::~Employee()
{
   cout << "~Employee() called for " << firstName
        << ' ' << lastName << endl;
   delete [] firstName;  // recapture memory
   delete [] lastName;   // recapture memory
   --count;  // decrement static count of employees
}

// Return first name of employee
char *Employee::getFirstName() const
{
   char *tempPtr = new char[strlen(firstName) + 1];
   assert(tempPtr != 0);   // ensure memory allocated
   strcpy(tempPtr, firstName);
   return tempPtr;
}
```

Fig. 17.9 Cómo utilizar un miembro de datos estático para llevar cuenta del número de objetos de una clase (parte 2 de 5).

```
// Return last name of employee
char *Employee::getLastName() const
{
   char *tempPtr = new char[strlen(lastName) + 1];
   assert(tempPtr != 0);   // ensure memory allocated
   strcpy(tempPtr, lastName);
   return tempPtr;
}
```

Fig. 17.9 Cómo utilizar un miembro de datos estático para llevar cuenta del número de objetos de una clase (parte 3 de 5).

```
// FIG17_9.CPP
// Driver to test the employee class
#include <iostream.h>
#include "employ1.h"

main()
{
   cout << "Number of employees before instantiation is "
        << Employee::getCount() << endl;   // use class name

   Employee *e1Ptr = new Employee("Susan", "Baker");
   Employee *e2Ptr = new Employee("Robert", "Jones");

   cout << "Number of employees after instantiation is "
        << e1Ptr->getCount() << endl;

   cout << "\nEmployee 1: "
        << e1Ptr->getFirstName()
        << " " << e1Ptr->getLastName()
        << "\nEmployee 2: "
        << e2Ptr->getFirstName()
        << " " << e2Ptr->getLastName() << "\n\n";

   delete e1Ptr;   // recapture memory
   delete e2Ptr;   // recapture memory

   cout << "Number of employees after deletion is "
        << Employee::getCount() << endl;

   return 0;
}
```

Fig. 17.9 Cómo utilizar un miembro de datos estático para llevar cuenta del número de objetos de una clase (parte 4 de 5).

Una función miembro puede ser declarada **static** si no tiene acceso a miembros de clase no estáticos. A diferencia de las funciones miembro no estáticas, una función miembro estática no tiene un apuntador **this**, porque los miembros de datos estáticos y las funciones miembro estáticos existen en forma independiente a cualquier objeto de la clase.


```
Number of employees before instantiation is 0
Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.
Number of employees after instantiation is 2

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() calle for Susan Baker
~Employee() calle for Robert Jones
Number of employees after deletion is 0
```

Fig. 17.9 Cómo utilizar un miembro de datos estático para llevar cuenta del número de objetos de una clase (parte 5 de 5).

Error común de programación 17.9

Referirse al apuntador **this** *desde dentro de una función miembro estática.*

Error común de programación 17.10

Declarar una función miembro estática **const**.

Observación de ingeniería de software 17.8

Los miembros de datos estáticos y las funciones miembro estáticas existen y pueden ser utilizados, aún si no se han producido objetos de dicha clase.

17.8 Abstracción de datos y ocultamiento de información

Las clases ocultan por lo regular sus detalles de puesta en práctica a los clientes (es decir, usuarios) de las clases. Esto se conoce como ocultamiento de la información. Por consiguiente, por ejemplo, el programador puede crear un clase de pila y ocultar a sus usuarios la puesta en práctica de la pila. Mediante arreglos o listas enlazadas las pilas pueden ser puestas en práctica en forma fácil. Un cliente de una clase de pila no necesita saber cómo ha sido puesta en práctica la misma. Lo que en realidad interesa al usuario es que cuándo en la pila se coloquen elementos de datos que éstos serán recuperados en orden de últimas entradas, primeras salidas (LIFO). Este concepto se conoce como *abstracción de datos*, y las clases de C++ definen *tipos de datos abstractos (ADT,* por *abstract data types)*. Aunque los usuarios pudieran conocer los detalles de cómo se pone en práctica una clase, los usuarios no deben escribir código que dependa de dichos detalles. Esto significa que una clase particular (como una que ponga en práctica una pila, junto con sus operaciones **push** y **pop**), puede ser reemplazada por otra versión, sin afectar al resto del sistema, siempre y cuando la interfaz pública de dicha clase no cambie.

La tarea de un lenguaje de alto nivel es crear una presentación conveniente para uso de los programadores. No existe una forma aceptada de presentación y esta es una razón por la cual existen tantos lenguajes de programación. La programación orientada a objetos de C++, presenta sólo otra alternativa.

La mayor parte de los lenguajes de programación enfatizan acciones. En estos lenguajes, los datos existen en apoyo de acciones que los programas necesitan llevar a cabo. De todas formas, los datos son "menos interesantes" que las acciones. Los datos son "corrientes". Sólo existen unos

cuantos tipos de datos incorporados, y para los programadores resulta difícil crear sus propios tipos de datos.

En C++ y en el estilo de programación orientado a objetos, esta panorámica cambia. C++ aumenta la importancia de los datos. En C++ la actividad primordial es la creación de nuevos tipos de datos (es decir clases) y expresar las interacciones entre objetos de dichos tipos de datos.

Para ir en esa dirección, la comunidad de los lenguajes de programación necesitaba formalizar algunos conceptos relacionados con los datos. La formalización a la que nos estamos refiriendo es el concepto de tipo de datos abstracto (ADT por abstract data type). ADT recibe tanta atención, hoy en día, como tuvo durante las dos últimas décadas la programación estructurada. ADT no sustituye la programación estructurada. Más bien, provee una formalización adicional, que puede mejorar aún más el proceso de desarrollo de programas.

¿Qué es un tipo de datos abstracto? Veamos el tipo incorporado `int`. Lo que viene a la mente es el concepto de un entero en matemáticas, pero en una computadora `int` no es precisamente lo que es un entero en matemáticas. En particular, por lo regular los `int` de computadora son muy limitados en tamaño. Por ejemplo, en una máquina de 32 bits `int` pudiera estar limitado al rango desde -2 mil millones hasta +2 mil millones. Si el resultado de un cálculo cae fuera de este rango, ocurrirá un error y la máquina responderá de alguna manera dependiente de la máquina. Los enteros matemáticos no tienen este problema. Por lo tanto, el concepto de un `int` de computadora es solamente una aproximación al concepto de un entero del mundo real. Lo mismo es cierto tratándose de `float`.

Inclusive `char` es también una aproximación. A menudo los valores del tipo `char` son patrones de 8 bits, que no se parecen en nada a los caracteres que se supone deben representar como una `Z`, una `z`, un signo de dólares (`$`), un dígito (por ejemplo, `5`), y así en lo sucesivo. En la mayor parte de las computadoras los valores del tipo `char` están bastante limitados, en comparación con el rango de los caracteres del mundo real. El conjunto de caracteres ASCII de 7 bits proporciona 127 distintos valores de caracteres. Esto resulta en totalidad inadecuado para representar lenguajes como el japonés y el chino, que requieren de miles de caracteres.

El punto es que inclusive los tipos de datos incorporados, proporcionados en lenguajes de programación como C++, son sólo aproximaciones o modelos de conceptos y comportamientos del mundo real. Hasta este momento hemos aceptado como es a `int`, pero ahora el lector tiene que pensar con una nueva perspectiva. Los tipos como `int`, `float`, `char` y otros son todos ejemplos de *tipos de datos abstractos*. Son en esencia formas de representar conceptos del mundo real, a cierto nivel satisfactorio de precisión, dentro de un sistema de computación.

Un tipo de datos abstracto de hecho captura dos conceptos, es decir una representación de datos, así como aquellas operaciones permitidas sobre dichos datos. Por ejemplo, en C++, el concepto de `int` define la adición, substracción, multiplicación, división y operaciones de módulo, pero queda indefinida la división entre cero; y estas operaciones permitidas se llevan a cabo de una forma que resulta dependiente de los parámetros de la máquina, como son el tamaño de la palabra fija del sistema de computación subyacente. En C++, para poner en práctica tipos de datos abstractos, el programador utiliza las clases.

17.8.1 Ejemplo: tipo de datos abstracto de arreglo

En el capítulo 6, analizamos los arreglos. Un arreglo no es mucho más que un apuntador. Si el programador es cuidadoso, esta capacidad primitiva es aceptable para efectuar operaciones de arreglos. Existen muchas operaciones que sería bueno ejecutar con arreglos, pero que en C++ no están incorporadas. Con C++, el programador puede desarrollar un arreglo ADT, que es preferible a los arreglos "en bruto". La clase arreglo puede proveer muchas nuevas capacidades como

- Verificación del rango de los subíndices.
- Un rango arbitrario de subíndices.
- Asignación de arreglo.
- Comparación de arreglo.
- Entrada/salida de arreglo.
- Los arreglos conocen su tamaño.

La debilidad aquí es que estamos creando un tipo de datos personalizado, no estándar, que es poco probable que esté disponible precisamente de esa forma, en la mayor parte de las puestas en práctica de C++. El uso de la programación orientada a objetos y de C++ está aumentando con rapidez. Resulta crucial que la profesión de la programación propugne por una normalización y distribución a gran escala de bibliotecas de clases, para poder aprovechar todo el potencial de la orientación a objetos.

C++ tiene un pequeño conjunto de tipos incorporados. ADT amplía el lenguaje de programación base.

Observación de ingeniería de software 17.9

El programador puede crear nuevos tipos mediante el uso del mecanismo de clases. Estos nuevos tipos pueden ser diseñados para ser usados tan convenientemente como los tipos incorporados. Por lo tanto, C++ resulta un lenguaje extensible. A pesar de que mediante estos nuevos tipos el lenguaje es fácil de ampliar, el lenguaje base mismo no es modificable.

Los nuevos ADT creados en entornos C++ pueden ser propiedad de un individuo, de pequeños grupos o de empresas. Los ADT también pueden ser colocados en bibliotecas de clases estándar destinadas a una amplia difusión. Esto no necesariamente promueve la normalización, aunque de hecho es probable que emergerán estándares. El valor pleno de C++ únicamente será comprendido cuando bibliotecas de clase sustanciales y estándar resulten ampliamente disponibles. Un proceso institucional debe ser instaurado para alentar el desarrollo de bibliotecas estándar. En los Estados Unidos, a menudo dicha normalización ocurre a través de ANSI, el American National Standards Institute. ANSI está desarrollando actualmente una versión estándar de C++. Independiente de cómo aparezcan estas bibliotecas en última instancia, el lector que aprenda C++ y programación orientada a objetos, estará listo para rápidamente aprovechar los nuevos tipos de desarrollo de software orientado a componentes, que serán posibles con las bibliotecas de ADT.

17.8.2 Ejemplo: tipo de datos abstracto de cadena

Intencionalmente C++ es un lenguaje escueto, que proporciona a los programadores sólo aquellas capacidades fundamentales necesarias para elaborar un amplio rango de sistemas. Los diseñadores del lenguaje no deseaban crear sobrecargas de rendimiento. Su objetivo era que C++ fuese apropiado para la programación de sistemas, que demandan que los programas se ejecuten con eficiencia. Es cierto, entre los tipos incorporados de C++, hubiera sido posible incluir un tipo de cadena, pero en vez de ello los diseñadores optaron incluir un mecanismo mediante clases para la creación y puesta en práctica de dichos ADT. En el capítulo 18 desarrollaremos nuestro propio ADT para cadenas.

17.8.3 Ejemplo: tipo de datos abstracto de cola

Cada uno de nosotros de vez en cuando tiene que esperar en fila. Una fila de espera también se llama una *cola*. Esperamos en fila en la caja de salida de un supermercado, esperamos en fila para

obtener gasolina, esperamos en fila para subirnos al autobús, esperamos en fila en la autopista para pagar el peaje, y los estudiantes saben demasiado bien sobre tener que esperar en fila durante la época de registro para obtener las materias que desean. Los sistemas de cómputo utilizan de forma interna muchas filas de espera, y necesitamos escribir programas que simulen lo que las colas hacen y son.

Una cola es un bonito ejemplo de un tipo de datos abstracto. Una cola ofrece a sus clientes un comportamiento bien comprendido. Los clientes ponen cosas en una cola uno a la vez utilizando una operación *enqueue*, y los clientes obtienen estas cosas de regreso sobre demanda uno a la vez utilizando la operación *dequeue*. Conceptualmente, una cola puede hacerse muy larga. Una cola real, por lo regular, es finita. Los elementos de una cola son devueltos en un orden de *primeras entradas, primeras salidas (FIFO por first-in, first-out)*.

La cola oculta una representación de datos internos, que de alguna forma lleva control de los elementos actualmente esperando en fila, y ofrece un conjunto de operaciones a sus clientes, es decir *enqueue* y *dequeue*. Los clientes no tienen que preocuparse respecto a la puesta en práctica de la cola. Los clientes solo desean que la cola funcione "como anunciado". Cuando un cliente pone un nuevo elemento en cola, ésta deberá aceptar dicho elemento y colocarlo internamente en algún tipo de estructura de primeras entradas, primeras salidas. Cuando el cliente desee el siguiente elemento de la parte frontal de la cola, la cola deberá quitar el elemento de su representación interna y entregar el elemento al mundo exterior en un orden fijo, es decir, el elemento que ha estado por más tiempo en la cola, deberá ser el siguiente que, en la siguiente operación *dequeue*, será devuelto.

El ADT de la cola garantiza la integridad de su estructura interna de datos. Los clientes no pueden manipular en directo esta estructura de datos. Solo el ADT de la cola tiene acceso a sus datos internos. Los clientes pueden causar sólo operaciones permitidas, para ejecutarse sobre la representación de los datos; las operaciones no contempladas en la interfaz pública del ADT son rechazadas por el ADT de alguna forma apropiada. Esto podría significar la emisión de un mensaje de error, la terminación de la ejecución o sólo ignorar la solicitud de operación.

17.9 Clases contenedor e iteradores

Entre los tipos más populares de clases son las *clases contenedor* (también llamadas *clases de colección*), es decir, clases diseñadas para contener colecciones de objetos. Las clases contenedor proporcionan por lo común servicios como es inserción, borrado, búsqueda, clasificación, prueba de un elemento verificando membresía dentro de la clase, y otras similares. Los arreglos y las listas enlazadas son ejemplos de clases contenedor.

Es común asociar *objetos de iterador* o solo *iteradores* con la clase de colección. Un iterador es un objeto que regresa el siguiente elemento de una colección. Una vez que se ha escrito un iterador para una clase, obtener el siguiente elemento de la clase puede ser expresado en forma simple. Los iteradores se escriben como amigos de las clases, a través de las cuales hacen la iteración. De la misma forma que un libro se comparte entre varias personas y podría tener a la vez varios marcatextos, una clase contenedor puede tener varios iteradores operando sobre ella simultáneamente.

17.10 Clases plantilla

En el capítulo 15 analizamos funciones plantilla y cómo ayudan a la reutilización del software. Cerramos este capítulo con un análisis de las *clases plantilla*.

Es posible comprender que una pila es algo independiente del tipo de elementos que se colocan dentro de la pila. Pero cuando se trata de hecho programar una pila, se debe proporcionar un tipo

de datos. Esto crea una maravillosa oportunidad para la reutilización del software. Lo que necesitamos es un medio de describir el concepto genérico de pila y de producir clases que sean copias de esta clase genérica, pero de tipo específico. Esta es la capacidad proporcionada en C++ por las clases plantilla.

Observación de ingeniería de software 17.10

Las clases plantilla fomentan la reutilización del software.

Las clases plantilla se pusieron a disposición con el compilador de C++ de AT&T Versión 3, por lo que son una adición relativa reciente al lenguaje.

Las clases plantilla a menudo se conocen como *tipos parametrizados,* porque requieren de uno o más parámetros de tipo para especificar cómo personalizar la especificación de plantilla genérica.

El programador que desee utilizar clases plantilla, sólo escribe una definición de clase plantilla genérica. Cada vez que el programador necesite una nueva clase, el programador utiliza una notación concisa y simple, y el compilador escribe el código fuente para la clase que especifica el programador. Una clase plantilla de pila, por ejemplo podría entonces convertirse en base para crear muchas clases de pilas (como sería "una pila de `float`", "una pila de `int`", "una pila de `char`", etcétera) utilizadas en un programa.

Note la definición de la clase plantilla de pila en la figura 17.10. Se parece a una definición de clase convencional, excepto que está precedida por el encabezado

```
template <class T>
```

para indicar que se trata de una definición de clase plantilla que recibe un parámetro de tipo `T` indicando el tipo de la clase de pila que el programador desea crear. El tipo de elemento a almacenarse en esta pila se menciona sólo genéricamente como `T` a todo lo largo del encabezado de clase de pila y de las definiciones de función miembro.

```
// TSTACK1.H
// Simple template class Stack

#ifndef TSTACK1_H
#define TSTACK1_H

template <class T>
class Stack {
public:
   Stack(int = 10);       // Constructor with default size 10
   ~Stack() { delete [] stackPtr; } // Destructor
   int push(const T&);    // Push an element onto the stack
   int pop(T&);           // Pop an element off the stack
   int isEmpty() const { return top == -1; } // 1 if empty
   int isFull() const { return top == size - 1; } // 1 if full
private:
   int size;              // # of elements in the stack
   int top;               // location of the top element
   T *stackPtr;           // pointer to the stack
};
```

Fig. 17.10 Definición de la clase plantilla `Stack` (parte 1 de 4).

```
// Constructor with default size 10
template <class T>
Stack<T>::Stack(int s)
{
   size = s;
   top = -1;  // Empty stack
   stackPtr = new T[size];
}

// Push an element onto the stack
// return 1 if successful, 0 otherwise
template <class T>
int Stack<T>::push(const T &item)
{
   if (!isFull()) {
      stackPtr[++top] = item;
      return 1;  // push successful
   }
   return 0;     // push unsuccessful
}

// Pop an element off the stack
template <class T>
int Stack<T>::pop(T &popValue)
{
   if (!isEmpty()) {
      popValue = stackPtr[top--];
      return 1;  // pop successful
   }
   return 0;     // pop unsuccessful
}

#endif
```

Fig. 17.10 Definición de la clase plantilla **Stack** (parte 2 de 4).

Ahora veamos el manejador de prueba (la función **main**) correspondiente a la clase plantilla de pila. El manejador empieza produciendo el objeto **floatStack** del tamaño 5. Este objeto se declara ser de la clase **Stack<float>**. El compilador asocia el tipo **float** con el tipo parametrizado **T** en la plantilla para producir el código fuente para una clase de pila del tipo **float**.

El manejador a continuación inserta (push) los valores **float** 1.1, 2.2, 3.3, 4.4 y 5.5 sobre **floatStack**. El ciclo de inserción termina cuando el manejador intenta insertar un sexto valor sobre **floatStack** (mismo que ya está lleno porque fue creado con 5 elementos).

Ahora el manejador extrae (pop) los 5 valores de la pila (en orden LIFO). El manejador intenta extraer un sexto valor, pero **floatStack** está vacío, por lo que el ciclo de extracción termina.

A continuación, el manejador produce una pila de enteros con la declaración

```
Stack<int>    intStack;
```

Dado que no se especifica tamaño, el tamaño por omisión será de 10, tal y como se especifica en el constructor por omisión. Otra vez, el manejador cicla la inserción de valores sobre **intStack**, hasta que éste queda lleno, y a continuación cicla la extracción de valores de **intStack** hasta que queda vacío.


```
// FIG17_10.CPP
// Test driver for Stack template

#include <iostream.h>
#include "tstack1.h"

main()
{
   Stack<float> floatStack(5);
   float f = 1.1;
   cout << "Pushing elements onto floatStack\n";

   while (floatStack.push(f)) { // success (1 returned)
      cout << f << ' ';
      f += 1.1;
   }

   cout << "\nStack is full. Cannot push " << f
        << "\n\nPopping elements from floatStack" << endl;

   while (floatStack.pop(f))  // success (1 returned)
      cout << f << ' ';

   cout << "\nStack is empty. Cannot pop" << endl;

   Stack<int> intStack;
   int i = 1;
   cout << "\nPushing elements onto intStack\n";

   while (intStack.push(i)) { // success (1 returned)
      cout << i << ' ';
      i++;
   }

   cout << "\nStack is full. Cannot push " << i
        << "\n\nPopping elements from intStack" << endl;

   while (intStack.pop(i))  // success (1 returned)
      cout << i << ' ';

   cout << "\nStack is empty. Cannot pop" << endl;
   return 0;
}
```

Fig. 17.10 Manejador para la clase plantilla **Stack** (parte 3 de 4).

Las definiciones de función miembro por fuera del encabezado de clase plantilla cada una de ellas empieza con el encabezado

```
template <class T>
```

Entonces cada definición se parece a una definición convencional, a excepción que el tipo de elemento de pila siempre se lista genéricamente como del tipo parametrizado **T**. Como siempre,

```
Pushing elements onto floatStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from floatStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

Fig. 17.10 Manejador para la clase plantilla `Stack` (parte 4 de 4).

se utiliza el operador de resolución de alcance binario para unir cada definición de función miembro al alcance correcto de la clase. En este caso, el nombre de clase es **`Stack<T>`**. Cuando **`floatStack`** es producido para ser del tipo **`Stack<float>`**, para representar la pila el constructor **`Stack`** crea un arreglo de elementos del tipo **`T`**. El tipo **`float`** sustituye a **`T`** y el enunciado

```
stackPtr = new T[size];
```

es generado por el compilador en la versión **`Stackat<float>`**, como

```
stackPtr = new float[size];
```

Los tipos parametrizados (es decir, las plantillas) pueden tomar más de un parámetro, como en

```
template <class T, class S, class R
```

Cada parámetro debe estar precedido por la palabra reservada **`class`**.

Resumen

- La palabra reservada **`const`** puede ser utilizada para especificar que un objeto no es modificable y que cualquier intento de modificarlo es un error.
- El compilador de C++ no permite llamadas de función miembro no **`const`** para un objeto **`const`**.
- Las funciones miembro **`const`** no pueden modificar un objeto.
- Una función se especifica como **`const`** tanto en su declaración como en su definición.
- Una función miembro **`const`** puede ser homónima con una versión no **`const`**. El compilador escogerá qué función miembro homónima utilizar en forma automática, basado en si el objeto ha sido declarado **`const`** o no.
- Un objeto **`const`** debe ser inicializado.

- Cuando una clase contenga objetos miembro **`const`**, los inicializadores miembros deben ser proporcionados al constructor.
- Las clases pueden estar compuestas de objetos de otras clases.
- Los objetos miembros son construidos antes de que sean construidos sus objetos de clase que los encierran.
- Si un inicializador miembro no es proporcionado para un objeto miembro, se llamará al constructor por omisión del objeto miembro.
- Una función amigo de una clase es una función definida por fuera de dicha clase y tiene derecho de acceso a miembros **`private`** y **`protected`** de la clase.
- Las declaraciones de amistad pueden ser colocadas en cualquier parte de la definición de clase.
- Cada función homónima que se pretende sea un amigo debe ser declarada de forma explícita como amigo de la clase.
- Cada objeto mantiene un apuntador a sí mismo, llamado apuntador **`this`**, que se convierte en un argumento implícito en todas las referencias a miembros dentro de dicho objeto. Este apuntador **`this`** es utilizado de forma implícita para referenciar tanto las funciones miembro como los miembros de datos de un objeto.
- Cada objeto puede determinar su propia dirección utilizando la palabra reservada **`this`**.
- El apuntador **`this`** pude ser utilizado explícitamente, pero se utiliza de manera implícita más a menudo.
- El operador **`new`** crea en forma automática un objeto del tamaño correcto, y regresa un apuntador del tipo correcto. En C++, para liberar el espacio correspondiente a este objeto, usted utiliza el operador **`delete`**.
- Un miembro de datos estático representa información "accesible a toda la clase". La declaración de un miembro estático empieza con la palabra reservada **`static`**.
- Los miembros de datos estáticos tienen alcance de clase.
- Los miembros estáticos de una clase son accesibles a través de un objeto de dicha clase, o pueden ser accesibles a través del nombre de la clase mediante el uso del operador de resolución de alcance.
- Una función miembro puede ser declarada estática, si no tiene acceso a miembros de clase no estáticos. A diferencia de las funciones miembro no estáticas, una función miembro estática no tiene apuntador **`this`**. Esto es debido a que los miembros de datos estáticos y las funciones miembro estáticas existen en forma independiente de cualquiera de los objetos de una clase.
- Las clases plantilla proporcionan una forma genérica de describir el concepto de una clase y de producir clases que son copias de esta clase genérica, pero son de tipo específico.
- Las clases plantilla a menudo se llaman tipos parametrizados, porque requieren de uno o más parámetros de tipo para especificar cómo personalizar la especificación de plantilla genérica.
- Las definiciones de clases plantilla empiezan con la línea

  ```
  template <class T>
  ```

  en la línea que antecede a la definición de clase. Puede existir más de un tipo parametrizado. Si es así, estarán separados por comas y cada tipo estará precedido por la palabra reservada **`class`**.

- Una clase plantilla se produce especificando el tipo de la clase que sigue al nombre de la clase, como sigue:

```
ClassName<type> objectName;
```

El compilador sustituirá `type` a todo lo largo de la definición de clase y de la definición de la función miembro correspondiente por el tipo parametrizado de la clase.

Terminología

operador de resolución de alcance binario (`::`)
llamadas de función de miembro encadenado
alcance de clase
composición
función miembro `const`
objeto `const`
constructor
clases contenedor
constructor por omisión
destructor por omisión
operador `delete`
operador `delete [ ]`
destructor
objetos dinámicos
extensibilidad
clase `friend`
función `friend`
iterador
operador de selección de miembro (`.`)
especificadores de acceso de miembro
inicializador de miembro
objeto miembro
constructor de objeto miembro
clase anidada
operador `new`
tipo parametrizado
operador selector de miembro de apuntador (`->`)
apuntador a miembro
principio de mínimo privilegio
miembro de datos estático
función miembro estática
clase plantilla
apuntador `this`

Errores comunes de programación

17.1 Definir como `const` una función miembro que modifique un miembro de datos de un objeto.
17.2 Definir como `const` una función miembro que llama a una función miembro no `const`.
17.3 Llamar a una función miembro no `const` en relación con un objeto `const`.
17.4 Intentar modificar un objeto `const`.
17.5 No proporcionar un inicializador de miembro para un objeto miembro `const`.
17.6 No proporcionar un constructor por omisión para un objeto miembro, cuando no se proporciona inicializador miembro para dicho objeto miembro. Esto puede dar como resultado un objeto miembro no inicializado.
17.7 Intentar utilizar el operador de selección de miembro (`.`) con un apuntador a un objeto (el operador de selección de miembro sólo puede ser utilizado con un objeto o con una referencia a un objeto).
17.8 Mezclar asignación dinámica de memoria del tipo `new` y `delete` con asignación dinámica de memoria del tipo `malloc` y `free`: el espacio creado por `malloc` no podrá ser liberado por `delete`; los objetos creados mediante `new` no podrán ser borrados por `free`.
17.9 Referirse al apuntador `this` desde dentro de una función miembro estática.
17.10 Declarar una función miembro estática `const`.

Prácticas sanas de programación

17.1 Declare como `const` todas las funciones miembro que se pretenda utilizar con objetos `const`.
17.2 Coloque en la clase todas las definiciones de amistad en primer término, después del encabezado de clase, y no las anteceda con ningún especificador de acceso de miembros.

17.3 A pesar de que los programas C++ pueden contener almacenamiento creado por `malloc` y borrados por `free`, y objetos creados por `new` y borrados por `delete`, lo mejor es utilizar solo `new` y `delete`.

Sugerencias de rendimiento

17.1 Inicialice de forma explícita los objetos miembro mediante inicializadores miembro. Esto elimina la sobrecarga de una doble inicialización de objetos miembro una vez cuando se llame al constructor por omisión del objeto miembro, y una segunda vez cuando se utilicen las funciones set para inicializar dicho objeto miembro.
17.2 Por razones de economía de almacenamiento, existe sólo una copia de cada función miembro por clase, y esta función miembro es invocada para todos los objetos de dicha clase. Por otra parte, cada objeto tiene su propia copia de los miembros de datos de la clase.
17.3 Cuando una copia de los datos sea suficiente, utilice miembros de datos estáticos a fin de ahorrar almacenamiento.

Observaciones de ingeniería de software

17.1 Declarar un objeto como `const` ayuda a que se cumpla el principio del mínimo privilegio. Cualquier intento accidental de modificar dicho objeto será detectado en tiempo de compilación, en vez de que causar errores en tiempo de ejecución.
17.2 Una función miembro `const` puede ser homónima en una versión no `const`. El compilador selecciona de forma automática la función miembro homónima basándose en el objeto que ha sido declarado `const` o no.
17.3 Tanto los objetos `const` como las "variables" `const` necesitan ser inicializadas con sintaxis de inicializador miembro. Las asignaciones no son permitidas.
17.4 Una forma de reutilización del software es la composición, en la cual una clase tiene como miembros objetos de otras clases.
17.5 Si un objeto tiene varios objetos miembro, está indefinido el orden en el cual los objetos miembro serán construidos. No escriba código que dependa de que los constructores de objetos miembro ejecuten su trabajo en un orden específico.
17.6 Los conceptos de acceso de miembros correspondientes a `private`, `protected` y `public` no tienen relación con las declaraciones de amistad, por lo que las declaraciones de amistad pueden ser colocadas en cualquier parte de la definición de clase.
17.7 Algunas personas en la comunidad OOP (Programación orientada a objetos) sienten que la "amistad" corrompe el ocultamiento de la información y debilita el valor del enfoque de diseño orientado a objetos.
17.8 Los miembros de datos estáticos y las funciones miembro estáticas existen y pueden ser utilizados, aun si no se han producido objetos de dicha clase.
17.9 El programador puede crear nuevos tipos mediante el uso del mecanismo de clases. Estos nuevos tipos pueden ser diseñados para ser usados tan conveniente como los tipos incorporados. Por lo tanto; C++ resulta un lenguaje extensible. A pesar de que mediante estos nuevos tipos el lenguaje es fácil de ampliar, el lenguaje base mismo no es modificable.
17.10 Las clases plantilla fomentan la reutilización del software.

Ejercicios de autoevaluación

17.1 Llene cada uno de los siguientes espacios en blanco:
a) Se utiliza sintaxis _____ para inicializar los miembros constantes de una clase.
b) En una clase debe declararse un _____ para tener acceso a los miembros de datos `private` de dicha clase.

c) El operador _____ asigna dinámicamente memoria para un objeto de un tipo especifico y regresa un _____ a dicho tipo.
d) Un objeto constante debe de ser _____; no puede ser modificado después de creado.
e) Un miembro de datos _____ representa información para todo el ámbito de la clase.
f) Todo objeto mantiene un apuntador a sí mismo llamado el apuntador _____.
g) _____ proporcionan un medio de describir el concepto genérico de una clase.
h) La palabra reservada _____ especifica que un objeto o una variable no es modificable.
i) Si no se proporciona un inicializador de miembro para un objeto miembro de una clase, se llama al _____ del objeto.
j) Una función miembro puede ser **static** si no tiene acceso a los miembros de clase_____.
k) Las funciones amigo pueden tener acceso a los miembros _____ y _____ de una clase.
l) Las clases plantilla se conocen a veces como tipos _____.
m) Los objetos miembro se construyen _____ que el objeto de clase que los contiene.
n) El operador _____ recupera memoria asignada previamente por **new**.

17.2 Encuentre el error en cada uno de los siguientes y explique cómo corregirlo.

a)
```
char *string;
string = new char[20];
free(string);
```

b)
```
class Example {
public:
   Example(int y = 10) { data = y; }
   int getIncrementedData() const { return ++data; }
   static int getCount()
   {
      cout << "Data is " << data << endl;
      return count;
   }
private:
   int data;
   static int getCount;
};
```

Respuestas a los ejercicios de autoevaluación

17.1 a) Inicializador de miembro. b) **friend** .c) **new**, apuntador. d) inicializado. e) **static**. f) **this**. g) clases plantilla. h) **const**. i) constructor por omisión. j) no **static**. k) **private, protected**. l) parametrizado. m) antes. n) **delete**.

17.2 a) Error: la memoria asignada dinámicamente por **new** es borrada por la función de biblioteca estándar de C **free**.
Corrección: utilice el operador **delete** de C++ para recuperar la memoria. La asignación dinámica de memoria de tipo C no debe de ser mezclada con los operadores de C++ **new** y **delete**.

b) Error: la definición de clase correspondiente a **Example** tiene dos errores. el primero ocurre en la función **getIncrementedData**. La función es declarada **const**, pero modifica el objeto.
Corrección: Para corregir el primer error, elimine la palabra reservada **const** de la definición de **getIncrementedData**.

Error: el segundo error ocurre en la función **getCount**. Esta función es declarada **static**, por lo que no se le permite acceso a cualquier miembro no **static** de la clase.

Corrección: para corregir el segundo error, elimine la línea de salida de la definición de **getCount**.

Ejercicios

17.3 Compare la asignación dinámica de memoria mediante los operadores de C++ **new** y **delete**, con la asignación dinámica de memoria utilizando las funciones de la biblioteca estándar de C **malloc** y **free**.

17.4 Explique el concepto de amistad en C++. Explique los aspectos negativos de la amistad, tal y como se describen en el texto.

17.5 ¿Puede una definición correcta de clase **Time** incluir ambos constructores siguientes?

```
Time (int h = 0, int m = 0, int s = 0);
Time();
```

17.6 ¿Qué ocurre cuando un tipo de regreso, inclusive el tipo **void**, es especificado para un constructor o destructor?

17.7 Cree una clase **Date** con las siguientes capacidades:

a) Extraer la fecha en formatos múltiples como

```
DDD YYYY
MM/DD/YY
June 14, 1992
```

b) Utilizar constructores homónimos para crear objetos **Date** inicializados con fechas de los formatos de la parte (a).
c) Crear un constructor **Date** que lea la fecha del sistema utilizando las funciones estándar de biblioteca del encabezado **time.h** y que defina los miembros **Date**.

En el capítulo 18, podremos crear operadores para probar la igualdad de dos fechas y para comparar fechas, a fin de determinar si una fecha es anterior a otra o posterior a otra.

17.8 Utilizando como guía el programa de procesamiento de lista de la figura 12.3, construya una clase **List**. La clase deberá proporcionar las mismas capacidades que el ejemplo. Escriba un programa manejador para probar por completo dicha clase.

17.9 Utilizando como guía el programa de procesamiento de colas de la figura 12.13, construya una clase **Queue**. La clase deberá proporcionar las mismas capacidades que el ejemplo. Escriba un programa manejador para probar completamente dicha clase.

17.10 Utilizando como punto de partida la clase **List** desarrollada en el ejercicio 17.8, cree una clase plantilla **List**, que sea capaz de producir objetos **List** que contengan muchos tipos de datos diferentes. Escriba un programa manejador para probar completamente la clase plantilla.

17.11 Utilizando como punto de partida la clase **Queue** desarrollada en el ejercicio 17.9, cree una clase plantilla **Queue** que sea capaz de producir objetos **Queue** que contengan muchos tipos de datos diferentes. Escriba un programa manejador para probar por completo la clase plantilla.

18

Homonimia de operadores

Objetivos

- Comprender cómo redefinir operadores para que funcionen con nuevos tipos.
- Comprender cómo convertir objetos de una clase a otra clase.
- Aprender cuándo efectuar, y cuándo no efectuar, la homonimia de operadores.
- Estudiar varias clases interesantes que utilizan operadores homónimos.

La diferencia entre construcción y creación es exactamente esto: que una cosa construida puede ser querida sólo después de construida; pero una cosa creada es querida antes de que exista.
Gilbert Keith Chesterton
Prefacio a Dickens, Pickwick Papers

El dado ha sido lanzado.
Julius Caesar

En realidad jamás nuestro doctor operaba, a menos de que fuese necesario. El era precisamente así. Si no necesitaba el dinero, no ponía una mano sobre usted.
Herb Shriner

Sinopsis

18.1 Introducción

En los capítulos 16 y 17, presentamos los fundamentos de las clases C++ y el concepto de tipos de datos abstractos (ADT). Las manipulaciones sobre los objetos de clase (es decir, sobre ejemplos de ADT) se llevaron a cabo enviando mensajes a los objetos (bajo forma de llamadas de funciones miembro). Para ciertos tipos de clases esta notación de llamada de función resulta engorrosa, en especial para clases matemáticas. Sería agradable, para estos tipos de clases, utilizar el extenso repertorio de operadores incorporados de C++ concebidos para especificar manipulaciones de objetos. En este capítulo, mostramos cómo habilitar a los operadores de C++ para funcionar con objetos de clase. Este proceso se conoce como homonimia de operador. Es un proceso directo y natural extender C++ con estas nuevas capacidades.

En C++ el operador `<<` es utilizado para varios fines como operador de inserción de flujos y como operador de desplazamiento a la izquierda. Este es un ejemplo de *homonimia de operador*. Similar, `>>` también tiene su homónimo; es utilizado tanto como operador de extracción de flujo como operador de desplazamiento a la derecha. En la biblioteca de clases de C++, ambos operadores tienen homónimos. El lenguaje de C++ mismo hace la homonimia de + y de —. Estos operadores funcionan en forma diferente dependiendo del contexto en aritmética de enteros, aritmética de punto flotante y aritmética de apuntadores.

En general, C++ le permite al programador hacer homónimos de la mayor parte de los operadores, haciéndolos sensibles al contexto dentro del cual son utilizados. El compilador generará el código apropiado, basándose en la forma en la cual el operador sea utilizado. Algunos operadores se hacen homónimos en forma frecuente, en especial el operador de asignación y varios

operadores aritméticos, como + y -. La tarea llevada a cabo por los operadores homónimos, también pueden ser ejecutada mediante llamadas de función explícitas, pero la notación de operador es por lo general más legible.

Analizaremos cuándo utilizar homonimia de operadores y cuando no utilizarla. Mostraremos cómo hacer la homonimia de operadores, y presentaremos muchos programas completos utilizando operadores homónimos.

18.2 Fundamentos de la homonimia de operadores

La programación en C++ es un proceso sensible a los tipos y enfocado a los tipos. Los programadores pueden utilizar tipos incorporados y pueden definir nuevos tipos. Los tipos incorporados pueden ser utilizados aprovechando la extensa colección de operadores de C++. Los operadores proporcionan a los programadores una forma de notación concisa para expresar manipulaciones de objetos de tipos incorporados.

Los programadores pueden también utilizar operadores con tipos definidos por usuario. Aunque C++ no permite que se creen operadores nuevos, sí permite que se haga la homonimia de algunos operadores existentes, de tal forma que estos operadores sean utilizados con objetos de clase, teniendo los operadores un significado apropiado a los nuevos tipos. Esta es una de las características más poderosas de C++. La homonimia de operadores contribuye a la extensibilidad de C++, lo que seguro es uno de los atributos más llamativos de este lenguaje.

Práctica sana de programación 18.1

Utilice la homonimia de operadores cuando ésta haga más claro un programa si efectúa las mismas operaciones mediante llamadas de función explícitas.

Práctica sana de programación 18.2

Evite un uso excesivo o inconsistente de la homonimia de operadores, ya que ello puede hacer que un programa resulte críptico o difícil de leer.

A pesar de que la homonimia de operadores pudiera sonar como una capacidad exótica, la mayor parte de los programadores utilizan de forma implícita operadores homónimos en forma regular. Por ejemplo, el operador más (+) opera de forma bien diferente sobre enteros, flotantes y dobles. Pero, con todo, más (+) funciona bien con variables del tipo `int`, `float`, `double` y varios otros tipos incorporados, porque en el lenguaje C++ mismo se hace la homonimia del operador más (+).

Se hace la homonimia de operadores escribiendo una definición de función (con encabezado y cuerpo) como usted lo haría normalmente, excepto que ahora el nombre de la función se convierte en la palabra clave `operator`, seguida por el símbolo correspondiente al operador homónimo. Por ejemplo, para hacer el homónimo del operador de adición se utilizaría el nombre de función `operator+`.

Para utilizar un operador sobre objetos de clase, dicho operador *deberá* ser un homónimo con dos excepciones. El operador de asignación (=) puede ser utilizado sin homónimo con cualquier clase. Cuando no se incluye un operador de asignación homónimo, el comportamiento por omisión es una *copia a nivel de miembro* de los miembros de datos de la clase. Pronto veremos que dicha copia a nivel de miembro por omisión es peligrosa, tratándose de clases con miembros que apunten a almacenamiento dinámicamente asignado; para dichas clases por lo regular haremos la homonimia del operador de asignación. El operador de dirección (&) también puede ser utilizado sin homonimia con objetos de cualquier clase; sólo devuelve la dirección del objeto en memoria. También se puede hacer la homonimia de este operador.

La homonimia es lo más apropiado para clases matemáticas. A menudo éstas requieren que un conjunto substancial de operadores sean homónimos, para asegurar paralelismo con la forma en que estas clases matemáticas son manejadas en el mundo real. Por ejemplo, para una clase de números complejos sería poco usual hacer la homonimia de sólo la adición, dado que con números complejos también son muy utilizados otros operadores aritméticos.

C++ es un lenguaje rico en operadores. Los programadores de C++ comprenden el significado y el contexto de cada operador. Por lo tanto, cuando en relación con nuevas clases se trata de hacer la homonimia de operadores, los programadores probablemente harán selecciones razonables.

El punto de la homonimia de operadores es proporcionar las mismas expresiones concisas para tipos definidos por usuario, que las que C++ proporciona con su rica colección de operadores, en relación con tipos incorporados. La homonimia de operadores, sin embargo, no es automática; para llevar a cabo las operaciones deseadas el programador deberá escribir funciones de homonimia de operadores. Algunas veces estas funciones se realizan mejor como funciones miembro; algunas veces se realizan mejor como funciones amigo; y en ocasiones pueden ser hechas como funciones no miembro, y no amigo.

Un uso equivocado de la homonimia sería hacer la homonimia del operador + para llevar a cabo operaciones de tipo substracción o hacer la homonimia del operador / para ejecutar operaciones de tipo multiplicación. Un uso como éste de la homonimia puede hacer un programa confuso.

Práctica sana de programación 18.3

Haga la homonimia de operadores para llevar a cabo la misma función o funciones muy similares sobre objetos de clase que dichos operadores ejecutan sobre objetos de tipos incorporados.

Práctica sana de programación 18.4

Antes de escribir programas en C++ utilizando operadores homónimos, consulte los manuales de C++ correspondientes a su compilador, para informarse de las varias restricciones y requisitos únicos a operadores particulares.

18.3 Restricciones sobre la homonimia de operadores

Se puede hacer la homonimia de la mayor parte de los operadores de C++. Estos se muestran en la figura 18.1. En la figura 18.2 se muestran los operadores que no pueden tener homónimos.

La precedencia de un operador no puede ser modificada por la homonimia. Esto puede llevar a situaciones incómodas en las cuales un operador tiene un homónimo, pero de forma tal, que su precedencia fija es inapropiada. Sin embargo, se pueden utilizar paréntesis dentro de una expresión, a fin de obligar el orden de evaluación de operadores homónimos.

Operadores que pueden tener homónimos

+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete

Fig. 18.1 Operadores que pueden tener homónimos.

Operadores que no pueden tener homónimos

.	.*	::	?:	sizeof

Fig. 18.2 Operadores que no pueden tener homónimos.

La asociatividad de un operador tampoco puede ser modificada por la homonimia. Para hacer la homonimia de un operador no pueden utilizarse argumentos por omisión.

No es posible modificar el número de operandos que toma un operador: los operadores unarios homónimos se conservan como unarios; los operadores binarios homónimos se conservan como binarios. El único operador ternario de C++, (?:), no puede tener homónimo. Los operadores &, *, + y - cada uno de ellos tienen versiones unaria y binaria; cada una de estas versiones unaria y binaria pueden tener homónimos en forma separada.

No es posible crear operadores nuevos; sólo se puede hacer la homonimia de operadores existentes. Esto impide que el programador utilice notaciones populares como el operador **, utilizado en BASIC para la exponenciación.

Error común de programación 18.1

Intentar crear operadores nuevos.

El significado del funcionamiento de un operador sobre objetos de tipos incorporados no puede ser modificado por la homonimia de operadores. El programador no puede, por ejemplo, cambiar cómo + añade dos enteros. La homonimia de operador funciona sólo con objetos de tipos definidos por el usuario o por una combinación de un objeto de tipo definido por el usuario y un objeto de tipo incorporado.

Error común de programación 18.2

Intentar modificar cómo funciona un operador con objetos de tipos incorporados.

Observación de ingeniería de software 18.1

En una función operador, por lo menos un argumento debe ser un objeto de clase o una referencia a un objeto de clase. Esto impedirá que los programadores modifiquen cómo funcionan los operadores sobre objetos de tipos incorporados.

Al hacer la homonimia de (), [], ->, o = la función de homonimia de operador deberá ser declarada como un miembro de clase. Para los demás operadores, las funciones de homonimia de operador pueden ser amigo.

La homonimia de un operador de asignación y de un operador de adición, para permitir enunciados como

```
object2 = object2 + object1
```

no implica que el operador += quede también de forma automática homónimo para permitir enunciados como

```
object2 += object1;
```

Sin embargo; este comportamiento puede ser conseguido haciendo la homonimia explícita del operador += correspondiente a dicha clase.

Error común de programación 18.3

Suponer que hacer la homonimia de un operador (como +) hace automáticamente la homonimia de sus operadores relacionados (como +=). Los operadores sólo pueden tener homónimos en forma explícita; la homonimia implícita no existe.

18.4 Funciones operador como miembros de clase en comparación con funciones amigo

Las funciones operador pueden ser funciones miembro o funciones no miembro; las funciones no miembro por lo regular son funciones amigo. Las funciones miembro utilizan el apuntador **this** en forma implícita, para obtener uno de sus argumentos de objeto de clase. En una llamada de función no miembro, este argumento de clase deberá estar listado en forma explícita.

Independiente que una función de operador esté puesta en práctica como función miembro o como función no miembro, aun así en expresiones, el operador será utilizado de la misma forma. Por lo tanto, ¿cuál de las puestas en práctica es la mejor?

Cuando una función operador es puesta en práctica como función miembro, el operando más a la izquierda (o único) deberá ser un objeto de clase (o una referencia a un objeto de clase) de la clase del operador. Si el operando izquierdo tiene que ser un objeto de clase diferente o de tipo incorporado, esta función de operador deberá ser puesta en práctica como función no miembro, exactamente como lo hicimos al hacer la homonimia de << y >> como operadores de inserción y de extracción de flujo, respectivamente. Una función operador puesta en práctica como una función no miembro necesita ser un amigo, si dicha función debe tener acceso directo a miembros privados o protegidos de dicha clase.

El operador homónimo << debe tener un operando izquierdo del tipo **ostream &** (como **cout** en la expresión **cout << classObject**), por lo que deberá ser función no miembro. De igual forma, el operador homónimo >> deberá tener un operando izquierdo del tipo **istream &** (como **cin** en la expresión **cin >> classObject**), de forma tal, que también él deberá de ser una función miembro. También, cada una de estas funciones de operador homónimas requiere tener acceso a los miembros de datos privados del objeto de clase que se está extrayendo o introduciendo, por lo que, por razones de rendimiento, esas funciones operador homónimas por lo regular se hacen funciones amigo de la clase.

Sugerencia de rendimiento 18.1

Es posible hacer la homonimia de un operador como función no miembro, y no amigo, pero dicha función con la necesidad de tener acceso a datos privados o protegidos de una clase requeriría utilizar las funciones set o get proporcionadas en la interfaz pública de la clase. La sobrecarga de llamar a estas funciones causaría bajo rendimiento.

Las funciones miembro operador son llamadas sólo cuando el operando izquierdo de un operador binario es de forma específica un objeto de dicha clase o cuando el único operando de un operador unario es un objeto de dicha clase.

Otra razón por la cual uno podría elegir una función no miembro para hacer la homonimia de un operador, sería para permitir que el operador fuera conmutativo. Por ejemplo, suponga que tenemos un objeto, **number**, del tipo **long int** y un objeto **bigInteger1**, de la clase **HugeInteger** (una clase en la cual los enteros pueden ser de manera arbitraria extensos, en vez de quedar limitados por el tamaño de palabra de máquina del hardware subyacente; en los

ejercicios del capítulo se desarrolla la clase **HugeInteger**). El operador de adición (+) produce un objeto temporal **HugeInteger** como la suma de un **HugeInteger** y de un **long int** (como en la expresión **bigInteger1 + number**), o como la suma de un **long int** y un **HugeInteger** (como en la expresión **number + bigInteger1**). Por lo tanto, necesitamos que el operador de adición sea conmutativo (exactamente como normalmente es). El problema estriba en que, si debe hacer la homonimia del operador como función miembro, el objeto de clase debe aparecer a la izquierda de la adición. Por lo tanto, hacemos la homonimia del operador como amigo, para permitir que **HugeInteger** aparezca a la derecha de la adición. La función **operator+**, que se ocupa de **HugeInteger** a la izquierda, aún puede ser una función miembro.

18.5 Cómo hacer la homonimia de operadores de inserción y de extracción de flujo

C++ tiene la capacidad de introducir y de extraer los tipos de datos estándar utilizando los operadores de extracción de flujo >> y de inserción de flujo <<. Se hace la homonimia de estos operadores (incluidos en las bibliotecas de clase proporcionadas con los compiladores C++) para procesar cada tipo de datos estándar, incluyendo cadenas y direcciones de memoria. Los operadores de inserción y de extracción de flujo también pueden tener homónimos para ejecutar entradas y salidas para tipos definidos por usuario. En la figura 18.3 se demuestra la homonimia de los operadores de extracción y de inserción de flujo para manejar datos de una clase de número de teléfono definida por usuario llamada **PhoneNumber**. Este programa supone que los números telefónicos están introducidos de forma correcta. Dejamos para los ejercicios la inclusión de la verificación de errores.

La función operador de extracción de flujo (**operator>>**) toma como argumentos una referencia **istream** (**input**) y una referencia (**num**) a un tipo definido por usuario (**PhoneNumber**), y devuelve una referencia **istream**. En la figura 18.3 se utiliza la función de operador **operator>>** para introducir números telefónicos de la forma

```
(800) 555-1212
```

en objetos de la clase **PhoneNumber**. Cuando el compilador ve en **main** la expresión

```
cin >> phone
```

genera la llamada de función

```
operator>>(cin, phone);
```

Cuando esta llamada se ejecuta, el parámetro **input** se convierte en un seudónimo para **cin** y el parámetro **num** se convierte en un seudónimo para **phone**. La función operador utiliza la función miembro **istream**, de nombre **getline**, para leer como cadenas las tres porciones del número telefónico a los miembros **areaCode**, **exchange** y **line** del objeto referenciado **PhoneNumber** (**num** en la función operador y **phone** en **main**). La función **getline** explica en detalle en el capítulo 21. Los caracteres paréntesis, espacio y guión son pasados por alto, llamando a la función miembro **istream**, de nombre **ignore**, que descarta el número especificado de caracteres del flujo de entrada (un carácter por omisión). La función **operator>>** devuelve la referencia **istream** de nombre **input** (es decir, **cin**). Esto permite que se *concatenen* operaciones de entrada sobre objetos **PhoneNumber** con operaciones de entrada sobre otros objetos **PhoneNumber** o sobre objetos de otros tipos de datos. Por ejemplo, se podrían introducir dos objetos **PhoneNumber** como sigue:

```
cin >> phone1 >> phone2;
```

```
// FIG18_3.CPP
// Overloading the stream-insertion and
// stream-extraction operators.
#include <iostream.h>

class PhoneNumber {
   friend ostream &operator<<(ostream &, const PhoneNumber &);
   friend istream &operator>>(istream &, PhoneNumber &);
private:
   char areaCode[4];  // 3-digit area code and null
   char exchange[4];  // 3-digit exchange and null
   char line[5];      // 4-digit line and null
};

// Overloaded stream insertion operator (cannot be
// a member function).
ostream &operator<<(ostream &output, const PhoneNumber &num)
{
   output << "(" << num.areaCode << ") "
          << num.exchange << "-" << num.line;

   return output;      // enables cout << a << b << c;
}

// overloaded stream extraction operator
istream &operator>>(istream &input, PhoneNumber &num)
{
   input.ignore();                    // skip (
   input.getline(num.areaCode, 4); // input area code
   input.ignore(2);                   // skip ) and space
   input.getline(num.exchange, 4); // input exchange
   input.ignore();                    // skip dash (-)
   input.getline(num.line, 5);     // input line

   return input;       // enables cin >> a >> b >> c;
}

main()
{
   PhoneNumber phone; // create object phone

   cout << "Enter a phone number in the "
        << "form (123) 456-7890:\n";

   // cin >> phone invokes operator>> function by
   // issuing the call operator>>(cin, phone).
   cin >> phone;

   // cout << phone invokes operator<< function by
   // issuing the call operator<<(cout, phone).
   cout << "The phone number entered was:\n"
        << phone << endl;
   return 0;
}
```

Fig. 18.3 Operadores de inserción y de extracción de flujo definidos por usuario (parte 1 de 2).


```
Enter a phone number in the form (123) 456-7890:
(800) 555-1212
The phone number entered was:
(800) 555-1212
```

Fig. 18.3 Operadores de inserción y de extracción de flujo definidos por usuario (parte 2 de 2).

Primero, se ejecutaría la expresión **cin >> phone1** mediante la llamada

```
operator>>(cin, phone1);
```

Esta llamada a continuación regresaría **cin** como el valor de **cin >> phone1**, por lo que la porción restante de la expresión se interpretaría solo como **cin >> phone2**.

El operador de inserción de flujo toma como argumentos una referencia **ostream** (**output**) y una referencia (**num**) a un tipo definido por usuario (**PhoneNumber**), y devuelve una referencia **ostream**. La función **operator<<** muestra objetos del tipo **PhoneNumber**.

Cuando en **main** el compilador ve la expresión

```
cout << phone
```

generará la llamada de función

```
operator<<(cout, phone);
```

La función **operator<<** muestra las porciones del número telefónico como cadenas, porque están almacenadas en formato de cadena (la función miembro **istream** de nombre **getline** almacena un carácter nulo después de que termina su entrada).

Note que las funciones **operator>>** y **operator<<** se declaran en **classPhoneNumber** como funciones no miembro, amigo. Estos operadores deben ser no miembro, porque en cada caso el objeto de la clase **PhoneNumber** aparece como el operando derecho del operador; a fin de hacer la homonimia del operador como función miembro —el operando de clase debe aparecer a la izquierda. Los operadores de entrada y de salida homónimos deben ser declarados como amigo, si han de tener acceso directo a los miembros de clase no públicos.

Observación de ingeniería de software 18.2

Se pueden añadir nuevas capacidades de entrada/salida a C++ correspondientes a tipos definidos por usuario, sin modificar las declaraciones o los miembros de datos privados correspondientes a la clase **ostream** *o a la clase* **istream**. *Esto promueve la extensibilidad del lenguaje de programación C++ uno de los aspectos más atractivos de C++.*

18.6 Homonimia de operadores unarios

Se puede hacer la homonimia de un operador unario para una clase como función miembro no estática sin argumentos, o como función no miembro con un argumento; dicho argumento debe ser un objeto de la clase o una referencia a un objeto de la clase.

Más adelante en este capítulo haremos la homonimia del operador unario ! para probar si una cadena está vacía. Al hacer la homonimia de un operador unario como !, en forma de función miembro no estática sin argumentos, si **s** es un objeto de clase, cuando el compilador vea la expresión **!s** generará la llamada **s.operator! ()**. El operando **s** es el objeto de clase s para el cual se está invocando la función miembro **operator!**:

```
class String {
public:
   int operator!() const;
   ...
};
```

Existan dos formas distintas en que se puede hacer la homonimia de un operador unario como ! en forma de función no miembro con un argumento —con un argumento que es un objeto o con un argumento que es una referencia a un objeto. Si **s** es un objeto de clase, entonces **!s** se tratará como si se hubiera escrito la llamada **operator! (s)**.

```
class String {
   friend int operator!(const String &);
   ...
};
```

Práctica sana de programación 18.5

Al hacer la homonimia de operadores unarios, es preferible hacer la funciones operador miembros de clase en vez de funciones amigo no miembro. Esta es una solución más limpia. Las funciones amigo y las clases amigo deberán evitarse, salvo que sean en lo absoluto necesarias. La utilización de amigos viola el encapsulado de una clase.

18.7 Homonimia de operadores binarios

Se puede hacer la homonimia de un operador binario como función miembro no estática con un argumento o como, función no miembro con dos argumentos (uno de dichos argumentos debiendo ser un objeto de clase o una referencia a un objeto de clase).

Más adelante en este capítulo, haremos la homonimia de **+=** para indicar concatenación de dos objetos de cadena. Al hacer la homonimia del operador binario **+=** como una función miembro no estática de una clase **String** con un argumento, si **y** y **z** son objetos de clase, entonces **y += z** será tratado como si se hubiera escrito **y.operator+=(z)**.

```
class String {
public:
   String &operator+=(const String &);
   ...
};
```

Se puede hacer la homonimia del operador binario **+=** como función no miembro con dos argumentos uno de los cuales debe ser un objeto de clase o una referencia a un objeto de clase. Si **y** y **z** son objetos de clase, entonces **y += z** será tratado como si dentro del programa la llamada **operator+=(y, z)** hubiera sido escrita


```
class String {
   friend String &operator+=(String &, const String &);
   ...
};
```

18.8 Estudio de caso: una clase Array

En C y en C++ la notación de arreglo es solo una alternativa a los apuntadores, por lo que los arreglos tienen mucho potencial para errores. Por ejemplo, un programa con facilidad puede "salirse" de un arreglo, porque C y C++ no verifican si los subíndices caen fuera del rango de un arreglo. Los arreglos del tamaño *n* deberá numerar sus elementos 0, ..., *n* - 1; no se permiten rangos con subíndices distintos. Un arreglo completo no puede ser introducido o extraído de una vez; cada elemento del arreglo debe ser leído o escrito en forma individual. No se pueden comparar dos arreglos mediante operadores de igualdad u operadores relacionales. Cuando se pasa un arreglo a una función de uso general, diseñada para manejar arreglos de cualquier tamaño, como argumento adicional deberá pasarse el tamaño del arreglo. Un arreglo no puede ser asignado a otro arreglo utilizando el operador u operadores de asignación. Estas y otras capacidades ciertamente parecerían como "naturales" para el manejo de arreglos, pero C y C++ no tienen estas capacidades. Sin embargo, C++ sí proporciona los medios para poner en práctica estas capacidades para arreglos, mediante los mecanismos de la homonimia de operadores.

En este ejemplo, desarrollamos una clase de arreglo que lleva a cabo verificación de rangos, con el fin de asegurarse que los subíndices se conservan dentro de los límites del arreglo. La clase permite que, mediante el operador de asignación, un objeto de arreglo sea asignado a otro. Los objetos de la clase de arreglo de forma automática conocen su tamaño, por lo que al pasar el arreglo a una función dicho tamaño no necesita ser pasado como argumento. Mediante los operadores de extracción y de inserción de flujo, respectivamente, es posible introducir o extraer arreglos completos. Se pueden llevar a cabo comparaciones de arreglos utilizando los operadores de igualdad == y !=. Nuestra clase de arreglo utiliza un miembro estático para llevar control del número de objetos de arreglo que en el programa han sido producidos. Este ejemplo aumentará su aprecio a la abstracción de datos. Es probable que desee sugerir varias mejoras a esta clase de arreglo. El desarrollo de clases es una actividad interesante e intelectual fascinante.

El programa de la figura 18.4 demuestra la clase **Array** y sus operadores homónimos. Primero recorramos el programa manejador en **main**. A continuación veamos la definición de clase y cada una de las funciones miembro de la clase y las definiciones de las funciones amigo. El programa produce dos objetos de clase **Array integers1**, con siete elementos, e **integers2**, con un tamaño por omisión de diez elementos (valor por omisión especificado por el constructor de **Array**). El miembro de datos estático **arrayCount** de la clase **Array** contiene el número de objetos **Array** producidos durante la ejecución del programa. La función miembro estática **getArrayCount** devuelve este valor. La función miembro **getSize** devuelve el tamaño del arreglo **integers1**. El programa extrae el tamaño del arreglo **integers1**, y a continuación extrae el arreglo mismo, utilizando el operador de inserción de flujo homónimo, para confirmar que los elementos del arreglo fueron inicializados de forma correcta por el constructor. A continuación, el tamaño del arreglo **integers2**, y después se extrae el arreglo mismo, utilizando el operador de inserción de flujo homónimo.

Al usuario se le pide que introduzca 16 enteros. Para leer dichos valores a ambos arreglos se utiliza el operador de extracción de flujo homónimo, mediante el enunciado

```
cin >> integers1 >> integers2;
```

Los primeros siete valores se almacenan en **integers1** y los restantes en **integers2**. Para confirmar que la entrada fue llevada a cabo correctamente, los dos arreglos son extraídos mediante el operador de inserción de flujo.

A continuación el programa prueba el operador de desigualdad homónimo evaluando la condición.

```
integers1 != integers2
```

y el programa informa que en efecto los arreglos no son iguales.

El programa produce un tercer arreglo, llamado **integers3**, y lo inicializa con el arreglo **integers1**. El programa extrae el tamaño del arreglo **integers3** y a continuación extrae el arreglo mismo, utilizando el operador de inserción de flujo homónimo, para confirmar que los elementos del arreglo fueron inicializados de forma correcta por el constructor.

Después el programa prueba el operador de asignación homónimo (=) con la expresión **integers1=integers2**. Ambos arreglos a continuación son impresos, para confirmar que la asignación fue correcta. Es interesante notar que **integers1** originalmente contenía 7 enteros y para contener una copia de los 10 elementos de **integers2** necesitaba ser redimensionado. Como veremos, el operador de asignación homónimo ejecuta este redimensionamiento de forma transparente para el llamador del operador.

A continuación el programa usa el operador de igualdad homónimo (==) para ver si después de la asignación los objetos **integers1=integers2** son de hecho idénticos.

Entonces el programa utiliza el operador de subíndice homónimo para hacer la referencia al elemento **integers1[5]** un elemento dentro del rango de **integers1**. Este nombre con subíndice se utiliza después como valor a la derecha (*rvalue*) para imprimir el valor en **integers1[5]**, y entonces el nombre se utiliza como valor a la izquierda (*lvalue*) en el lado izquierdo del enunciado de asignación, para recibir un nuevo valor, 1000, correspondiente a **integers1[5]**. Note que el **operator[]** provee la referencia y asigna el valor.

El programa a continuación intenta asignar le valor 1000 a **integers1[15]** un elemento fuera de rango. El programa detecta este error y termina en forma anormal.

De forma interesante, el operador de subíndice de arreglo **[]** no está restringido para uso en arreglos; puede ser utilizado para seleccionar elementos de otros tipos de clases contenedor, como son listas enlazadas, cadenas, diccionarios y demás. También, ya no es necesario que los subíndices sean enteros; se pueden utilizar caracteres o cadenas, por ejemplo.

Ahora que hemos visto cómo funciona este programa, vayamos recorriendo el encabezado de clase y las definiciones de funciones miembro. Las líneas

```
int *ptr;                    // pointer to first element of array
int size;                    // size of the array
static int arrayCount;       // # of Arrays instantiated
```

representa los miembros de datos privados de la clase. El arreglo está formado de un apuntador, **ptr** (al tipo apropiado, en este caso **int**), un miembro **size**, que indica el número de elementos en el arreglo, y un miembro estático **arrayCount**, que indica el número de objetos de arreglo que han sido producidos.

Las líneas

```
friend ostream &operator<<(ostream &, const Array &);
friend istream &operator>>(istream &, Array &);
```

declaran el operador de inserción de flujo homónimo y el operador de extracción de flujo homónimo como amigo de la clase **Array**. Cuando el compilador ve una expresión como


```
cout << arrayObject
```

invoca la función **operator<<** mediante la generación de la llamada

```
operator<<(cout, arrayObject)
```

Cuando el compilador ve una expresión como

```
cin >> arrayObject
```

invoca a la función **operator>>** mediante la generación de la llamada

```
operator>>(cin, arrayObject)
```

Notamos otra vez que esas funciones operador no pueden ser miembros de la clase **Array**, porque el objeto **Array** siempre es mencionado del lado derecho de los operadores de inserción y de extracción de flujo. La función **operator<<** imprime el número de elementos indicados por **size** en el arreglo almacenado en **ptr**. La función **operator>>** introduce directamente al arreglo apuntado por **ptr**. Cada una de estas funciones operador devuelven una referencia apropiada, para permitir llamadas concatenadas.

La línea

```
Array(int arraySize = 10)            // default constructor
```

declara al constructor por omisión correspondiente a la clase y especifica que el tamaño del arreglo se preestablece en 10 elementos. Cuando el compilador ve una declaración como

```
Array integers1(7);
```

o su forma equivalente

```
Array integers1 = 7;
```

invoca al constructor por omisión. La función miembro constructor por omisión **Array** incrementa **arrayCount**, copia el argumento en el miembro de datos **size**, utiliza a **new** para obtener el espacio para contener la representación interna de este arreglo y asigna el apuntador regresado por **new** al miembro de datos **ptr**, utiliza **assert** para probar que **new** tuvo éxito, y a continuación utiliza un ciclo **for** para inicializar a cero todos los elementos del arreglo. Sería perfectamente razonable tener una clase **Array** que no inicializara sus miembros si, por ejemplo, dichos miembros tuvieran que ser leídos en algún momento posterior.

La línea

```
Array(const Array &);                          // copy constructor
```

es un *constructor de copia*. Inicializa un objeto **Array** haciendo una copia de un objeto existente **Array**. Dicha copia deberá ser efectuada con cuidado, a fin de evitar el error de que ambos objetos **Array** apunten al mismo almacenamiento asignado dinámicamente, lo que sería exactamente el mismo problema que ocurriría mediante la copia a nivel de miembro por omisión. Los constructores de copia son invocados siempre que se requiera una copia de un objeto, como es tratándose de llamada por valor, al devolver un objeto de una función llamada, o al inicializar un objeto como una copia de otro objeto de la misma clase. En una declaración se llama al constructor de copia cuando un objeto de la clase **Array** es producido e inicializado con otro objeto de la clase **Array**, como ocurre en la declaración siguiente:

```
Array integers3(integers1);
```

o en la declaración equivalente

```
Array integers3 = integers1);
```

La función miembro constructor de copia **Array** incrementa **arrayCount**, copia **size** del arreglo utilizado para la inicialización en el miembro de datos **size**, utiliza **new** para obtener el espacio para contener la representación interna de este arreglo y asigna el apuntador regresado por **new** al miembro de datos **ptr**, utiliza **assert** para probar que **new** tuvo éxito, y después utiliza un ciclo **for** para copiar todos los elementos del arreglo inicializador a este arreglo. Es importante notar que si el constructor de copia simplemente hubiera copiado **ptr** del objeto fuente al objeto destino **ptr**, entonces ambos objetos hubieran señalado al mismo almacenamiento asignado dinámicamente. El primer destructor a ejecutarse hubiera entonces borrado el almacenamiento dinámicamente asignado, y el **ptr** del otro objeto hubiera quedado sin definición, una situación que seguramente causaría un error serio en tiempo de ejecución.

Práctica sana de programación 18.6

Un destructor, el operador de asignación, y un constructor de copia para una clase por lo general se proporcionan en grupo.

La línea

```
~Array();                                   // destructor
```

declara al destructor de la clase. Cuando se termine la vida de un objeto de la clase **Array**, el destructor será invocado en forma automática. El destructor decrementa **arrayCount** y a continuación utiliza **delete** para recuperar el almacenamiento dinámico creado por **new** en el constructor.

La línea

```
int getSize() const;                        // return size
```

declara a una función que lee el tamaño del arreglo.

La línea

```
const Array &operator=(const Array &);      // assign arrays
```

declara la función operador de asignación homónimo para la clase. Cuando el compilador ve una expresión como

```
integers1 = integers2;
```

invoca a la función **operator=** mediante la generación de la llamada

```
integers1.operator=(integers2)
```

La función miembro **operator=** prueba buscando autoasignación. Si se está intentando hacer una autoasignación, la asignación es pasada por alto (es decir, el objeto ya es él mismo). Si no se trata de una autoasignación, entonces la función miembro utiliza **delete** para recuperar el espacio originalmente asignado en el arreglo destino, copia **size** del arreglo fuente a **size** del arreglo destino, utiliza **new** para asignar dicha cantidad de espacio para el arreglo destino y coloca

el apuntador devuelto por **new** en el miembro **ptr** del arreglo, utiliza **assert** para verificar que **new** tuvo éxito, y después utiliza un ciclo **for** para copiar los elementos del arreglo, correspondientes al arreglo fuente, al arreglo destino. Independiente de que se trate de una autoasignación o no, a continuación la función miembro devuelve el objeto actual (es decir, ***this**) como referencia constante; esto habilita las asignaciones **Array** concatenadas como **x = y = z**. Si hubiera sido omitida la prueba de autoasignación, la función miembro hubiera empezado borrando el espacio del arreglo destino. Dado que en una autoasignación esto es también el arreglo fuente, el arreglo hubiera sido destruido.

Error común de programación 18.4

No probar la existencia y evitar la autoasignación cuando se hace la homonimia del operador de asignación de una clase que incluya un apuntador a almacenamiento dinámico.

Error común de programación 18.5

No proporcionar un operador de asignación homónimo y un constructor de copia para una clase, cuando los objetos de dicha clase contengan apuntadores a almacenamiento dinámicamente asignado.

Observación de ingeniería de software 18.3

Es posible evitar que un objeto de clase sea asignado a otro. Esto se lleva a cabo definiendo el operador de asignación como miembro privado de la clase.

La línea

```
int operator==(const Array &) const: // compare equal
```

declara al operador de igualdad homónimo (==) para la clase. Cuando el compilador ve la expresión

```
integers1 == integers2
```

en **main**, invoca la función miembro **operator==** mediante la generación de la llamada

```
integers1.operator==(integers2)
```

La función miembro **operator==** de inmediato devuelve 0 (falso) si son diferentes los miembros **size** de los arreglos. De lo contrario, la función miembro compara cada par de elementos. Si todos son iguales, devuelve 1 (verdadero). El primer par de elementos que sea distinto hará que se devuelva de inmediato 0.

La línea

```
int operator!=(const Array &) const; // compare !equal
```

declara el operador de desigualdad homónimo (!=) para la clase. Invocará a la función miembro **operator!=** y operará en forma similar a la función de operador de igualdad homónimo.

La línea

```
int &operator[](int);                       //subscript operator
```

declara al operador de subíndice homónimo correspondiente a la clase. Cuando el compilador ve la expresión

```
integers1[5]
```

en **main**, el compilador invoca la función miembro homónima **operator[]** generando la llamada

```
integers1.operator[](5)
```

La función miembro **operator[]** prueba si el subíndice está dentro de rango, y si no es así, el programa termina en forma anormal. Si el subíndice está dentro de rango, devuelve el elemento apropiado del arreglo como referencia, de manera que pueda ser utilizado como valor a la izquierda (*lvalue*) (por ejemplo, en el lado izquierdo de un enunciado de asignación).

La línea

```
static int getArrayCount();              // return count of Arrays
```

declara la función miembro estática **getArrayCount** que devuelve el valor del miembro de datos estático **arrayCount**, aún si no existen objetos de la clase **Array**.

```
// ARRAY1.H
// Simple class Array (for integers)
#ifndef ARRAY1_H
#define ARRAY1_H

#include <iostream.h>

class Array {
   friend ostream &operator<<(ostream &, const Array &);
   friend istream &operator>>(istream &, Array &);
public:
   Array(int = 10);                          // default constructor
   Array(const Array &);                     // copy constructor
   ~Array();                                 // destructor
   int getSize() const;                      // return size
   const Array &operator=(const Array &);    // assign arrays
   int operator==(const Array &) const;      // compare equal
   int operator!=(const Array &) const;      // compare !equal
   int &operator[](int);                     // subscript operator
   static int getArrayCount();               // return count of
                                             // arrays instantiated
private:
   int *ptr; // pointer to first element of array
   int size; // size of the array
   static int arrayCount;  // # of Arrays instantiated
};

#endif
```

Fig. 18.4 Definición de la clase **Array** (parte 1 de 7).


```
// ARRAY1.CPP
// Member function definitions for class Array
#include <iostream.h>
#include <stdlib.h>
#include <assert.h>
#include "array1.h"

// Initialize static data member at file scope
int Array::arrayCount = 0;    // no objects yet

// Return the number of Array objects instantiated
int Array::getArrayCount() { return arrayCount; }

// Default constructor for class Array
Array::Array(int arraySize)
{
   ++arrayCount;                    // count one more object
   size = arraySize;                // default size is 10
   ptr = new int[size];             // create space for array
   assert(ptr != 0);  // terminate if memory not allocated

   for (int i = 0; i < size; i++)
      ptr[i] = 0;                   // initialize array
}

// Copy constructor for class Array
Array::Array(const Array &init)
{
   ++arrayCount;                    // count one more object
   size = init.size;                // size this object
   ptr = new int[size];             // create space for array
   assert(ptr != 0);  // terminate if memory not allocated

   for (int i = 0; i < size; i++)
      ptr[i] = init.ptr[i];         // copy init into object
}

// Destructor for class Array
Array::~Array()
{
   --arrayCount;                    // one fewer objects
   delete [] ptr;                   // reclaim space for array
}

// Get the size of the array
int Array::getSize() const { return size; }

// Overloaded subscript operator
int &Array::operator[](int subscript)
{
   // check for subscript out of range error
   assert(0 <= subscript && subscript < size);

   return ptr[subscript];   // reference return creates lvalue
}
```

Fig. 18.4 Definiciones de funciones miembro para la clase **Array** (parte 2 de 7).

```
// Determine if two arrays are equal and
// return 1 if true, 0 if false.
int Array::operator==(const Array &right) const
{
   if (size != right.size)
      return 0;     // arrays of different sizes

   for (int i = 0; i < size; i++)
      if (ptr[i] != right.ptr[i])
         return 0; // arrays are not equal

   return 1;        // arrays are equal
}

// Determine if two arrays are not equal and
// return 1 if true, 0 if false.
int Array::operator!=(const Array &right) const
{
   if (size != right.size)
      return 1;     // arrays of different sizes

   for (int i = 0; i < size; i++)
      if (ptr[i] != right.ptr[i])
         return 1;       // arrays are not equal

   return 0;        // arrays are equal
}

// Overloaded assignment operator
const Array &Array::operator=(const Array &right)
{
   if (&right != this) {     // check for self-assignment
      delete [] ptr;         // reclaim space
      size = right.size;     // resize this object
      ptr = new int[size];   // create space for array copy
      assert(ptr != 0);      // terminate if memory not allocated

      for (int i = 0; i < size; i++)
         ptr[i] = right.ptr[i];  // copy array into object
   }

   return *this;    // enables x = y = z;
}

// Overloaded input operator for class Array;
// inputs values for entire array.
istream &operator>>(istream &input, Array &a)
{
   for (int i = 0; i < a.size; i++)
      input >> a.ptr[i];

   return input;    // enables cin >> x >> y;
}
```

Fig. 18.4 Definiciones de funciones miembro para la clase **Array** (parte 3 de 7).


```
// Overloaded output operator for class Array
ostream &operator<<(ostream &output, const Array &a)
{
   for (int i = 0; i < a.size; i++) {
      output << a.ptr[i] << ' ';

      if ((i + 1) % 10 == 0)
         output << endl;
   }

   if (i % 10 != 0)
      output << endl;

   return output;    // enables cout << x << y;
}
```

Fig. 18.4 Definiciones de función miembro para la clase **Array** (parte 4 de 7.

```
// FIG18_4.CPP
// Driver for simple class Array
#include <iostream.h>
#include "array1.h"

main()
{
   // no objects yet
   cout << "# of arrays instantiated = "
        << Array::getArrayCount() << '\n';

   // create two arrays and print Array count
   Array integers1(7), integers2;
   cout << "# of arrays instantiated = "
        << Array::getArrayCount();

   // print integers1 size and contents
   cout << "\n\nSize of array integers1 is "
        << integers1.getSize()
        << "\nArray after initialization:\n"
        << integers1;

   // print integers2 size and contents
   cout << "\nSize of array integers2 is "
        << integers2.getSize()
        << "\nArray after initialization:\n"
        << integers2;

   // input and print integers1 and integers2
   cout << "\nInput 17 integers:\n";
   cin >> integers1 >> integers2;
   cout << "After input, the arrays contain:\n"
        << "integers1: " << integers1
        << "integers2: " << integers2;
```

Fig. 18.4 Manejador para la clase **Array** (parte 5 de 7).

```
   // create array integers3 using integers1 as an
   // initializer; print size and contents
   Array integers3(integers1);

   cout << "\nSize of array integers3 is "
        << integers3.getSize()
        << "\nArray after initialization:\n"
        << integers3;

   // use overloaded inequality (!=) operator
   cout << "\nEvaluating: integers1 != integers2\n";
   if (integers1 != integers2)
      cout << "They are not equal\n\n";

   // use overloaded assignment (=) operator
   cout << "Assigning integers2 to integers1:\n";
   integers1 = integers2;
   cout << "integers1: " << integers1
        << "integers2: " << integers2;

   // use overloaded equality (==) operator
   cout << "\nEvaluating: integers1 == integers2\n";
   if (integers1 == integers2)
      cout << "They are equal\n\n";

   // use overloaded subscript operator to create rvalue
   cout << "integers1[5] is " << integers1[5] << endl;

   // use overloaded subscript operator to create lvalue
   cout << "Assigning 1000 to integers1[5]\n";
   integers1[5] = 1000;
   cout << "integers1: " << integers1;

   // attempt to use out of range subscript
   cout << "\nAttempt to assign 1000 to integers1[15]\n";
   integers1[15] = 1000;  // ERROR: out of range

   return 0;
}
```

Fig. 18.4 Manejador para la clase **Array** (parte 6 de 7).

18.9 Conversión entre tipos

La mayor parte de los programas procesan información en una variedad de tipos. Algunas veces todas las operaciones "se mantienen dentro de un tipo". Por ejemplo, añadir un entero a un entero produce un entero (siempre y cuando el resultando no sea tan grande que no pueda ser representado como un entero). Pero, a menudo es necesario convertir datos de un tipo a datos de otro tipo. Esto puede ocurrir en asignaciones, en cálculos, al pasar valores a funciones, y al devolver valores de funciones. El compilador sabe cómo llevar a cabo conversiones entre tipos incorporados. Los programadores pueden obligar a conversiones entre tipos incorporados mediante las conversiones explícitas casting.


```
# of arrays instantiated = 0
# of arrays instantiated = 2

Size of array integers1 is 7
Array after initialization:
0 0 0 0 0 0 0

Size of array integers2 is 10
Array after initialization:
0 0 0 0 0 0 0 0 0 0

Input 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
After input, the arrays contain:
integers1: 1 2 3 4 5 6 7
integers2: 8 9 10 11 12 13 14 15 16 17

Size of array integers3 is 7
Array after initialization:
1 2 3 4 5 6 7

Evaluating: integers1 != integers2
They are not equal

Assigning integers2 to integers1:
integers1: 8 9 10 11 12 13 14 15 16 17
integers2: 8 9 10 11 12 13 14 15 16 17

Evaluating: integers1 == integers2
They are equal

integers1[5] is 13
Assigning 1000 to integers1[5]
integers1: 8 9 10 11 12 13 14 15 16 17

Attempt to assign 1000 to integers1[15]
Assertion failed: 0 <= subscript && subscript < size,
   file ARRAY1.CPP, line 98
Abnormal program termination
```

Fig. 18.4 Salida del manejador para la clase **Array** (parte 7 de 7).

Pero ¿qué pasa en relación con los tipos definidos por usuario? El compilador no puede saber en forma automática cómo convertir entre tipos definidos por usuario y tipos incorporados. El programador deberá especificar en forma explícita cómo deberán ocurrir dichas conversiones. Estas conversiones pueden ser ejecutadas mediante *constructores de conversión*, es decir, constructores de un solo argumento, que sólo conviertan objetos de otros tipos en objetos de una clase particular. Más adelante en este capítulo utilizaremos un constructor de conversión para convertir las cadenas **char** * normales en objetos **String** de C++.

Un *operador de conversión* (también conocido como *operador de conversión explicita cast*) puede ser utilizado para convertir un objeto de una clase en un objeto de otra clase o en un objeto

de un tipo incorporado. Este operador de conversión debe ser una función miembro no estática; este tipo de operador de conversión no puede ser una función amigo.

El prototipo de función

```
operator char *() const;
```

declara una función operador de conversión explicita cast homónima para crear un objeto temporal **char*** partiendo de un objeto de un tipo definido por usuario. Una función operador de conversión explicita cast homónima no especifica un tipo de regreso —el tipo de regreso es el tipo al cual se está convirtiendo el objeto. Si **s** es un objeto de clase, cuando el compilador vea la expresión **(char *) s** el compilador generará la llamada **s.operator char * ()**. El operando **s** es el objeto de clase **s** para el cual se invoca la función miembro **operator char ***.

Las funciones operador de conversión explícita cast homónimas pueden ser definidas para la conversión de objetos de tipos definidos por usuario a tipos incorporados o a objetos de otros tipos definidos por usuario. Los prototipos de función

```
operator int() const;
operator otherClass() const;
```

declaran funciones operador de conversión explícita cast homónimas para la conversión de un objeto de un tipo definido por usuario a un entero o para convertir un objeto de un tipo definido por usuario a un objeto de un tipo definido por usuario **otherClass**.

Una de las características maravillosas de los operadores de conversión explícita (cast) y de los constructores de conversión es que, cuando es necesario, el compilador puede llamar estas funciones en forma automática para crear objetos temporales. Por ejemplo, si en un programa aparece un objeto **s** de una clase **String** definida por usuario, en una posición donde se espera un **char *** normal, tal como

```
cout << s;
```

el compilador llama la función de operador de conversión explícita (cast) homónima **operator char *** para convertir el objeto en un **char *** y utiliza el **char *** resultante dentro de la expresión.

18.10 Estudio de caso: una clase String

Como un ejercicio de recapitulación a nuestro estudio de la homonimia construiremos una clase que maneje la creación y manipulación de cadenas. Ni C ni C++ tienen un tipo de datos de cadenas incorporado. Pero C++ nos permite añadir como clase a nuestro propio tipo de cadenas, y —mediante los mecanismos de la homonimia— proporciona un conjunto de operadores para manejar convenientemente las cadenas. Las varias partes de la figura 18.5 incluyen un encabezado de clase, definiciones de funciones miembros, y un programa manejador para probar nuestra nueva clase **String**.

Primero, presentaremos el encabezado para la clase **String**. Analizamos los datos privados utilizados para representar objetos **String**. A continuación, recorremos la interfaz pública de la clase, analizando cada uno de los servicios que la misma proporciona.

A continuación, recorreremos el programa manejador en **main**. Analizaremos el estilo de codificación al cual "aspiramos", es decir, los tipos de expresiones operador intensivo que nos gustaría tener la capacidad de escribir, con objetos de nuestra nueva clase **String** y con el conjunto de operadores homónimos de la clase.

A continuación, analizamos las definiciones de funciones miembro correspondientes a la clase **String**. Para cada uno de los operadores homónimos, mostramos el código en el programa manejador que invoca a la función operador homónima, y explicamos cómo funciona la función operador homónima.

Empezamos con la representación interna de un **String**. Las líneas

```
private:
   char *sPtr;                    // pointer to start of string
   int length;                    // string length
```

declaran los miembros de datos privados de la clase. Un objeto **String** tiene un apuntador a su almacenamiento dinámico asignado representando la cadena de caracteres, y tiene un campo de longitud que representa el número de caracteres en la cadena, excluyendo el carácter nulo al final de la cadena de caracteres.

Ahora recorramos el archivo de cabecera de la clase **String** de la figura 18.5. Las líneas

```
friend ostream &operator<<(ostream &, const String &);
friend istream &operator>>(istream &, String &);
```

declaran la función operador de inserción de flujo homónima **operator<<** y la función operador de extracción de flujo homónima **operator>>** como amigo de la clase. La puesta en práctica de las funciones anteriores es simple.

La línea

```
String(const char * = ""); // conversion constructor
```

declara un *constructor de conversión*. Este constructor toma un argumento **char *** (que por omisión es la cadena vacía) y produce un objeto **String** que incluye la misma cadena de caracteres. Cualquier constructor de un argumento puede ser considerado como si fuera un constructor de conversión. Como veremos, estos constructores son útiles cuando estamos haciendo asignaciones de cadenas de caracteres a objetos **String**. El constructor de conversión convierte la cadena convencional en un objeto **String**, que a continuación es asignado al objeto destino **String**. La disponibilidad de este constructor de conversión significa que para asignar en forma específica las cadenas de caracteres a los objetos **String** no es necesario proveer un operador de asignación homónimo. El compilador invoca en forma automática el constructor de conversión, a fin de crear un objeto temporal **String** que contenga la cadena de caracteres. Entonces, se invoca el operador de asignación homónimo para asignar el objeto temporal **String** a otro objeto **String**. Note que cuando C++ utiliza de esta forma un constructor de conversión, sólo puede aplicar un constructor para intentar hacer coincidir las necesidades del operador de asignación homónimo. No es posible hacer coincidir las necesidades del operador homónimo llevando a cabo una serie de conversiones implícitas, definidas por el usuario. El constructor de conversión **String** podría ser invocado en una declaración como **String s1 ("happy")**. La función miembro constructor de conversión calcula la longitud de la cadena de caracteres y asigna este cálculo al miembro de datos privado **length**, utiliza **new** para asignar suficiente espacio al miembro de datos privado **sPtr**, utiliza **assert** para probar que **new** tuvo éxito, y si así fue, utiliza **strcpy** para copiar la cadena de caracteres al objeto.

La línea

```
String(const String &);              // copy constructor
```

es un constructor de copia. Inicializa un objeto **String** haciendo una copia de un objeto existente **String**. Esta copia deberá ser efectuada con cuidado para evitar el problema de tener ambos objetos **String** señalando al mismo almacenamiento asignado dinámicamente, exactamente el problema que ocurriría tratándose de una copia a nivel de miembro por omisión. El constructor de copia opera de forma similar al constructor de conversión, excepto que simplemente copia el miembro **length** del objeto fuente **String** al objeto destino **String**. Note que el constructor de copia genera un nuevo espacio para la cadena de caracteres interna del objeto destino. Si solo copiara **sPtr** en el objeto fuente al **sPtr** del objeto destino, entonces ambos objetos señalarían al mismo almacenamiento dinámico asignado. El primer destructor que se ejecutase borraría entonces el almacenamiento dinámicamente asignado y el **sPtr** del otro objeto quedaría entonces sin definir, una situación que probablemente causaría un error serio en tiempo de ejecución.

La línea

```
~String();                                  // destructor
```

declara al destructor correspondiente a la clase **String**. El destructor utiliza **delete** para reclamar el almacenamiento dinámico obtenido por **new** en los constructores para proveer el espacio correspondiente a la cadena de caracteres.

La línea

```
const String &operator=(const String &);    // assignment
```

declara al operador de asignación homónimo. Cuando el compilador ve una expresión como **string1 = string2**, genera la llamada de función

```
string1.operator=(string2);
```

La función operador de asignación homónimo **operator=** prueba por si existe autoasignación, exactamente de la misma forma que lo hizo el constructor de copia. Si se trata de una autoasignación, la función sólo regresa, dado que el objeto ya es él mismo. Si esta prueba se omitiera, la función de inmediato borraría el espacio en el objeto destino y se perdería la cadena de caracteres. Suponiendo que no se trata de una autoasignación, la función no borra el espacio, copia el campo de longitud del objeto fuente al objeto destino, genera un nuevo espacio para el objeto destino, utiliza **assert** para determinar si **new** tuvo éxito, y entonces utiliza **strcpy** para copiar la cadena de caracteres del objeto fuente al objeto destino. Independiente de que se trate o no de una autoasignación, se devuelve *** this** para permitir asignaciones concatenadas.

La línea

```
String &operator+=(const String &);          // concatenation
```

declara al operador de concatenación de cadenas homónimo. Cuando el compilador ve la expresión **s1 += s2** en **main**, genera la llamada de función **s1.operator+=(s2)**. La función **operator+=** crea un apuntador temporal para contener el apuntador de cadena de caracteres del objeto actual, en tanto pueda ser borrada la memoria de la cadena de caracteres, calcula la longitud combinada de la cadena concatenada, utiliza **new** para reservar espacio para la cadena, utiliza **assert** para probar si **new** tuvo éxito, utiliza **strcpy** para copiar la cadena original al nuevo espacio asignado, utiliza **strcat** para concatenar la cadena de caracteres del objeto fuente al nuevo espacio asignado, utiliza **delete** para recuperar el espacio ocupado por la cadena de

caracteres original de este objeto, y devuelve *** this** como un **String &** para permitir la concatenación de los operadores **+=**.

¿Será necesario tener un segundo operador de concatenación homónimo para poder hacer la concatenación de un **String** y de un **char ***? No, el constructor de conversión **const char *** convierte una cadena convencional en un objeto **String**, que a continuación coincide con el operador de concatenación homónimo. C++ puede llevar a cabo estas conversiones en un nivel de profundidad, para facilitar una coincidencia. C++ también puede, antes de llevar a cabo la conversión entre un tipo incorporado y una clase, ejecutar una conversión implícita definida por compilador entre tipos incorporados.

Sugerencia de rendimiento 18.2

Usar el operador de concatenación homónimo += que toma un solo argumento del tipo **const char *** *da resultados más eficaces que primero tener que efectuar la conversión implícita y a continuación, la concatenación. Las conversiones implícitas requieren de menos código y causan menos errores.*

La línea

```
int operator!() const;                      // is String empty?
```

declara el operador de negación homónimo. Este operador se usa por lo común con las clases de cadenas, para probar si una cadena está vacía. Por ejemplo, cuando el compilador ve la expresión **!string1**, genera la llamada de función

```
string1.operator!()
```

Esta función sencillamente devuelve el resultado de probar si **length** es igual a cero.

Las líneas

```
int operator==(const String &) const;    // test s1 == s2
int operator!=(const String &) const;    // test s1 != s2
int operator<(const String &) const;     // test s1 < s2
int operator>(const String &) const;     // test s1 > s2
int operator>=(const String &) const;    // test s1 >= s2
int operator<=(const String &) const;    // test s1 <= s2
```

declaran los operadores de igualdad homónimos y los operadores relacionales homónimos, correspondientes a la clase **String**. Todos ellos son similares, por lo que analizaremos un ejemplo, simplemente la homonimia del operador **>=**. Cuando el compilador ve la expresión **string1 >= string2**, generará la llamada de función

```
string1.operator>=(string2)
```

misma que devolverá **1** (verdadero) si **string1** es mayor que o igual a **string2**. Cada uno de estos operadores utiliza **strcmp** para comparar las cadenas de caracteres en los objetos **String**.

La línea

```
char &operator[](int);                      // return char reference
```

declara el operador de subíndice homónimo. Cuando el compilador ve una expresión como **string1[0]**, generará la llamada de función **string1.operator[](0)**. La función **ope-**

rator[] primero utiliza **assert** para llevar a cabo una verificación de rango sobre el subíndice; si el subíndice está fuera de rango, el programa imprimirá un mensaje de error y terminará en forma anormal. Si el subíndice está dentro de rango, el operador **operator[]** devuelve el carácter apropiado como un **char &** del objeto **String**; este **char &** puede ser utilizado como un valor a la izquierda (*lvalue*) para modificar el carácter designado del objeto **String**.

La línea

```
String &operator() (int, int);    // return a substring
```

declara el *operador de llamada de función homónimo*. En las clases de cadenas, es común hacer la homonimia de este operador para seleccionar una subcadena de un objeto **String**. Los dos parámetros enteros especifican la posición inicial y la longitud de la subcadena que se está seleccionando de **String**. Si la posición inicial está fuera de rango o si es negativa la longitud de la subcadena, se generará un mensaje de error. Por regla convencional, si la longitud de la subcadena es 0, entonces la subcadena será seleccionada hasta el final del objeto **String**. Por

```
// STRING2.H
// Definition of a String class
#ifndef STRING1_H
#define STRING1_H

#include <iostream.h>

class String {
   friend ostream &operator<<(ostream &, const String &);
   friend istream &operator>>(istream &, String &);
public:
   String(const char * = ""); // conversion constructor
   String(const String &);    // copy constructor
   ~String();                 // destructor
   const String &operator=(const String &);  // assignment
   String &operator+=(const String &);    // concatenation
   int operator!() const;                 // is String empty?
   int operator==(const String &) const;  // test s1 == s2
   int operator!=(const String &) const;  // test s1 != s2
   int operator<(const String &)  const;  // test s1 < s2
   int operator>(const String &)  const;  // test s1 > s2
   int operator>=(const String &) const;  // test s1 >= s2
   int operator<=(const String &) const;  // test s1 <= s2
   char &operator[](int);         // return char reference
   String &operator()(int, int); // return a substring
   int getLength() const;        // return string length
private:
   char *sPtr;                   // pointer to start of string
   int length;                   // string length
};

#endif
```

Fig. 18.5 Definición de una clase **String** básica (parte 1 de 8).

ejemplo, suponga que **string1** es un objeto **String** que contiene la cadena de caracteres **"AEIOU"**. Cuando el compilador ve la expresión **string1(2, 2)**, genera la llamada de función **string1.operator() (2, 2)**. Cuando esta llamada se ejecuta, se produce un objeto temporal **String**, que contiene la cadena **"IO"** y devuelve una referencia a este objeto. Hacer la homonimia del operador de llamada de función **()** resulta poderosa porque las funciones pueden tomar listas de parámetros arbitrariamente largas y complejas. Por lo tanto, podemos utilizar esta capacidad para muchos fines interesantes. Otro uso del operador de llamada de función es una notación alterna de subscripción de los arreglos: en vez de utilizar la notación de dobles corchetes de C, que resultan torpes tratándose de dobles arreglos, como en **a[b][c]**, por ejemplo, algunos programadores prefieren hacer la homonimia del operador de llamada de función para permitir la notación **a(b, c)**. El operador de llamada de función homónimo solamente puede ser una función miembro no estática. Este operador se utiliza sólo cuando "el nombre de función" es un objeto de la clase **String**.

La línea

```
int getLength() const;            // return string length
```

declara una función que devuelve la longitud del objeto **String**. Note que esta función obtiene la longitud, regresando el valor de los datos privados de la clase **String**.

Llegado a este punto, el lector deberá recorrer el código en **main**, examinar la ventana de salida, y verificar el uso de cada operador homónimo.

```
// STRING2.CPP
// Member function definitions for class String

#include <iostream.h>
#include <string.h>
#include <assert.h>
#include "string2.h"

// Conversion constructor: Convert char * to String
String::String(const char *s)
{
   cout << "Conversion constructor: " << s << endl;
   length = strlen(s);              // compute length
   sPtr = new char[length + 1]; // allocate storage
   assert(sPtr != 0);  // terminate if memory not allocated
   strcpy(sPtr, s);                 // copy literal to object
}

// Copy constructor
String::String(const String &copy)
{
   cout << "Copy constructor: " << copy.sPtr << endl;
   length = copy.length;            // copy length
   sPtr = new char[length + 1]; // allocate storage
   assert(sPtr != 0);               // ensure memory allocated
   strcpy(sPtr, copy.sPtr);         // copy string
}
```

Fig. 18.5 Definiciones de funciones miembro para la clase **String** (parte 2 de 8).

```
// Destructor
String::~String()
{
   cout << "Destructor: " << sPtr << endl;
   delete [] sPtr;               // reclaim string
}

// Overloaded = operator; avoids self assignment
const String &String::operator=(const String &right)
{
   cout << "operator= called" << endl;

   if (&right != this) {            // avoid self assignment
      delete [] sPtr;               // prevents memory leak
      length = right.length;        // new String length
      sPtr = new char[length + 1]; // allocate memory
      assert(sPtr != 0);            // ensure memory allocated
      strcpy(sPtr, right.sPtr);     // copy string
   }
   else
      cout << "Attempted assignment of a String to itself\n";

   return *this;   // enables concatenated assignments
}

// Concatenate right operand to this object and
// store in this object.
String &String::operator+=(const String &right)
{
   char *tempPtr = sPtr;         // hold to be able to delete
   length += right.length;       // new String length
   sPtr = new char[length + 1]; // create space
   assert(sPtr != 0);   // terminate if memory not allocated
   strcpy(sPtr, tempPtr);        // left part of new String
   strcat(sPtr, right.sPtr);     // right part of new String
   delete [] tempPtr;            // reclaim old space
   return *this;                 // enables concatenated calls
}

// Is this String empty?
int String::operator!() const { return length == 0; }

// Is this String equal to right String?
int String::operator==(const String &right) const
{ return strcmp(sPtr, right.sPtr) == 0; }

// Is this String not equal to right String?
int String::operator!=(const String &right) const
{ return strcmp(sPtr, right.sPtr) != 0; }

// Is this String less than right String?
int String::operator<(const String &right) const
{ return strcmp(sPtr, right.sPtr) < 0; }
```

Fig. 18.5 Definiciones de funciones miembro para la clase **String** (parte 3 de 8).


```
// Is this String greater than right String?
int String::operator>(const String &right) const
{ return strcmp(sPtr, right.sPtr) > 0; }

// Is this String greater than or equal to right String?
int String::operator>=(const String &right) const
{ return strcmp(sPtr, right.sPtr) >= 0; }

// Is this String less than or equal to right String?
int String::operator<=(const String &right) const
{ return strcmp(sPtr, right.sPtr) <= 0; }

// Return a reference to a character in a String.
char &String::operator[](int subscript)
{
   // First test for subscript out of range
   assert(subscript >= 0 && subscript < length);

   return sPtr[subscript];  // creates lvalue
}

// Return a substring beginning at index and
// of length subLength as a reference to a String object.
String &String::operator()(int index, int subLength)
{
   // ensure index is in range and substring length >= 0
   assert(index >= 0 && index < length && subLength >= 0);

   String *subPtr = new String;   // empty String
   assert(subPtr != 0);      // ensure new String allocated

   // determine length of substring
   int size;

   if ((subLength == 0) || (index + subLength > length))
      size = length - index + 1;
   else
      size = subLength + 1;

   // allocate memory for substring
   delete subPtr->sPtr;
   subPtr->length = size;
   subPtr->sPtr = new char[size];
   assert(subPtr->sPtr != 0);  // ensure space allocated

   // copy substring into new String
   for (int i = index, j = 0; i < index + size - 1; i++, j++)
      subPtr->sPtr[j] = sPtr[i];

   subPtr->sPtr[j] = '\0';  // terminate the new String
   return *subPtr;          // return new String
}
```

Fig. 18.5 Definiciones de funciones miembro para la clase **String** (parte 4 de 8).

```
// Return string length
int String::getLength() const { return length; }

// Overloaded output operator
ostream &operator<<(ostream &output, const String &s)
{
   output << s.sPtr;
   return output;    // enables concatenation
}

// Overloaded input operator
istream &operator>>(istream &input, String &s)
{
   static char temp[100];   // buffer to store input

   input >> temp;
   s = temp;          // use String class assignment operator
   return input;      // enables concatenation
}
```

Fig. 18.5 Definiciones de funciones miembro para la clase **String** (parte 5 de 8).

```
// FIG18_5.CPP
// Driver for class String
#include <iostream.h>
#include "string2.h"

main()
{
   String s1("happy"), s2(" birthday"), s3;

   // test overloaded equality and relational operators
   cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
        << "\"; s3 is empty\n"
        << "The results of comparing s2 and s1:"
        << "\ns2 == s1 yields " << (s2 == s1)
        << "\ns2 != s1 yields " << (s2 != s1)
        << "\ns2 >  s1 yields " << (s2 > s1)
        << "\ns2 <  s1 yields " << (s2 < s1)
        << "\ns2 >= s1 yields " << (s2 >= s1)
        << "\ns2 <= s1 yields " << (s2 <= s1) << "\n\n";

   // test overloaded String empty (!) operator
   cout << "Testing !s3:\n";
   if (!s3) {
      cout << "s3 is empty; assigning s1 to s3;\n";
      s3 = s1;                // test overloaded assignment
      cout << "s3 is \"" << s3 << "\"\n\n";
   }
```

Fig. 18.5 Manejador para probar la clase **String** (parte 6 de 8).


```
   // test overloaded String concatenation operator
   cout << "s1 += s2 yields s1 = ";
   s1 += s2;                  // test overloaded concatenation
   cout << s1 << "\n\n";

   // test conversion constructor
   cout << "s1 += \" to you\" yields\n";
   s1 += " to you";           // test conversion constructor
   cout << "s1 = " << s1 << "\n\n";

   // test overloaded function call operator () for substring
   cout << "The substring of s1 starting at\n"
        << "location 0 for 14 characters, s1(0, 14), is: "
        << s1(0, 14) << "\n\n";

   // test substring "to-end-of-String" option
   cout << "The substring of s1 starting at\n"
        << "location 15, s1(15, 0), is: "
        << s1(15, 0) << "\n\n";  // 0 means "to end of string"

   // test copy constructor
   String *s4Ptr = new String(s1);
   cout << "*s4Ptr = " << *s4Ptr << "\n\n";

   // test assignment (=) operator with self-assignment
   cout << "assigning *s4Ptr to *s4Ptr\n";
   *s4Ptr = *s4Ptr;           // test overloaded self-assignment
   cout << "*s4Ptr = " << *s4Ptr << endl;

   // test destructor
   delete s4Ptr;

   // test using subscript operator to create lvalue
   s1[0] = 'H';
   s1[6] = 'B';
   cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
        << s1 << "\n\n";

   // test subscript out of range
   cout << "Attempt to assign 'd' to s1[30] yields:\n";
   s1[30] = 'd';      // ERROR: subscript out of range

   return 0;
}
```

Fig. 18.5 Manejador para probar la clase **String** (parte 7 de 8).

18.11 Homonimia de ++ y --

Todos los operadores de incremento y de decremento —preincremento, postincremento, predecremento y postdecremento pueden tener homónimos. Veremos cómo el compilador puede conocer cual es la versión prefija o postfija del operador de incremento o de decremento que está siendo utilizada.

```
Conversion constructor: happy
Conversion constructor: birthday
Conversion constructor:
s1 is "happy"; s2 is "birthday"; s3 is empty
The results of comparing s2 and s1:
s2 == s1 yields 0
s2 != s1 yields 1
s2 >  s1 yields 0
s2 <  s1 yields 1
s2 >= s1 yields 0
s2 <= s1 yields 1

Testing !s3:
s3 is empty; assigning s1 to s3;
operator= called
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += "to you" yields
Conversion constructor: to you
Destructor: to you
s1 = happy birthday to you

Conversion constructor:
The substring of s1 starting at
location 0 for 14 characters, s1(0, 14), is: happy birthday

Conversion constructor:
The substring of s1 starting at
location 15, s1(15, 0), is: to you

Copy constructor: happy birthday to you
*s4Ptr = happy birthday to you

ssigning *s4Ptr to *s4Ptr
operator= called
Attempted assignment of a String to itself
*s4Ptr = happy birthday to you
Destructor: happy birthday to you

s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you

Attempt to assign 'd' to s1[30] yields:
Assertion failed: subscript >= 0 && subscript < length,
   file STRING2.CPP, line 98
Abnormal program termination
```

Fig. 18.5 Salida del manejador para probar la clase **String** (parte 8 de 8).

Para hacer la homonimia del operador incremental para permitir el uso tanto de preincremento como de postincremento, cada función de operador homónimo debe tener una firma distinta, de tal forma que el compilador sea capaz de determinar cuál es la versión de ++ que se desea. Se hace la homonimia de las versiones prefijas de la misma manera que cualquier operador unario prefijo.

Suponga, por ejemplo, que en el objeto **d1** de **Date** deseamos añadir 1 al día. Cuando el compilador ve la expresión preincremental

```
++d1
```

el compilador genera la llamada de función miembro

```
d1.operator++()
```

cuyo prototipo sería

```
Date operator++();
```

Si el preincremento es puesto en práctica como una función no miembro, cuando el compilador vea la expresión

```
++d1
```

el compilador generará la llamada de función

```
operator++(d1)
```

cuyo prototipo debería ser declarado en la clase **Date** como

```
friend Date operator++(Date &);
```

Hacer la homonimia del operador de postincremento resulta un pequeño desafío, porque el compilador debe ser capaz de distinguir las firmas de las funciones de operador de preincremento y postincremento homónimas. La regla convencional que ha sido adoptada en C++, es que cuando el compilador vea la expresión de postincremento

```
d1++
```

generará la llamada de función miembro

```
d1.operator++(0)
```

cuyo prototipo sería

```
Date operator++(int)
```

Aquí el 0 es un valor estricto de relleno, con el propósito que sea distinguible la lista de argumentos de **operator++** utilizada para postincrementar, de la lista de argumentos de **operator++** utilizada para preincrementar.

Si el postincremento es puesto en práctica como función no miembro, cuando el compilador vea la expresión

```
d1++
```

el compilador generará la llamada de función

```
operator++(d1, 0)
```

cuyo prototipo sería

```
friend Date operator++(Date &, int);
```

Otra vez, el argumento `0 int` es utilizado por el compilador para que la lista de argumentos de **operator++** utilizado para postincrementar, resulte distinguible de la lista de argumentos para preincrementar. Todo lo que hemos dicho en esta sección para la homonimia de operadores de preincremento y postincremento es aplicable a la homonimia de los operadores de predecremento y posdecremento. En la sección siguiente, examinaremos una clase **Date**, que utiliza operadores de preincremento y postincremento homónimos.

18.12 Estudio de caso: una clase Date

En la figura 18.6 se ilustra una clase **Date**. La clase utiliza operadores de preincremento y de postincremento homónimos, para añadir 1 al día en un objeto **Date**.

La clase proporciona las siguientes funciones miembro: un operador de inserción de flujo homónimo, un constructor por omisión, una función **setDate**, una función operador de preincremento homónima, una función operador de postincremento homónima, y un operador de asignación de adición homónimo (+=).

El programa manejador en **main** crea los objetos de fecha **d1** que se inicializan a January 1, 1990; **d2**, que se inicializan a December 27, 1992; y **d3** que se inicializan a una fecha inválida. El constructor **Date** llama a **setDate** para validar el mes, día y año específico. Si el mes es inválido, se define como 1. Un año inválido se define como 1900. Un día inválido se define como 1.

El programa manejador extrae cada uno de los objetos construidos **Date** utilizando el operador de inserción de flujo homónimo. El operador homónimo += es utilizado para añadir 7 días a **d2**. A continuación la función **setDate** es usada para definir **d3** a February 28, 1992. Después, un nuevo objeto **Date**, **d4**, se define a March 18, 1969. Entonces mediante el operador de preincremento homónimo, **d4** se incrementa en 1. La fecha es impresa, antes y después del preincremento, para confirmar que se hizo correctamente. Por ultimo, **d4** es incrementado utilizando el operador de postincremento homónimo. La fecha es impresa antes y después del postincremento, a fin de confirmar que funcionó correctamente.

La homonimia del operador de preincremento es directa. Para efectuar el incremento real el programa llama a la función de utilería **helpIncrement**. Esta función debe ocuparse de los desbordamientos que ocurren cuando intentamos aumentarle un día a un mes que ya está en su valor máximo. Estos excedentes requieren que el mes se incremente. Si el mes es ya el mes 12, el año también deberá ser incrementado.

El operador de preincremento homónimo devuelve una *copia* incrementada del objeto real. Esto ocurre porque el objeto real, *** this**, es regresado como un **Date**. Esto de hecho invoca al constructor de copia.

Resulta un poco más complejo hacer la homonimia del operador de postincremento. A fin de emular el efecto de postincremento, debemos regresar una copia del objeto **Date** sin incrementar. Al introducir el **operator++**, guardamos en **temp** el objeto actual (*** this**). A continuación llamamos a **helpIncrement** para incrementar el objeto. Por último regresamos la copia sin incrementar del objeto en **temp**.


```
// DATE1.H
// Definition of class Date
#ifndef DATE1_H
#define DATE1_H
#include <iostream.h>

class Date {
   friend ostream &operator<<(ostream &, const Date &);
public:
   Date(int m = 1, int d = 1, int y = 1900);  // def. constructor
   void setDate(int, int, int);  // set the date
   Date operator++();            // preincrement
   Date operator++(int);         // postincrement
   Date &operator+=(int);        // add days, modify object
private:
   int month;
   int day;
   int year;
   static int days[];      // array of days per month
   void helpIncrement();   // utility function
};

#endif
```

Fig. 18.6 Definición de la clase **Date** (parte 1 de 6).

```
// DATE1.CPP
// Member function definitions for Date class
#include <iostream.h>
#include "date1.h"

// Initialize static member at file scope;
// one class-wide copy.
int Date::days[] = {0, 31, 28, 31, 30, 31, 30,
                    31, 31, 30, 31, 30, 31};

// Date constructor
Date::Date(int m, int d, int y) { setDate(m, d, y); }

// Set the date
void Date::setDate(int mm, int dd, int yy)
{
   month = (mm >= 1 && mm <= 12) ? mm : 1;
   year = (yy >= 1900 && yy <= 2100) ? yy : 1900;

   // test for a leap year
   if (month == 2 && year % 4 == 0 && (year % 400 == 0 ||
                                       year % 100 != 0))
      day = (dd >= 1 && dd <= 29) ? dd : 1;
   else
      day = (dd >= 1 && dd <= days[month]) ? dd : 1;
}
```

Fig.18.6 Definición de función miembro para la clase **Date** (parte 2 de 6).

```
// Preincrement operator overloaded as a member function.

Date Date::operator++()
{
   helpIncrement();
   return *this;  // value return; not a reference return
}

// Postincrement operator overloaded as a member function.
// Note that the dummy integer parameter does not have a
// parameter name.

Date Date::operator++(int)
{
   Date temp = *this;
   helpIncrement();

   // return non-incremented, saved, temporary object
   return temp;   // value return; not a reference return
}

// Add a specific number of days to a date

Date &Date::operator+=(int additionalDays)
{
   for (int i = 1; i <= additionalDays; i++)
      helpIncrement();

   return *this;    // enables concatenation
}

// Function to help increment the date

void Date::helpIncrement()
{
   // test for a leap year first
   if (month == 2 && day < 29 && year % 4 == 0 &&
       (year % 400 == 0 || year % 100 != 0))
      ++day;
   else if (day < Date::days[month])  // not last day of month
      ++day;
   else if (month < 12) {  // last day of month < December
      ++month;
      day = 1;
   }
   else {                  // December 31
      month = 1;
      day = 1;
      ++year;
   }
}
```

Fig. 18.6 Definición de función miembro para la clase **Date** (parte 3 de 6).


```
// Overloaded output operator
ostream &operator<<(ostream &output, const Date &d)
{
   static char *monthName[13] = {"", "January",
      "February", "March", "April", "May", "June",
      "July", "August", "September", "October",
      "November", "December"};

   output << monthName[d.month] << ' '
          << d.day << ", " << d.year;

   return output;   // enables concatenation
}
```

Fig. 18.6 Definición de función miembro para la clase **Date** (parte 4 de 6).

```
// FIG18_6.CPP
// Driver for class Date
#include <iostream.h>
#include "date1.h"

main()
{
   Date d1, d2(12, 27, 1992), d3(0, 99, 8045);
   cout << "d1 is " << d1
        << "\nd2 is " << d2
        << "\nd3 is " << d3;

   cout << "\n\nd2 += 7 is " << (d2 += 7) << "\n\n";

   d3.setDate(2, 28, 1992);
   cout << "  d3 is " << d3;
   cout << "\n++d3 is " << ++d3 << "\n\n";

   Date d4(3, 18, 1969);

   cout << "Testing the preincrement operator:\n"
        << "  d4 is " << d4
        << "\n++d4 is " << ++d4
        << "\n  d4 is " << d4;

   cout << "\n\nTesting the postincrement operator:\n"
        << "  d4 is " << d4
        << "\nd4++ is " << d4++
        << "\n  d4 is " << d4 << endl;

   return 0;
}
```

Fig. 18.6 Manejador para la clase **Date** (parte 5 de 6).

```
d1 is January 1, 1900
d2 is December 27, 1992
d3 is January 1, 1900

d2 += 7 is January 3, 1993

  d3 is February 28, 1992
++d3 is February 29, 1992

Testing the preincrement operator:
  d4 is March 18, 1969
++d4 is March 19, 1969
  d4 is March 19, 1969

Testing the postincrement operator:
  d4 is March 19, 1969
d4++ is March 19, 1969
  d4 is March 20, 1969
```

Fig. 18.6 Salida correspondiente al manejador de la clase **Date** (parte 6 de 6).

Resumen

- En C++ el operador << es utilizado para varios fines, como operador de inserción de flujo y como operador de desplazamiento a la izquierda. Esto es un ejemplo de homonimia de operadores. De igual forma, >> también tiene homónimo; se utiliza tanto como operador de extracción de flujo, como operador de desplazamiento a la derecha.
- En forma más general, C++ le permite al programador hacer la homonimia de la mayor parte de los operadores, para que sean sensibles al contexto en el cual están siendo utilizados. El compilador genera el código apropiado basándose en la forma en la cual cada operador es utilizado.
- La homonimia de operadores contribuye a la extensibilidad de C++.
- Se lleva a cabo la homonimia de operadores al escribir una definición de función (incluyendo un encabezado y un cuerpo). El nombre de la función se convierte en la palabra reservada **operator**, seguida por el símbolo correspondiente al operador homónimo.
- Para utilizar un operador sobre objetos de clase, dicho operador *debe* ser un operador homónimo con dos excepciones. El operador de asignación (=) puede ser utilizado con cualquier clase para llevar a cabo una copia a nivel de miembro por omisión, sin necesidad de tener homónimo. El operador de dirección (&) también puede ser utilizado sobre objetos de cualquier clase sin tener homónimo; sólo devuelve la dirección en la memoria del objeto.
- Al hacer la homonimia de operadores la idea subyacente es disponer de las mismas expresiones concisas para tipos definidos por usuario que C++ proporciona con su rica colección de operadores para tipos incorporados.
- La mayor parte de los operadores de C++ pueden tener homónimos.

- La homonimia de un operador no puede modificar su precedencia y asociatividad.
- En un operador homónimo no se pueden utilizar argumentos por omisión.
- No es posible modificar el número de operandos que un operador toma: los operadores unarios homónimos se conservan como operadores unarios; los operadores binarios homónimos se conservan como operadores binarios.
- No es posible crear símbolos para operadores nuevos; sólo se puede hacer la homonimia de operadores existentes.
- El significado de cómo opera un operador sobre objetos de tipos incorporados no puede ser modificado mediante la homonimia.
- Al hacer la homonimia de `()`, `[]`, `->`, o `=`, la función operador homónima debe ser declarada como miembro de clase.
- Las funciones operador pueden ser funciones miembro o funciones no miembro.
- Cuando una función operador es puesta en práctica como función miembro, el operando más a la izquierda debe ser un objeto de clase (o una referencia a un objeto de clase) de la clase del operador.
- Si el operando a la izquierda tiene que ser un objeto de clase distinta, esta función operador debe ser puesta en práctica como una función no miembro.
- Las funciones miembro de operador son llamadas sólo cuando el operando izquierdo de un operador binario es en específico un objeto de dicha clase o cuando el único operando de un operador unario es un objeto de dicha clase.
- Se podría escoger una función no miembro para hacer la homonimia de un operador, a fin de permitir que el operador sea conmutativo.
- Se puede hacer la homonimia de un operador unario como función miembro no estática sin argumentos o como función no miembro con un argumento; dicho argumento debe ser un objeto de un tipo definido por usuario o una referencia a un objeto de un tipo definido por usuario.
- Se puede hacer la homonimia de un operador binario como función miembro no estática con un argumento, o como función no miembro con dos argumentos (uno de los cuales debe ser un objeto de clase o una referencia a un objeto de clase).
- El operador de subíndice de arreglo `[]` no está restringido en su uso sólo con arreglos; puede ser utilizado para seleccionar elementos de otros tipos de clases contenedor como son listas enlazadas, cadenas, diccionarios y demás. Además, ya no es necesario que los subíndices sean enteros, también pueden ser utilizados caracteres o cadenas, por ejemplo.
- Un constructor de copia se utiliza para inicializar un objeto con otro objeto de la misma clase. Los constructores de copia también son invocados siempre que se requiera de la copia de un objeto, como es el caso en la llamada por valor o al regresar un valor de una función llamada.
- El compilador no sabe en forma automática cómo convertir entre tipos definidos por usuario y tipos incorporados. El programador debe especificar en forma explícita cómo ocurrirán dichas conversiones. Estas conversiones pueden ser llevadas a cabo mediante constructores de conversión (es decir, constructores de un argumento) que sólo convierten objetos de otros tipos en objetos de una clase particular.
- Un operador de conversión (u operador cast) puede ser utilizado para convertir un objeto de una clase a un objeto de otra o a un objeto de un tipo incorporado. Dicho operador de conversión

debe ser una función miembro no estática; este tipo de operador de conversión no puede ser una función amigo.

- Un constructor de conversión es un constructor de un argumento, que es utilizado para convertir el argumento en el objeto de la clase del constructor. El compilador puede llamar de forma implícita a un constructor de este tipo.
- El operador de asignación es el operador homónimo más frecuente. Por lo regular se utiliza para asignar un objeto a otro objeto de la misma clase, pero mediante el uso de los constructores de conversión; también puede ser utilizado para asignaciones entre clases.
- Si no se define un operador de asignación homónimo, la asignación se permitirá, pero será por omisión a una copia a nivel de miembro de cada miembro de datos. En algunos casos esto es aceptable. En el caso de objetos que contengan apuntadores a almacenamiento dinámico asignado, la copia a nivel de miembro dará como resultado dos objetos distintos apuntando al mismo almacenamiento asignado dinámicamente. Cuando se llame el destructor correspondiente a cualquiera de estos objetos, se liberará el almacenamiento dinámico asignado. Si el otro objeto entonces quiere referirse a dicho almacenamiento, el resultado quedará indefinido.
- Para hacer la homonimia del operador de incremento y permitir el uso tanto de preincremento como de postincremento, cada función de operador homónima debe tener una signatura distinta o diferente, de tal forma, que el compilador pueda ser capaz de determinar qué versión de `++` debe utilizar. Las versiones prefijas se hacen homónimas de igual forma que cualquier operador unario prefijo. El proporcionar una firma única a la función de operador de postincremento se consigue incluyendo un segundo argumento que debe ser del tipo `int`. De hecho, el usuario no proporciona un valor para este argumento entero especial. Aparece ahí simplemente para ayudar al compilador a distinguir entre versiones prefijas y postfijas de los operadores de incremento y de decremento.

Terminología

tipo incorporado
clase `Array`
clase `Date`
clase `String`
llamadas de función concatenadas
resolución de conversión de ambigüedad
constructor de conversión
función de conversión
operador de conversión
conversiones entre clases de tipos diferentes
conversiones entre tipos y clases incorporados
constructor de copia
conversiones explícitas de tipo (mediante cast)
operador homónimo de función amigo
homonimia de función
conversiones implícitas de tipo
operador homónimo de función miembro
operadores no sujetos de homonimia
palabra reservada `operator`
homonimia de operadores
operadores como funciones
operadores capaces de homonimia
operador homónimo `==`
operador homónimo `!=`
operador homónimo `<`
operador homónimo `<=`
operador homónimo `>`
operador homónimo `>=`
operador de asignación homónimo (`=`)
operador homónimo `[]`
hacer la homonimia
homonimia de un operador binario
homonimia de un operador unario
homonimia de un operador postfijo
homonimia de un operador prefijo
conversión definida por usuario
tipo definido por usuario

Errores comunes de programación

18.1 Intentar crear operadores nuevos.
18.2 Intentar modificar cómo funciona un operador con objetos de tipos incorporados.
18.3 Suponer que hacer la homonimia de un operador (como +) hace de forma automática la homonimia de sus operadores relacionados (como +=). Los operadores sólo pueden tener homónimos en forma explícita; la homonimia implícita no existe.
18.4 No probar la existencia y evitar la autoasignación cuando se hace la homonimia del operador de asignación de una clase que incluya un apuntador a almacenamiento dinámico.
18.5 No proporcionar un operador de asignación homónimo y un constructor de copia para una clase, cuando los objetos de dicha clase contengan apuntadores a almacenamiento dinámico asignado.

Prácticas sanas de programación

18.1 Utilice la homonimia de operadores cuando ésta haga más claro un programa si efectúa las mismas operaciones mediante llamadas de función explícitas.
18.2 Evite un uso excesivo o inconsistente de la homonimia de operadores, ya que ello puede hacer que un programa resulte críptico o difícil de leer.
18.3 Haga la homonimia de operadores para llevar a cabo la misma función o funciones bastante similares sobre objetos de clase que dichos operadores ejecutan sobre objetos de tipos incorporados.
18.4 Antes de escribir programas en C++ utilizando operadores homónimos, consulte los manuales de C++ correspondientes a su compilador, para informarse de las varias restricciones y requisitos únicos a operadores particulares.
18.5 Al hacer la homonimia de operadores unarios, es preferible hacer la funciones operador miembros de clase en vez de funciones amigo no miembro. Esta es una solución más limpia. Las funciones amigo y las clases amigo deberán evitarse, salvo que sean en lo absoluto necesarias. La utilización de amigos viola el encapsulado de una clase.
18.6 Un destructor, el operador de asignación, y un constructor de copia para una clase por lo general se proporcionan en grupo.

Sugerencias de rendimiento

18.1 Es posible hacer la homonimia de un operador como función no miembro, y no amigo, pero dicha función con la necesidad de tener acceso a datos privados o protegidos de una clase requeriría utilizar las funciones set o get proporcionadas en la interfaz pública de la clase. La sobrecarga de llamar a estas funciones causaría bajo rendimiento.
18.2 Usar el operador de concatenación homónimo `+=` que toma un solo argumento del tipo `const char *` da resultados más eficaces que primero tener que efectuar la conversión implícita y a continuación la concatenación. Las conversiones implícitas requieren de menos código y causan menos errores.

Observaciones de ingeniería de software

18.1 En una función operador, por lo menos un argumento debe ser un objeto de clase o una referencia a un objeto de clase. Esto impedirá que los programadores modifiquen cómo funcionan los operadores sobre objetos de tipos incorporados.
18.2 Se pueden añadir nuevas capacidades de entrada/salida a C++ correspondientes a tipos definidos por usuario, sin modificar las declaraciones o los miembros de datos privados correspondientes a la clase `ostream` o a la clase `istream`. Esto promueve la extensibilidad del lenguaje de programación C++ —uno de los aspectos más atractivos de C++.

18.3 Es posible evitar que un objeto de clase sea asignado a otro. Esto se lleva a cabo definiendo el operador de asignación como miembro privado de la clase.

Ejercicios de autoevaluación

18.1 Llene cada uno de los siguientes espacios en blanco:

a) Suponga que `a` y `b` son variables enteras y que formamos la suma `a + b`. Ahora suponga que `c` y `d` son variables de punto flotante y que formamos la suma `c + d`. Los dos operadores `+` están siendo utilizados aquí con claridad para fines distintos. Esto es un ejemplo de _____.

b) La palabra reservada _____ introduce una definición de función de operador homónimo.

c) Para utilizar operadores sobre objetos de clase, deben tener homónimos a excepción de los operadores _____ y _____.

d) La _____ y _____ de un operador no pueden ser modificadas mediante la homonimia.

18.2 Explique los varios significados de los operadores `<<` y `>>` en C++.

18.3 ¿En que contexto pudiera ser utilizado el nombre `operator/` en C++?.

18.4 (Verdadero/falso). En C ++ solo los operadores existentes pueden tener homónimos.

18.5 ¿Cómo se compara la precedencia de un operador homónimo en C++ con la precedencia del operador original?.

Respuestas a los ejercicios de autoevaluación

18.1 a) homonimia de operador. b) `operator`. c) asignación (=), dirección(`&`). d) precedencia, asociatividad.

18.2 El operador `>>` es, dependiendo del contexto, a la vez operador de desplazamiento a la derecha y operador de extracción de flujo. El operador `<<` es a la vez operador de desplazamiento a la izquierda y operador de inserción de flujo, dependiendo del contexto.

18.3 Para la homonimia de operadores: sería el nombre de una función que proporcionaría una nueva versión del operador `/`.

18.4 Verdadero.

18.5 Idéntico.

Ejercicios

18.6 De tantos ejemplos como pueda de la homonimia de operadores implícita en C. De tantos ejemplos como pueda de la homonimia de operadores implícita en C++. De un ejemplo razonable de una situación en la cual pudiera usted desear hacer la homonimia explícita de un operador en C++.

18.7 Los operadores C++ que no pueden tener homónimos son _____, _____, _____, _____ y _____.

18.8 La concatenación de cadenas requiere de dos operandos —las dos cadenas que han de concatenarse. En el texto mostramos cómo poner en práctica un operador de concatenación homónimo, que concatena el segundo objeto `String` a la derecha del primer objeto `String`, modificando así el primer objeto `String`. En algunas aplicaciones, es deseable producir un objeto concatenado `String` sin modificar los dos argumentos `String`. Diseñe un `operator+` para permitir operaciones como

```
string1 = string2 + string3;
```

18.9 *(Ejercicio de homonimia de operadores final.)* Para poder valorizar el cuidado que debe ponerse en la selección de operadores para su homonimia, liste cada uno de los operadores que se puedan hacer

homónimos en C++ y por cada uno de ellos liste un significado posible (o varios, si es apropiado) para cada una de las distintas clases que ha estudiado en este curso. Sugerimos que pruebe:

a) Arreglos
b) Pilas
c) Cadenas

Una vez hecho lo anterior, comente sobre qué operadores parecen tener significado para una amplia variedad de clases. ¿Qué operadores parecen tener poco valor para homonimia? ¿Qué operadores parecen ambiguos?

18.10 Ahora ejecute el proceso descrito en el programa anterior pero a la inversa. Liste cada uno de los operadores capaces de homonimia en C++. Para cada uno de ellos, liste lo que usted sienta es quizás la "operación óptima" para la cual dicho operador debería ser utilizado. Si existen varias operaciones excelentes, lístelas todas.

18.11 *(Proyecto).* C++ es un lenguaje en evolución, y continuamente se están desarrollando nuevos lenguajes. ¿Qué operadores adicionales recomendaría usted añadir a C++ o a un lenguaje similar a C++, que apoyasen tanto a la programación procedural como a la programación orientada a objetos? Escriba una cuidadosa justificación. Pudiera inclusive considerar enviar sus sugerencias al Comité ANSI C++.

18.12 Un buen ejemplo de la homonimia del operador de llamada de función `()` es permitir la forma más común de dobles subíndices de arreglo. En vez de decir

```
chessBoard[row][column]
```

en relación con un arreglo de objetos, haga la homonimia del operador de llamada de función para que pueda utilizarse la forma alterna

```
chessBoard(row, column)
```

18.13 Haga la homonimia del operador de subíndice para regresar un miembro dado en una lista enlazada.

18.14 Haga la homonimia del operador de subíndice para regresar el elemento más grande de una colección, el segundo más grande, el tercero más grande, etcétera.

18.15 Considere la clase `Complex` mostrada en la figura 18.7. La clase permite operaciones de los *números* llamados *complejos*. Estos son números de forma `realPart + imaginaryPart * i`, donde *i* tiene el valor:

$$\sqrt{-1}$$

a) Modifique la clase, para permitir la entrada y salida de números complejos, mediante los operadores homónimos `>>` y `<<`, respectivamente (deberá eliminar la función de impresión de la clase).

b) Haga la homonimia del operador de multiplicación, para permitir la multiplicación de números complejos, como en álgebra.

c) Haga la homonimia de los operadores `==` y `!=`, para permitir comparaciones de números complejos.

18.16 Una máquina con enteros de 32 bits puede representar enteros en el rango de aproximadamente -2 mil millones hasta +2 mil millones. Esta restricción de tamaño fijo rara vez causa problemas. Pero existen muchas aplicaciones en las cuales nos gustaría ser capaces de utilizar un rango mucho más amplio de enteros. Para esto fue construido C++, es decir para crear nuevos tipos poderosos de datos. Considere la clase `HugeInt` de la figura 18.8. Estudie cuidadosamente esta clase, y después

a) Describa con precisión cómo funciona.

b) ¿Qué restricciones tiene la clase?

c) Modifique la clase para que pueda procesar enteros arbitrariamente grandes. (Sugerencia: utilice una lista enlazada para representar a `HugeInt`.)

d) Haga la homonimia del operador de multiplicación `*`.

e) Haga la homonimia del operador de división `/`.

f) Haga la homonimia de todos los operadores relacionales y de igualdad.

```
// COMPLEX1.H
// Definition of class Complex
#ifndef COMPLEX1_H
#define COMPLEX1_H

class Complex {
public:
   Complex(double = 0.0, double = 0.0);       // constructor
   Complex operator+(const Complex &) const; // addition
   Complex operator-(const Complex &) const; // subtraction
   Complex &operator=(const Complex &);       // assignment
   void print() const;                        // output
private:
   double real;      // real part
   double imaginary; // imaginary part
};

#endif
```

Fig. 18.7 Definición de la clase **Complex** (parte 3 de 5).

```
// COMPLEX1.CPP
// Member function definitions for class Complex
#include <iostream.h>
#include "complex1.h"

// Constructor
Complex::Complex(double r, double i)
{
   real = r;
   imaginary = i;
}

// Overloaded addition operator
Complex Complex::operator+(const Complex &operand2) const
{
   Complex sum;
   sum.real = real + operand2.real;
   sum.imaginary = imaginary + operand2.imaginary;
   return sum;
}

// Overloaded subtraction operator
Complex Complex::operator-(const Complex &operand2) const
{
   Complex diff;
   diff.real = real - operand2.real;
   diff.imaginary = imaginary - operand2.imaginary;
   return diff;
}
```

Fig. 18.7 Definiciones de función miembro para la clase **Complex** (parte 2 de 5).


```
// Overloaded = operator
Complex& Complex::operator=(const Complex &right)
{
   real = right.real;
   imaginary = right.imaginary;
   return *this;   // enables concatenation
}

// Display a Complex object in the form: (a, b)
void Complex::print() const
   { cout << '(' << real << ", " << imaginary << ')'; }
```

Fig. 18.7 Definiciones de función miembro para la clase **Complex** (parte 3 de 5).

```
// FIG18_7.CPP
// Driver for class Complex
#include <iostream.h>
#include "complex1.h"

main()
{
   Complex x, y(4.3, 8.2), z(3.3, 1.1);

   cout << "x: ";
   x.print();
   cout << "\ny: ";
   y.print();
   cout << "\nz: ";
   z.print();

   x = y + z;
   cout << "\n\nx = y + z:\n";
   x.print();
   cout << " = ";
   y.print();
   cout << " + ";
   z.print();

   x = y - z;
   cout << "\n\nx = y - z:\n";
   x.print();
   cout << " = ";
   y.print();
   cout << " - ";
   z.print();
   cout << '\n';

   return 0;
}
```

Fig. 18.7 Manejador para la clase **Complex** (parte 4 de 5).

```
x: (0, 0)
y: (4.3, 8.2)
z: (3.3, 1.1)

x = y + z:
(7.6, 9.3) = (4.3, 8.2) + (3.3, 1.1)

x = y - z:
(1, 7.1) = (4.3, 8.2) - (3.3, 1.1)
```

Fig. 18.7 Salida del manejador para la clase **Complex** (parte 5 de 5).

```
// HUGEINT1.H
// Definition of the HugeInt class
#ifndef HUGEINT1_H
#define HUGEINT1_H

#include <iostream.h>

class HugeInt {
   friend ostream &operator<<(ostream &, HugeInt &);
public:
   HugeInt(long = 0);             // conversion constructor
   HugeInt(const char *);         // conversion constructor
   HugeInt operator+(HugeInt &); // addition
private:
   short integer[30];
};

#endif
```

Fig. 18.8 Clase de grandes enteros definida por usuario (parte 1 de 4).

```
// HUGEINT1.CPP
// Member and friend function definitions for class HugeInt
#include <string.h>
#include "hugeint1.h"

// Conversion constructor
HugeInt::HugeInt(long val)
{
   for (int i = 0; i <= 29; i++)
      integer[i] = 0;    // initialize array to zero

   for (i = 29; val != 0 && i >= 0; i--) {
      integer[i] = val % 10;
      val /= 10;
   }
}
```

Fig. 18.8 Clase de grandes enteros definida por usuario (parte 2 de 4).


```
HugeInt::HugeInt(const char *string)
{
   int i, j;

   for (i = 0; i <= 29; i++)
      integer[i] = 0;

   for (i = 30 - strlen(string), j = 0; i <= 29; i++, j++)
      integer[i] = string[j] - '0';
}

// Addition
HugeInt HugeInt::operator+(HugeInt &op2)
{
   HugeInt temp;
   int carry = 0;

   for (int i = 29; i >= 0; i--) {
      temp.integer[i] = integer[i] + op2.integer[i] + carry;

      if (temp.integer[i] > 9) {
         temp.integer[i] %= 10;
         carry = 1;
      }
      else
         carry = 0;
   }

   return temp;
}

ostream& operator<<(ostream &output, HugeInt &num)
{
   for (int i = 0; (num.integer[i] == 0) && (i <= 29); i++)
      ; // skip leading zeros

   if (i == 30)
      output << 0;
   else
      for ( ; i <= 29; i++)
         output << num.integer[i];

   return output;
}
```

Fig. 18.8 Clase de grandes enteros definida por usuario (parte 3 de 4).

18.17 Cree una clase **rationalNumber** (fracciones) con las siguientes capacidades:

a) Cree un constructor que impida la existencia de un denominador 0 en una fracción, que simplifique las fracciones que no estén en forma simplificada, y que evite denominadores negativos.

b) Haga la homonimia de los operadores de suma, resta, multiplicación y división para esta clase.

c) Haga la homonimia de los operadores relacionales y de igualdad para esta clase.

```
// FIG18_8.CPP
// Test driver for HugeInt class
#include <iostream.h>
#include "hugeint1.h"

main()
{
   HugeInt n1(7654321), n2(7891234),
           n3("99999999999999999999999999999"),
           n4("1"), n5;

   cout << "n1 is " << n1 << "\nn2 is " << n2
        << "\nn3 is " << n3 << "\nn4 is " << n4
        << "\nn5 is " << n5 << "\n\n";

   n5 = n1 + n2;
   cout << n1 << " + " << n2 << " = " << n5 << "\n\n";

   cout << n3 << " + " << n4 << "\n= " << (n3 + n4) << "\n\n";

   n5 = n1 + 9;
   cout << n1 << " + " << 9 << " = " << n5 << "\n\n";

   n5 = n2 + "10000";
   cout << n2 << " + " << "10000" << " = " << n5 << "\n\n";

   return 0;
}
```

```
n1 is 7654321
n2 is 7891234
n3 is 99999999999999999999999999999
n4 is 1
n5 is 0

7654321 + 7891234 = 15545555

99999999999999999999999999999 + 1
= 100000000000000000000000000000

7654321 + 9 = 7654330

7891234 + 10000 = 7901234
```

Fig. 18.8 Clase de grandes enteros definida por usuario (parte 4 de 4).

18.18 El operador **sizeof** en un objeto **String** sólo devuelve el tamaño de los miembros de datos del objeto **String**; no incluye la longitud del espacio **String** asignado por **new**. Haga la homonimia del operador **sizeof** para que regrese el tamaño del objeto **String**, incluyendo la memoria dinámicamente asignada.

18.19 Estudie las funciones de biblioteca de manejo de cadenas de C y ponga en práctica cada una de las funciones como parte de la clase **String**. A continuación utilice estas funciones para llevar a cabo manipulaciones de texto.

18.20 Desarrolle la clase **Polynomial**. La representación interna de un **Polynomial** es una lista de términos enlazados. Cada término tiene un coeficiente y un exponente. El término

$$2x^4$$

tiene un coeficiente de 2 y un exponente de 4. Desarrolle una clase completa que contenga las funciones destructor y constructor apropiadas así como las funciones *set* y *get*. La clase también deberá proporcionar las siguientes capacidades de operadores homónimos:

a) Haga la homonimia del operador de suma (+) para poder sumar dos **Polynomial**.
b) Haga la homonimia del operador de resta (-) para substraer dos **Polynomials**.
c) Haga la homonimia del operador de asignación, para asignar un **Polynomial** a otro.
d) Haga la homonimia del operador de multiplicación (*) para multiplicar dos **Polynomials**.
e) Haga la homonimia del operador de asignación de adición (+=), del operador de asignación de substracción (-=) y del operador de asignación de multiplicación (*=).

19

Herencia

Objetivos

- Ser capaz de crear nuevas clases heredando de clases existentes.
- Comprender como la herencia fomenta la reutilización del software.
- Comprender los conceptos de clases base y clases derivadas.
- Ser capaz de utilizar herencia múltiple para derivar una clase a partir de varias clases base.

No diga que conoce por completo a otro, hasta que con él haya dividido una herencia.
Johann Kasper Lavater

Este método es para definir como el número de una clase la clase de todas las clases similares a la clase dada.
Bertrand Russell

Un mazo de cartas fue diseñado como la más pura de las jerarquías, siendo cada una de las cartas superior a todas las que están debajo de ella, y lacayo de las que están por encima.
Ely Culbertson

Aunque es muy bueno heredar una biblioteca, mejor aún es reunirla.
Augustine Birrell

Ahorre autoridad base de libros de terceros.
William Shakespeare
Love's Labour's Lost

Sinopsis

19.1 Introducción

En éste y en el capítulo siguiente analizamos dos de las capacidades de mayor importancia que proporciona la programación orientada a objetos *herencia* y *polimorfismo*. La herencia es una forma de *reutilización del software*, en la cual se crean clases nuevas a partir de clases existentes, mediante la absorción de sus atributos y comportamientos, y embelleciendo éstos con las capacidades que las clases nuevas requieren. La reutilización del software ahorra tiempo en el desarrollo de programas, fomenta la reutilización de software de alta calidad, probado y depurado, y reduce problemas en un sistema después de convertido en funcional. Estas son excitantes posibilidades. El polimorfismo nos permite escribir programas en forma general, a fin de procesar una amplia variedad de clases relacionadas tanto existentes como aún sin especificar. La herencia y el polimorfismo son técnicas efectivas para enfrentarse con la complejidad del software.

Al crear una clase nueva, en vez de escribir en su totalidad miembros de datos y funciones miembro nuevos, el programador puede determinar que la clase nueva debe *heredar* los miembros de datos y las funciones miembro provenientes de una *clase base* ya definida. La clase nueva se

conoce como *clase derivada*. Cada clase derivada misma se convierte en candidato a clase base, para alguna clase derivada futura. Mediante la *herencia única*, una clase es derivada de una única clase base. Con la *herencia múltiple* una clase derivada hereda de múltiples clases base (posiblemente no relacionadas). A menudo una clase derivada añade miembros de datos y funciones miembro propias, por lo que en general una clase derivada es más grande que su clase base. Una clase derivada es más específica que su clase base y representa un grupo más pequeño de objetos. Con la herencia única, la clase derivada se inicia en esencia de la misma forma que la clase base. La fuerza real de la herencia proviene de la capacidad de definir en la clase derivada, adiciones, reemplazos o refinamientos de las características heredadas de la clase base.

Cada objeto de una clase derivada también es un objeto de la clase base de dicha clase derivada. Sin embargo; la inversa no es verdad, los objetos de la clase base no son objetos de la clase derivada de dicha clase base. Nos aprovecharemos de esta relación "un objeto de clase derivada es un objeto de clase base" para ejecutar algunas manipulaciones interesantes. Por ejemplo, podemos unir una gran variedad de objetos distintos relacionados mediante la herencia en una lista enlazada de objetos de clase base. Esto permite que una variedad de objetos se procesen de forma general. Como veremos en éste y en el siguiente capítulo, esto es un impulso clave a la programación orientada a objetos.

En este capítulo añadimos una nueva forma de control de acceso de miembros, es decir el acceso protegido. Para el acceso a los miembros de clase base protegidos, las clases derivadas y sus amigos reciben un trato favorecido en relación con otras funciones.

La experiencia en la elaboración de sistemas de software indica que grandes porciones de código se refieren a casos especiales muy relacionados. En dichos sistemas se hace difícil visualizar la "imagen general" porque el diseñador y el programador se han visto preocupados por los casos especiales. La programación orientada a objetos proporciona varias formas de "ver el bosque a través de los árboles" —un proceso conocido a veces como *abstracción*.

Si un programa está lleno de casos especiales muy relacionados, entonces es común ver enunciados `switch`, que distinguen entre casos especiales, que proporcionan la lógica de proceso necesaria para manejar cada caso en forma individual. En el capítulo 20, mostraremos cómo utilizar la herencia y el polimorfismo para reemplazar la lógica `switch` por una lógica más simple.

Distinguimos entre *relaciones "es un"* y *relaciones "tiene un"*. "Es un" es herencia. En una relación "es un", un objeto de un tipo de clase derivada también puede ser tratado como un objeto del tipo de clase base. "Tiene un" es composición (véase la figura 17.4). En una relación "tiene un" un objeto de clase tiene como miembros uno o más objetos de otras clases.

Una clase derivada no puede tener acceso a los miembros privados de su clase base; permitirlo violaría el encapsulado de la clase base. Una clase derivada puede, sin embargo, tener acceso a los miembros públicos y protegidos de su clase base. Los miembros de clase base que no deban ser accesibles a una clase derivada mediante la herencia, en la clase base se declararán privados. Una clase derivada puede tener acceso a los miembros privados de la clase base mediante funciones de acceso, provistas en la interfaz pública de la clase base. Un problema, tratándose de la herencia, es que una clase derivada puede heredar funciones miembro públicas que no requiera, y que expresamente no tendría. Cuando un miembro de clase base no es apropiado para una clase derivada, dicho miembro puede ser redefinido en la clase derivada con una puesta en práctica apropiada.

De lo más excitante resulta la idea que las clases nuevas pueden heredar de *bibliotecas de clase* existentes. Las organizaciones desarrollan sus propias bibliotecas de clase y pueden aprovechar de otras bibliotecas disponibles en todo el mundo. La perspectiva es que, en algún momento, el software será elaborado a partir de *componentes reutilizables estandarizados*, de la

misma forma que, hoy día, a menudo es construido el hardware. Esto ayudará a estar a la altura del reto de desarrollar el software más y más poderoso que en el futuro necesitaremos.

19.2 Clases base y clases derivadas

A menudo un objeto de clase en realidad también "es un" objeto de otra clase. Un rectángulo *es un* cuadrilátero (como lo es un cuadrado, un paralelogramo y un trapezoide). Entonces, la clase **Rectangle** puede decirse que *hereda* de la clase **Quadrilateral**. En este contexto, la clase **Quadrilateral** se conoce como *clase base* y la clase **Rectangle** se conoce como *clase derivada*. Un rectángulo *es un* tipo específico de un cuadrilátero, pero sería incorrecto decir que un cuadrilátero *es un* rectángulo. En la figura 19.1 se muestran varios ejemplos sencillos de herencia.

Otros lenguajes de programación orientados a objetos, como Smalltalk, utilizan diferentes terminologías: en la herencia, la clase base se conoce como *superclase* y la clase derivada como *subclase*. Dado que por lo general la herencia produce clases derivadas de tamaño mayor que las clases base, los términos superclase y subclase parecen inapropiados; evitaremos utilizar estos términos.

La herencia forma estructuras jerárquicas de apariencia arborescente. Una clase base existe en una relación jerárquica con sus clases derivadas. Ciertamente una clase puede existir por sí misma, pero es cuando una clase es utilizada mediante el mecanismo de la herencia, que ésta se convierte en clase base, proveyendo atributos y comportamientos a otras clases, o la clase se convierte en una clase derivada, que hereda dichos atributos y comportamientos.

Desarrollemos una jerarquía simple de herencia. Una comunidad universitaria típica está formada por miles de personas que son miembros de la misma. Estas personas son los empleados y los estudiantes. Los empleados son o miembros de la facultad o personal de oficinas. Los miembros de la facultad son o administradores (rectores o coordinadores de departamentos) o la facultad de profesores. Esto da como resultado la jerarquía de herencia que se muestra en la figura 19.2.

Clase base	Clases derivadas
Student	CommuterStudent
	ResidentStudent
Shape	Circle
	Triangle
	Rectangle
Loan	CarLoan
	HomeImprovementLoan
	MortgageLoan
Employee	FacultyMember
	StaffMember
Account	CheckingAccount
	SavingsAccount

Fig. 19.1 Algunos ejemplos simples de herencia.

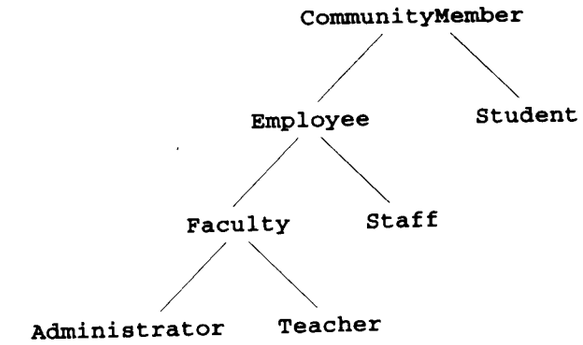


Fig. 19.2 Una jerarquía de herencia para los miembros de una comunidad universitaria.

Otra jerarquía de herencia sustancial es la jerarquía **Shape** de la figura 19.3. Un comentario común entre aquellos alumnos que estudian por primera vez la programación orientada a objetos, es que en el mundo real existen ejemplos abundantes de jerarquías. Es simple que a estos alumnos jamás se les ha pedido que categoricen el mundo real de esta forma, por lo que deben hacer algunos ajustes en su forma de pensar.

Para especificar que la clase **CommissionWorker** se deriva de la clase **Employee**, típicamente la clase **CommissionWorker** sería definida como sigue:

```
class CommissionWorker : public Employee {
   ...
};
```

Esto se conoce como *herencia pública* y es el tipo que más probablemente utilice el lector. También analizaremos *herencia privada* y *herencia protegida*. En el caso de la herencia pública, los miembros públicos y protegidos de la clase base son heredados como miembros públicos y protegidos de la clase derivada, respectivamente.

Es posible tratar en forma similar a los objetos de la clase base y a los objetos de la clase derivada. El estado común es expresado en los atributos y comportamientos de la clase base. Los objetos de cualquier clase derivada pública originados de una clase base común, pueden todos ellos ser tratados como objetos de dicha clase base.

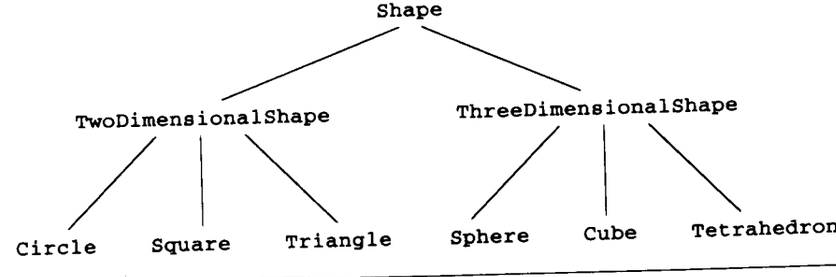


Fig. 19.3 Una porción de la jerarquía de clase **Shape**.

Veremos muchos ejemplos en los cuales aprovecharemos esta relación con una facilidad de programación no disponible en lenguajes no orientados a los objetos como es C.

19.3 Miembros protegidos

Los miembros públicos de una clase base son accesibles a todas las funciones en el programa. Los miembros privados de clase base son accesibles sólo para las funciones miembro y los amigos de la clase base.

El acceso protegido sirve como nivel intermedio de protección entre el acceso público y el acceso privado. Los miembros protegidos de clase base son accesibles sólo por miembros y amigos de la clase base, y por miembros y amigos de las clases derivadas.

Los miembros de clases derivadas pueden hacer referencia a miembros públicos y protegidos de la clase base sólo utilizando los nombres de los miembros. No es necesario utilizar el operador de resolución de alcance. Se supone el objeto actual.

19.4 Cómo hacer la conversión explícita (cast) de apuntadores de clase base a apuntadores de clase derivada

Un objeto de clase derivada pública también puede ser tratado como objeto de su clase base correspondiente. Esto permite algunas manipulaciones interesantes. Por ejemplo, a pesar del hecho que objetos de una variedad de clases derivadas de clase base particular pudieran ser muy distintos unos de los otros, aún así, podemos crear una lista enlazada de los mismos otra vez, siempre y cuando los tratemos como objetos de clase base. Pero lo inverso no es cierto: un objeto de clase base no es también automáticamente un objeto de clase derivada.

Error común de programación 19.1

Puede causar errores tratar un objeto de clase base como si fuera un objeto de clase derivada.

El programa podrá, sin embargo; utilizar un especificador de conversión explícita (cast) para convertir uno de clase base a un apuntador de clase derivada. Pero tenga cuidado si dicho apuntador debe ser desreferenciado, entonces primero deberá hacer que señale a un objeto del tipo de clase derivada.

Error común de programación 19.2

Efectuar la conversión explícita de un apuntador de clase base que señala a un objeto de clase base a un apuntador de clase derivada y a continuación, hacer referencia a miembros de clase derivada que no existen en dicho objeto.

Nuestro primer ejemplo aparece en la figura 19.4. partes 1 hasta la 5. Las partes 1 y 2 muestran la definición de clase **Point** y las definiciones de funciones miembro **Point**. Las partes 3 y 4 muestran la definición de clase **Circle** y la definición de función miembro **Circle**. La parte 5 muestra un programa manejador, en el cual demostramos la asignación de apuntadores de clase derivada a apuntadores de clase base, y la conversión explícita (cast) de apuntadores de base clase a apuntadores de clase derivada.

Examinemos primero la definición de clase **Point** (figura 19.4, parte 1). La interfaz pública a **Point** tiene las funciones miembro **setPoint, getX**, y **getY**. Los miembros de datos **x** y **y** de **Point** se definen como **protected**. Esto impide que los usuarios de los objetos **Point** tengan acceso directo a los datos, pero permite que las clases que se deriven de **Point** tengan acceso directo a los miembros de datos heredados. Si los datos fueran definidos como **private**, para tener acceso a los datos tendrían que ser utilizadas las funciones miembro públicas de **Point**.

Note que la función de operador de inserción de flujo homónima **Point** (figura 19.4 parte 2), es capaz de hacer referencia directa a las variables **x** y **y** porque éstas son miembros protegidos de la clase **Point**. Note también que es necesario hacer referencia a **x** y a **y** mediante objetos como en **p.x** y **p.y**. Esto es debido a que la función de operador de inserción de flujo homónima no es una función miembro de la clase **Circle**, sino que es un amigo de la clase.

```
// POINT.H
// Definition of class Point
#ifndef POINT_H
#define POINT_H

class Point {
   friend ostream &operator<<(ostream &, const Point &);
public:
   Point(float = 0, float = 0);       // default constructor
   void setPoint(float, float);       // set coordinates
   float getX() const { return x; }   // get x coordinate
   float getY() const { return y; }   // get y coordinate
protected:          // accessible by derived classes
   float x, y;      // x and y coordinates of the Point
};

#endif
```

Fig. 19.4 Definición de la clase **Point** (parte 1 de 5).

```
// POINT.CPP
// Member functions for class Point
#include <iostream.h>
#include "point.h"

// Constructor for class Point
Point::Point(float a, float b)
{
   x = a;
   y = b;
}

// Set x and y coordinates of Point
void Point::setPoint(float a, float b)
{
   x = a;
   y = b;
}

// Output Point (with overloaded stream insertion operator)
ostream &operator<<(ostream &output, const Point &p)
{
   output << '[' << p.x << ", " << p.y << ']';

   return output;   // enables concatenated calls
}
```

Fig. 19.4 Definiciones de función miembro para la clase **Point** (parte 2 de 5).

```
// CIRCLE.H
// Definition of class Circle
#ifndef CIRCLE_H
#define CIRCLE_H

#include <iostream.h>
#include <iomanip.h>
#include "point.h"

class Circle : public Point {  // Circle inherits from Point
   friend ostream &operator<<(ostream &, const Circle &);
public:
   // default constructor
   Circle(float r = 0.0, float x = 0, float y = 0);

   void setRadius(float);      // set radius
   float getRadius() const;    // return radius
   float area() const;         // calculate area
protected:
   float radius;
};

#endif
```

Fig. 19.4 Definición de la clase **Circle** (parte 3 de 5).

```
// CIRCLE.CPP
// Member function definitions for class Circle
#include "circle.h"

// Constructor for Circle calls constructor for Point
// with a member initializer then initializes radius.
Circle::Circle(float r, float a, float b)
   : Point(a, b)       // call base-class constructor
{ radius = r; }

// Set radius of Circle
void Circle::setRadius(float r) { radius = r; }

// Get radius of Circle
float Circle::getRadius() const { return radius; }

// Calculate area of Circle
float Circle::area() const
{ return 3.14159 * radius * radius; }

// Output a Circle in the form:
// Center = [x, y]; Radius = #.##
ostream &operator<<(ostream &output, const Circle &c)
{
   output << "Center = [" << c.x << ", " << c.y
          << "]; Radius = " << setiosflags(ios::showpoint)
          << setprecision(2) << c.radius;

   return output;    // enables concatenated calls
}
```

Fig. 19.4 Definición de función miembro para la clase **Circle** (parte 4 de 5).


```
// FIG19_4.CPP
// Casting base-class pointers to derived-class pointers

#include <iostream.h>
#include <iomanip.h>
#include "point.h"
#include "circle.h"

main()
{
   Point *pointPtr, p(3.5, 5.3);
   Circle *circlePtr, c(2.7, 1.2, 8.9);

   cout << "Point p: " << p << "\nCircle c: " << c << endl;

   // Treat a Circle as a Circle (with some casting)
   pointPtr = &c;   // assign address of Circle to pointPtr
   circlePtr = (Circle *) pointPtr;  // cast base to derived
   cout << "\nArea of c (via circlePtr): "
        << circlePtr->area() << endl;

   // DANGEROUS: Treat a Point as a Circle
   pointPtr = &p;   // assign address of Point to pointPtr
   circlePtr = (Circle *) pointPtr;  // cast base to derived
   cout << "\nRadius of object circlePtr points to: "
        << circlePtr->getRadius() << endl;

   return 0;
}
```

```
Point p: [3.5, 5.3]
Circle c: Center = [1.2, 8.9]; Radius = 2.70

Area of c (via circlePtr): 22.90

Radius of object ciclePtr points to: 4.02e-38
```

Fig. 19.4 Conversión explícita de apuntadores de clase base a apuntadores de clase derivada (parte 5 de 5).

La clase **Circle** (figura 19.4, parte 3) hereda de la clase **Point** mediante herencia pública. Esto se especifica en la primera línea de la definición de clase

```
class Circle : public Point {     // Circle inherits from Point
```

Los dos puntos (**:**) en el encabezado de la definición de clase indican herencia. La palabra reservada **public** indica el tipo de herencia (vea la sección 19.7). Todos los miembros de la clase **Point** son heredados a la clase **Circle**. Esto significa que la interfaz pública a **Circle** incluye la funciones miembro públicas de **Point**, así como las funciones miembros **Circle** de nombre **area, setRadius**, y **getRadius**.

El constructor **Circle** (figura 19.4, parte 4), deberá invocar al constructor **Point** para inicializar la porción de clase base de un objeto **Circle**. Esto se lleva a cabo con la sintaxis de inicializador de miembros que se presentó en el capítulo 17, como sigue

```
Circle::Circle(float r, float a, float b)
   : Point(a, b)       // call base-class constructor
```

La segunda línea del encabezado de función constructor invoca al constructor **Point** por su nombre. Los valores **a** y **b** son pasados del constructor **Circle** al constructor **Point**, a fin de inicializar los miembros de clase base **x** e **y**. Note que la función de operador de inserción de flujo homónima es capaz de hacer referencia directa a las variables **x** y **y**, porque estos son miembros protegidos de la clase base **Point**. Note también que es necesario hacer referencia a **x** y a **y** mediante objetos como en **c.x** y en **c.y**. Esto es debido a que la función del operador de inserción de flujo homónima no es un miembro de la clase **Circle**, sino un amigo de la clase.

El programa manejador (figura 19.4, parte 5) crea **pointPtr** como un apuntador a un objeto **Point** y produce el objeto **p** de la clase **Point**, y a continuación crea **circlePtr**, como un apuntador a un objeto **Circle** y produce el objeto **c** de **Circle**. Los objetos **Point** y **Circle** son extraídos utilizando sus operadores de inserción de flujo homónimos, para demostrar que fueron inicializados en forma correcta. A continuación, el manejador demuestra la asignación de un apuntador de clase derivada (la dirección del objeto **c**) a un apuntador de clase base de nombre **pointPtr** y la conversión explícita (cast) de **pointPtr** de regreso a **Circle ***. El resultado de la operación de conversión explícita (cast) se asigna a **circlePtr**. El área del objeto **Circle** de nombre **c** es extraído vía **circlePtr**. Esto da como resultado un valor válido de superficie, porque los apuntadores están siempre señalando a un objeto **Circle**. Siempre resulta válido asignar un apuntador de clase derivada a un apuntador de clase base, porque un objeto de clase derivada *es un* objeto de clase base. El apuntador de clase base sólo "ve" la parte correspondiente de clase base del objeto de clase derivada. El compilador lleva a cabo una conversión implícita del apuntador de clase derivada a un apuntador de clase base. Un apuntador de clase base no puede ser asignado en forma directa a un apuntador de clase derivada, porque sería una asignación inherente peligrosa los apuntadores de clase derivada se espera estén apuntados a objetos de clase derivada. En este caso el compilador no llevará a cabo una conversión implícita. El uso de una conversión explícita (cast) informará al compilador que el programador sabe que este tipo de conversión de apuntador es peligrosa y que el programador asume la responsabilidad de utilizar adecuadamente dicho apuntador.

A continuación, el manejador demuestra la asignación del apuntador de clase base (la dirección del operador **p**) al apuntador de clase base de nombre **pointPtr** y la conversión explícita (cast) de **pointPtr**, de regreso a **Circle ***. El resultado de la operación de conversión explícita (cast) es asignado a **circlePtr**. El radio del objeto al cual **circlePtr** señala (objeto **p** de **Point**) es extraído vía **circlePtr**. Esto da como resultado un valor inválido, porque los apuntadores están siempre señalando a un objeto **Point**. Y un objeto **Point** no tiene un miembro **radius**. Por lo tanto, el programa extraerá cualquier valor que en ese momento exista en memoria en la posición que **circlePtr** espera que esté el miembro de datos **radius**.

19.5 Cómo utilizar funciones miembro

Cuando se forma una clase derivada a partir de una clase base, las funciones miembro de la clase derivada pudiera ser necesario que contengan ciertos miembros de clase base.

Observación de ingeniería de software 19.1

Una clase derivada no puede tener acceso directo a miembros privados de su clase base.

Este es un aspecto crucial de ingeniería de software en C++. Si una clase derivada podría tener acceso a los miembros privados de la clase base, esto violaría el encapsulado de la clase base. El ocultamiento de los miembros privados es gran ayuda en la prueba, depuración y modificación correcta de los sistemas. Si una clase derivada tuviera acceso a los miembros privados de su clase base, sería entonces posible para las clases derivadas de dicha clase derivada también tener acceso a dichos datos, y así en lo sucesivo. Esto propagaría el acceso de lo que se supone deben ser datos privados y a todo lo largo de la jerarquía de la clase se perderían los beneficios del encapsulado.

19.6 Cómo redefinir los miembros de clase base en una clase derivada

Una clase derivada puede redefinir una función miembro de clase base. Cuando en la clase derivada dicha función es mencionada por su nombre, de forma automática se selecciona la versión de la clase derivada. Puede utilizarse el operador de resolución de alcance, para tener acceso a la versión de clase base partiendo de la clase derivada.

Error común de programación 19.3

Cuando se redefine una función miembro de clase base en una clase derivada, es común hacer que la versión de clase derivada llame a la versión de clase base y haga algún trabajo adicional. Causa recursión infinita no utilizar el operador de resolución de alcance para hacer referencia a la función miembro de la clase base, porque la función miembro de la clase derivada, de hecho, está llamando a sí misma.

Práctica sana de programación 19.1

Al heredar capacidades que no son necesarias en la clase derivada, dichas capacidades se deben enmascarar mediante redefinición de las funciones.

Observación de ingeniería de software 19.2

Una redefinición de una función miembro de clase base en una clase derivada no necesariamente debe tener la misma signatura que la función miembro de clase base.

Veamos por ejemplo una clase **Employee** simplificada. Almacena el **firstName** y el **lastName** del empleado. Esta información es común a todos los empleados, incluyendo a aquellos de clases derivadas de la clase **Employee**. De la clase **Employee** derivan ahora las clases **Hourlyworker**, **PieceWorker**. **Boss**, y **CommissionWorker**. Al **Hourlyworker** se le paga por hora, "una vez y media" por las horas extra en exceso de 40 horas por semana. Al **PieceWorker** se le paga una tasa fija por cada elemento producido y para simplificar, suponga que esta persona fabrica un tipo de elemento, por lo que los miembros de datos privados son el número de elementos producidos y la tasa correspondiente por cada elemento. El **Boss** recibe un salario fijo por semana. El **CommissionWorker** recibe un reducido salario base semanal fijo, más un porcentaje fijo de las ventas brutas efectuadas por dicha persona durante la semana. A fin de simplificar, ahora estudiaremos sólo la clase **Employee** y la clase derivada **Hourlyworker**.

Nuestro siguiente ejemplo aparece en la figura 19.5, partes 1 hasta la 5. Las parte 1 y 2 muestran la definición de clase **Employee** y las definiciones de función miembro **Employee**. Las partes 3 y 4 muestran la definición de clase **Hourlyworker** y la definición de función miembro **Hourlyworker**. La parte 5 muestra un programa manejador para la jerarquía de herencia **Employee/Hourlyworker**, que sólo produce un objeto **Hourlyworker**, lo inicializa y llama a la función miembro **print** de **Hourlyworker**, con el fin de extraer los datos del objeto.

```
// EMPLOY.H
// Definition of class Employee
#ifndef EMPLOY_H
#define EMPLOY_H

class Employee {
public:
   Employee(const char*, const char*);  // constructor
   void print() const;  // output first and last name
   ~Employee();         // destructor
private:
   char *firstName;     // dynamically allocated string
   char *lastName;      // dynamically allocated string
};

#endif
```

Fig. 19.5 Definición de la clase **Employee** (parte 1 de 5).

```
// EMPLOY.CPP
// Member function definitions for class Employee
#include <string.h>
#include <iostream.h>
#include <assert.h>
#include "employ.h"

// Constructor dynamically allocates space for the
// first and last name and uses strcpy to copy
// the first and last names into the object.
Employee::Employee(const char *first, const char *last)
{
   firstName = new char[ strlen(first) + 1 ];
   assert(firstName != 0); // terminate if memory not allocated
   strcpy(firstName, first);

   lastName = new char[ strlen(last) + 1 ];
   assert(lastName != 0);  // terminate if memory not allocated
   strcpy(lastName, last);
}

// Output employee name
void Employee::print() const
{ cout << firstName << ' ' << lastName; }

// Destructor deallocates dynamically allocated memory
Employee::~Employee()
{
   delete [] firstName;    // reclaim dynamic memory
   delete [] lastName;     // reclaim dynamic memory
}
```

Fig. 19.5 Definiciones de funciones miembro para la clase **Employee** (parte 2 de 5).


```
// HOURLY.H
// Definition of class HourlyWorker
#ifndef HOURLY_H
#define HOURLY_H

#include "employ.h"

class HourlyWorker : public Employee {
public:
   HourlyWorker(const char*, const char*, float, float);
   float getPay() const;  // calculate and return salary
   void print() const;    // redefined base-class print
private:
   float wage;            // wage per hour
   float hours;           // hours worked for week
};

#endif
```

Fig. 19.5 Definición de la clase **HourlyWorker** (parte 3 de 5).

```
// HOURLY_B.CPP
// Member function definitions for class HourlyWorker
#include <iostream.h>
#include <iomanip.h>
#include "hourly.h"

// Constructor for class HourlyWorker
HourlyWorker::HourlyWorker(const char *first, const char *last,
                           float initHours, float initWage)
   : Employee(first, last)   // call base-class constructor
{
   hours = initHours;
   wage = initWage;
}

// Get the HourlyWorker's pay
float HourlyWorker::getPay() const { return wage * hours; }

// Print the HourlyWorker's name and pay
void HourlyWorker::print() const
{
   cout << "HourlyWorker::print()\n\n";

   Employee::print();   // call base-class print function

   cout << " is an hourly worker with pay of"
        << " $" << setiosflags(ios::showpoint)
        << setprecision(2) << getPay() << endl;
}
```

Fig. 19.5 Definiciones de funciones miembro para la clase **HourlyWorker** (parte 4 de 5).

```
// FIG19_5.CPP
// Redefining a base-class member function in a
// derived class.
#include <iostream.h>
#include "hourly.h"

main()
{
   HourlyWorker h("Bob", "Smith", 40.0, 7.50);
   h.print();
   return 0;
}
```

```
HourlyWorker::print()

Bob Smith is an hourly worker with pay of $300.00
```

Fig. 19.5 Cómo redefinir una función miembro de clase base en una clase derivada (parte 5 de 5).

La definición de clase **Employee** (figura 19.5 parte 1) está formada de dos miembros de datos privados **char *** que son **firstName** y **lastName** y de tres funciones miembro un constructor, un destructor y **print**. La función constructor (figura 19.5 part 2) recibe dos cadenas y asigna dinámicamente arreglos de caracteres para almacenar las cadenas. Note que es utilizado el macro **assert** (analizado en el capítulo 14, "Temas avanzados"), para determinar si se asignó memoria a **firstName** y **lastName**. Si no es así, el programa termina con mensaje de error indicando la condición probada, el número de línea en el cual aparece dicha condición, y el archivo en el cual la condición está almacenada. Dado que los datos de **Employee** son **private**, el único acceso a los datos es mediante la función miembro **print**, que sólo extrae el nombre y el apellido del empleado. La función destructor devuelve al sistema la memoria dinámicamente asignada.

La clase **HourlyWorker** (figura 19.5 parte 3) hereda de la clase **Employee** mediante herencia pública. Otra vez, esto queda especificado en la primera línea de la definición de clase, como sigue:

```
class HourlyWorker : public Employee
```

La interfaz pública de **HourlyWorker** incluye la función **Employee print** y las funciones miembro de **HourlyWorker**, de nombre **getPay** y **print**. Note que la clase **HourlyWorker** define su propia función **print**. Por lo tanto, la clase **HourlyWorker** tiene acceso a dos funciones **print**. La clase **HourlyWorker** también contiene los miembros de datos privados **wage** y **hours**, para el cálculo del salario semanal del empleado.

El constructor **HourlyWorker** (figura 19.5 parte 4) utiliza sintaxis de inicializador de miembro para pasar las cadenas **first** y **last** al constructor **Employee**, de forma tal, que puedan ser inicializados los miembros de la clase base, y a continuación inicializa a los miembros **wage** y **hours**. La función miembro **getPay** calcula el salario de **HourlyWorker**.

La función miembro de **HourlyWorker**, de nombre **print**, es un ejemplo de la función miembro de clase base, redefinida en una clase derivada. A menudo se redefinen las funciones miembro de clase base en una clase derivada, a fin de dar más funcionalidad. Las funciones redefinidas a veces llaman a la versión de clase base de la función para llevar a cabo parte de la

nueva tarea. En este ejemplo, la función de clase derivada **print** llama la función de clase base **print**, para extraer el nombre del empleado (la clase base **print** es la única función con acceso a los datos privados de la clase base). La función de clase derivada **print** también extrae la paga del empleado. Note cómo es llamada la versión de clase base de la función **print**

```
Employee::print();
```

Dado que tanto la función de clase base como la función de clase derivada tienen el mismo nombre, la función de clase base debe ser antecedida por su nombre de clase y por el operador de resolución de alcance. De lo contrario, la función correspondiente a la versión de la clase derivada será llamada, lo que resultaría una recursión infinita (la función **HourlyWorker print** se llamaría a sí misma).

19.7 Clases base públicas, protegidas y privadas

Al derivar una clase a partir de una clase base, la clase base puede ser heredada como **public**, **protected**, o **private**. La herencia protegida y la herencia privada son raras y deberían ser utilizadas sólo con extremo cuidado; en los ejemplos de este libro solo utilizamos herencia pública.

Al derivar una clase a partir de una clase base pública, los miembros públicos de la clase base se convierten en miembros públicos de la clase derivada, y los miembros protegidos de la clase base se convierten en miembros protegidos de la clase derivada. Los miembros privados de una clase base nunca son accesibles en forma directa desde una clase derivada.

Al derivar una clase a partir de una clase base protegida, los miembros públicos y protegidos de la clase base se convierten en miembros protegidos de la clase derivada.

Cuando se deriva una clase a partir de una clase base privada, los miembros públicos y protegidos de la clase base se convierten en miembros privados de la clase derivada.

19.8 Clases base directas y clases base indirectas

Una clase base puede ser una *clase base directa* de una clase derivada, o una clase base puede ser una *clase base indirecta* de una clase derivada. Un clase base directa de una clase derivada es listada en forma explícita en el encabezado de dicha clase derivada cuando se declara esta clase derivada. Una clase base indirecta no se lista en forma explícita en el encabezado de la clase derivada; más bien la clase base indirecta se hereda desde varios niveles arriba en la jerarquía de la clase.

19.9 Cómo utilizar constructores y destructores en clases derivadas

Dado que una clase derivada hereda los miembros de su clase base, cuando es producido un objeto de una clase derivada, el constructor de la clase base deberá de ser llamado, para inicializar los miembros de la clase base del objeto de la clase derivada. El constructor de la clase derivada puede llamar de forma implícita al constructor de la clase base o en el constructor de la clase derivada un *inicializador de clase base* (mismo que utiliza la sintaxis de inicializador de miembro que ya hemos visto) puede ser proveído, a fin de llamar de manera explícita al constructor de la clase base.

Los constructores de la clase base y los operadores de asignación de la clase base no son heredados por las clases derivadas. Sin embargo, los constructores y los operadores de asignación de las clases derivadas, pueden llamar a los constructores y a los operadores de asignación de la clase base.

Un constructor de clase derivada siempre llamará primero al constructor correspondiente de su clase base para inicializar los miembros de la clase base de la clase derivada. Si el constructor de la clase derivada ha sido omitido, el constructor por omisión de la clase derivada llamará al constructor de la clase base. Los destructores son llamados en orden inverso a las llamadas de constructor, por lo que un destructor de clase derivada será llamado antes de su destructor de clase base.

Observación de ingeniería de software 19.3

Cuando en una clase derivada se cree un objeto, primero se ejecutará el constructor de clase base, después se ejecutarán los constructores de los objetos miembro correspondientes a la clase derivada, y después se ejecutará el constructor de la clase derivada. Los destructores serán llamados en orden inverso en el cual fueron llamados sus correspondientes constructores.

El estándar de C++ especifica que el orden en que son construidos los objetos miembros es el orden en el cual han sido declarados dichos objetos dentro de la definición de clase. No afecta a la construcción el orden en el cual los inicializadores de miembro están listados. En la herencia, los constructores de clase base son llamados en el orden en el cual se haya especificado la herencia en la definición de clase derivada. El orden en el cual se han especificado los constructores de clase base en el constructor de clase derivada, no afecta a la construcción.

Error común de programación 19.4

Escribir un programa que dependa de que los objetos miembros sean construidos en un orden particular, puede producir errores sutiles.

El programa de la figura 19.6 demuestra el orden en que los constructores y destructores de clase base y de clase derivada son llamados. El programa está formado de cinco partes. Las partes 1 y 2 muestran una clase **Point** simple que contiene un constructor, un destructor y miembros de datos protegidos **x** y **y**. Tanto el constructor como el destructor ambos imprimen el objeto **Point** para el cual son invocados. Las partes 3 y 4 muestran una clase simple **Circle** derivada de **Point** mediante herencia pública y que contiene un constructor, un destructor y el miembro de datos privado **radius**. Tanto el constructor como el destructor imprimen el objeto **Circle** para el cual fueron invocados. El constructor **Circle** también invoca al constructor **Point** utilizando sintaxis de inicializador de miembro y pasa los valores **a** y **b**, de tal forma que los miembros de datos de clase base puedan ser inicializados.

En la parte 5 se muestra un programa manejador para esta jerarquía **Point/Circle**. El programa empieza produciendo un objeto **Point** en su propio alcance dentro de **main**. El objeto entra y sale de alcance de inmediato, por lo que tanto el constructor como el destructor **Point** son llamados. El programa a continuación produce el objeto **Circle** de nombre **circle1**. Esto invoca al constructor **Point** para llevar a cabo una salida con los valores pasados a partir del constructor **Circle**, y a continuación ejecuta la salida especificada en el constructor **Circle**. A continuación se produce el objeto **Circle** de nombre **circle2**. Otra vez, tanto el constructor **Point** como el **Circle** son llamados. Note que se ejecuta el cuerpo del constructor **Point** antes del cuerpo del constructor **Circle**. Se alcanza el fin de **main**, por lo que deberán ser llamados los destructores tanto para **circle1** como para **circle2**. Los destructores son llamados en orden inverso a sus constructores correspondientes. Por lo tanto el destructor **Circle** y el destructor **Point** son llamados en este orden para el objeto **circle2**, y a continuación el destructor **Circle** y el destructor **Point** son llamados en ese orden para el objeto **circle1**.


```
// POINT2.H
// Definition of class Point
#ifndef POINT2_H
#define POINT2_H

class Point {
public:
   Point(float = 0.0, float = 0.0);  // default constructor
   ~Point();     // destructor
protected:       // accessible by derived classes
   float x, y;   // x and y coordinates of Point
};

#endif
```

Fig. 19.6 Definición de clase **Point** (parte 1 de 5).

```
// POINT2.CPP
// Member function definitions for class Point
#include <iostream.h>
#include "point2.h"

// Constructor for class Point
Point::Point(float a, float b)
{
   x = a;
   y = b;

   cout << "Point  constructor: "
        << '[' << x << ", " << y << ']' << endl;
}

// Destructor for class Point
Point::~Point()
{
   cout << "Point  destructor:  "
        << '[' << x << ", " << y << ']' << endl;
}
```


Fig. 19.6 Definiciones de funciones miembro para la clase **Point** (parte 2 de 5).

19.10 Conversión implícita de objeto de clase derivada a objeto de clase base

A pesar del hecho que un objeto de clase derivada también "es un" objeto de clase base, el tipo de la clase derivada y el tipo de la clase base son distintos. Los objetos de clase derivada pueden ser tratados como objetos de clase base. Tiene sentido este tipo de asignación, porque la clase derivada tiene miembros que corresponden a cada uno de los miembros de la clase base —recuerde que por lo regular la clase derivada tiene más miembros que la clase base. En la otra dirección la asignación no es permitida, porque asignar un objeto de clase base a un objeto de clase derivada dejaría sin definición a los miembros adicionales de la clase derivada. Aunque tal asignación no

```
// CIRCLE2.H
// Definition of class Circle
#ifndef CIRCLE2_H
#define CIRCLE2_H

#include "point2.h"
#include <iomanip.h>

class Circle : public Point {
public:
   // default constructor
   Circle(float r = 0.0, float x = 0, float y = 0);

   ~Circle();       // destructor
private:
   float radius;    // radius of Circle
};

#endif
```

Fig. 19.6 Definición de clase `Circle` (parte 3 de 5).

```
// CIRCLE2.CPP
// Member function definitions for class Circle
#include "circle2.h"

// Constructor for Circle calls constructor for Point
Circle::Circle(float r, float a, float b)
   : Point(a, b)    // call base-class constructor
{
   radius = r;

   cout << "Circle constructor: radius is "
        << radius << " [" << a << ", " << b << ']' << endl;
}

// Destructor for class Circle
Circle::~Circle()
{
   cout << "Circle destructor:  radius is "
        << radius << " [" << x << ", " << y << ']' << endl;
}
```

Fig. 19.6 Definiciones de funciones miembro para la clase `Circle` (parte 4 de 5).

es permitida en forma "natural", se puede legitimar proporcionando un operador de asignación y/o un constructor de conversión correctamente homónimos.

Error común de programación 19.5

Asignar un objeto de clase derivada a un objeto de una clase base correspondiente, y a continuación intentar hacer referencia a miembros de sólo la clase derivada en el objeto nuevo de clase.


```
// FIG19_6.CPP
// Demonstrate when base-class and derived-class
// constructors and destructors are called.
#include <iostream.h>
#include "point2.h"
#include "circle2.h"

main()
{
   // Show constructor and destructor calls for Point
   {
      Point p(1.1, 2.2);
   }

   cout << endl;
   Circle circle1(4.5, 7.2, 2.9);
   cout << endl;
   Circle circle2(10, 5, 5);
   cout << endl;
   return 0;
}
```

```
Point  constructor: [1.1, 2.2]
Point  destructor:  [1.1, 2.2]

Point  constructor: [7.2, 2.9]
Circle constructor: radius is 4.5 [7.2, 2.9]

Point  constructor: [5, 5]
Circle constructor: radius is 10 [5, 5]

Circle destructor: radius is 10 [5, 5]
Point  destructor: [5, 5]
Circle destructor: radius is 4.5 [7.2, 2.9]
Point  destructor: [7.2, 2.9]
```

Fig. 19.6 Orden en el cual son llamados los constructores y destructores de clase base y clase derivada (parte 5 de 5).

Un apuntador a un objeto de clase derivada puede ser convertido en forma implícita en un apuntador a un objeto de una clase base, porque un objeto de una clase derivada es un objeto de una clase base.

Existen cuatro formas posibles para mezclar y hacer coincidir apuntadores de clase base y apuntadores de clase derivada con objetos de clase base y objetos de clase derivada:

1. Es simple hacer referencia a un objeto de clase base con un apuntador de clase base.
2. También es simple hacer referencia a un objeto de clase derivada con un apuntador de clase derivada.
3. Es seguro hacer referencia a un objeto de clase derivada con un apuntador de clase base, porque el objeto de clase derivada es también un objeto de su base clase. El programa

sólo puede referirse a miembros de clase base. Si mediante el apuntador de clase base el programa hace referencia a miembros sólo de clase derivada, el compilador informará de un error de sintaxis.

4. Es un error de sintaxis hacer referencia a un objeto de clase base mediante un apuntador de clase derivada. El apuntador de clase derivada deberá ser convertido primero explícitamente en un apuntador de clase base.

Error común de programación 19.6

Puede causar error convertir explícitamente (cast) un apuntador de clase base a un apuntador de clase derivada, si a continuación dicho apuntador es utilizado para hacer referencia a un objeto de clase base que no tenga los miembros deseados de la clase derivada.

Independiente de lo conveniente que resulte tratar los objetos de la clase derivada como si fueran objetos de clase base, y hacerlo manipulando todos estos objetos mediante apuntadores de clase base, existe un problema. En un sistema de nóminas, por ejemplo, nos gustaría ser capaces de recorrer una lista enlazada de empleados y calcular la paga semanal para cada persona. Pero utilizar apuntadores de clase base sólo le permite al programa llamar la rutina de cálculo de nóminas de clase base (si en verdad tal rutina existe en la clase base). Necesitamos una forma para invocar para cada objeto la rutina de cálculo de nóminas adecuada, sea éste un objeto de clase base o un objeto de clase derivada, y hacerlo con facilidad utilizando el apuntador de clase base. La solución es utilizar funciones virtuales y polimorfismo, como se analiza en el capítulo 20.

19.11 Ingeniería de software con herencia

Podemos utilizar la herencia para *personalizar* el software existente. Heredamos los atributos y comportamientos de una clase existente, y a continuación añadimos o retiramos atributos y comportamientos para personalizar la clase a fin de que cumpla con nuestras necesidades. Esto en C++ se efectúa sin que la clase derivada tenga acceso al código fuente de la clase base, pero la clase derivada sí debe tener la capacidad de enlazarse con el código objeto de la clase base. Esta poderosa capacidad resulta atractiva para los fabricantes independientes de software (ISV). Los ISV pueden desarrollar clases propietarias para venta o para licencia, y poner estas clases a disposición de los usuarios en formato de código objeto. Los usuarios pueden entonces con rápidez derivar de estas clases de biblioteca clases nuevas, sin necesidad de tener acceso al código fuente propiedad de los ISV. Todo lo que los ISV tienen que suministrar junto con el código objeto son los archivos de cabecera.

Puede ser difícil para los estudiantes darse cuenta de los problemas a los que los diseñadores y los instaladores se enfrentan tratándose de proyectos de software a gran escala. Las personas con experiencia en estos proyectos invariablemente declararán que, para mejorar el proceso de desarrollo de software, resulta clave fomentar la reutilización del software. La programación orientada a objetos en general, y C++ en particular, alienta y fomenta la reutilización del software.

Es la disponibilidad de bibliotecas de clases sustanciales y útiles la que proporciona el máximo de beneficios en la reutilización del software mediante la herencia. Conforme crezca el interés en C++, aumentará el interés en las bibliotecas de clases. Igual que con la llegada de la computadora personal, el software listo para usarse, producido por fabricantes independientes de software, se convirtió en una industria con un crecimiento explosivo, también, por lo tanto, será la creación y la venta de bibliotecas de clases. Partiendo de estas bibliotecas los diseñadores de aplicaciones construirán sus aplicaciones, y los diseñadores de bibliotecas se verán premiados al empacarse dichas bibliotecas junto con las aplicaciones. Las bibliotecas actualmente incluidas con los

compiladores de C++ tienden a ser más bien de uso general y de alcance limitado. Lo que estamos viendo venir es un compromiso masivo a nivel mundial para el desarrollo de bibliotecas de clases y para una gran variedad de campos de aplicaciones.

Observación de ingeniería de software 19.4

Crear una clase derivada no afecta el código fuente o el código objeto de su clase base; mediante la herencia se conserva la integridad de la clase base.

Una clase base define un estado común. Todas las clases derivadas de una clase base heredan las capacidades de dicha clase base. En el proceso de diseño orientado a objetos, el diseñador busca el estado común y lo extrae para formar clases base deseables. Las clases derivadas después son personalizadas, más allá de las capacidades heredadas de las clases base.

De igual forma que el diseñador de sistemas no orientados a objetos busca evitar una proliferación innecesaria de funciones, el diseñador de sistemas orientados a objetos debería evitar una proliferación innecesaria de clases. Dicha proliferación de clases crea problemas de administración y puede dificultar la reutilización del software, debido a que para el reutilizador potencial de una clase resulta más difícil localizar dicha clase en una gran colección. La contrapartida es crear menos clases, capaz cada una de ellas de proporcionar funcionalidad adicional. Para ciertos reutilizadores dichas clases podrían resultar demasiado ricas; esta funcionalidad excesiva pudiera ser enmascarada, "aminorando" entonces dichas clases a fin de que cumplan con sus necesidades.

Sugerencia de rendimiento 19.1

Si las clases producidas mediante la herencia son más grandes de lo requerido, pudiera dar como resultado un desperdicio de memoria y de recursos de proceso.

Note que puede resultar confuso leer un conjunto de declaraciones de clase derivada, porque no se muestran los miembros heredados. Pero aún así en las clases derivadas los miembros heredados están presentes.

Observación de ingeniería de software 19.5

En un sistema orientado a objetos, las clases a menudo están muy relacionadas. "Disgregue y elimine" atributos y comportamientos comunes y colóquelos en una clase base. A continuación utilice la herencia para formar clases derivadas.

Observación de ingeniería de software 19.6

Una clase derivada contiene los atributos y comportamientos de su clase base. Una clase derivada también puede contener atributos y comportamientos adicionales. Con la herencia, la clase base puede ser compilada independientemente de la clase derivada. Para tener la capacidad de combinar estos atributos y comportamientos adicionales con la clase base para formar la clase derivada basta compilar los atributos y comportamientos incrementados de la clase derivada.

Observación de ingeniería de software 19.7

Las modificaciones a una clase base no requieren que sea modificada la clase derivada, siempre y cuando la interfaz pública de la clase base se mantenga sin modificación. Pudiera sin embargo ser necesario recompilar las clases derivadas.

19.12 Composición en comparación con herencia

Hemos discutido relaciones *es un* basadas en la herencia. También hemos analizado relaciones *tiene un* (y hemos visto ejemplos en capítulos anteriores) en las cuales un objeto puede tener otros

objetos —como miembros mediante la *composición* de clases existentes estas relaciones crean nuevas clases. Por ejemplo, dadas las clases **Employee**, **BirthDate**, y **TelephoneNumber**, no es correcto decir que un **Employee** *es un* **BirthDate** o que un **Employee** *es un* **TelephoneNumber**. Pero es correcto decir que un **Employee** *tiene un* **BirthDate** y un **Employee** *tiene un* **TelephoneNumber**.

Recuerde que el orden en el cual los objetos miembros se construyen se define como el orden en el cual ha sido declarado dicho objeto. Los constructores de clase base serán llamados en el orden en el cual ha sido especificada la herencia en la clase derivada.

Observación de ingeniería de software 19.8

Las modificaciones a una clase miembro no requieren que su clase compuesta que la encierra sea modificada, siempre y cuando se mantenga sin modificación la interfaz pública de la clase miembro. Note que pudiera ser necesario recompilar la clase compuesta.

19.13 Relaciones "utiliza un" y "conoce un"

La herencia y la composición, al crear nuevas clases que tengan mucho en común con clases existentes, cada una de ellas fomentan la reutilización del software. Existen otras formas de utilizar los servicios de las clases. Aunque un objeto persona no es un automóvil, y un objeto persona no contiene un automóvil, un objeto persona *utiliza un automóvil*. Una función utiliza un objeto, sólo al emitir una llamada de función a una función miembro de dicho objeto.

Un objeto puede estar *consciente de* otro objeto. Las redes de conocimiento a menudo tienen estas relaciones. Para estar consciente de un objeto, otro objeto puede contener un apuntador o una referencia a dicho objeto. En este caso, se dice que un objeto tiene una relación de conciencia con el otro objeto.

19.14 Estudio de caso: Point, Circle, Cylinder

Veamos ahora el ejercicio recapitulativo correspondiente a este capítulo. Veremos una jerarquía de punto, círculo y cilindro. Primero desarrollaremos y utilizaremos la clase **Point** (figura 19.7). Después presentaremos un ejemplo en el cual de la clase **Point** derivaremos la clase **Circle** (figura 19.8). Por último, presentaremos un ejemplo en el cual de la clase **Circle** derivaremos la clase **Cylinder** (figura 19.9).

La figura 19.7 muestra la clase **Point**. La Parte 1 de la figura 19.7 muestra la definición de la clase **Point**. Note que los miembros de datos de **Point** están protegidos. Entonces, cuando de la clase **Point** derivemos la clase **Circle**, las funciones miembro de la clase **Circle** podrán hacer referencia directa a las coordenadas **x** y **y**, en vez de tener que utilizar funciones de acceso. Esto pudiera resultar en un mejor rendimiento.

En la figura 19.7 parte 2, se muestran las definiciones de funciones miembro correspondientes a la clase **Point**. En la figura 19.7 parte 3, se muestra el programa manejador para la clase **Point**. Note que **main** deberá utilizar las funciones de acceso **getX** y **getY** para leer los valores de los miembros de dato protegidos **x** e **y**; recuerde que los miembros de dato protegidos son accesibles sólo a miembros y amigos de su clase y a miembros y amigos de sus clases derivadas.

Nuestro siguiente ejemplo se muestra en la figura 19.8 partes 1 a 3. La definición de clase **Point** y las definiciones de función miembro correspondientes de la figura 19.7 son utilizadas aquí otra vez. Las partes 1 a 3 muestran la definición de clase **Circle**, las definiciones de función miembro **Circle**, y el programa manejador, respectivamente. Note que la clase **Circle** hereda de la clase **Point** mediante herencia pública. Esto significa que la interfaz pública de **Circle**


```
// POINT2.H
// Definition of class Point
#ifndef POINT2_H
#define POINT2_H

class Point {
   friend ostream &operator<<(ostream &, const Point &);
public:
   Point(float = 0, float = 0);       // default constructor
   void setPoint(float, float);       // set coordinates
   float getX() const { return x; }   // get x coordinate
   float getY() const { return y; }   // get y coordinate
protected:         // accessible to derived classes
   float x, y;     // coordinates of the point
};

#endif
```

Fig. 19.7 Definición de clase **Point** (parte 1 de 3).

```
// POINT2.CPP
// Member functions for class Point
#include <iostream.h>
#include "point2.h"

// Constructor for class Point
Point::Point(float a, float b)
{
   x = a;
   y = b;
}

// Set the x and y coordinates
void Point::setPoint(float a, float b)
{
   x = a;
   y = b;
}

// Output the Point
ostream &operator<<(ostream &output, const Point &p)
{
   output << '[' << p.x << ", " << p.y << ']';

   return output;             // enables concatenation
}
```

Fig. 19.7 Funciones miembro para la clase **Point** (parte 2 de 3).

incluye las funciones miembro **Point**, así como las funciones miembro **Circle** de nombre **setRadius**, **getRadius**, y **area**. Note que la función de operador de inserción de flujo homónimo **Circle** puede hacer referencia directa a las variables **x** y **y**, porque éstas son

```
// FIG19_7.CPP
// Driver for class Point
#include <iostream.h>
#include "point2.h"

main()
{
   Point p(7.2, 11.5);    // instantiate Point object p

   // protected data of Point inaccessible to main
   cout << "X coordinate is " << p.getX()
        << "\nY coordinate is " << p.getY();

   p.setPoint(10, 10);
   cout << "\n\nThe new location of p is " << p << endl;

   return 0;
}
```

```
X coordinate is 7.2
Y coordinate is 11.5

The new location of p is [10, 10]
```

Fig. 19.7 Manejador para la clase **Point** (parte 3 de 3).

miembros protegidos de la clase base **Point**. Note también que es necesario hacer referencia a **x** y a **y** mediante objetos, como en **c.x** y en **c.y**. Esto es debido a que la función de operador de inserción de flujo homónima no es una función miembro de la clase **Circle**, pero un amigo de la clase. El programa manejador produce un objeto de la clase **Circle**, a continuación utiliza funciones *get* para obtener la información sobre el objeto **Circle**. Otra vez, **main** no es una función miembro ni amigo de la clase **Circle**, por lo cual no puede hacer referencia directa a los datos protegidos de la clase **Circle**. El programa manejador utiliza entonces la función *set* de nombre **setRadius** y **setPoint** para redefinir el radio y las coordenadas del centro del círculo. Por último, el manejador hace algo en especial interesante. Inicializa la variable de referencia **pRef** del tipo "referencia a un objeto **Point**" (**Point &**), al objeto **c** de **Circle**. El manejador a continuación imprime **pRef**, el cual, a pesar del hecho que es inicializado con un objeto **Circle**, "piensa" que es un objeto **Point**, por lo que el objeto **Circle** de hecho se imprime como un objeto **Point**.

Nuestro último ejemplo se muestra en la figura 19.9, partes 1 a 3. Las definiciones de la clase **Point** y de la clase **Circle**, y las definiciones de funciones miembro correspondientes a las figuras 19.7 y 19.8 son vueltas a usar aquí. Las partes 1 a 3 muestran la definición de clase **Cylinder**, las definiciones de función miembro **Cylinder**, y un programa manejador, respectivamente. Note que la clase **Cylinder** hereda de la clase **Circle** mediante herencia pública. Esto significa que la interfaz pública a **Cylinder** incluye las funciones miembro **Circle**, así como las funciones miembro **Cylinder**, de nombre **setHeight**, **getHeigtht**, **area** (redefinidos a partir de **Circle**) y **volume**. Note que la función de operador de inserción de flujo homónima **Cylinder** es capaz de hacer referencia directa a las variables **x**, **y**, y **radius**, porque


```
// CIRCLE2.H
// Definition of class Circle
#ifndef CIRCLE2_H
#define CIRCLE2_H

#include "point2.h"

class Circle : public Point {
   friend ostream &operator<<(ostream &, const Circle &);
public:
   // default constructor
   Circle(float r = 0.0, float x = 0, float y = 0);
   void setRadius(float);       // set radius
   float getRadius() const;     // return radius
   float area() const;          // calculate area
protected:          // accessible to derived classes
   float radius;    // radius of the Circle
};

#endif
```

Fig. 19.8 Definición de la clase **Circle** (parte 1 de 3).

```
// CIRCLE2.CPP
// Member function definitions for class Circle
#include <iostream.h>
#include <iomanip.h>
#include "circle2.h"

// Constructor for Circle calls constructor for Point
// with a member initializer and initializes radius
Circle::Circle(float r, float a, float b)
   : Point(a, b)        // call base-class constructor
{ radius = r; }

// Set radius
void Circle::setRadius(float r) { radius = r; }

// Get radius
float Circle::getRadius() const { return radius; }

// Calculate area of Circle
float Circle::area() const
   { return 3.14159 * radius * radius; }

// Output a circle in the form:
// Center = [x, y]; Radius = #.##
ostream &operator<<(ostream &output, const Circle &c)
{
   output << "Center = [" << c.x << ", " << c.y
          << "]; Radius = " << setiosflags(ios::showpoint)
          << setprecision(2) << c.radius;

   return output;    //  enables concatenated calls
}
```

Fig. 19.8 Definiciones de función miembro para la clase **Circle** (parte 2 de 3).

```
// FIG19_8.CPP
// Driver for class Circle
#include <iostream.h>
#include "point2.h"
#include "circle2.h"

main()
{
   Circle c(2.5, 3.7, 4.3);

   cout << "X coordinate is " << c.getX()
        << "\nY coordinate is " << c.getY()
        << "\nRadius is " << c.getRadius();

   c.setRadius(4.25);
   c.setPoint(2, 2);
   cout << "\n\nThe new location and radius of c are\n"
        << c << "\nArea " << c.area() << endl;

   Point &pRef = c;
   cout << "\nCircle printed as a Point is: "
        << pRef << endl;

   return 0;
}
```

```
X coordinate is 3.7
Y coordinate is 4.3
Radius is 2.5

The new location and radius of c are
Center = [2, 2]; Radius = 4.25
Area: 56.74

Circle printed as a Point is: [2.00, 2.00]
```

Fig. 19.8 Manejador para la clase **Circle** (parte 3 de 3).

estos son miembros protegidos de la clase base **Circle** (**x** y **y** fueron heredados por **Circle** a partir de **Point**). Note también que es necesario hacer referencia a **x**, **y** y **radius** mediante objetos, como en **c.x**, **c.y** y **c.radius**. Esto es debido a que la función de operador de inserción de flujo homónima no es una función miembro de la clase **Cylinder**, pero es un amigo de la clase. El programa manejador produce un objeto de la clase **Cylinder** y a continuación utiliza las funciones *get* para obtener la información respectiva del objeto **Cylinder**. Otra vez, **main** no es ni una función miembro ni un amigo de la clase **Cylinder**, por lo que no puede hacer referencia directa a los datos protegidos de la clase **Cylinder**. El programa manejador a continuación utiliza funciones *set* de nombre **setHeigt**, **setRadius** y **setPoint** para redefinir la altura, el radio y las coordenadas del cilindro. Por último, el manejador hace algo en especial interesante. Inicializa la variable de referencia **pRef** del tipo "referencia al objeto **Point**" (**Point &**) al objeto **cyl** de **Cylinder**. A continuación imprime **pRef** el cual, a pesar

del hecho que ha sido inicializado con un objeto **Cylinder**, "piensa" que se trata de un objeto **Point**, por lo tanto, el objeto **Cylinder** de hecho imprime un objeto **Point**. El manejador a continuación inicializa la variable de referencia **cRef** del tipo "referencia a un objeto **Circle**" (**Circle &**), al objeto **cyl** de **Cylinder**. A continuación imprime **cRef** el cual, a pesar del hecho de que ha sido inicializado con un objeto **Cylinder**, "piensa" que es un objeto **Circle**, por lo que el objeto **Cylinder** de hecho se imprime como un objeto **Circle**. También es extraída el área del **Circle**.

Este ejemplo demuestra muy bien la herencia pública y la definición y referencia de miembros de datos protegidos. A estas alturas el lector deberá sentirse familiarizado con los fundamentos de la herencia. En el siguiente capítulo mostraremos cómo programar con jerarquías de herencia de forma general mediante el polimorfismo. La abstracción de datos, la herencia y el polimorfismo forman el núcleo de la programación orientada a objetos.

19.15 Herencia múltiple

Hasta ahora en el capítulo hemos analizado la herencia simple, en la cual cada clase es derivada de una clase base. Una clase puede ser derivada a partir de más de una clase base; esta derivación se conoce como *herencia múltiple*. La herencia múltiple significa que una clase derivada hereda los miembros de varias clases base. Esta poderosa capacidad fomenta formas interesantes de reutilización de software, pero puede causar una variedad de problemas de ambigüedad.

Práctica sana de programación 19.2

La herencia múltiple, si se utiliza de forma adecuada, es una capacidad poderosa. La herencia múltiple deberá ser utilizada cuando exista una relación "es una" entre un nuevo tipo y dos o más tipos existentes (es decir, tipo A "es un" tipo B y "es un" tipo C).

```
// CYLINDR2.H
// Definition of class Cylinder
#ifndef CYLINDR2_H
#define CYLINDR2_H
#include "circle2.h"

class Cylinder : public Circle {
   friend ostream& operator<<(ostream&, const Cylinder&);
public:
   // default constructor
   Cylinder(float h = 0.0, float r = 0.0,
            float x = 0.0, float y = 0.0);
   void setHeight(float);   // set height
   float getHeight() const; // return height
   float area() const;      // calculate and return area
   float volume() const;    // calculate and return volume
protected:
   float height;            // height of the Cylinder
};

#endif
```

Fig. 19.9 Definición de clase **Cylinder** (parte 1 de 3).

```
// CYLINDR2.CPP
// Member and friend function definitions for class Cylinder
#include <iostream.h>
#include <iomanip.h>
#include "cylindr2.h"

// Cylinder constructor calls Circle constructor
Cylinder::Cylinder(float h, float r, float x, float y)
   : Circle(r, x, y)   // call base-class constructor
{ height = h; }

// Set height of Cylinder
void Cylinder::setHeight(float h) { height = h; }

// Get height of Cylinder
float Cylinder::getHeight() const { return height; }

// Calculate area of Cylinder (i.e., surface area)
float Cylinder::area() const
{
   return 2 * Circle::area() +
          2 * 3.14159 * radius * height;
}

// Calculate volume of Cylinder
float Cylinder::volume() const
   { return 3.14159 * radius * radius * height; }

// Output Cylinder dimensions
ostream& operator<<(ostream &output, const Cylinder& c)
{
   output << "Center = [" << c.x << ", " << c.y
          << "]; Radius = " << setiosflags(ios::showpoint)
          << setprecision(2) << c.radius
          << "; Height = " << c.height;

   return output;   // enables concatenated calls
}
```

Fig. 19.9 Definiciones de función miembro y de función amigo para la clase **Cylinder** (parte 2 de 3).

Considere el ejemplo de herencia múltiple de la figura 19.10. La clase **Base1** contiene un miembro de datos protegido **int value**. **Base1** contiene un constructor que define **value** y la función miembro pública **getData**, que lee **value**.

La clase **Base2** es similar a la clase **Base1**, excepto que sus datos protegidos son **char letter**. **Base2** también tiene una función miembro pública **getData**, pero su función lee el valor de **char letter**.

La clase **Derived** es heredada tanto de la clase **Base1** como de la clase **Base2** mediante herencia múltiple. **Derived** tiene el miembro de datos privados **float real**, y tiene la función miembro pública **getReal**, que lee el valor de **float real**.


```
// FIG19_9.CPP
// Driver for class Cylinder
#include <iostream.h>
#include <iomanip.h>
#include "point2.h"
#include "circle2.h"
#include "cylindr2.h"

main()
{
   // create Cylinder object
   Cylinder cyl(5.7, 2.5, 1.2, 2.3);

   // use get functions to display the Cylinder
   cout << "X coordinate is " << cyl.getX()
        << "\nYcoordinate is " << cyl.getY()
        << "\nRadius is " << cyl.getRadius()
        << "\nHeight is " << cyl.getHeight();

   // use set functions to change the Cylinder's attributes
   cyl.setHeight(10);
   cyl.setRadius(4.25);
   cyl.setPoint(2, 2);
   cout << "\n\nThe new location, radius, "
        << "and height of cyl are:\n" << cyl << endl << "Area: "
        << cyl.area() << "   Volume: " << cyl.volume();

   // display the Cylinder as a Point
   Point &pRef = cyl;   // pRef "thinks" it is a Point
   cout << "\n\nCylinder printed as a Point is: " << pRef;

   // display the Cylinder as a Circle
   Circle &cRef = cyl;  // cRef "thinks" it is a Circle
   cout << "\n\nCylinder printed as a Circle is:\n" << cRef
        << "\nArea: " << cRef.area() << endl;
   return 0;
}
```

```
X coordinate is 1.2
Y coordinate is 2.3
Radius is 2.5
Height is 5.7

The new location, radius, and height of cyl are:
Center = [2, 2]; Radius = 4.25; Height = 10.00
Area: 380.53   Volume: 567.45

Cylinder printed as a Point is: [2.00, 2.00]

Cylinder printed as a Circle is:
Center = [2.00, 2.00]; Radius = 4.25
Area: 56.74
```

Fig. 19.9 Manejador para la clase **Cylinder** (parte 3 de 3).

```
// BASE1.H
// Definition of class Base1
#ifndef BASE1_H
#define BASE1_H

class Base1 {
public:
   Base1(int x) { value = x; }
   int getData() const { return value; }
protected:      // accessible to derived classes
   int value;   // inherited by derived class
};

#endif
```

Fig. 19.10 Definición de la clase **Base1** (parte 1 de 6).

```
// BASE2.H
// Definition of class Base2
#ifndef BASE2_H
#define BASE2_H

class Base2 {
public:
   Base2(char c) { letter = c; }
   char getData() const { return letter; }
protected:        // accessible to derived classes
   char letter;   // inherited by derived class
};

#endif
```

Fig. 19.10 Definición de clase **Base2** (parte 2 de 6).

Note qué directo es indicar herencia múltiple haciendo seguir los dos puntos (**:**) después de **class Derived** con una lista separada por comas de clases base públicas. Note también que para cada una de sus clases base, el constructor **Derived** llama a los constructores de clase base **Base1** y **Base2**, mediante la sintaxis de inicializador de miembro.

El operador de inserción de flujo homónimo para **Derived** utiliza la notación punto sobre el objeto derivado **d** para imprimir **value**, **letter** y **real**. Esta función de operador es un amigo de **Derived**, por lo que **operator<<** puede tener acceso directo al miembro de datos privado **real** de **Derived**. También, dado que este operador es un amigo de una clase derivada, puede tener acceso a los miembros protegidos **value** y **letter** de **Base1** y de **Base2**, respectivamente.

Ahora examinemos el programa manejador en **main**. Creamos el objeto **b1** de clase **Base1** y lo inicializamos al valor **int 10**. Creamos el objeto **b2** de clase **Base2** y lo inicializamos al valor **char 'Z'**. Entonces, creamos el objeto **d** de la clase **Derived** y lo inicializamos para contener el valor **int 7**, el valor **char 'A'** y el valor **float 3.5**.

El contenido de cada uno de estos objetos de clase base es impreso utilizando ligaduras estáticas. Aún cuando existen dos funciones **getData**, no hay ambigüedad en las llamadas, porque hacen referencia directa a la versión **b1** del objeto de **getData** y a la versión **b2** de **getData**.


```
// DERIVED.H
// Definition of class Derived which inherits
// multiple base classes (Base1 and Base2).
#ifndef DERIVED_H
#define DERIVED_H

#include <iostream.h>
#include "base1.h"
#include "base2.h"

// multiple inheritance
class Derived : public Base1, public Base2 {
   friend ostream &operator<<(ostream &, const Derived &);
public:
   Derived(int, char, float);
   float getReal() const;
private:
   float real;    // derived class's private data
};

#endif
```

Fig. 19.10 Definición de clase **Derived** (parte 3 de 6).

```
// DERIVED.CPP
// Member function definitions for class Derived
#include "derived.h"

// Constructor for Derived calls constructors for
// class Base1 and class Base2.
Derived::Derived(int i, char c, float f)
   : Base1(i), Base2(c)  // call both base-class constructors
{ real = f; }

// Return the value of real
float Derived::getReal() const { return real; }

// Display all the data members of Derived
ostream &operator<<(ostream &output, const Derived &d)
{
   output << "    Integer: " << d.value
          << "\n  Character: " << d.letter
          << "\nReal number: " << d.real;

   return output;   // enables concatenated calls
}
```

Fig. 19.10 Definición de funciones miembro para la clase **Derived** (parte 4 de 6).

A continuación imprimimos el contenido del objeto **d** de **Derived** mediante ligadura estática. Pero aquí sí existe un problema de ambigüedad, porque este objeto contiene dos funciones **getData**, una heredada de **Base1** y la otra de **Base2**. Este problema es fácil de resolver mediante

```
// FIG19_10.CPP
// Driver for multiple inheritance example
#include <iostream.h>
#include "base1.h"
#include "base2.h"
#include "derived.h"

main()
{
   Base1 b1(10), *base1Ptr;   // create base-class object
   Base2 b2('Z'), *base2Ptr;  // create other base-class object
   Derived d(7, 'A', 3.5);    // create derived-class object

   // print data members of base class objects
   cout << "Object b1 contains integer "
        << b1.getData()
        << "\nObject b2 contains character "
        << b2.getData()
        << "\nObject d contains:\n" << d;

   // print data members of derived class object
   // scope resolution operator resolves getData ambiguity
   cout << "\n\nData members of Derived can be"
        << " accessed individually:\n"
        << "    Integer: " << d.Base1::getData()
        << "\n  Character: " << d.Base2::getData()
        << "\nReal number: " << d.getReal() << "\n\n";

   cout << "Derived can be treated as an "
        << "object of either base class:\n";

   // treat Derived as a Base1 object
   base1Ptr = &d;
   cout << "base1Ptr->getData() yields "
        << base1Ptr->getData();

   // treat Derived as a Base2 object
   base2Ptr = &d;
   cout << "\nbase2Ptr->getData() yields "
        << base2Ptr->getData() << endl;

   return 0;
}
```

Fig. 19.10 Manejador para el ejemplo de herencia múltiple (parte 5 de 6).

el operador de resolución de alcance binario, como **d.Base1::getData()** para imprimir el **int** en **value**, y **d.Base2::getData()** para imprimir el **char** en **letter**. En **real** el valor **float** es impreso sin ambigüedades mediante la llamada **d.getReal()**.

A continuación demostramos que las relaciones *es una* de herencia simple también son aplicables a la herencia múltiple. Asignamos la dirección del objeto derivado **d** al apuntador de clase base **base1Ptr**, e imprimimos **int value** invocando la función miembro **getData** de **Base1** desde **base1Ptr**. A continuación asignamos la dirección del objeto derivado **d** al


```
Object b1 contains integer 10
Object b2 contains character Z
Object d contains:
    Integer: 7
  Character: A
Real number: 3.5

Data members of Derived can be accessed individually:
    Integer: 7
  Character: A
Real number: 3.5

Derived can be treated as an object of either base class:
base1Ptr->getData() yields 7
base2Ptr->getData() yields A
```

Fig. 19.10 Manejador para el ejemplo de herencia múltiple (parte 6 de 6).

apuntador de clase base **base2Ptr** e imprimimos **char letter** invocando la función miembro **getData** de **Base2** partiendo de **base2Ptr**.

Este ejemplo mostró la mecánica de la herencia múltiple e introdujo un problema simple de ambigüedad. La herencia múltiple es un tema complejo, tratado en mayor detalle en textos de C++, como nuestro C++ *How to Program* (Prentice-Hall, 1994).

Resumen

- Una de las claves al poder de la programación orientada a objetos es conseguir la reutilización del software mediante la herencia.
- El programador puede determinar que la nueva clase debe heredar los miembros de datos y las funciones miembro de una clase base previa definida. En este caso, la nueva clase se conoce como la clase derivada.
- Con la herencia simple, una clase se deriva de una sola clase base. En la herencia múltiple, una clase derivada hereda de múltiples clases base (posiblemente no relacionadas).
- Una clase derivada por lo regular añade miembros de datos y funciones miembro propias, por lo que una clase derivada en general tiene una definición mayor que su clase base. Una clase derivada es más específica que su clase base y normalmente representa menos objetos.
- Una clase derivada no puede tener acceso a los miembros privados de su clase base; permitirlo violaría el encapsulado de la clase base. Una clase base puede, sin embargo, tener acceso a los miembros públicos y protegidos de su clase base.
- Al constructor de clase derivada siempre llamará primero al constructor de su clase base, a fin de crear y de inicializar los miembros de la clase base de la clase derivada.

- Los destructores serán llamados en orden inverso a las llamadas de constructor, por lo que un destructor de clase derivada será llamado antes del destructor de su clase base.
- La herencia permite la reutilización del software, lo que ahorra tiempo de desarrollo y fomenta el uso de software de alta calidad previamente probado y depurado.
- La herencia puede ser realizada a partir de bibliotecas de clases existentes.
- En algún momento el software será construido en su mayor parte partiendo de componentes estándar reutilizables, exactamente de la misma forma que como hoy en día se construye la mayor parte del hardware.
- El responsable de una puesta en práctica de una clase derivada no necesita tener acceso al código fuente de una clase base, pero necesita conocer la interfaz de la clase base y ser capaz de enlazar con el código objeto de la clase base.
- Un objeto de una clase derivada puede ser tratado como un objeto de su correspondiente clase base pública. Sin embargo, lo inverso no es cierto.
- Una clase base existe en una relación jerárquica con sus clases derivadas.
- Una clase puede existir por sí misma. Cuando dicha clase es utilizada con el mecanismo de herencia, se convierte ya sea en clase base, que proporciona atributos y comportamientos a otras clases, o la clase se convierte en una clase derivada, que hereda dichos atributos y comportamientos.
- Una jerarquía de herencia puede tener una profundidad arbitraria dentro de las limitaciones físicas de un sistema particular.
- Las jerarquías son herramientas útiles para comprender y administrar la complejidad. Siendo que el software se está haciendo cada vez más complejo, C++ proporciona mecanismos para apoyar estructuras jerárquicas mediante la herencia y el polimorfismo.
- Se puede utilizar una conversión explícita (cast) para convertir un apuntador de clase base a un apuntador de clase derivada. Si dicho apuntador debe ser desreferenciado, primero deberá hacerse que señale a un objeto del tipo de la clase derivada.
- El acceso protegido sirve como un nivel intermedio de protección entre el acceso público y el acceso privado. Se puede tener acceso a los miembros protegidos de una clase base por los miembros y amigos de la clase base y por los miembros y amigos de las clases derivadas; ninguna otra de las funciones pueden tener acceso a los miembros protegidos de la clase base.
- Los miembros protegidos pueden ser utilizados para extender privilegios a clases derivadas, en tanto que niegan dichos privilegios a funciones no de clase y no amigos.
- La herencia múltiple produce jerarquías arborescentes. Estas gráficas no tienen ciclos porque todas las flechas apuntan en la misma dirección. Dichas gráficas se conocen como gráficas acíclicas dirigidas (DAG).
- Al derivar una clase de una clase base, la clase base puede ser declarada **`public`**, **`protected`**, o **`private`**.
- Al derivar una clase de una clase base **`public`**, los miembros **`public`** de la clase base se convierten en miembros **`public`** de la clase derivada, y los miembros **`protected`** de la clase base se convierten en miembros **`protected`** de la clase derivada.

- Al derivar una clase de una clase base **`protected`**, los miembros **`public`** y **`protected`** de la clase base se convierten en miembros **`protected`** de la clase derivada.
- Al derivar una clase de la clase base **`private`**, los miembros **`public`** y **`protected`** de la clase base se convierten en miembros **`private`** de la clase derivada.
- Una clase base puede ser o una clase base directa de una clase derivada o una clase base indirecta de una clase derivada. Una clase base directa estará de forma explícita listada al declararse la clase derivada. Una clase base indirecta no está listada explícitamente; más bien es herencia a partir de varios niveles hacia arriba en la estructura de jerarquía arbórea.
- Cuando un miembro de clase base es inapropiado para una clase derivada, podemos redefinir dicho miembro en la clase derivada.
- Es importante distinguir entre relaciones "es un" y relaciones "tiene un". En una relación "tiene un", un objeto de clase tiene un objeto de otra clase como un miembro. En una relación "es un", un objeto de una clase de tipo derivado también puede ser tratado como un objeto de tipo de clase base. "Es un" es herencia. "Tiene un" es composición.
- Un objeto de clase derivada puede ser asignado a un objeto de clase base. Este tipo de asignación tiene sentido por que la clase derivada tiene miembros que corresponden a cada uno de los miembros de la clase base. Un apuntador a un objeto de clase derivada puede ser convertido implícitamente en un apuntador a un objeto de clase base.
- Es posible convertir un apuntador de clase base a un apuntador de clase derivada utilizando conversión explícita (cast). El destino debe ser un objeto de clase derivada.
- Una clase base especifica un estado común. Todas la clases derivadas de una clase base heredan las capacidades de dicha clase base. En el proceso de diseño orientado a objetos, el diseñador busca el estado común y lo segrega para formar clases base deseables. Las clases derivadas a continuación son personalizadas más allá de las capacidades heredadas de la clase base.
- Puede ser confuso leer un conjunto de declaraciones de clase derivada, porque no todos los miembros de la clase derivada están presentes en dichas declaraciones. En particular, los miembros heredados no están listados en las declaraciones de clase derivada, pero estos miembros están de hecho presentes en dichas clases derivadas.
- Las relaciones "tiene un" son ejemplos de creación de nuevas clases mediante composición de clase existentes.
- Los constructores de objeto miembro son llamados en el orden en el cual dichos objetos han sido declarados. En la herencia, los constructores de clase base son llamados en el orden en el cual se especificó la herencia, y antes del constructor de la clase derivada.
- Tratándose de un objeto de clase derivada, primero se llama al constructor de clase base, y a continuación el constructor de clase derivada.
- Cuando un objeto de clase derivada sale de alcance, se llaman a los destructores en orden inverso a la de los constructores —primero se llama al destructor de la clase derivada y a continuación al destructor de la clase base.
- Una clase puede ser derivada de más de una clase base; dicha derivación se llama herencia múltiple.

- Indique herencia múltiple haciendo seguir al indicador de herencia de dos puntos (`:`) con una lista separada por comas de clases base.
- El constructor de clase derivada llama a los constructores de clase base para cada una de sus clases base, mediante la sintaxis de inicializador de miembro.

Terminología

abstracción
ambigüedad en herencia múltiple
clase base
constructor por omisión de clase base
constructor de clase base
destructor de clase base
inicializador de clase base
apuntador de clase base
jerarquía de clase
biblioteca de clase
cliente de una clase
composición
personalizar software
clase derivada
constructor de clase derivada
destructor de clase derivada
apuntador de clase derivada
clase base directa
gráfica acíclica dirigida (DAG)
amigos de una clase base
amigos de una clase derivada
relación *tiene un*
relación jerárquica
clase base indirecta
error de recursión infinito
herencia
relación *es un*
relación *conoce a*
control de acceso de miembro
clase miembro
objeto miembro
herencia múltiple
programación orientada a objetos (OOP)
apuntador a un objeto de clase base
apuntador a un objeto de clase derivada
clase base privada
herencia privada
clase base protegida
derivación protegida
palabra reservada `protected`
miembro protegido de una clase
clase base pública
herencia pública
redefinición de un miembro de clase base
herencia simple
reutilización de software
componentes estándar de software
subclase
superclase
relación *utiliza un*

Errores comunes de programación

19.1 Puede causar errores tratar un objeto de clase base como si fuera un objeto de clase derivada.

19.2 Efectuar la conversión explícita de un apuntador de clase base que señala a un objeto de clase base a un apuntador de clase derivada y a continuación hacer referencia a miembros de clase derivada que no existen en dicho objeto.

19.3 Cuando se redefine una función miembro de clase base en una clase derivada, es común hacer que la versión de clase derivada llame a la versión de clase base y haga algún trabajo adicional. Causa recursión infinita no utilizar el operador de resolución de alcance para hacer referencia a la función miembro de la clase base, porque la función miembro de la clase derivada, de hecho, está llamando a sí misma.

19.4 Escribir un programa que dependa de que los objetos miembros sean construidos en un orden particular, puede producir errores sutiles.

19.5 Asignar un objeto de clase derivada a un objeto de una clase base correspondiente, y a continuación intentar hacer referencia a miembros de sólo la clase derivada en el objeto nuevo de clase base.

19.6 Puede causar error convertir en forma explícita (cast) un apuntador de clase base a un apuntador de clase derivada, si a continuación dicho apuntador es utilizado para hacer referencia a un objeto de clase base que no tenga los miembros deseados de la clase derivada.

Prácticas sanas de programación

19.1 Al heredar capacidades que no son necesarias en la clase derivada, dichas capacidades se deben enmascarar mediante redefinición de las funciones.

19.2 La herencia múltiple, si se utiliza adecuadamente, es una capacidad poderosa. La herencia múltiple deberá ser utilizada cuando exista una relación "es una" entre un nuevo tipo y dos o más tipos existentes (es decir, tipo A "es un" tipo B y "es un" tipo C).

Sugerencia de rendimiento

19.1 Si las clases producidas mediante la herencia son más grandes de lo requerido, pudiera dar como resultado un desperdicio de memoria y de recursos de proceso.

Observaciones de ingeniería de software

19.1 Una clase derivada no puede tener acceso directo a miembros privados de su clase base.

19.2 Una redefinición de una función miembro de clase base en una clase derivada no necesaria debe tener la misma signatura que la función miembro de clase base.

19.3 Cuando en una clase derivada se cree un objeto, primero se ejecutará el constructor de clase base, después se ejecutarán los constructores de los objetos miembros correspondientes a la clase derivada, y después se ejecutará el constructor de la clase derivada. Los destructores serán llamados en orden inverso en el cual fueron llamados sus correspondientes constructores.

19.4 Crear una clase derivada no afecta el código fuente o el código objeto de su clase base; mediante la herencia se conserva la integridad de la clase base.

19.5 En un sistema orientado a objetos, las clases a menudo están íntimamente relacionadas. "Disgregue y elimine" atributos y comportamientos comunes y colóquelos en una clase base. A continuación utilice la herencia para formar clases derivadas.

19.6 Una clase derivada contiene los atributos y comportamientos de su clase base. Una clase derivada también puede contener atributos y comportamientos adicionales. Con la herencia, la clase base puede ser compilada independiente de la clase derivada. Para tener la capacidad de combinar estos atributos y comportamientos adicionales con la clase base para formar la clase derivada basta compilar los atributos y comportamientos incrementados de la clase derivada.

19.7 Las modificaciones a una clase base no requieren que sea modificada la clase derivada, siempre y cuando la interfaz pública de la clase base se mantenga sin modificación. Pudiera sin embargo, ser necesario recompilar las clases derivadas.

19.8 Las modificaciones a una clase miembro no requieren que su clase compuesta que la encierra sea modificada, siempre y cuando se mantenga sin modificación la interfaz pública de la clase miembro. Note que pudiera ser necesario recompilar la clase compuesta.

Ejercicios de autoevaluación

19.1 Llene cada uno de los siguientes espacios en blanco:

a) Si la clase `Alpha` hereda de la clase `Beta`, la clase `Alpha` se llama la clase _____ y la clase `Beta` se llama la clase _____.

b) C++ proporciona _____, lo que permite que una clase derivada herede de muchas clases base, aún si dichas clases base no están relacionadas.
c) La herencia permite _____ lo que ahorra tiempo en el desarrollo, y fomenta el uso de software de alta calidad previamente probado.
d) Un objeto de una clase _____ puede ser tratado como si fuera un objeto de su correspondiente clase _____.
e) Para convertir un apuntador de clase base a un apuntador de clase derivada deberá utilizarse un _____ porque el compilador considera lo anterior una operación peligrosa.
f) Los tres especificadores de acceso de miembros son _____, _____ y _____.
g) Al derivar una clase de una clase base con herencia `public`, los miembros `public` de la clase se convierten en miembros _____ de la clase derivada, y los miembros `protected` de la clase base se convierten en miembros _____ de la clase derivada.
h) Al derivar una clase de una clase base con herencia `protected`, los miembros `public` de la clase base se convierten en miembros _____ de la clase derivada, y los miembros `protected` de la clase base e convierten en miembros _____ de la clase derivada.
i) Una relación "tiene un" entre clases representa _____ y una relación "es un" entre clases representa _____.

Respuesta a los ejercicios de autoevaluación

19.1 a) derivado, base. b) herencia múltiple. c) reutilización del software. d) derivada, base. e) conversión explícita (cast). f) `public, protected, private`. g) `public protected`. h) `protected`, `protected`. i) composición, herencia.

Ejercicios

19.2 Considere la clase `bicycle`. En base en sus conocimientos de algunos componentes básicos y comunes de las bicicletas, muestre una jerarquía de clases en la cual la clase `bicycle` hereda de otras clases, las cuales a su vez también heredan de otras clases. Analice la producción de varios objetos de la clase `bicycle`. Analice la herencia correspondiente a la clase `bicycle` para otras clases derivadas íntimamente relacionadas.

19.3 Defina brevemente cada uno de los términos siguientes: herencia, herencia múltiple, clase base y clase derivada.

19.4 Analice el por qué se considera peligroso para el compilador la conversión del apuntador de clase base a un apuntador de clase derivada.

19.5 Señale todas las formas que le vengan a la mente tanto de dos como de tres dimensiones, y organice estas formas en una jerarquía de formas. Su jerarquía deberá tener una clase base `Shape`, a partir de la cual se derivarán la clase `TwoDimensionalShape` y la clase `ThreeDimensionalShape`. Una vez desarrollada la jerarquía, defina cada una de las clases en la jerarquía. Utilizaremos esta jerarquía en los ejercicios del capítulo 20 para procesar todas las formas como objetos de la clase base `Shape`. Esta es una técnica conocida como polimorfismo.

19.6 Un tipo popular de clase es la que se llama una clase colección o una clase contenedor. Una clase de este tipo contiene elementos de otras clases. Algunos tipos de clases de colección son los arreglos, las pilas, las colas, las listas enlazadas, los árboles, las cadenas de caracteres, los conjuntos, las bolsas, los diccionarios, las tablas, etcétera. Una clase contenedor o de colección proporciona en forma típica servicios como es insertar un elemento, borrarlo y buscarlo, combinar dos conjuntos o colecciones, determinar la intersección de dos colecciones (es decir, los elementos que sean comunes a ambas), imprimir una colección, encontrar el elemento más grande en la colección, encontrar el elemento más pequeño en la colección, encontrar la suma de los elementos de la colección, etcétera.

a) Haga una lista de todos los tipos de clases de colección que se le puedan ocurrir (incluyendo aquellos que ya hemos mencionado).
b) Arregle estas clases de colección en una jerarquía de clases. Quizás desee distinguir entre colecciones ordenadas y colecciones desordenadas.
c) Ponga en práctica todas las clases que pueda, utilizando la herencia para minimizar la cantidad de código nuevo que deba escribir para la creación de cada una de las nuevas clases.
d) Escriba un programa manejador que pruebe cada una de las clases en su jerarquía de herencia.

20

Funciones virtuales y polimorfismo

Objetivos

- Comprender el concepto de polimorfismo.
- Comprender cómo declarar y utilizar las funciones virtuales para conseguir el polimorfismo.
- Comprender la diferencia entre clases abstractas y clases concretas.
- Aprender a declarar funciones virtuales puras a fin de crear clases abstractas.
- Valorizar cómo el polimorfismo consigue que los sistemas sean extensibles y mantenibles.

¡Oh! tiene usted una repetición maldecible, y es ciertamente capaz de corromper a un santo.
William Shakespeare
Henry IV, Part I, 1, ii

Proposiciones generales no deciden los casos concretos.
Oliver Wendell Holmes

Un filósofo de imponente estatura no piensa en el vacío. Inclusive sus más abstractas ideas están, hasta cierto punto, condicionadas por lo que es o no sabido en la era en que vive.
Alfred North Whitehead

Sinopsis

20.1 Introducción

Mediante las *funciones virtuales* y el *polimorfismo*, se hace posible diseñar y poner en práctica sistemas que son de mayor facilidad extensibles. Se pueden escribir programas para procesar objetos de clases existentes y de clases que aun no existan cuando el programa esté en desarrollo. Si esas clases deben ser derivadas de clases base que el programa conozca, éste puede proporcionar un marco general para manipular los objetos de las clases base, y los objetos de las clases derivadas encajarán bien dentro de este marco. De hecho, basándose en este principio, productos completos son elaborados, conocidos como marcos de aplicación.

20.2 Campos de tipo y enunciados switch

Una forma de tratar los objetos de muchos tipos distintos es utilizar un enunciado **`switch`**, a fin de tomar la acción apropiada sobre cada objeto, basándose en su tipo. Al utilizar la lógica **`switch`** se presentan muchos problemas. Se le podría olvidar al programador llevar a cabo la correspondiente prueba de tipo, cuando sea requerida. También se le podría olvidar probar todos los casos posibles, incluidos en un **`switch`**. Si se modifica un sistema basado en **`switch`**, el programador podría olvidar insertar los nuevos casos en los enunciados **`switch`** existentes. Toda modificación a los enunciados **`switch`** efectuados para manejar nuevos tipos, exigen que sea modificado cada enunciado **`switch`** del sistema; rastrear dichos enunciados puede resultar tardado y sujeto a errores.

Como veremos, las funciones virtuales y la programación polimórfica puede eliminar la necesidad de la lógica **`switch`**. El programador puede utilizar el mecanismo de las funciones virtuales para ejecutar en forma automática la lógica equivalente, evitando por lo tanto los tipos de errores asociados típicamente con la lógica **`switch`**.

Observación de ingeniería de software 20.1

Una consecuencia interesante del uso de funciones virtuales y polimorfismo es que los programas adquieren una apariencia simplificada. Contienen menos lógica de bifurcación, prefiriendo un código secuencial más sencillo. Esta simplificación facilita probar, depurar y mantener el programa.

20.3 Funciones virtuales

Suponga un conjunto de clases de forma, como **`Circle`**, **`Triangle`**, **`Rectangle`**, **`Square`**, etcétera, todas ellas derivadas de la clase base **`Shape`**. En la programación orientada a objetos, cada una de estas clases estaría investida con la capacidad de dibujarse a sí misma. Aunque cada clase tiene su propia función **`draw`**, la función **`draw`** correspondiente a cada forma es bien distinta. Cuando se dibuja una forma, cualquiera que ésta sea, sería agradable tener la capacidad de tratar todas estas formas en una genérica, como objetos de la clase base **`Shape`**. Entonces al dibujar cualquier forma, sólo llamaríamos a la función **`draw`** de la clase base **`Shape`**, y dejaríamos que el programa determinase en forma dinámica (es decir, en tiempo de ejecución) cual de las funciones **`draw`**, de las clases derivadas, utilizar.

Para habilitar este tipo de comportamiento, declaramos **`draw`** en la clase base en forma de una *función virtual*, a continuación, en cada una de las clases derivadas *redefinimos* **`draw`**, a fin de que se dibuje la forma apropiada. Se declara una función virtual en la clase base precediendo el prototipo de función con la palabra reservada **`virtual`**.

Observación de ingeniería de software 20.2

Una vez declarada una función como virtual, se conserva como virtual a lo largo de toda la jerarquía de herencia, a partir de dicho punto.

Práctica sana de programación 20.1

A pesar que ciertas funciones en forma implícita son virtuales, en razón de una declaración ya hecha en la jerarquía de clase, algunos programadores prefieren declarar estas funciones virtuales en forma explícita en cada uno de los niveles de la jerarquía, a fin de promover claridad en el programa.

Observación de ingeniería de software 20.3

Cuando una clase derivada decide no definir una función virtual, la clase derivada sólo heredará la función virtual de la clase base inmediata.

Si en la clase base la función **`draw`** ha sido declarada **`virtual`**, y si entonces utilizamos un apuntador de clase base para señalar al objeto de la clase derivada y utilizando este apuntador invocamos la función **`draw`**, el programa seleccionará de manera dinámica (es decir, en tiempo de ejecución) la función correcta **`draw`** de la clase derivada. Esto se conoce como *ligadura dinámica* (vea las figuras 20.1 y 20.2).

Observación de ingeniería de software 20.4

Las funciones virtuales redefinidas deben tener el mismo tipo de regreso y la misma firma que la función virtual base.

Error común de programación 20.1

Resultará un error de sintaxis redefinir una clase derivada como función virtual de clase base, sin asegurarse que la función derivada tenga el mismo tipo de regreso y signatura que la versión de clase base.

Si una función virtual es llamada haciendo referencia a un objeto específico por nombre, y utilizando el operador de selección miembro punto, la referencia se resuelve en tiempo de compilación (esto se llama *ligadura estática*) y la función virtual llamada es aquella definida para, (o heredada por) la clase de dicho objeto en particular.

La homonimia no utiliza ligadura dinámica. Más bien, la homonimia es resuelta en tiempo de compilación al seleccionar la definición de función con una firma que coincida con la llamada de función (utilizando posiblemente una conversión implícita cast de tipo para que la coincidencia ocurra). Esto corresponde a ligadura estática.

20.4 Clases base abstractas y clases concretas

Cuando pensamos sobre una clase como un tipo, suponemos que serán producidos objetos de dicho tipo. Existen, sin embargo, muchas situaciones en las cuales resulta útil definir clases para las cuales el programador no tiene intención de producir ningún objeto. Estas clases se conocen como *clases abstractas*. Dado que en situaciones de herencia son utilizadas como clases base, a menudo nos referiremos a ellas como *clases base abstractas*. A partir de una clase base abstracta no se pueden producir objetos.

El único fin de una clase abstracta es proporcionar una clase base apropiada, a partir de la cual las clases pueden heredar interfaz y/o puesta en práctica. Las clases a partir de las cuales los objetos se pueden producir, se conocen como *clases concretas*.

Podríamos tener una clase base abstracta **`TwoDimensionalObject`** y derivar clases concretas, como **`Square`**, **`Circle`**, **`Triangle`**, etcétera. También podríamos tener una clase base abstracta **`ThreeDimensionalObject`** y derivar clases concretas como **`Cube`**, **`Sphere`**, **`Cylinder`**, **`Pyramid`**, etcétera. Estas clases base abstractas resultan demasiado genéricas para definir objetos reales; antes de pensar en la producción de objetos necesitamos ser más específicos. Y esto es lo que hacen las clases concretas; dan la definición que convierte en razonable la producción de objetos.

Una clase con funciones virtuales se hace abstracta al declarar uno o más de sus funciones virtuales como puras. Una *función virtual pura* es una que en su declaración contenga un *inicializador de* **`= 0`**, como en

```
virtual float earnings() const = 0;     // pure virtual
```

Observación de ingeniería de software 20.5

Si una clase se deriva de una clase con una función virtual pura, y para dicha función virtual pura no se ha dado definición en la clase derivada, entonces la función virtual también se conserva pura en la clase derivada. Por consecuencia, también la clase derivada será una clase abstracta.

Error común de programación 20.2

Es un error de sintaxis intentar producir un objeto a partir de una clase abstracta (es decir, de una clase que contenga una o más funciones virtuales puras).

Una jerarquía no necesita contener ninguna clase abstracta, pero como veremos, mucho sistemas de calidad orientados a objetos tienen jerarquías de clase encabezadas por una clase abstracta. En algunos casos, las clases abstractas forman los primeros niveles superiores de la jerarquía. Un buen ejemplo de lo anterior es la jerarquía de forma. La jerarquía podría ser encabezada por la clase base abstracta **`Shape`**. En el siguiente nivel inferior, podríamos tener dos

clases base más, de tipo abstracto, es decir **`TwoDimensionalShape`** y **`ThreeDimensionalShape`**. El siguiente nivel hacia abajo empezaría a definir clases concretas correspondientes a formas de dos dimensiones, como círculos, cuadrados y clases concretas correspondientes a formas tridimensionales, como esferas y cubos.

20.5 Polimorfismo

C++ permite el *polimorfismo* —la capacidad de objetos de clases diferentes, relacionados mediante la herencia, a responder de forma distinta a una misma llamada de función miembro. Si, por ejemplo, la clase **`Rectangle`** es derivada de la clase **`Quadrilateral`**, entonces un objeto **`Rectangle`** *es una* versión más específica de un objeto **`Quadrilateral`**. Una operación (como el cálculo del perímetro o de la superficie) que pueda ser efectuada en un objeto de la clase **`Quadrilateral`**, también podrá ser ejecutada en un objeto de la clase **`Rectangle`**.

El polimorfismo se pone en práctica vía las funciones virtuales. Cuando se hace una solicitud, a través de un apuntador de clase base (o una referencia) para usar una función virtual, C++ escoge la función redefinida correcta, en la clase derivada apropiada, asociada con el objeto.

Algunas veces, una función miembro no virtual es definida en una clase base y redefinida en una derivada. Si una función miembro como ésta es llamada mediante un apuntador de clase base, se utilizará la versión de clase base. Si la función miembro es llamada mediante un apuntador de clase derivada, se utilizará la versión de clase derivada. Esto es comportamiento no polimórfico.

Mediante el uso de las funciones virtuales y del polimorfismo, una llamada de función miembro puede hacer que ocurran distintas acciones, dependiendo del tipo del objeto que reciba la llamada. Esto le da al programador una capacidad de expresión tremenda. En las siguientes secciones veremos ejemplos del poder del polimorfismo y de las funciones virtuales.

Observación de ingeniería de software 20.6

Mediante las funciones virtuales y el polimorfismo, el programador puede ocuparse de generalidades y dejar que en tiempo de ejecución, el entorno se preocupe de lo específico. El programador puede dirigir una amplia variedad de objetos haciendo que se comporten de formas apropiadas a dichos objetos incluso sin conocer los tipos de los mismos.

Observación de ingeniería de software 20.7

El polimorfismo fomenta la extensibilidad: software escrito para invocar comportamiento polimórfico se escribe en forma independiente del tipo de los objetos a los cuales los mensajes son enviados. Por lo tanto, nuevos tipos de objetos, que pudieran responder a mensajes existentes, pueden ser añadidos en dicho sistema sin modificar el sistema base. A excepción de la parte de código cliente que produce nuevos objetos, los programas no necesitan ser recompilados.

Observación de ingeniería de software 20.8

Una clase abstracta define una interfaz para los distintos miembros de una jerarquía de clase. La clase abstracta contiene funciones virtuales puras, que serán definidas en las clases derivadas. Mediante el polimorfismo todas las funciones en la jerarquía pueden utilizar esta misma interfaz.

A pesar de que no podemos producir objetos de clases base abstractas, sí *podemos* declarar apuntadores a clases base abstractas. Dichos apuntadores podrán ser utilizados después para permitir manipulaciones polimórficas de objetos de clases derivadas, cuando dichos objetos son producidos a partir de clases concretas.

Consideremos aplicaciones del polimorfismo y de las funciones virtuales. Un administrador de pantalla necesita mostrar una variedad de objetos, incluyendo nuevos tipos de objetos, que serán añadidos al sistema, aún después de que haya sido escrito el administrador de pantalla. El sistema pudiera necesitar mostrar varias formas (es decir, la clase base es **Shape**) como cuadrados, círculos, triángulos, rectángulos y similar (cada una de estas clases de formas es derivada de la clase base **Shape**). El administrador de pantalla utiliza apuntadores de clase base para administrar todos los objetos a mostrar. Para dibujar cualquier objeto (independiente del nivel en la jerarquía de herencia en el cual aparezca dicho objeto), el administrador de pantalla utiliza un apuntador de clase base a dicho objeto, y envía al objeto un mensaje **draw**. En la clase base **Shape** la función **draw** ha sido declarada como virtual pura y en cada una de las clases derivadas ha sido redefinida. Cada objeto sabe como dibujarse a sí mismo. El administrador de pantalla no tiene que preocuparse sobre estos detalles; le indica a cada objeto que se dibuje a sí mismo.

El polimorfismo es en particular efectivo para la puesta en práctica de sistemas de software en capas. Por ejemplo, en los sistemas operativos, cada tipo de dispositivo físico pudiera operar en forma bastante distinta a los demás. Independiente de lo anterior, órdenes o comandos para *leer* o *escribir* datos de y hacia los dispositivos pueden tener una cierta uniformidad. El mensaje *write* enviado a un objeto manejador de dispositivo, necesita ser interpretado en específico en el contexto de dicho manejador de dispositivo, y de la forma en que dicho manejador de dispositivo manipula dispositivos de un tipo específico. Pero la llamada *write* por sí misma en realidad no es distinta de la llamada *write* para cualquier otro dispositivo —simplemente colocar cierto número de bytes de la memoria en dicho dispositivo. Un sistema operativo orientado a objetos pudiera utilizar una clase base abstracta para proporcionar una interfaz apropiada para todos los manejadores de dispositivo. Entonces, mediante herencia, a partir de dicha clase base abstracta, se formarían clases derivadas, todas operando en forma similar. Las capacidades (es decir, la interfaz pública) ofrecida por los manejadores de dispositivo se dan en la clase base abstracta como funciones virtuales puras. Las puestas en práctica de estas funciones virtuales se dan en las clases derivadas, correspondientes a los tipos específicos de manejadores de dispositivo.

En el capítulo 17, introdujimos el concepto de los iteradores. Es común definir una *clase iterador*, que recorra todos los objetos de una colección. Si por ejemplo, usted desea imprimir una lista de objetos en una lista enlazada, se puede producir un objeto iterador, que, cada vez que se llame al iterador, devolverá el siguiente elemento de la lista enlazada. Los iteradores son por lo general usados en la programación polimórfica para recorrer una lista enlazada de objetos, correspondiente a varios niveles de una jerarquía. Los apuntadores a una lista como ésta, serían todos apuntadores de clase base.

20.6 Estudio de caso: un sistema de nóminas utilizando polimorfismo

Usemos funciones virtuales y polimorfismo para llevar a cabo cálculos de nómina, basados en el tipo de un empleado (figura 20.1). Utilizamos una clase base **Employee**. Las clases derivadas de **Employee** son **Boss**, mismo que se le paga un salario semanal fijo, independiente del número de horas trabajadas, **CommissionWorker**, que obtiene un salario base fijo más un porcentaje de las ventas, **PieceworkWorker**, a quien se le paga según el número de elementos producidos, y **HourlyWorker**, que cobra por hora y recibe paga por tiempo extraordinario.

Una llamada de función **earnings** es ciertamente aplicable en forma genérica a todos los empleados. Pero la forma en que se calculen los ingresos de cada persona dependerá de la clase del empleado y estas clases son todas derivadas de la clase base **Employee**. Por lo tanto, en la clase base **Employee**, **earnings** es declarado **virtual**, y se dan puestas en práctica apropiadas

de **earnings** para cada una de las clases derivadas. A continuación, a fin de calcular los ingresos de cada empleado, el programa sólo utiliza un apuntador de clase base al objeto de dicho empleado, e invoca la función **earnings**. En un sistema real de nóminas, los distintos objetos "empleado" pudieran estar almacenados en una lista enlazada, en la cual, cada uno de los apuntadores es del tipo **Employee ***. El programa entonces sólo recorrería la lista enlazada, nodo por nodo, utilizando los apuntadores **Employee *** a fin de invocar la función **earnings** sobre cada objeto.

Veamos la clase **Employee** (figura 20.1, parte 1 y 2). Las funciones miembro públicas incluyen un constructor, que toma como argumentos el nombre y el apellido; un destructor, que recupera la memoria asignada dinámicamente; una función *get*, que devuelve el nombre; una función *get*, que devuelve el apellido; y, por último, dos funciones virtuales puras **earnings** y **print**. ¿Por qué estas funciones son virtuales puras? La respuesta es que no tiene sentido hacer puestas en práctica de dichas funciones en la clase **Employee**. Al hacer estas funciones virtuales puras, estamos indicando que proporcionaremos puestas en práctica en las clase derivada, pero no en la clase base misma. No es posible calcular los ingresos de un empleado genérico —primero tenemos que saber qué tipo de empleado es— ni tampoco, a partir de un empleado genérico, podemos imprimir el tipo de empleado.

La clase **Boss** (figura 20.1, parte 3 y 4) es derivada de **Employee** mediante herencia pública. Las funciones miembro públicas incluyen un constructor, que toma como argumentos un nombre, un apellido y un salario semanal, y pasa el nombre y el apellido al constructor **Employee** para inicializar los miembros **firstName** y **lastName** de la parte de clase base del objeto de clase derivada; una función *set*, para asignar un valor nuevo al miembro de datos privado **weeklySalary**; una función virtual **earnings**, que define cómo calcular los ingresos **Boss**; y una función virtual **print**, que extrae el tipo del empleado, seguido por el nombre del mismo.

```
// EMPLOY2.H
// Abstract base class Employee
#ifndef EMPLOY2_H
#define EMPLOY2_H

class Employee {
public:
   Employee(const char *, const char *);
   ~Employee();
   const char *getFirstName() const;
   const char *getLastName() const;

   // Pure virtual functions make
   // Employee an abstract base class.
   virtual float earnings() const = 0; // pure virtual
   virtual void print() const = 0;     // pure virtual
private:
   char *firstName;
   char *lastName;
};

#endif
```

Fig. 20.1 Clase base abstracta **Employee** (parte 1 de 12).

```
// EMPLOY2.CPP
// Member function definitions for
// abstract base class Employee.
//
// Note: No definitions given for pure virtual functions.
#include <iostream.h>
#include <string.h>
#include <assert.h>
#include "employ2.h"

// Constructor dynamically allocates space for the
// first and last name and uses strcpy to copy
// the first and last names into the object.
Employee::Employee(const char *first, const char *last)
{
   firstName = new char[ strlen(first) + 1 ];
   assert(firstName != 0);     // test that new worked
   strcpy(firstName, first);

   lastName = new char[ strlen(last) + 1 ];
   assert(lastName != 0);      // test that new worked
   strcpy(lastName, last);
}

// Destructor deallocates dynamically allocated memory
Employee::~Employee()
{
   delete [] firstName;
   delete [] lastName;
}

// Return a pointer to the first name
const char *Employee::getFirstName() const
{
   // Const prevents caller from modifying private data.
   // Caller should copy returned string before destructor
   // deletes dynamic storage to prevent undefined pointer.

   return firstName;   // caller must delete memory
}

// Return a pointer to the last name
const char *Employee::getLastName() const
{
   // Const prevents caller from modifying private data.
   // Caller should copy returned string before destructor
   // deletes dynamic storage to prevent undefined pointer.

   return lastName;   // caller must delete memory
}
```

Fig. 20.1 Definiciones de función miembro para la clase base abstracta **Employee** (parte 2 de 12).


```
// BOSS1.H
// Boss class derived from Employee
#ifndef BOSS1_H
#define BOSS1_H
#include "employ2.h"

class Boss : public Employee {
public:
   Boss(const char *, const char *, float = 0.0);
   void setWeeklySalary(float);
   virtual float earnings() const;
   virtual void print() const;
private:
   float weeklySalary;
};

#endif
```

Fig. 20.1 Clase **Boss** derivada de la clase base abstracta **Employee** (parte 3 de 12).

```
// BOSS1.CPP
// Member function definitions for class Boss
#include <iostream.h>
#include "boss1.h"

// Constructor function for class Boss
Boss::Boss(const char *first, const char *last, float s)
   : Employee(first, last)  // call base-class constructor
{ weeklySalary = s > 0 ? s : 0; }

// Set the Boss's salary
void Boss::setWeeklySalary(float s)
{ weeklySalary = s > 0 ? s : 0; }

// Get the Boss's pay
float Boss::earnings() const { return weeklySalary; }

// Print the Boss's name
void Boss::print() const
{
   cout << "\n             Boss: " << getFirstName()
        << ' ' << getLastName();
}
```

Fig. 20.1 Definiciones de función miembro para la clase **Boss** (parte 4 de 12).

La clase `CommissionWorker` (figura 20.1, parte 5 de 6) se deriva de **`Employee`** mediante herencia pública. Las funciones miembro públicas incluyen un constructor, que toma como argumentos un nombre, un apellido, un salario, una comisión y una cantidad de elementos vendidos y pasa el nombre y el apellido al constructor **`Employee`**, a fin de inicializar los miembros **`firstName`** y **`lastName`** de la parte de clase base del objeto de clase derivada; funciones *set*, para asignar nuevos valores a los miembros de datos privados **`salary`**, **`commission`** y **`quantity`**;

```
// COMMIS1.H
// CommissionWorker class derived from Employee
#ifndef COMMIS1_H
#define COMMIS1_H
#include "employ2.h"

class CommissionWorker : public Employee {
public:
   CommissionWorker(const char *, const char *,
                    float = 0.0, float = 0.0, int = 0);
   void setSalary(float);
   void setCommission(float);
   void setQuantity(int);
   virtual float earnings() const;
   virtual void print() const;
private:
   float salary;     // base salary per week
   float commission; // amount per item sold
   int quantity;     // total items sold for week
};

#endif
```

Fig. 20.1 Clase **CommissionWorker** derivada de la clase base abstracta **Employee** (parte 5 de 12).

una función virtual **earnings**, que define cómo calcular los ingresos **CommissionWorker**; y una función virtual **print**, que extrae el tipo del empleado seguido por el nombre del mismo.

La clase **PieceWorker** (figura 20.1, partes 7 y 8) se deriva de **Employee** mediante herencia pública. Las funciones miembro públicas incluyen un constructor, que toma como argumentos un nombre, un apellido, un salario por pieza y una cantidad de elementos producidos, y pasa el nombre y apellido al constructor **Employee**, a fin de inicializar los miembros **firstName** y **lastName** de la parte de clase base del objeto de clase derivada; funciones *set*, para asignar nuevos valores a los miembros de datos privados **wagePerPiece** y **quantity**; una función virtual **earnings**, que define cómo calcular los ingresos **PieceWorker**; y una función virtual **print**, que extrae el tipo del empleado seguido por el nombre del mismo.

La clase **HourlyWorker** (figura 20.1, partes 9 y 10) se deriva de **Employee** mediante herencia pública. Las funciones miembro públicas incluyen un constructor, que toma como argumentos un nombre, apellido, un salario, el número de horas trabajadas y pasa el nombre y el apellido al constructor **Employee**, a fin de inicializar los miembros **firstName** y **lastName** de la parte de clase base del objeto de clase derivada; funciones *set*, a fin de asignar nuevos valores a los miembros de datos privados **wage** y **hours**; una función virtual **earnings**, que define cómo calcular los ingresos **HourlyWorker**; y una función virtual **print**, que extrae el tipo del empleado seguido por el nombre del mismo.

El programa manejador (figura 20.1 partes 11 y 12) empieza declarando el apuntador de clase base, **ptr**, del tipo **Employee ***. Cada uno de los tres segmentos de código en **main** es similar, por lo que analizaremos sólo el primer segmento, que trata del objeto **Boss**. La línea

```
Boss b("John", "Smith", 800.00);
```



```
// COMMIS1.CPP
// Member function definitions for class CommissionWorker
#include <iostream.h>
#include "commis1.h"

// Constructor for class CommissionWorker
CommissionWorker::CommissionWorker(const char *first,
   const char *last, float s, float c, int q)
   : Employee(first, last)  // call base-class constructor
{
   salary = s > 0 ? s : 0;
   commission = c > 0 ? c : 0;
   quantity = q > 0 ? q : 0;
}

// Set CommissionWorker's weekly base salary
void CommissionWorker::setSalary(float s)
{ salary = s > 0 ? s : 0; }

// Set CommissionWorker's commission
void CommissionWorker::setCommission(float c)
{ commission = c > 0 ? c : 0; }

// Set CommissionWorker's quantity sold
void CommissionWorker::setQuantity(int q)
{ quantity = q > 0 ? q : 0; }

// Determine CommissionWorker's earnings
float CommissionWorker::earnings() const
{ return salary + commission * quantity; }

// Print the CommissionWorker's name
void CommissionWorker::print() const
{
   cout << "\nCommission worker: " << getFirstName()
        << ' ' << getLastName();
}
```

Fig. 20.1 Definiciones de función miembro para la clase **CommissionWorker** (parte 6 de 12).

produce el objeto de clase derivada **b** de la clase **Boss** y proporciona varios argumentos de constructor incluyendo un nombre, un apellido y un salario fijo semanal. La línea

```
ptr = &b;  // base-class pointer to derived-class object
```

coloca la dirección de la clase derivada del objeto **b** en el apuntador de clase base **ptr**. Esto es precisamente lo que debemos hacer para utilizar comportamiento polimórfico. La línea

```
ptr->print();                                  // dynamic binding
```

invoca a la función miembro **print** del objeto al cual señala **ptr**. Dado que **print** ha sido declarado como una función virtual de la clase base, el sistema invoca la función **print** del

```
// PIECE1.H
// PieceWorker class derived from Employee
#ifndef PIECE1_H
#define PIECE1_H
#include "employ2.h"

class PieceWorker : public Employee {
public:
   PieceWorker(const char *, const char *,
               float = 0.0, int = 0);
   void setWage(float);
   void setQuantity(int)
   virtual float earnings() const;
   virtual void print() const;

private:
   float wagePerPiece; // wage for each piece output
   int quantity;       // output for week
};

#endif
```

Fig. 20.1 Clase **PieceWorker** derivada de la clase base abstracta **Employee** (parte 7 de 12).

objeto de clase derivada —otra vez precisamente lo que llamamos comportamiento polimórfico. Esta llamada de función es un ejemplo de ligadura dinámica— la función es invocada a través de un apuntador de clase base, por lo que la decisión de cuál función deberá invocarse se pospone hasta el tiempo de ejecución. La línea

```
cout << " earned $" << ptr->earnings(); // dynamic binding
```

invoca la función miembro **earnings** del objeto al cual señala **ptr**. Dado de que **earnings** ha sido declarada una función virtual de la clase base, el sistema invoca a la función **earnings** del objeto de clase derivada. También éste es un ejemplo de ligadura dinámica. La línea

```
b.print();                          // static binding
```

invoca en forma explícita a la función **Boss** de la función miembro **print**, mediante el uso del operador de selección miembro punto sobre el objeto **b** específico de **Boss**. Esto es un ejemplo de ligadura estática, porque en tiempo de compilación el tipo del objeto para el cual se llama a la función es ya conocido. Esta llamada ha sido incluida para efectos de comparación con el fin de ilustrar que la función **print** correcta es invocada mediante el uso de la ligadura dinámica. La línea

```
cout << " earned $" << b.earnings();   // static binding
```

invoca de manera explícita la versión **Boss** de la función **earnings**, mediante el uso del operador de selección de miembro punto sobre el objeto **b** específico de **Boss**. También este es un ejemplo de ligadura estática. También se incluye esta llamada para efectos de comparación, a fin de ilustrar que la función **earnings** correcta es invocada utilizando ligadura dinámica.


```
// PIECE1.CPP
// Member function definitions for class PieceWorker

#include <iostream.h>
#include "piece1.h"

// Constructor for class PieceWorker
PieceWorker::PieceWorker(const char *first, const char *last,
                         float w, int q)
   : Employee(first, last)  // call base-class constructor
{
   wagePerPiece = w > 0 ? w : 0;
   quantity = q > 0 ? q : 0;
}

// Set the wage
void PieceWorker::setWage(float w)
{ wagePerPiece = w > 0 ? w : 0; }

// Set the number of items output
void PieceWorker::setQuantity(int q)
{ quantity = q > 0 ? q : 0; }

// Determine the PieceWorker's earnings
float PieceWorker::earnings() const
{ return quantity * wagePerPiece; }

// Print the PieceWorker's name
void PieceWorker::print() const
{
   cout << "\n     Piece worker: " << getFirstName()
        << ' ' << getLastName();
}
```

Fig. 20.1 Definiciones de función miembro para la clase **PieceWorker** (parte 8 de 12).

20.7 Clases nuevas y ligadura dinámica

El polimorfismo y las funciones virtuales ciertamente funcionarían bien en un mundo en el cual todas las clases posibles fueran conocidas con antelación. Pero también funcionan cuando en forma regular a los sistemas se les añaden nuevos tipos de clases.

Se les hace sitio a las nuevas clases mediante la ligadura dinámica (también conocido como *ligadura tardía*). Para que sea compilada una llamada de función virtual, no es necesario conocer el tipo de un objeto en tiempo de compilación. La llamada de función virtual se hace coincidir en tiempo de ejecución con la función miembro del objeto llamado.

Un programa administrador de pantalla puede ahora manejar nuevos objetos de exibición, conforme sean añadidos al sistema, sin necesidad de recompilación. La llamada de función **draw** se conserva idéntica. Los nuevos objetos mismos contienen sus propias capacidades de dibujo. Esto facilita añadir con un impacto mínimo nuevas capacidades a sistemas. También promueve la reutilización del software.

```
// HOURLY1.H
// Definition of class HourlyWorker
#ifndef HOURLY1_H
#define HOURLY1_H
#include "employ2.h"

class HourlyWorker : public Employee {
public:
   HourlyWorker(const char *, const char *,
                float = 0.0, float = 0.0);
   void setWage(float);
   void setHours(float);
   virtual float earnings() const;
   virtual void print() const;
private:
   float wage;   // wage per hour
   float hours;  // hours worked for week
};

#endif
```

Fig. 20.1 La clase **HourlyWorker** derivada de la clase base abstracta **Employee** (parte 9 de 12).

```
// HOURLY1.CPP
// Member function definitions for class HourlyWorker
#include <iostream.h>
#include "hourly1.h"

// Constructor for class HourlyWorker
HourlyWorker::HourlyWorker(const char *first, const char *last,
                           float w, float h)
   : Employee(first, last)   // call base-class constructor
{
   wage = w > 0 ? w : 0;
   hours = h >= 0 && h < 168 ? h : 0;
}

// Set the wage
void HourlyWorker::setWage(float w) { wage = w > 0 ? w : 0; }

// Set the hours worked
void HourlyWorker::setHours(float h)
{ hours = h >= 0 && h < 168 ? h : 0; }

// Get the HourlyWorker's pay
float HourlyWorker::earnings() const { return wage * hours; }

// Print the HourlyWorker's name
void HourlyWorker::print() const
{
   cout << "\n    Hourly worker: " << getFirstName()
        << ' ' << getLastName();
}
```

Fig. 20.1 Definiciones de función miembro para las clases **HourlyWorker** (parte 10 de 12).


```
// FIG20_1.CPP
// Driver for Employee hierarchy

#include <iostream.h>
#include <iomanip.h>
#include "employ2.h"
#include "boss1.h"
#include "commis1.h"
#include "piece1.h"
#include "hourly1.h"

main()
{
   // set output formatting
   cout << setiosflags(ios::showpoint) << setprecision(2);

   Employee *ptr;  // base-class pointer

   Boss b("John", "Smith", 800.00);
   ptr = &b;  // base-class pointer to derived-class object
                                          // dynamic binding
   ptr->print();                          // dynamic binding
   cout << " earned $" << ptr->earnings(); // dynamic binding
   b.print();                             // static binding
   cout << " earned $" << b.earnings();    // static binding

   CommissionWorker c("Sue", "Jones", 200.0, 3.0, 150);
   ptr = &c;  // base-class pointer to derived-class object
                                          // dynamic binding
   ptr->print();                          // dynamic binding
   cout << " earned $" << ptr->earnings(); // static binding
   c.print();                             // static binding
   cout << " earned $" << c.earnings();    // static binding

   PieceWorker p("Bob", "Lewis", 2.5, 200);
   ptr = &p;  // base-class pointer to derived-class object
                                          // dynamic binding
   ptr->print();                          // dynamic binding
   cout << " earned $" << ptr->earnings(); // static binding
   p.print();                             // static binding
   cout << " earned $" << p.earnings();    // static binding

   HourlyWorker h("Karen", "Price", 13.75, 40);
   ptr = &h;  // base-class pointer to derived-class object
                                          // dynamic binding
   ptr->print();                          // dynamic binding
   cout << " earned $" << ptr->earnings(); // static binding
   h.print();                             // static binding
   cout << " earned $" << h.earnings();    // static binding

   cout << endl;

   return 0;
}
```

Fig. 20.1 Jerarquía de derivación de clase "empleado" que utiliza una clase base abstracta (parte 11 de 12).

```
          Boss: John Smith earned $800.00
          Boss: John Smith earned $800.00
Commission worker: Sue Jones earned $650.00
Commission worker: Sue Jones earned $650.00
     Piece worker: Bob Lewis earned $500.00
     Piece worker: Bob Lewis earned $500.00
    Hourly worker: Karen Price earned $550.00
    Hourly worker: Karen Price earned $550.00
```

Fig. 20.1 Jerarquía de derivación de clase "empleado" que utiliza una clase base abstracta (parte 12 de 12).

La ligadura dinámica permite que fabricantes independientes de software (*ISV*), distribuyan software sin tener que revelar secretos propietarios. Estas distribuciones de software pueden estar formadas de solo los archivos de cabecera y archivos objeto. No es necesario revelar el código fuente. Los desarrolladores de software entonces pueden utilizar la herencia, para derivar nuevas clases a partir de aquellas proporcionadas por los ISV. El software, que funciona con las clases que proporcionan los ISV, continuará funcionando con las clases derivadas y utilizará (vía ligaduras dinámicas) las funciones virtuales redefinidas proporcionadas en dichas clases.

La ligadura dinámica requiere que en tiempo de ejecución, se encamine la llamada a una función miembro virtual a la versión de función virtual apropiada para la clase. Se pone en práctica una *tabla de funciones virtuales* llamada la *vtable*, en forma de un arreglo que contiene apuntadores de función. Cada clase que contenga funciones virtuales tiene una *vtable*. Para cada función virtual dentro de la clase, la *vtable* tiene una entrada que contiene un apuntador de función a la versión de la función virtual para uso de un objeto de dicha clase. La función virtual a usar para una clase particular podría ser la función definida en dicha clase, o podría ser una función heredada, ya sea directa o indirecta, de una clase base más alta dentro de la jerarquía.

Cuando una clase base proporciona una función miembro y la declara **`virtual`**, las clases derivadas pueden redefinir la función virtual, pero no están obligadas a redefinirla. Por lo tanto, una clase derivada puede utilizar la versión de clase base de una función miembro virtual, y esto quedaría indicado en la *vtable*.

Cada objeto de una clase con funciones virtuales contiene un apuntador a la *vtable* para dicha clase. Este apuntador no es accesible para el programador. Se obtiene el apuntador apropiado de función en la *vtable* y se desreferencia para completar la llamada en tiempo de ejecución.

Esta búsqueda y desreferenciación de apuntador en *vtable* requiere de una sobrecarga nominal en tiempo de ejecución.

Sugerencia de rendimiento 20.1

El polimorfismo, tal y como se pone en práctica mediante funciones virtuales y ligaduras dinámicas, resulta eficiente. Los programadores pueden utilizar estas capacidades con un impacto solo nominal sobre el rendimiento del sistema.

Sugerencia de rendimiento 20.2

Las funciones virtuales y las ligaduras dinámicas permiten la programación polimórfica, en contraste con la programación lógica **switch**. *Los compiladores optimizantes de C++ por lo regular generan código que se ejecuta con una eficiencia por lo menos igual a la de la lógica basada en* **switch** *codificada manualmente.*

20.8 Destructores virtuales

Puede ocurrir un problema, al utilizar el polimorfismo para procesar objetos dinámicamente asignados de una jerarquía de clase. Si es destruido, aplicando el operador **`delete`** a un apuntador de clase base al objeto, la función destructor de clase base será llamada sobre el objeto. Esto ocurrirá independiente del tipo del objeto al cual está señalado el apuntador de clase base e independiente del hecho que el destructor de cada clase tenga un nombre distinto.

Existe una solución simple para este problema —declarar virtual el destructor de clase base. Esto hará virtuales automáticamente a todos los destructores de clase derivada, a pesar de que no tengan el mismo nombre que el destructor de clase base. Ahora, si un objeto en la jerarquía es destruido en forma explícita, mediante la aplicación del operador **`delete`** a un apuntador de clase base a un objeto de clase derivada, se llamará al destructor de la clase apropiada.

Práctica sana de programación 20.2

Si una clase tiene funciones virtuales, incluya un destructor virtual, inclusive si para dicha clase no se requiere uno. Las clases que se deriven de esta clase pudieran contener destructores que deberán ser llamados en forma correcta.

Error común de programación 20.3

Declarar un constructor como función virtual. Los constructores no pueden ser virtuales.

20.9 Estudio de caso: cómo heredar interfaz, y puesta en práctica

Nuestro siguiente ejemplo (figura 20.2) vuelve a examinar la jerarquía de punto, círculo y cilindro del capítulo anterior, excepto que ahora encabezamos la jerarquía con la clase base abstracta **`Shape`**. **`Shape`** tiene una función virtual pura —**`printShapeName`**— de tal forma que **`Shape`** es una clase base abstracta. **`Shape`** contiene otras dos funciones virtuales, es decir **`area`** y **`volume`**, cada una de las cuales tiene una puesta en práctica que regresa un valor cero. **`Point`** hereda estas puestas en práctica de **`Shape`**. Esto tiene sentido, porque tanto el área como el volumen de un punto son cero. **`Circle`** hereda la función **`volume`** de **`Point`**, pero **`Circle`** proporciona su propia puesta en práctica correspondiente a la función **`area`**. **`Cylinder`** proporciona sus propias puestas en práctica, tanto para la función **`area`** como para la función **`volume`**.

Note que aunque **`Shape`** es una clase base abstracta, aún así contiene puestas en práctica de dichas funciones miembro, y dichas puestas en práctica son heredables. La clase **`Shape`** proporciona una interfaz heredable en la forma de tres funciones virtuales, que pueden ser contenidas por todos los miembros de la jerarquía. La clase **`Shape`** también proporciona algunas puestas en práctica, que podrán utilizar las clases derivadas en los primeros pocos niveles de la jerarquía.

Este estudio de caso hace énfasis en el hecho de que una clase puede heredar la interfaz y/o la puesta en práctica de una clase base. Las jerarquías diseñadas para la herencia de puestas en práctica tienden a tener su funcionalidad alta dentro de la jerarquía. Las jerarquías diseñadas para la herencia de interfaz tienden a tener su funcionalidad baja dentro de la jerarquía.

La clase base **`Shape`** (figura 20.2, parte 1) está formada de tres funciones virtuales públicas y no contiene ningun dato. La función **`printShapeName`** es virtual pura, por lo que es redefinida en cada una de las clases derivadas. Las funciones **`area`** y **`volume`** se definen para devolver **`0.0`**. Estas funciones se redefinen en las clases derivadas cuando es apropiado para dichas clases el tener un cálculo distinto para **`area`** y/o un cálculo distinto de **`volume`**.

```
// SHAPE.H
// Definition of abstract base class Shape
#ifndef SHAPE_H
#define SHAPE_H

class Shape {
public:
   virtual float area() const { return 0.0; }
   virtual float volume() const { return 0.0; }
   virtual void printShapeName() const = 0; // pure virtual
};

#endif
```

Fig. 20.2 Definición de la clase base abstracta **Shape** (parte 1 de 9).

La clase **Point** (figura 20.2 partes 2 y 3) es derivada de **Shape** con herencia pública. Un **Point** no tiene ni área ni volumen, por lo que las funciones miembro de clase base **area** y **volume** aquí no se redefinen —se heredan tal y como están definidas en **Shape**. La función **printShapeName** es una puesta en práctica de una función virtual, que en la clase base fue definida como una virtual pura. Otras funciones miembro incluyen una función *set*, a fin de asignar coordenadas nuevas **x** e **y** a un **Point** y función *get*, para regresar las coordenadas **x** e **y** de un **Point**

La clase **Circle** (figura 20.2 partes 4 y 5) es derivada de **Point** con herencia pública. Un **Circle** no tiene volumen, por lo que la función miembro de clase base **volume** no se redefine aquí —es heredada de **Shape** (y subsecuentemente en **Point**). Un **Circle** tiene área, por lo que en esta clase se redefine la función **area**. La función **printShapeName** es una puesta en

```
// POINT1.H
// Definition of class Point
#ifndef POINT1_H
#define POINT1_H
#include <iostream.h>
#include "shape.h"

class Point : public Shape {
   friend ostream &operator<<(ostream &, const Point &);
public:
   Point(float = 0, float = 0);  // default constructor
   void setPoint(float, float);
   float getX() const { return x; }
   float getY() const { return y; }
   virtual void printShapeName() const { cout << "Point: "; }
private:
   float x, y;    // x and y coordinates of Point
};

#endif
```

Fig. 20.2 Definición de clase **Point** (parte 2 de 9).


```
// POINT1.CPP
// Member function definitions for class Point
#include <iostream.h>
#include "point1.h"

Point::Point(float a, float b)
{
   x = a;
   y = b;
}

void Point::setPoint(float a, float b)
{
   x = a;
   y = b;
}

ostream &operator<<(ostream &output, const Point &p)
{
   output << '[' << p.x << ", " << p.y << ']';

   return output;   // enables concatenated calls
}
```

Fig. 20.2 Definiciones de función miembro para la clase **Point** (parte 3 de 9).

```
// CIRCLE1.H
// Definition of class Circle
#ifndef CIRCLE1_H
#define CIRCLE1_H
#include "point1.h"

class Circle : public Point {
   friend ostream &operator<<(ostream &, const Circle &);
public:
   // default constructor
   Circle(float r = 0.0, float x = 0.0, float y = 0.0);

   void setRadius(float);
   float getRadius() const;
   virtual float area() const;
   virtual void printShapeName() const { cout << "Circle: "; }
private:
   float radius;   // radius of Circle
};

#endif
```

Fig. 20.2 Definición de clase **Circle** (parte 4 de 9).

práctica de una función virtual, que en la clase base fue definida como una virtual pura. Si esta función no se redefine aquí, sería heredada la versión **Point** de la función.

```
// CIRCLE1.CPP
// Member function definitions for class Circle
#include <iostream.h>
#include <iomanip.h>
#include "circle1.h"

// Constructor for Circle call constructor for Point
Circle::Circle(float r, float a, float b)
   : Point(a, b)  // call base-class constructor
{ radius = r > 0 ? r : 0; }

// Set radius
void Circle::setRadius(float r) { radius = r > 0 ? r : 0; }

// Get radius
float Circle::getRadius() const { return radius; }

// Calculate area of a Circle
float Circle::area() const { return 3.14159 * radius * radius; }

// Output a circle in the form: Center=[x, y]; Radius=#.##
ostream &operator<<(ostream &output, const Circle &c)
{
   output << '[' << c.getX() << ", " << c.getY()
          << "]; Radius = " << setiosflags(ios::showpoint)
          << setprecision(2) << c.radius;

   return output;   // enables concatenated calls
}
```

Fig. 20.2 Definiciones de función miembro para la clase **Circle** (parte 5 de 9).

Otras funciones miembro incluyen una función *set,* para asignar un nuevo **radius** a un **Circle** y una función *get,* para devolver **radius** de un **Circle**.

La clase **Cylinder** (figura 20.2 partes 6 y 7) es derivada de **Circle** con herencia pública. Un **Cylinder** tiene área y volumen, por lo que las funciones **area** y **volume** ambas se redefinen en esta clase. La función **printShapeName** es una puesta en práctica de una función virtual, que en la clase base fue definida como virtual pura. Si aquí no se redefine esta función, sería heredada la versión **Circle** de la función. Otras funciones miembro incluyen una función *set,* para asignar un nuevo **radius** a un **Circle** y una función *get,* para devolver **radius** de un **Circle**.

El programa manejador (figura 20.2 partes 8 y 9) empieza produciendo el objeto **Point** de la clase **point**, el objeto **circle** de la clase **Circle** y el objeto **cylinder** de clase **Cylinder**. Se invoca la función **printShapeName** para cada uno de los objetos y cada objeto es extraído con su operador de inserción de flujo homónimo, a fin de ilustrar que los objetos han sido inicializados en forma correcta. A continuación, se declara el apuntador **ptr** del tipo **Shape ***. Se utiliza este apuntador para señalar a cada uno de los objetos de la clase derivada. Primero se asigna la dirección de **point** a **ptr** y se efectúan las siguientes llamadas

```
ptr->printShapeName()
ptr->area()
ptr->volume()
```



```
// CYLINDER.H
// Definition of class Cylinder
#ifndef CYLINDER_H
#define CYLINDER_H
#include "circle1.h"

class Cylinder : public Circle {
   friend ostream &operator<<(ostream &, const Cylinder &);
public:
   // default constructor
   Cylinder(float h = 0.0, float r = 0.0,
            float x = 0.0, float y = 0.0);

   void setHeight(float);
   virtual float area() const;
   virtual float volume() const;
   virtual void printShapeName() const { cout << "Cylinder: "; }
private:
   float height;   // height of Cylinder
};

#endif
```

Fig. 20.2 Definición de clase **Cylinder** (parte 6 de 9).

para invocar estas funciones para el objeto al cual apunta **ptr**. La salida ilustra que las funciones han sido de forma correcta invocadas para **point**— **"Point: "** es extraído y el área y el volumen son ambos **0.00**. A continuación, se asigna a **ptr** la dirección del objeto **circle** y se invocan las mismas funciones. La salida ilustra que las funciones han sido correctamente invocadas para el objeto **circle** se extrae— **"Circle: "** el área de **circle** se calcula, y el volumen es **0.00**. Por último, **ptr** se asigna a la dirección del objeto **cylinder** y se invocan las mismas funciones. La salida ilustra que las funciones han sido invocadas correctamente para el objeto **cylinder**, se extrae **cylinder**— **"Cylinder: "** se calcula el área de **cylinder** y se calcula el volumen de **cylinder**. Todas las llamadas de función **printShapeName**, **area** y **volume** se resuelven en tiempo de ejecución mediante ligaduras dinámicas.

Resumen

- Con las funciones virtuales y el polimorfismo, se hace posible diseñar y poner en práctica sistemas que son más extensibles. Se pueden escribir programas para procesar objetos de tipos que pudieran no existir cuando el programa está aún en desarrollo.
- Las funciones virtuales y la programación polimórfica pueden eliminar la necesidad de la lógica **switch**. El programador puede utilizar el mecanismo de la funciones virtuales para llevar a cabo de manera automática la lógica equivalente y, por lo tanto, evitando los tipos de errores típicamente asociados con la lógica **switch**. Un código cliente que tome decisiones sobre tipos de objetos y representaciones indicará un diseño de clase pobre.
- Se declara una función virtual haciendo preceder en la clase base el prototipo de función por la palabra reservada **virtual**.

```
// CYLINDR1.CPP
// Member function definitions for class Cylinder
#include <iostream.h>
#include <iomanip.h>
#include "cylindr1.h"

Cylinder::Cylinder(float h, float r, float x, float y)
   : Circle(r, x, y)  // call base-class constructor
{ height = h > 0 ? h : 0; }

void Cylinder::setHeight(float h)
{ height = h > 0 ? h : 0; }

float Cylinder::area() const
{
   // surface area of Cylinder
   return 2 * Circle::area() +
          2 * 3.14159 * Circle::getRadius() * height;
}

float Cylinder::volume() const
{
   float r = Circle::getRadius();
   return 3.14159 * r * r * height;
}

ostream &operator<<(ostream &output, const Cylinder& c)
{
   output << '[' << c.getX() << ", " << c.getY()
          << "]; Radius = " << setiosflags(ios::showpoint)
          << setprecision(2) << c.getRadius()
          << "; Height = " << c.height;

   return output;   // enables concatenated calls
}
```

Fig. 20.2 Definiciones de función miembro para la clase `Cylinder` (parte 7 de 9).

- Si así lo desean, las clases derivadas pueden proveer sus propias puestas en práctica de una función virtual de clase base, pero si no es así, entonces se utilizará la puesta en práctica de la clase base.
- Si se llama una función virtual haciendo referencia a un objeto específico por su nombre y utilizando el operador de selección miembro punto, dicha referencia se resuelve en tiempo de compilación (esto se conoce como *ligadura estática*) y la función virtual llamada es aquella definida (o heredada) por la clase de dicho objeto particular.
- Existen muchas situaciones, sin embargo, en las cuales resulta útil definir clases para las cuales el programador no tiene jamás intención de producir objeto alguno. Estas clases se conocen como clases abstractas. Dado que en situaciones de herencia éstas son utilizadas como clases base, nos referiremos por lo general a ellas como clases base abstractas. No se pueden producir objetos de una clase base abstracta.


```
// FIG20_2.CPP
// Driver for point, circle, cylinder hierarchy
#include <iostream.h>
#include <iomanip.h>
#include "shape.h"
#include "point1.h"
#include "circle1.h"
#include "cylindr1.h"

main()
{
   // create some shape objects
   Point point(7, 11);
   Circle circle(3.5, 22, 8);
   Cylinder cylinder(10, 3.3, 10, 10);

   point.printShapeName();      // static binding
   cout << point << endl;

   circle.printShapeName();     // static binding
   cout << circle << endl;

   cylinder.printShapeName();   // static binding
   cout << cylinder << "\n\n";

   cout << setiosflags(ios::showpoint) << setprecision(2);
   Shape *ptr;                  // create base-class pointer

   // aim base-class pointer at derived-class Point object
   ptr = &point;
   ptr->printShapeName();       // dynamic binding
   cout << "x = " << point.getX() << "; y = " << point.getY()
        << "\nArea = " << ptr->area()
        << "\nVolume = " << ptr->volume() << "\n\n";

   // aim base-class pointer at derived-class Circle object
   ptr = &circle;
   ptr->printShapeName();       // dynamic binding
   cout << "x = " << circle.getX() << "; y = " << circle.getY()
        << "\nArea = " << ptr->area()
        << "\nVolume = " << ptr->volume() << "\n\n";

   // aim base-class pointer at derived-class Cylinder object
   ptr = &cylinder;
   ptr->printShapeName();       // dynamic binding
   cout << "x = " << cylinder.getX()
        << "; y = " << cylinder.getY()
        << "\nArea = " << ptr->area()
        << "\nVolume = " << ptr->volume() << endl;

   return 0;
}
```

Fig. 20.2 Manejador para la jerarquía punto, círculo y cilindro (parte 8 de 9).

```
Point: [7, 11]
Cicle: [22, 8]; Radius = 3.50
Cylinder: [10, 10]; Radius = 3.30; Height = 10.00

Point: x = 7.00; y = 11.00
Area = 0.00
Volume = 0.00

Cicle: x = 22.00; y = 8.00
Area = 38.48
Volume = 0.00

Cylinder: x = 10.00; y = 10.00
Area = 275.77
Volume = 342.12
```

Fig. 20.2 Manejador para la jerarquía punto, círculo y cilindro (parte 9 de 9).

- Las clases a partir de las cuales se pueden producir objetos se conocen como clases concretas.
- Una clase con funciones virtuales se puede convertir en abstracta al declarar como pura una o más de sus funciones virtuales. Una función virtual pura es una que tenga un inicializador de = 0 dentro de su declaración.
- Si una clase es derivada de una clase con una función virtual pura sin proporcionar en la clase derivada una definición para dicha función virtual pura, entonces dicha función virtual se conserva como pura en la clase derivada. Y en consecuencia, la clase derivada también es una clase abstracta.
- C++ permite el polimorfismo —la habilidad de los objetos de clases distintas, emparentados mediante la herencia, de responder en forma diferente a la misma llamada de función miembro.
- El polimorfismo se pone en práctica vía funciones virtuales.
- Cuando se hace una solicitud mediante un apuntador de clase base para el uso de una función virtual, C++ escoge la función redefinida correcta, en la clase derivada apropiada asociada con dicho objeto.
- Mediante el uso de las funciones virtuales y del polimorfismo, una llamada de función miembro puede causar distintas acciones, dependiendo del tipo de objeto que reciba la llamada.
- Aunque no podemos producir objetos de clases base abstractas, *podemos* declarar apuntadores a clases base abstractas. Estos apuntadores pueden entonces utilizarse para permitir manipulaciones polimórficas de objetos de clase derivada, cuando dichos objetos sean producidos a partir de clases concretas.
- Continuamente se están añadiendo nuevos tipos de clase a los sistemas. Se les da sitio a la nuevas clases mediante la ligadura dinámica (también llamada ligadura tardía). No es necesario conocer en tiempo de compilación el tipo de un objeto, para que una llamada de función virtual sea compilada. La llamada de función virtual se hace coincidir en tiempo de ejecución con la función miembro del objeto llamado.

- La ligadura dinámica permite que fabricantes independientes de software (ISV) distribuyan software sin tener que revelar secretos propietarios. Las distribuciones de software pueden estar sólo constituidas de archivos de cabecera y archivos objeto. No es necesario revelar código fuente. A partir de las clases proveídas por los ISV, los desarrolladores de software podrán entonces utilizar la herencia para derivar nuevas clases. Aquel software que funcione con las clases proveídas por los ISV, continuará funcionando con las clases derivadas y utilizará (vía la ligadura dinámica) las funciones virtuales redefinidas proporcionadas en esas clases.
- La ligadura dinámica requiere que, en tiempo de ejecución, se encamine la llamada a una función miembro virtual, a la versión de función virtual apropiada para dicha clase. Se pone en operación una tabla de función virtual, conocida como la vtable, como un arreglo que contiene apuntadores de función. Cada clase, que contenga funciones virtuales, tiene una vtable. Para cada función virtual dentro de la clase, la vtable tiene una entrada que incluye un apuntador de función a la versión de la función virtual a utilizar para un objeto de dicha clase. La función virtual a utilizar para una clase particular pudiera ser la función definida en dicha clase, o pudiera ser una función heredada, ya sea directa o indirecta de una clase base más alta, dentro de la jerarquía.
- Cuando una clase base proporciona una función miembro y la declara **`virtual`**, las clases derivadas pueden redefinir la función virtual, pero no están obligadas a hacerlo. Por lo tanto, una clase derivada puede utilizar la versión de clase base de un función miembro virtual y esto quedaría indicado en la vtable.
- Cada objeto de una clase con funciones virtuales contiene un apuntador a la vtable para dicha clase. Este apuntador no es accesible para el programador. El apuntador de función apropiado para la vtable es obtenido y desreferenciado para completar la llamada en tiempo de ejecución. Esta búsqueda y desreferenciación de apuntadores en la vtable requiere de una sobrecarga nominal en tiempo de ejecución, usualmente menor que la del mejor código cliente posible.
- Declare el destructor de clase base virtual, si la clase contiene funciones virtuales. Esto hará virtuales a todos los destructores de clase derivada, aunque no tengan el mismo nombre que el destructor de clase base. Si un objeto en la jerarquía es explícitamente destruido mediante la aplicación del operador **`delete`** a un apuntador de clase base a un objeto de clase derivada, el destructor de la clase apropiada será llamado.

Terminología

clase base abstracta
clase abstracta
función de clase base **`virtual`**
jerarquía de clase
convertir objeto de clase derivada a objeto de clase base
convertir apuntador de clase derivada a apuntadoy de clase base
clase derivada
constructor de clase derivada
clase base directa
ligadura dinámica
ligadura temprana
eliminación de enunciados **`switch`**
conversión explícita de apuntadores
extensibilidad
conversión implícita de apuntador
clase base indirecta
herencia
ligadura tardía
programación orientada a objetos (OOP)
apuntador a una clase base
apuntador a una clase derivada
apuntador a una clase abstracta

polimorfismo
función virtual pura (`=0`)
función virtual redefinida
referencia a una clase base
referencia a una clase derivada
referencia a una clase abstracta
reutilización del software
ligadura estática
lógica `switch`
destructor virtual
función virtual
palabra reservada `virtual`

Errores comunes de programación

20.1 Resultará un error de sintaxis redefinir una clase derivada como función virtual de clase base, sin asegurarse que la función derivada tenga el mismo tipo de regreso y signatura que la versión de clase base.

20.2 Es un error de sintaxis intentar producir un objeto a partir de una clase abstracta (es decir, de una clase que contenga una o más funciones virtuales puras).

20.3 Declarar un constructor como función virtual. Los constructores no pueden ser virtuales.

Prácticas sanas de programación

20.1 A pesar que ciertas funciones en forma implícita son virtuales, en razón de una declaración hecha anteriormente en la jerarquía de clase, algunos programadores prefieren declarar estas funciones virtuales en forma explícita en cada uno de los niveles de la jerarquía, a fin de promover claridad en el programa.

20.2 Si una clase tiene funciones virtuales, incluya un destructor virtual, inclusive si para dicha clase no se requiere uno. Las clases que se deriven de esta clase pudieran contener destructores que deberán ser llamados en forma correcta.

Sugerencias de rendimiento

20.1 El polimorfismo, tal y como se pone en práctica mediante funciones virtuales y ligaduras dinámicas, resulta eficiente. Los programadores pueden utilizar estas capacidades con un impacto sólo nominal sobre el rendimiento del sistema.

20.2 Las funciones virtuales y las ligaduras dinámicas permiten la programación polimórfica, en contraste con la programación lógica `switch`. Los compiladores optimizantes de C++ por lo regular generan código que se ejecuta con una eficiencia por lo menos igual a la de la lógica basada en `switch` codificada manualmente.

Observaciones de ingeniería de software

20.1 Una consecuencia interesante del uso de funciones virtuales y polimorfismo es que los programas adquieren una apariencia simplificada. Contienen menos lógica de bifurcación, prefiriendo un código secuencial más sencillo. Esta simplificación facilita probar, depurar y mantener el programa.

20.2 Una vez declarada una función como virtual, se conserva como virtual a lo largo de toda la jerarquía de herencia, a partir de dicho punto.

20.3 Cuando una clase derivada decide no definir una función virtual, la clase derivada heredará la función virtual de la clase base inmediata.

20.4 Las funciones virtuales redefinidas deben tener el mismo tipo de regreso y la misma signatura que la función virtual base.

20.5 Si una clase se deriva de una clase con una función virtual pura, y para dicha función virtual pura no se ha dado definición en la clase derivada, entonces la función virtual también se conserva pura en la clase derivada. Por consecuencia, también la clase derivada será una clase abstracta.

20.6 Mediante las funciones virtuales y el polimorfismo, el programador puede ocuparse de generalidades y dejar que en tiempo de ejecución, el entorno se preocupe de lo específico. El programador puede dirigir una amplia variedad de objetos haciendo que se comporten de formas apropiadas a dichos objetos incluso sin conocer los tipos de los mismos.

20.7 El polimorfismo fomenta la extensibilidad: software escrito para invocar comportamiento polimórfico se escribe en forma independiente del tipo de los objetos a los cuales los mensajes son enviados. Por lo tanto, nuevos tipos de objetos, que pudieran responder a mensajes existentes, pueden ser añadidos en dicho sistema sin modificar el sistema base. A excepción de la parte de código cliente que produce nuevos objetos, los programas no necesitan ser recompilados.

20.8 Una clase abstracta define una interfaz para los distintos miembros de una jerarquía de clase. La clase abstracta contiene funciones virtuales puras, que serán definidas en las clases derivadas. Mediante el polimorfismo todas las funciones en la jerarquía pueden utilizar esta misma interfaz.

Ejercicios de autoevaluación

20.1 Llene cada uno de los siguientes espacios vacíos:

a) Utilizando la herencia y el polimorfismo se ayuda a eliminar la lógica _____.
b) Una función virtual pura se define colocando una _____ al fin de su prototipo en la definición de clase.
c) Si una clase contiene una o más funciones virtuales puras, es una _____.
d) Una llamada de función resuelta en tiempo de compilación se conoce como ligadura _____.
e) Una llamada de función resuelta en tiempo de ejecución se conoce como ligadura _____

Respuestas a los ejercicios de autoevaluación

20.1 a) `switch`. b) `= 0`. c) clase base abstracta. d) estática. e) dinámica.

Ejercicios

20.2 ¿Qué son las funciones virtuales? Describa alguna circunstancia en la cual las funciones virtuales serían apropiadas.

20.3 Dado que los constructores no pueden ser virtuales, describa algún procedimiento mediante el cual usted podría conseguir un efecto similar.

20.4 En el ejercicio 19.5 usted desarrolló una jerarquía de clase `Shape` y definió las clases dentro de la jerarquía. Modifique dicha jerarquía, de tal forma que la clase `Shape` sea una clase base abstracta que contenga la interfaz a la jerarquía. Derive `TwoDimensionalShape` y `ThreeDimensionalShape` de la clase `Shape` —estas clases también deberán ser abstractas. Utilice una función virtual `print` para extraer el tipo y dimensiones de cada clase. También incluya las funciones virtuales `area` y `volume` de tal forma que estos cálculos puedan ser ejecutados para objetos de cada clase concreta dentro de la jerarquía. Escriba un programa manejador que pruebe la jerarquía de clase `Shape`.

20.5 Utilizando la clase de plantilla `List` que se desarrolló en el ejercicio 17.10, cree una lista enlazada de apuntadores a objetos `Shape` (es decir, apuntadores a cualquier forma dentro de la jerarquía). Proporcione un operador de inserción de flujo homónimo para la clase `Shape`, que sólo llame la función virtual pura `print`. Utilice la capacidad de impresión de la lista enlazada para recorrer la lista y extraer cada objeto.

21

Flujo de entrada/salida de C++

Objetivos

- Comprender cómo utilizar el flujo de entrada/salida orientada a objetos de C++.
- Ser capaz de darle formato a las entradas y a las salidas.
- Comprender la jerarquía de clase de flujo de entradas/salidas.
- Comprender cómo introducir/extraer objetos de tipos definidos por usuario.
- Ser capaz de crear manipuladores de flujo definidos por usuario.
- Ser capaz de determinar el éxito o el fracaso de operaciones de entrada/salida.
- Ser capaz de ligar flujos de salida con flujos de entrada.

Al parecer, el estar consciente... no está dividido en pedazos
...Las metáforas mediante las cuales quedaría descrito
con más naturalidad son "río" o "flujo".
William James

Todas las noticias que merecen imprimirse.
Adolph S. Ochs

Sinopsis

21.1 Introducción

Las bibliotecas estándar de C++ proporcionan un conjunto extenso de capacidades de entrada/salida. Este capítulo analiza un conjunto de capacidades suficientes para llevar a cabo las operaciones de entradas y salidas más comunes y da un resumen general de las capacidades restantes.

Muchas de las características de entrada/salida descritas aquí son orientadas a objetos. El lector deberá encontrar de interés ver cómo dichas capacidades se ponen en práctica. Este estilo de entradas/salidas también hace mucho uso de características C++ como referencias, homonimia de funciones y homonimia de operadores.

Como veremos, C++ utiliza *entradas/salidas de tipo seguro*. Cada operación de entrada/salida se ejecuta de una manera que es sensible al tipo de los datos. Si una función de entrada/salida ha sido definida correctamente para manejar un tipo particular de datos, entonces dicha función es llamada en forma automática para manejar dicho tipo de datos. Si no existe coincidencia entre el tipo de los datos reales y una función para el manejo de dichos tipos de datos, se establece una indicación de error de compilación. Por lo tanto, no podrán introducirse datos equivocados dentro del sistema (como lo pueden hacer en C —una falla de C, que permite errores bastante sutiles y a menudo, extraños).

Los usuarios pueden especificar entradas/salidas de tipos definidos por usuario, así como de tipos estándar. Esta *extensibilidad* es una de las características más valiosas de C++.

Práctica sana de programación 21.1

En programas C++, utilice exclusivamente la forma de entradas/salidas de C++, a pesar del hecho que para los programadores C++ esté disponible el estilo de entradas/salidas de C.

Observación de ingeniería de software 21.1

El estilo de entradas/salidas de C++ es de tipo seguro.

Observación de ingeniería de software 21.2

C++ permite un tratamiento común de entradas/salidas de tipos predefinidos y de tipos definidos por usuario. Este tipo de estado común facilita el desarrollo de software en general y de la reutilización de software en particular.

21.2 Flujos

La entrada/salida en C++ ocurre en *flujos* de bytes. Un flujo es solo una secuencia de bytes. En operaciones de entrada, los bytes fluyen de un dispositivo (es decir, un teclado, una unidad de disco o una conexión de red) a la memoria principal. En operaciones de salida, los bytes fluyen de la memoria principal a un dispositivo (es decir, una pantalla de exhibición, una impresora, una unidad de disco o una conexión de red).

La aplicación asocia el significado con bytes. Los bytes pueden representar caracteres ASCII, datos en bruto de formato interno, imágenes gráficas, voz digitalizada, video digitalizado o cualquier otro tipo de información que pudiera una aplicación requerir.

El trabajo de los mecanismos de entrada y salida del sistema es mover los bytes de los dispositivos a la memoria y viceversa de forma consistente y confiable. Dichas transferencias a menudo, involucran movimientos mecánicos, como son el giro de un disco o de una cinta, o el tecleo en un teclado. El tiempo que, por lo general, toman esas transferencias es enorme en comparación con el tiempo que toma el procesador para la manipulación interna de los datos. Por lo tanto, para asegurar un rendimiento máximo, las operaciones de entrada/salida requieren una

cuidadosa planeación y sintonización. Temas como éste se discuten en detalle en los libros de texto de los sistemas operativos (De90).

C++ proporciona tanto capacidades de entradas y salidas de "bajo nivel" como de "alto nivel". Las capacidades de entrada/salida de bajo nivel (es decir, entradas/salidas sin formato) especifican en forma típica que cierto número de bytes sólo deben ser transferidos del dispositivo a la memoria o de la memoria al dispositivo. En este tipo de transferencias, el byte individual es el elemento de interés. Estas capacidades de bajo nivel, de hecho proporcionan transferencias de alta velocidad y volumen, pero estas capacidades no son particularmente convenientes para las personas.

Las personas prefieren una visión de mayor nivel de las entradas/salidas, es decir, *entradas/salidas con formato*, en el cual los bytes están agrupados en unidades significativas como enteros, números de punto flotante, caracteres, cadenas y tipos definidos por usuario. Estas capacidades, orientadas al tipo, son satisfactorias para la mayor parte de las entradas/salidas que no correspondan al proceso de archivos de alto volumen.

Sugerencia de rendimiento 21.1

Utilice entradas/salidas sin formato para un rendimiento máximo en procesos de alto volumen de archivos.

21.2.1 Archivos de cabecera de biblioteca iostream

La biblioteca **iostream** de C++ proporciona cientos de capacidades de entrada/salida. Varios archivos de cabecera contienen porciones de la interfaz de la biblioteca.

La mayor parte de los programas de C++ deben incluir el archivo de cabecera **iostream.h**, que contiene información básica requerida para todas las operaciones de flujo de entrada/salida. El archivo de cabecera **iostream.h** contiene los objetos **cin**, **cout**, **cerr** y **clog** los cuales, como veremos, corresponden al flujo de entrada estándar, flujo de salida estándar y flujo de error estándar. Se proporcionan tanto capacidades de entrada y salida con, como sin formato.

El encabezado **iomanip.h** contiene información útil para llevar a cabo entradas/salidas con formato, mediante *manipuladores de flujo parametrizados*.

El encabezado **fstream.h** contiene información de importancia para llevar a cabo las operaciones de proceso de archivos controlados por el usuario.

El encabezado **strstream.h** contiene información de importancia para llevar a cabo *formato en memoria* (también conocido como *formato en núcleo*). Esto se parece al proceso de archivos, pero en las operaciones "entrada y salida" se ejecutan hacia y desde arreglos de caracteres, en vez de hacia y desde archivos.

El encabezado **stdiostream.h** contiene información de importancia para aquellos programas que combinen los estilos de C y de C++ para entradas/salidas. Los programas nuevos deberían evitar entradas/salidas de estilo C, pero aquellas personas que deben modificar programas en C existentes, o que transforman programas de C a C++, pueden encontrar útiles estas capacidades.

Cada puesta en práctica de C++ por lo general, contiene otras bibliotecas relacionadas con entradas/salidas que proporcionan capacidades específicas a cada sistema, como el control de dispositivos de uso especial para entradas/salidas de audio y de video.

21.2.2 Clases y objetos de flujo de entrada/salida

La biblioteca **iostream** contiene muchas clases para el manejo una gran variedad de operaciones de entradas/salidas. La clase **istream** apoya operaciones de entrada de flujo. La clase **ostream** apoya operaciones de salida de flujo. La clase **iostream** apoya tanto operaciones de entrada como de salida de flujo.

Las clases **istream** y **ostream** cada una de ellas han sido derivadas mediante herencia simple de la clase base **ios**. La clase **iostream** es una derivación mediante herencia múltiple, tanto de la clase **istream** como de la clase **ostream**. Estas relaciones de herencia se resumen en la figura 21.1.

La homonimia de operadores permite una forma de notación conveniente para llevar a cabo entradas/salidas. Se hace la homonimia del operador de desplazamiento a la izquierda (<<) para designar salida de flujo y se conoce como *operador de inserción de flujo*. Se hace la homonimia del operador de desplazamiento a la derecha (>>) a fin de designar entrada de flujo y se conoce como *operador de extracción de flujo*. Estos operadores son utilizadas con los objetos de flujo estándar **cin**, **cout**, **cerr** y **clog**, y con objetos de flujo definidos por usuario.

cin es un objeto de la clase **istream** y se dice que está "ligado a" (o conectado con) el dispositivo de entrada estándar, que por lo regular es el teclado. El operador de extracción de flujo, tal y como se usa en el siguiente enunciado, hace que se introduzca un valor para la variable entera **grade** desde **cin** a la memoria

```
cin >> grade;
```

Note que la operación de extracción de flujo es "lo suficiente inteligente" para "saber" cuál es el tipo de los datos. Suponiendo que **grade** ha sido declarado en forma correcta, no es necesario especificar ningún tipo adicional de información para utilizar el operador de extracción de flujo (como sería el caso, incidentalmente, en entradas/salidas en estilo C).

cout es un objeto de la clase **ostream** y se dice que está "ligado con" el dispositivo de salida estándar, que por lo regular es la pantalla de exhibición. El operador de inserción de flujo, tal y como se usa en el enunciado siguiente, hace que se extraiga el valor de la variable entera **grade** de la memoria hacia el dispositivo de salida estándar.

```
cout << grade;
```

Note que el operador de inserción de flujo es "lo suficiente inteligente "para saber" el tipo de **grade** (suponiendo que éste haya sido declarado en forma correcta), por lo que no es necesario ningún tipo adicional de información para uso con el operador de inserción de flujo.

cerr es un objeto de la clase **ostream** y se dice que "está ligado con" el dispositivo de error estándar. Las salidas al objeto **cerr** no pasan por almacenamientos temporales o búfers. Esto significa que cada inserción a **cerr** hará que su salida aparezca de inmediato; esto resulta apropiado para una notificación instantánea de algún problema al usuario.

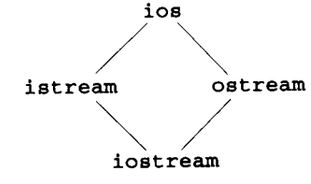


Fig. 21.1 Porción de la jerarquía de clase de flujo de entradas/salidas.

clog es un objeto de la clase **ostream** y también se dice que "está ligado con" el dispositivo de error estándar. Las salidas a **clog** sí pasan por almacenamiento temporal o búfer.

El procesamiento de archivo en C++ utiliza las clases **ifstream** para llevar a cabo operaciones de entrada de archivo, utiliza a **ofstream** para operaciones de salida de archivo, y **fstream** para operaciones de entrada/salida de archivo. La clase **ifstream** hereda de **istream**, la clase **ofstream** hereda de **ostream**, y la clase **fstream** hereda de **iostream**. Las relaciones de herencia de las clases relacionadas con entradas/salidas se resumen en la figura 21.2. En la jerarquía completa de clase de flujo de entradas/salidas en la mayor parte de las instalaciones existen muchas más clases incluidas, pero las clases que aquí se muestran proporcionan la gran mayoría de las capacidades que necesitarán la mayor parte de los programadores. Vea las referencias de biblioteca de clase correspondientes a su sistema C++ para más información de procesamiento de archivos.

21.3 Salida de flujo

La clase **ostream** de C++ da la capacidad para llevar a cabo salida con y sin formato. Las capacidades de salida incluyen: la salida de tipos de datos estándar mediante el operador de inserción de flujo; la salida de caracteres utilizando la función miembro **put**; salida sin formato mediante la función miembro **write** (sección 21.5); salida de enteros en formatos decimal, octal y hexadecimal (sección 21.6.1); salida de valores en punto flotante con distintas precisiones (sección 21.6.2), con puntos decimales obligados (21.7.2), en notación científica, y en notación fija (sección 21.7.6); salida de datos justificados en campos de anchos de campo designados (sección 21.7.3); salida de datos en campos rellenos de caracteres especiales (sección 21.7.4); y salida de letras mayúsculas en notación científica y hexadecimal (sección 21.7.7).

21.3.1 Operador de inserción de flujo

La salida de flujo puede ser ejecutada mediante el operador de inserción de flujo, es decir, el operador homónimo <<. Se hace la homonimia del operador << para extraer elementos de datos de tipos incorporados, para extraer cadenas y para extraer valores de apuntadores. En la sección 21.9, veremos como hacer la homonimia de << para extraer elementos de datos de tipos definidos por usuario. La figura 21.3 demuestra la salida de una cadena.

El ejemplo anterior utiliza un único enunciado de inserción de flujo a fin de mostrar la cadena de salida. Se pueden utilizar varios enunciados de inserción, como en la figura 21.4. Cuando este programa sea ejecutado, producirá la misma salida que el programa anterior.

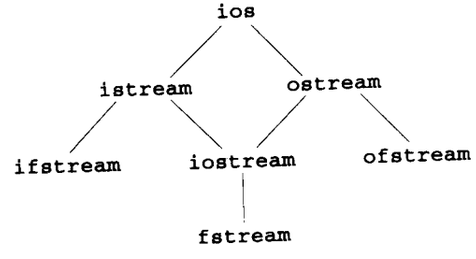


Fig. 21.2 Porción de la jerarquía de clase de flujo de entradas/salidas mostrando las clases de procesamiento de archivo clave.


```
// fig21_03.cpp
// Outputting a string using stream insertion.

#include <iostream.h>

main()
{
   cout << "Welcome to C++!\n";

   return 0;
}
```

```
Welcome to C++!
```

Fig. 21.3 Cómo extraer una cadena utilizando la inserción de flujo.

```
// fig21_04.cpp
// Outputting a string using two stream insertions.

#include <iostream.h>

main()
{
   cout << "Welcome to ";
   cout << "C++!\n";

   return 0;
}
```

```
Welcome to C++!
```

Fig. 21.4 Cómo extraer una cadena utilizando dos inserciones de flujo.

El efecto de la secuencia de escape **\n** (nueva línea) también se puede conseguir mediante el *manipulador de flujo* **endl** (terminar línea) como en la figura 21.5. El manipulador de flujo **endl** emite un carácter de nueva línea y, además, vacía el búfer de salida (es decir, hace que el búfer de salida sea extraído de inmediato, aunque todavía no esté lleno). El búfer de salida también puede ser vaciado, sólo mediante

```
cout << flush;
```

En la sección 21.6 se analizan en detalle los manipuladores de flujo.

Las expresiones pueden ser extraídas tal y como se muestra en la figura 21.6.

Práctica sana de programación 21.2

Al extraer expresiones, colóquelas entre paréntesis, para evitar problemas de precedencia de operadores entre los operadores de la expresión y el operador <<.

```
// fig21_05.cpp
// Using the endl stream manipulator.
#include <iostream.h>

main()
{
   cout << "Welcome to ";
   cout << "C++!";
   cout << endl;    // end line stream manipulator

   return 0;
}
```

```
Welcome to C++!
```

Fig. 21.5 Cómo utilizar el manipulador de flujo **endl**.

```
// fig21_06.cpp
// Outputting expression values.
#include <iostream.h>

main()
{
   cout << "47 plus 53 is ";
   cout << 47 + 53;       // expression
   cout << endl;

   return 0;
}
```

```
47 plus 53 is 100
```

Fig. 21.6 Cómo extraer valores de expresiones.

21.3.2 Cómo concatenar operadores de inserción/extracción de flujo

Los operadores homónimos << y >> cada uno de ellos puede ser utilizado en forma concatenada, tal y como se muestra en la figura 21.7.

Las múltiples inserciones de flujo de la figura 21.7 se ejecutan como si hubieran sido escritas:

```
(((cout << "47 plus 53 is ") << 47 + 53) << endl);
```

Esto está permitido porque el operador homónimo << devuelve una referencia a su objeto operando izquierdo, es decir, **cout**. Por lo tanto, la expresión entre paréntesis más a la izquierda

```
(cout << "47 plus 53 is ")
```



```
// fig21_07.cpp
// Concatenating the overloaded << operator.
#include <iostream.h>

main()
{
   cout << "47 plus 53 is " << 47 + 53 << endl;

   return 0;
}
```

```
47 plus 53 is 100
```

Fig. 21.7 Cómo concatenar el operador homónimo <<.

extrae la cadena especificada de caracteres y devuelve una referencia a **cout**. Esto permite que se evalúe la expresión entre paréntesis intermedia como

```
(cout << 47 + 53)
```

que extrae el valor entero **100** y devuelve una referencia a **cout**. A continuación se evalúa la expresión entre paréntesis más a la derecha como

```
cout << endl
```

que extrae una nueva línea, vacía a **cout** y devuelve una referencia a **cout**. Este último regreso no es utilizado.

21.3.3 Salida de variables char*

En entradas/salidas de estilo C, es necesario proporcionar información de tipo. C++ determina los tipos de datos en forma automática —lo que resulta una agradable mejoría respecto a C. Pero algunas veces esto "resulta un estorbo". Por ejemplo, sabemos que una cadena de caracteres es del tipo **char***. Suponga que deseamos imprimir el valor de dicho apuntador, es decir, la dirección en memoria del primer carácter de dicha cadena. Pero se ha hecho la homonimia del operador << para imprimir datos del tipo **char*** como si fuera una cadena terminada por **null**. La solución es hacer una conversión explícita (cast) del apuntador a **void*** (esto debe ser hecho a cualquier variable de apuntador a la cual el programador desee extraer como una dirección). En la figura 21.8 se demuestra la impresión de una variable **char*** tanto en formato de cadena como de dirección. Note que la dirección se imprime como un número hexadecimal (de base 16). En C++ los números hexadecimales empiezan con **0x**, o **0X**. En las secciones 21.6.1, 21.7.4, 21.7.5 y 21.7.7 ampliamos la información respecto a cómo controlar las bases de los números.

21.3.4 Extracción de caracteres mediante la función miembro put; cómo concatenar put

La función miembro **put** extrae un carácter como en

```
cout.put('A');
```

```
// fig21_08.cpp
// Printing the address stored in a char* variable

#include <iostream.h>

main()
{
   char *string = "test";

   cout << "Value of string is: " << string
        << "\nValue of (void *)string is: "
        << (void *)string << endl;

   return 0;
}
```

```
Value of string is: test
Value of (void *)string is: 0x00aa
```

Fig. 21.8 Cómo imprimir la dirección almacenada en una variable `char *`.

que exhibe **A** en pantalla. Las llamadas a `put` pueden ser concatenadas como en

```
cout.put('A').put('\n');
```

que extrae **A**, seguida por un carácter de nueva línea. La función `put` también puede ser llamada con una expresión de valor ASCII, como en `cout.put(65)`, que también extrae a **A**.

21.4 Entrada de flujo

Consideremos ahora entradas de flujo. Esto se puede ejecutar con el operador de extracción de flujo, es decir, con el operador homónimo >>. Este operador por lo general, pasa por alto los *caracteres de espacio en blanco* (como espacios en blanco, tabuladores y nuevas líneas) existentes en el flujo de entrada. Más tarde veremos cómo modificar este comportamiento. El operador de extracción de flujo devuelve cero (falso) cuando dentro de un flujo encuentra el final de archivo. Cada flujo contiene un conjunto de *bits de estado* que se utilizan para controlar el estado del flujo (es decir, estados de formato, de establecimiento de errores, etcétera). La extracción de flujo hace que para entradas de tipo incorrecto se active `failbit` del flujo, y si la operación falla hace que se active `badbit` del flujo. Veremos pronto cómo probar estos bits después de una operación de entrada/salida. En las sección 21.7 y 21.8 se analizan los bits de estado de flujo en detalle.

21.4.1 Operador de extracción de flujo

Para leer dos enteros, utilice el objeto `cin` y el operador de extracción de flujo >> homónimo, tal y como aparece en la figura 21.9.

La relativamente alta precedencia de los operadores >> y << pueden causar problemas. Por ejemplo, el programa de la figura 21.10 no se compilará en forma correcta sin los paréntesis que encierran la expresión condicional. El lector deberá de verificar lo anterior.


```
// fig21_09.cpp
// Calculating the sum of two integers
// input from the keyboard with cin
// and the stream extraction operator.
#include <iostream.h>

main()
{
   int x, y;

   cout << "Enter two integers: ";
   cin >> x >> y;
   cout << "Sum of " << x << " and " << y << " is: "
        << x + y << endl;

   return 0;
}
```

```
Enter two integers: 30 92
Sum of 30 and 92 is: 122
```

Fig. 21.9 Cómo calcular la suma de dos enteros introducidos desde el teclado mediante `cin` y el operador de extracción de flujo.

```
// fig21_10.cpp
// Revealing a precedence problem.
// Need parentheses around the conditional expression.
#include <iostream.h>

main()
{
   int x, y;

   cout << "Enter two integers: ";
   cin >> x >> y;
   cout << x << (x == y ? " is" : " is not")
        << " equal to " << y << endl;

   return 0;
}
```

```
Enter two integers: 7 5
7 is not equal to 5
```

```
Enter two integers: 8 8
8 is equal to 8
```

Fig. 21.10 Cómo evitar un problema de precedencia entre el operador de inserción de flujo y el operador condicional.

Error común de programación 21.1

Intentar leer de un ostream (o de cualquier otro flujo de solo salida).

Error común de programación 21.2

Intentar escribir a un istream (o a cualquier otro flujo de sólo entrada).

Error común de programación 21.3

Omitir paréntesis para obligar a una correcta precedencia al utilizar los operadores de inserción de flujo << o de extracción de flujo >> que tienen una relativamente alta precedencia.

Una forma popular para introducir una serie de valores es utilizar la operación de extracción de flujo en el encabezado de un ciclo **while**. La extracción devuelve falso (cero) cuando encuentra el fin de archivo. Considere el programa de la figura 21.11, mismo que encuentra la calificación más alta de un examen. Suponga que el número de calificaciones no es conocido con anticipación, y que el usuario escribirá un fin de archivo, a fin de indicar que todas las calificaciones han sido introducidas. La condición **while (cin >> grade)**, se convierte en cero (es decir, falso) cuando el usuario introduzca el fin de archivo.

```
// fig21_11.cpp
// Stream extraction operator returning
// false on end-of-file.
#include <iostream.h>

main()
{
   int grade, highestGrade = -1;

   cout << "Enter grade (enter end-of-file to end): ";
   while (cin >> grade) {
      if (grade > highestGrade)
         highestGrade = grade;

      cout << "Enter grade (enter end-of-file to end): ";
   }

   cout << "\nHighest grade is: " << highestGrade
        << endl;
   return 0;
}
```

```
Enter grade (enter end-of-file to end): 67
Enter grade (enter end-of-file to end): 87
Enter grade (enter end-of-file to end): 73
Enter grade (enter end-of-file to end): 95
Enter grade (enter end-of-file to end): 34
Enter grade (enter end-of-file to end): 99
Enter grade (enter end-of-file to end): ^Z

Highest grade is: 99
```

Fig. 21.11 El operador de extracción de flujo devolviendo falso al fin de archivo.

21.4.2 Funciones miembro get y getline

La función miembro **get** sin ningún argumento, introduce un carácter del flujo designado (incluso si se trata de un espacio en blanco) y devuelve dicho carácter como el valor de la llamada de función. Esta versión de **get** devuelve **EOF** cuando en el flujo se encuentra el fin de archivo. **EOF** por lo regular es **-1**, a fin de distinguirlo de valores de caracteres ASCII (esto de un sistema a otro pudiera variar).

En la figura 21.12 se demuestra el uso de las funciones miembro **eof** y **get** en el flujo de entrada **cin** y de la función miembro **put** en el flujo de salida **cout**. El programa primero imprime el valor de **cin.eof()**, es decir, **0**, (falso) a fin de mostrar que en **cin** no ha ocurrido el fin de archivo. El usuario escribe una línea de texto, seguida por un fin de archivo (**<ctrl>-z** seguido por **<return>**, en sistemas IBM PC y compatibles, y **<ctrl>-d**, en sistemas UNIX y Macintosh). El programa leerá cada uno de los caracteres y mediante la función miembro **put** lo extraerá a **cout**. Cuando se encuentra con el fin de archivo, se termina el **while**, y **cin.eof()** —ahora **1** (verdadero)— se vuelve a imprimir, a fin de demostrar que en **cin** se ha definido el fin de archivo. Note que este programa utiliza la versión de la función miembro **get** de **istream**, que no toma argumentos y que devuelve el carácter que se está introduciendo.

```
// fig21_12.cpp
// Using member functions get, put, and eof.

#include <iostream.h>

main()
{
   int c;

   cout << "Before input, cin.eof() is "
        << cin.eof() << '\n'
        << "Enter a sentence followed by end-of-file:\n";

   while ((c = cin.get()) != EOF)
      cout.put(c);

   cout << "\nAfter input, cin.eof() is "
        << cin.eof() << endl;

   return 0;
}
```

```
Before input, cin.eof() is 0
Enter a sentence followed by end-of-file:
Testing the get and put member functions^Z
Testing the get and put member functions
After input cin.eof() is 1
```

Fig. 21.12 Cómo utilizar las funciones miembro **get**, **put** y **eof**.

La función miembro **get** con un argumento de carácter, introduce el carácter siguiente del flujo de entrada (aun si se trata de un carácter en blanco). Esta versión de **get** devuelve falso cuando se encuentra el fin de archivo; de lo contrario esta versión de **get** devuelve una referencia al objeto **istream** para la cual se ha invocado a la función miembro **get**.

Una tercera versión de la función miembro **get** toma tres argumentos —un arreglo de caracteres, un límite de tamaño y un delimitador (con un valor por omisión de **'\n'**). Esta versión lee caracteres a partir del flujo de entrada. Lee hasta uno menos que el número máximo especificado de caracteres y termina, o termina en cuanto lea el delimitador. Se inserta un carácter nulo para terminar la cadena de caracteres, en el arreglo de caracteres que se utiliza como búfer por el programa. El delimitador no es colocado en el arreglo de caracteres, sino que es conservado en el flujo de entrada. En la figura 21.13 se compara la entrada utilizando **cin** con la extracción de flujo y la entrada con **cin.get**.

```
// fig21_13.cpp
// Contrasting input of a string with cin and cin.get.

#include <iostream.h>

const int SIZE = 80;

main()
{
   char buffer1[SIZE], buffer2[SIZE];

   cout << "Enter a sentence:\n";
   cin >> buffer1;
   cout << "\nThe string read with cin was:\n"
        << buffer1 << "\n\n";

   cin.get(buffer2, SIZE);
   cout << "The string read with cin.get was:\n"
        << buffer2 << endl;

   return 0;
}
```

```
Enter a sentence:
Contrasting string input with cin and cin.get

The string read with cin was:
Contrasting

The string read with cin.get was:
 string input with cin and cin.get
```

Fig. 21.13 Comparación de entradas de una cadena mediante **cin**, con la extracción de flujo y la entrada mediante **cin.get**.

La función miembro **getline** opera como la tercera versión de la función miembro **get** e inserta un carácter nulo después de la línea en el arreglo de caracteres. La función **getline** elimina el delimitador del flujo, pero no lo almacena en el arreglo de caracteres. El programa de la figura 21.14 demuestra el uso de la función miembro **getline** para la introducción de una línea de texto.

21.4.3 Otras funciones miembro istream (peek, putback, ignore)

La función miembro **ignore** pasa por alto un número designado de caracteres (un carácter por omisión) o da por terminado, al encontrar un delimitador designado (el delimitador por omisión es **EOF**).

La función miembro **putback** coloca el carácter previo obtenido por un **get** del flujo de entrada de regreso en dicho flujo. Esta función es útil para aquellas aplicaciones que rastrean un flujo de entrada buscando un campo que se inicie con un carácter específico. Cuando se introduce dicho carácter, la aplicación devuelve este carácter al flujo, a fin de que el carácter pueda ser incluido en los datos que serán introducidos.

La función miembro **peek** devuelve el carácter siguiente de un flujo de entrada, pero sin retirarlo del flujo.

```
// fig21_14.cpp
// Character input with member function getline.

#include <iostream.h>

const SIZE = 80;

main()
{
   char buffer[SIZE];

   cout << "Enter a sentence:\n";
   cin.getline(buffer, SIZE);

   cout << "\nThe sentence entered is:\n"
        << buffer << endl;

   return 0;
}
```

```
Enter a sentence:
Using the getline member function

The sentence entered is:
Using the getline member function
```

Fig. 21.14 Entrada de caracteres mediante la función miembro **getline**.

21.4.4 Entradas/salidas de tipo seguro

C++ ofrece *entradas/salidas de tipo seguro*. Se hace la homonimia de los operadores << y >> a fin de que acepten elementos de datos de tipos específicos. Si se procesan datos no esperados, se establecen varias banderas de error, mismas que el usuario puede probar, para determinar si una operación de entrada/salida ha tenido éxito o ha fracasado. De esta manera el programa "se conserva en control". Analizaremos estas banderas de error en la sección 21.8.

21.5 Entradas/salidas sin formato utilizando read, gcount, y write

Se lleva a cabo *entradas/salidas sin formato* mediante las funciones miembro **read** y **write**. Cada una de estas funciones introduce o extrae cierta cantidad de bytes de o desde un arreglo de caracteres existente en memoria. Estos bytes no contienen ningún tipo de formato. Sólo son introducidos o extraídos como bytes en bruto. Por ejemplo, la llamada

```
char buffer[] = "HAPPY BIRTHDAY";
cout.write(buffer, 10);
```

extrae los primeros 10 bytes de **buffer**. Dado que la cadena de caracteres se evalúa a la dirección del primero de los caracteres, la llamada

```
cout.write("ABCDEFGHIJKLMNOPQRSTUVWXYZ", 10);
```

exhibe los primeros 10 caracteres del alfabeto.

La función miembro **read** introduce un número designado de caracteres en un arreglo de caracteres. Si son leídos menos que el número designado de caracteres, se activa **failbit**. Pronto veremos cómo determinar si **failbit** ha sido activado (vea la sección 21.8).

En la figura 21.5 se demuestra el uso de las funciones miembro **read** y **gcount** de **istream**, y la función miembro **write** de **ostream**. El programa introduce 20 caracteres, (a partir de una secuencia de entrada más larga) en el arreglo de caracteres de nombre **buffer** mediante **read**, determina el número de caracteres introducidos mediante **gcount**, y extra los caracteres existentes en **buffer** mediante **write**

21.6 Manipuladores de flujo

C++ proporciona varios *manipuladores de flujo* que ejecutan tareas de formato. Los manipuladores de flujo ofrecen capacidades tales como definición de anchos de campo, definición de precisiones, establecimiento o eliminación de banderas de formato, establecimiento de caracteres de relleno en campos, vaciado de flujos, inserción de nueva línea en el flujo de salida y vaciado de flujo, inserción de un carácter nulo en el flujo de salida, y pasar por alto los espacios en blanco del flujo de entrada. Estas características se describen en las secciones siguientes.

21.6.1 Base de flujo integral: manipuladores de flujo dec, oct, hex, y setbase

Los enteros por lo general, se interpretan como valores decimales (de base 10). Para cambiar la base con la cual se interpretan los enteros dentro de un flujo, inserte el manipulador **hex** para definir la base a hexadecimal (de base 16), o el manipulador **oct** para definir a base octal (de base 8). Inserte el manipulador de flujo **dec** para devolver la base del flujo a decimal.


```
// fig21_15.cpp
// Unformatted I/O with the read,
// gcount and write member functions.

#include <iostream.h>

const int SIZE = 80;

main()
{
   char buffer[SIZE];

   cout << "Enter a sentence:\n";
   cin.read(buffer, 20);
   cout << "\nThe sentence entered was:\n";
   cout.write(buffer, cin.gcount());
   cout << endl;

   return 0;
}
```

```
Enter a sentence:
Using the read, write, and gcount member functions

The sentence entered was:
Using the read, writ
```

Fig. 21.15 Entradas/salidas sin formato mediante las funciones miembro **read**, **gcount** y **write**.

La base de un flujo también puede modificarse mediante el manipulador de flujo **setbase**, que toma un argumento entero **10**, **8** ó **16** para definir la base. Dado que **setbase** toma un argumento, se conoce como un *manipulador de flujo parametrizado*. El uso de **setbase** o de cualquier otro manipulador parametrizado requiere de la inclusión del archivo de cabecera **iomanip.h**. La base de flujo se conservará sin cambios hasta que sea modificada en forma explícita. En la figura 21.16 se muestra la utilización de los manipuladores de flujo **hex**, **oct**, **dec** y **setbase**.

21.6.2 Precisión de punto flotante (precision, setprecision)

Podemos controlar la *precisión* de los números en punto flotante, es decir, el número de dígitos a la derecha del punto decimal, ya sea mediante el manipulador de flujo **setprecision** o la función miembro **precision**. Una llamada a cualquiera de estas funciones define la precisión para todas las operaciones subsecuentes de salida, hasta la siguiente llamada definidora de precisión. La función miembro **precision** sin argumento, devuelve el ajuste actual de precisión. El programa de la figura 21.17 utiliza tanto la función miembro **precision**, como el manipulador **setprecision**, para imprimir una tabla mostrando la raíz cuadrada de **2** con precisiones variando desde **0** hasta **9**. Note que la precisión **0** tiene un significado especial. Restaura la *precisión por omisión* de valor **6**.

```
// fig21_16.cpp
// Using hex, oct, dec and setbase stream manipulators.
#include <iostream.h>
#include <iomanip.h>

main()
{
   int n;

   cout << "Enter a decimal number: ";
   cin >> n;

   cout << n << " in hexadecimal is: "
        << hex << n << '\n'
        << dec << n << " in octal is: "
        << oct << n << '\n'
        << setbase(10) << n << " in decimal is: "
        << n << endl;

   return 0;
}
```

```
Enter a decimal number: 20
20 in hexadecimal is: 14
20 in octal is: 24
20 in decimal is: 20
```

Fig. 21.16 Cómo utilizar los manipuladores de flujo `hex`, `oct`, `dec`, y `setbase`.

21.6.3 Ancho de campo (Setw, Width)

La función miembro **`width`** define el ancho de campo y devuelve el ancho anterior. Si los valores procesados son menores que el ancho de campo, se insertan *caracteres de llenado* como *relleno*. Un valor más ancho que el ancho designando no será truncado —se imprimirá el número completo. El ajuste de ancho se aplica sólo a la siguiente inserción o extracción; después y de forma implícita el ancho se define a **0**, es decir, los valores de salida sencillamente serán tan anchos como lo requieran. La función **`width`** sin argumento devuelve el ajuste presente.

Error común de programación 21.4

Cuando no se de un campo lo suficiente amplio para manejar salidas, dichas salidas se imprimirán del ancho que requieran, posiblemente causando salidas erróneas o difíciles de leer.

En la figura 21.18 se demuestra el uso de la función miembro **`width`** tanto en la entrada como en la salida. Note que en el caso de la entrada, se leerá un máximo de un carácter menos que el ancho establecido, porque se deja lugar para el carácter nulo a colocarse en la cadena de entrada. Recuerde que la extracción de flujo se da por terminada cuando se encuentra un espacio en blanco a la derecha. El manipulador de flujo **`setw`** también puede ser utilizado para definir el ancho de campo.


```
// fig21_17.cpp
// Controlling precision of floating point values
#include <iostream.h>
#include <iomanip.h>
#include <math.h>

main()
{
   double root2 = sqrt(2.0);

   cout << "Square root of 2 with precisions 0-9.\n"
        << "Precision set by the "
        << "precision member function:\n";

   for (int places = 0; places <= 9; places++) {
      cout.precision(places);
      cout << root2 << endl;
   }

   cout << "\nPrecision set by the "
        << "setprecision manipulator:\n";

   for (places = 0; places <= 9; places++)
      cout << setprecision(places) << root2 << endl;

   return 0;
}
```

```
Square root of 2 with precisions 0-9.
Precision set by the precision member function:
1.414214
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562

Precision set by the setprecision manipultor:
1.414214
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562
```

Fig. 21.17 Cómo controlar la precisión de valores de punto flotante.

```
// fig21_18.cpp
// Demonstrating the width member function

#include <iostream.h>

main()
{
   int w = 4;
   char string[10];

   cout << "Enter a sentence:\n";
   cin.width(5);

   while (cin >> string) {
      cout.width(w++);
      cout << string << '\n';
      cin.width(5);
   }

   return 0;
}
```

```
Enter a sentence:
This is a test of the width member function^Z
This
   is
    a
   test
      of
      the
      widt
         h
        memb
           er
          func
           tion
```

Fig. 21.18 Demostración de la función miembro **width**.

21.6.4 Manipuladores definidos por usuario

Los usuarios pueden crear su propios manipuladores de flujo. En la figura 21.19 se muestra la creación y el uso de los nuevos manipuladores de flujo **bell**, **ret** (retorno de carro), **tab** y **endLine**. Los usuarios también pueden crear sus propios manipuladores de flujo parametrizados —consulte sus manuales de instalación para encontrar las instrucciones de cómo llevar a cabo lo anterior.

21.7 Estados de formato de flujo

Varias *banderas de formato* especifican el tipo de formato a llevarse a cabo durante las operaciones de entrada/salida de flujo. Las funciones miembro **setf**, **unsetf** y **flags** controlan los ajustes de las banderas.


```
// fig21_19.cpp
// Testing user-defined, nonparameterized manipulators

#include <iostream.h>

// bell manipulator (using escape sequence \a)
ostream& bell(ostream& output)
{
   return output << '\a';
}

// ret manipulator (using escape sequence \r)
ostream& ret(ostream& output)
{
   return output << '\r';
}

// tab manipulator (using escape sequence \t)
ostream& tab(ostream& output)
{
   return output << '\t';
}

// endLine manipulator (using escape sequence \n
// and the flush member function)
ostream& endLine(ostream& output)
{
   return output << '\n' << flush;
}

main()
{
   cout << "Testing the tab manipulator:\n"
        << 'a' << tab << 'b' << tab << 'c' << endLine
        << "Testing the ret and bell manipulators:\n"
        << ".....";

   for (int i = 1; i <= 100; i++)
      cout << bell;

   cout << ret << "-----" << endLine;

   return 0;
}
```

```
Testing the tab manipulator:
a       b       c
Testing the ret and bell manipulators:
-----.....
```

Fig. 21.19 Cómo probar manipuladores definidos por usuario no parametrizados.

21.7.1 Banderas de estado de formato (setf, unsetf, flags)

Las banderas de estado de formato mostradas en la figura 21.20 se definen como una enumeración en la clase **ios** y son explicadas en la siguientes varias secciones.

Estas banderas pueden ser controladas por las funciones miembro **flags, setf** y **unsetf**, pero muchos programadores de C++ prefieren utilizar manipuladores de flujo (vea la sección 10.7.8). El programador puede utilizar la operación "OR a nivel de bits" **|**, para combinar varias opciones en un solo valor **int** (vea la figura 10.23). Llamar la función miembro **flags** para un flujo y especificar estas opciones de tipo "OR" define las opciones sobre dicho flujo y devuelve un valor que contiene las opciones anteriores. A menudo, este valor es guardado de tal forma que con este valor guardado pueda llamarse a **flags**, a fin de restaurar las opciones de flujo anteriores.

La función **flags** debe especificar un valor que represente los ajustes de todas las banderas. La función **setf** de un argumento, por otra parte, define una o más banderas "operadas con OR" o bien las opera con "OR" con los ajustes de banderas existentes a fin de formar un nuevo estado de formato.

El manipulador de flujo parametrizado **setiosflags** ejecuta las mismas funciones que la función **setf**. El manipulador de flujo **resetiosflags** lleva a cabo las mismas funciones que la función miembro **unsetf**. Para utilizar cualquiera de estos manipuladores de flujo, asegúrese de incluir **#include <iomanip.h>**.

La bandera **skipws** indica que **>>** deberá pasar por alto los espacios en blanco de un flujo de entrada. El comportamiento por omisión de **>>** es precisamente pasar por alto los espacios en blanco. Para modificar lo anterior, utilice la llamada **unsetf (ios::skipws)**. El manipulador de flujo **ws** pudiera también ser utilizado para definir que deben de saltarse los espacios en blanco.

21.7.2 Ceros a la derecha y puntos decimales (ios::showpoint)

La bandera **showpoint** se utiliza para obligar a un número de punto flotante a que se extraiga con punto decimal y ceros a la derecha. Sin la definición de **showpoint** un valor de punto flotante de **79.0** sería impreso como **79** y si **showpoint** estuviera definido sería impreso como **79.000000** (o con tantos ceros a la derecha como estuvieran especificados en la posición actual). El programa de la figura 21.21 muestra el uso de la función miembro **setf** para definir la bandera **showpoint** y controlar los ceros a la derecha y la impresión del punto decimal en el caso de valores de punto flotante.

```
ios::skipws
ios::left
ios::right
ios::internal
ios::dec
ios::oct
ios::hex
ios::showbase
ios::showpoint
ios::uppercase
ios::showpos
ios::scientific
ios::fixed
ios::unitbuf
ios::stdio
```

Fig. 21.20 Banderas de estado de formato.

21.7.3 Justificación (ios::left, los::right, ios::internal)

Las banderas **left** y **right** permiten que los campos sean justificados a la izquierda con caracteres de llenado a la derecha, o justificados a la derecha con caracteres de llenado a la izquierda, respectivamente. El carácter a utilizarse para el relleno se define mediante la función miembro **fill** o el manipulador de flujo parametrizado **setfill** (vea la sección 10.7.4). En la figura 21.22 se muestra el uso de los manipuladores **setw, setiosflags,** y **resetiosflags** así como las funciones miembro **setf** y **unsetf** para controlar la justificación a la derecha y a la izquierda de datos enteros dentro de un campo.

```
// fig21_21.cpp
// Controlling the printing of trailing
// zeros and decimal points when using
// floating point values with float values.

#include <iostream.h>
#include <iomanip.h>
#include <math.h>

main()
{
   cout << "cout prints 9.9900 as: " << 9.9900
        << "\ncout prints 9.9000 as: " << 9.9000
        << "\ncout prints 9.0000 as: " << 9.0000
        << "\n\nAfter setting the ios::showpoint flag\n";
   cout.setf(ios::showpoint);

   cout << "cout prints 9.9900 as: " << 9.9900
        << "\ncout prints 9.9000 as: " << 9.9000
        << "\ncout prints 9.0000 as: " << 9.0000 << endl;
   return 0;
}
```

```
cout prints 9.9900 as: 9.99
cout prints 9.9000 as: 9.9
cout prints 9.0000 as: 9

After setting the ios::showpoint flag
cout prints 9.9900 as: 9.990000
cout prints 9.9000 as: 9.900000
cout prints 9.0000 as: 9.000000
```

Fig. 21.21 Cómo controlar la impresión de ceros a la derecha y puntos decimales con valores de punto flotante.

```
// fig21_22.cpp
// Left-justification and right-justification.

#include <iostream.h>
#include <iomanip.h>

main()
{
   int x = 12345;

   cout << "Default is right justified:\n"
        << setw(10) << x
        << "\n\nUSING MEMBER FUNCTIONS\n"
        << "Use setf to set ios::left:\n" << setw(10);

   cout.setf(ios::left, ios::adjustfield);
   cout << x << "\nUse unsetf to restore default:\n";
   cout.unsetf(ios::left);
   cout << setw(10) << x
        << "\n\nUSING PARAMETERIZED STREAM MANIPULATORS"
        << "\nUse setiosflags to set ios::left:\n"
        << setw(10) << setiosflags(ios::left) << x
        << "\nUse resetiosflags to restore default:\n"
        << setw(10) << resetiosflags(ios::left)
        << x << endl;
   return 0;
}
```

```
Default is right justified:
    12345

USING MEMBER FUNCTIONS
Use setf to set ios::left:
12345
Use unsetf to restore default:
     12345

USING PARAMETERIZED STREAN MANIPULATORS
Use setiosflags to set ios::left:
12345
Use resetiosflags to restore default:
     12345
```

Fig. 21.22 Justificaciones a la izquierda y a la derecha.

La bandera **internal** indica que el signo de un número (o su base cuando la bandera **ios::showbase** está activa; vea la sección 21.7.5) deberá aparecer con justificación a la izquierda dentro de un campo, la magnitud del número deberá ser justificada a la derecha, y los espacios intermedios deberán ser rellenos con un carácter de llenado. Las banderas **left**, **right** e **internal** están contenidas en el miembro de datos estático **ios::adjustfield**. Al definir las banderas de justificación **left**, **right** o **internal**, el argumento **ios::adjustfield** deberá ser proporcionado como segundo argumento de **setf**. Esto activará a **setf** a fin de asegurarse que solamente una de las tres banderas de justificación esté activa (son mutuamente excluyentes). En la figura 21.23 se muestra el uso de los manipuladores de flujo **setiosflags** y **setw** para definir el espaciamiento interno. Note el uso de la bandera **ios::showpos** para obligar a la impresión del signo más.

21.7.4 Relleno (Fill, Setfill)

La *función miembro* **fill** especifica el carácter de llenado a utilizarse en campos ajustados; si no se especifica valor , se utilizarán espacios para el relleno. La función **fill** devuelve el carácter de relleno anterior. El *manipulador* **setfill** también define el carácter de relleno. En la figura 21.24 se demuestra el uso de la función miembro **fill** y del manipulador **setfill** para controlar la definición y redefinición del carácter de llenado.

21.7.5 Base de flujo integral (ios::dec, ios::oct, ios::hex, ios::showbase)

El *miembro estático* **ios::basefield** (que se usa de una manera similar que **ios::adjustfield** con **setf**), incluye los bits **oct**, **hex** y **dec** para especificar que los enteros deben ser tratados como valores octales, hexadecimales y decimales, en forma respectiva. Por omisión, las inserciones de flujo se hacen a valores decimales, si ninguno de estos bits está activo; el valor por omisión para las extracciones de flujo es procesar los datos en la misma forma en que han sido introducidos —los enteros que se inicien con **0** se tratan como valores octales, los enteros que se inicien con **0x** o **0X** se tratarán como valores hexadecimales, y todos los demás enteros serán tratados como valores decimales. Una vez especificada una base particular para un flujo, todos los enteros de dicho flujo serán procesados con dicha base, hasta que se especifique una nueva base, o hasta el final del programa.

```
// fig21_23.cpp
// Printing an integer with internal spacing and
// forcing the plus sign.

#include <iostream.h>
#include <iomanip.h>

main()
{
   cout << setiosflags(ios::internal | ios::showpos)
        << setw(10) << 123 << endl;

   return 0;
}
```

```
+       123
```

Fig. 21.23 Cómo imprimir un entero con espaciamiento interno y obligando a la impresión del signo más.

```
// fig21_24.cpp
// Using the fill member function and the setfill
// manipulator to change the padding character for
// fields larger than the values being printed.

#include <iostream.h>
#include <iomanip.h>

main()
{
   int x = 10000;

   cout << x
        << " printed as int right and left justified\n"
        << "and as hex with internal justification.\n"
        << "Using the default pad character (space):\n";
   cout.setf(ios::showbase);
   cout << setw(10) << x << '\n ;
   cout.setf(ios::left, ios::adjustfield);
   cout << setw(10) << x << '\n';
   cout.setf(ios::internal, ios::adjustfield);
   cout << setw(10) << hex << x << "\n\n";

   cout << "Using various padding characters:\n";
   cout.setf(ios::right, ios::adjustfield);
   cout.fill('*');
   cout << setw(10) << dec << x << '\n';
   cout.setf(ios::left, ios::adjustfield);
   cout << setw(10) << setfill('%') << x << '\n';
   cout.setf(ios::internal, ios::adjustfield);
   cout << setw(10) << setfill('^') << hex
        << x << endl;

   return 0;
}
```

```
10000 printed as int right and left justified
and as hex with internal justification.
Using the default pad character (space):
     10000
10000
0x   2710

Using various padding characters:
*****10000
10000%%%%%
0x****2710
```

Fig. 21.24 Cómo utilizar la función miembro **fill** y el manipulador **setfill** para modificar el carácter de relleno para campos mayores que los valores a imprimirse.

Active la bandera **showbase** para que se extraiga la base del valor completo. Los números decimales serán extraídos en forma normal, los números octales serán extraídos con un **0** a la izquierda, y los números hexadecimales serán extraídos ya sea con un **0x** a la izquierda o con un **0X** a la izquierda (la bandera **uppercase** determinará cuál de las opciones será seleccionada). En la figura 21.25 se demuestra el uso de la bandera **showbase** para obligar a que un entero se imprima en formatos decimal. octal y hexadecimales.

21.7.6 Números de punto flotante; notación científica (ios::scientific, ios::fixed)

La *bandera* ***ios::scientific*** y la *bandera* ***ios::fixed*** están incluidas en el *miembro de dato estático* ***ios::floatfield*** (que se usa en forma similar a **ios::adjustfield** e **ios::basefield** en **setf**). Se utilizan estas banderas para controlar el formato de salida de los números de punto flotante. La bandera **scientific** se activa para obligar a la salida de un número de punto flotante en formato científico. La bandera **fixed** obliga a un número de punto flotante a mostrar un número específico de dígitos (de acuerdo con lo especificado con la función miembro **precision**) a la derecha del punto decimal. Si estas banderas no están activadas, el valor del punto flotante determinará el formato de salida.

La llamada **cout.setf(0, ios::floatfield)** restaura el formato de sistema por omisión para la salida de número de punto flotante. En la figura 21.26 se demuestra la exhibición de los números de punto flotante, tanto en formatos fijos como científicos.

```
// fig21_25.cpp
// Using the ios::showbase flag

#include <iostream.h>
#include <iomanip.h>

main()
{
   int x = 100;

   cout << setiosflags(ios::showbase)
        << "Printing integers preceded by their base:\n"
        << x << '\n'
        << oct << x << '\n'
        << hex << x << endl;

   return 0;
}
```

```
Printing integers preceded by their base:
100
0144
0x64
```

Fig. 21.25 Cómo utilizar la bandera **ios::showbase**.

```
// fig21_26.cpp
// Displaying floating point values in system default,
// scientific, and fixed formats.

#include <iostream.h>

main()
{
   double x = .001234567, y = 1.946e9;

   cout << "Displayed in default format:\n"
        << x << '\t' << y << '\n';
   cout.setf(ios::scientific, ios::floatfield);
   cout << "Displayed in scientific format:\n"
        << x << '\t' << y << '\n';
   cout.unsetf(ios::scientific);
   cout << "Displayed in default format after unsetf:\n"
        << x << '\t' << y << '\n';
   cout.setf(ios::fixed, ios::floatfield);
   cout << "Displayed in fixed format:\n"
        << x << '\t' << y << endl;

   return 0;
}
```

```
Displayed in default format:
0.001235   1.946e+09
Displayed in scientific format:
1.234567e-03   1.946e+09
Displayed in default format after unsetf:
0.001235   1.946e+09
Displayed in fixed format:
0.001235   1946000000
```

Fig. 21.26 Cómo mostrar valores de punto flotante en formatos científicos, fijos y de sistema por omisión.

21.7.7 Control de mayúsculas/minúsculas (ios::uppercase)

La bandera **ios::uppercase** se activa para obligar la salida de una **X** o de una **E** en mayúsculas con enteros hexadecimales o con puntos con valores de punto flotante en notación científica, respectivamente (vea la figura 21.27). Cuando está activa, la bandera **ios::uppercase** hace que todas las letras en un valor hexadecimal aparezcan en mayúsculas.

21.7.8 Cómo activar y desactivar las banderas de formato (flags, setiosflags, resetiosflags)

La función miembro **flags**, sin argumento, sólo devuelve (como un valor **long**) los ajustes actuales de las banderas de formato. La función miembro **flags**, con un argumento **long**, define


```
// fig21_27.cpp
// Using the ios::uppercase flag
#include <iostream.h>
#include <iomanip.h>

main()
{
   cout << setiosflags(ios::uppercase)
        << "Printing uppercase letters in scientific\n"
        << "notation exponents and hexadecimal values:\n"
        << 4.345e10 << '\n'
        << hex << 123456789 << endl;

   return 0;
}
```

```
Printing uppercase letters in scientific
notation exponents and hexadecimal values:
4.345E+10
75BCD15
```

Fig. 21.27 Cómo utilizar la bandera **ios::uppercase**.

las banderas de formato según especificado en el argumento, y devuelve los ajustes de bandera anteriores. Cualquier otra bandera de formato que no se haya especificado en el argumento a **flags** será restablecida. El programa de la figura 21.28 demuestra el uso de la función miembro **flags** para definir un nuevo estado de formato, guardando el anterior, y a continuación restaurando los ajustes de formato original.

La función miembro **setf** define las banderas de formato incluidas en su argumento, y devuelve los ajustes previos de banderas como un valor **long**, como en

```
long previousFlagSettings =
   cout.setf(ios::showpoint | ios::showpos);
```

La función miembro **setf**, con dos argumentos **long**, como en

```
cout.setf(ios::left, ios::adjustfield);
```

primero desactiva los bits de **ios::adjustfield** y a continuación define la bandera **ios::left**. Esta versión de **setf** es utilizada con los campos de bits asociados con **ios::basefield** (representados por **ios::dec**, **ios::oct** e **ios::hex**), **ios::floatfield** (representado por **ios:scientific** e **ios::fixed**), e **ios::adjustfield** (representado por **ios::left**, **ios::right** e **ios::internal**).

La función miembro **unsetf** vuelve a activar las banderas designadas y devuelve el valor de las banderas existentes antes de que hayan sido reactivadas.

21.8 Estados de error de flujo

El estado de un flujo puede ser probado mediante bits en la clase **ios** —la clase base correspondiente a las clases **istream**, **ostream** e **iostream** que estamos utilizando para entradas/salidas.

```
// fig21_28.cpp
// Demonstrating the flags member function.
#include <iostream.h>

main()
{
   int i = 1000;
   double d = 0.0947628;

   cout << "The value of the flags variable is: "
        << cout.flags() << '\n'
        << "Print int and double in original format:\n"
        << i << '\t' << d << "\n\n";
   long originalFormat = cout.flags(ios::oct | ios::scientific);
   cout << "The value of the flags variable is: "
        << cout.flags() << '\n'
        << "Print int and double in a new format\n"
        << "specified using the flags member function:\n"
        << i << '\t' << d << "\n\n";
   cout.flags(originalFormat);
   cout << "The value of the flags variable is: "
        << cout.flags() << '\n'
        << "Print values in original format again:\n"
        << i << '\t' << d << endl;

   return 0;
}
```

```
The value of the flags variable is: 1
Print int and double in original format:
1000    0.094763

The value of the flags variable is: 4040
Print int and double in a new format
specified using the flags member function:
1750    9.47628e-02

The value of the flags variable is: 1
Print values in original format again:
1000    0.094763
```

Fig. 21.28 Demostración de la función miembro **flags**.

El **eofbit** queda automáticamente activado para un flujo de entrada cuando un fin de archivo es encontrado. Un programa puede utilizar la función miembro **eof** para determinar si en un flujo se ha encontrado un fin de archivo. La llamada

```
cin.eof()
```

devuelve verdadero si en **cin** se ha encontrado un fin de archivo, y falso de lo contrario.

El **failbit** queda activado para un flujo cuando en el mismo ocurre un error de formato, pero sin pérdida de caracteres. La función miembro **fail** determina si ha fallado la operación de un flujo; por lo regular es posible recuperarse de dichos errores.

El **badbit** se activa para un flujo cuando ocurre un error que resulte en pérdida de datos. La función miembro **bad** determina si ha fallado una operación de flujo. Estas fallas serias por lo regular no son recuperables.

El **goodbit** se activa en un flujo si para dicho flujo no se han activado ninguno de los bits **eofbit**, **failbit**, o **badbit**.

La función miembro **good** devuelve verdadero si las funciones **bad**, **fail** y **eof** devolvieran todas ellas falso. Las operaciones de entrada/salida deberían únicamente ser llevadas a cabo sobre flujos "buenos".

La función miembro **rdstate** devuelve el estado de error del flujo. Una llamada a **cout.rdstate**, por ejemplo, devolvería el estado del flujo que a continuación podría ser probado mediante un enunciado **switch** que examinase **ios::eofbit**, **ios::badbit**, **ios::failbit** e **ios::goodbit**. La manera preferida de probar el estado de un flujo es utilizando las funciones miembro **eof**, **bad**, **fail** y **good** —el uso de estas funciones no requiere que el programador esté familiarizado con bits de estado particulares.

La función miembro **clear** se utiliza por lo general, para restaurar el estado de un flujo a "bueno", de tal forma que puedan continuar las entradas/salidas sobre dicho flujo. El argumento por omisión para **clear** es **ios::goodbit**, por lo que el enunciado

```
cin.clear();
```

elimina o limpia **cin** y activa **goodbit** para el flujo. El enunciado

```
cin.clear(ios::failbit)
```

es el que verdaderamente activa **failbit**. El usuario quizás desease hacer esto al ejecutar una entrada en **cin** con un tipo definido por usuario y al encontrar un problema. En este contexto el nombre **clear** parece inapropiado, pero es correcto.

El programa de la figura 21.29 ilustra el uso de las funciones miembro **rdstate**, **eof**, **fail**, **bad**, **good**, y **clear**.

La función miembro **operator!** devuelve verdadero ya sea que **badbit** esté activo, **failbit** o ambos. La función miembro **operator void*** devuelve falso si **badbit**, **failbit** o ambos están activos. Estas funciones resultan útiles en el procesamiento de archivos, cuando se hacen pruebas de condiciones verdaderas/falsas dentro de la condición de una estructura de selección o de repetición.

21.9 Entradas/salidas de tipos definidos por usuario

C++ es capaz de introducir y extraer los tipos de datos estándar utilizando los operadores de extractor de flujo >> y de inserción de flujo <<. Estos operadores son homónimos, a fin de poder procesar cada tipo de datos estándar, incluyendo cadenas y direcciones en memoria. El programador puede hacer la homonimia de los operadores de inserción y de extracción de flujo, a fin de llevar a cabo entradas y salidas de tipos definidos por usuario. En la figura 21.30 se demuestra la homonimia de operadores de extracción y de inserción de flujo, a fin de manejar datos de una clase de número telefónicos definidos por usuario llamada **PhoneNumber**. Note que este programa supone que los números telefónicos están introducidos correctamente. Dejamos como ejercicio el incorporar la correspondiente verificación de errores.

```
// fig21_29.cpp
// Testing error states.

#include <iostream.h>

main()
{
   int x;
   cout << "Before a bad input operation:\n"
        << "cin.rdstate(): " << cin.rdstate()
        << "\n    cin.eof(): " << cin.eof()
        << "\n   cin.fail(): " << cin.fail()
        << "\n    cin.bad(): " << cin.bad()
        << "\n   cin.good(): " << cin.good()
        << "\n\nExpects an integer, but enter a character: ";
   cin >> x;

   cout << "\nAfter a bad input operation:\n"
        << "cin.rdstate(): " << cin.rdstate()
        << "\n    cin.eof(): " << cin.eof()
        << "\n   cin.fail(): " << cin.fail()
        << "\n    cin.bad(): " << cin.bad()
        << "\n   cin.good(): " << cin.good() << "\n\n";

   cin.clear();

   cout << "After cin.clear()\n"
        << "cin.fail(): " << cin.fail() << endl;

   return 0;
}
```

```
Before a bad input operation:
cin.rdstate(): 0
    cin.eof(): 0
   cin.fail(): 0
    cin.bad(): 0
   cin.good(): 1

Expects an integer, but enter a character: A

After a bad input operation:
cin.edstate(): 2
    cin.eof(): 0
   cin.fail(): 2
    cin.bad (): 0
   cin.good(): 0

After cin.clear()
cin.fail(): 0
```

Fig. 21.29 Cómo probar estados de error.

El operador de extracción de flujo toma como argumentos una referencia **istream** y una referencia a un tipo definido por usuario (en este caso **PhoneNumber**), y devuelve una referencia **istream**. En la figura 21.30, el operador de extracción de flujo homónimo es utilizado para introducir los números telefónicos de la forma

```
(800) 555-1212)
```

en objetos del tipo **PhoneNumber**. La función de operador lee las tres partes de un número telefónico en los miembros **areaCode**, **exchange** y **line** del objeto referenciado **PhoneNumber** (**num** en la función de operador y **phone** en **main**). Los caracteres de paréntesis, de espacio y de guiones serán descartados mediante el llamado a la función miembro **ignore** de **istream**. La función de operador devuelve la referencia **input** de **istream**. Al devolver la referencia de flujo, las operaciones de entrada en objetos **PhoneNumber** pueden ser concatenadas con operaciones de entrada sobre otros objetos **PhoneNumber** o con otros tipos de datos. Por ejemplo, dos objetos **PhoneNumber** podrían ser introducidos como sigue:

```
cin >> phone1 >> phone2;
```

El operador de inserción de flujo toma como argumentos una referencia **ostream** y una referencia a un tipo definido por usuario (en este caso **PhoneNumber**), y devuelve una referencia **ostream**. En la figura 21.30, el operador de inserción de flujo homónimo exhibe objetos del tipo **PhoneNumber**, de la misma forma en que fueron introducidos. La función **operator** exhibe las partes del número telefónico como cadenas, porque están almacenados en formato de cadena (recuerde que la función miembro **getline** de **istream** almacena un carácter nulo después de que termina su entrada).

Las funciones homónimas **operator** se declaran en **class PhoneNumber** como funciones **friend**. Los operadores de entrada y de salida homónimos deben de ser declarados como **friend**, para que puedan tener acceso a miembros de clase no públicos. Los amigos de una clase pueden tener acceso a los miembros de clase privados.

Observación de ingeniería de software 21.3

Se pueden añadir a C++ nuevas capacidades de entrada/salida para tipos definidos por usuario, sin modificar las declaraciones o los miembros de datos privados, ya sea para la clase **ostream** *o para la clase* **istream**. *Esto fomenta la extensibilidad del lenguaje de programación C++ —lo que es uno de los aspectos más atractivos de C++.*

21.10 Cómo ligar un flujo de salida con un flujo de entrada

Las aplicaciones interactivas generalmente involucran un **istream** como entrada y un **ostream** como salida. Cuando en la pantalla aparece un mensaje solicitando algo, el usuario responde introduciendo los datos apropiados. Obviamente, la solicitud debe aparecer antes de que se inicie la operación de entrada. Con almacenamiento temporal de salida, las salidas sólo aparecerán cuando el búfer esté lleno, cuando las salidas se vacíen en forma explícita por requerimientos del programa, o bien en forma automática al final del programa. C++ tiene la función miembro **tie** para sincronizar, es decir, "para ligar" la operación de un **istream** con un **ostream** para asegurar que las salidas aparezcan antes de sus entradas subsecuentes. Una llamada como

```
// fig21_30.cpp
// User-defined insertion and extraction operators

#include <iostream.h>

class PhoneNumber {
   friend ostream& operator<<(ostream&, PhoneNumber&);
   friend istream& operator>>(istream&, PhoneNumber&);
private:
   char areaCode[4];
   char exchange[4];
   char line[5];
};

ostream& operator<<(ostream& output, PhoneNumber& num)
{
   output << "(" << num.areaCode << ") "
          << num.exchange << "-" << num.line;

   return output;
}

istream& operator>>(istream& input, PhoneNumber& num)
{
   input.ignore();                    // skip (
   input.getline(num.areaCode, 4);
   input.ignore(2);                   // skip ) and space
   input.getline(num.exchange, 4);
   input.ignore();                    // skip -
   input.getline(num.line, 5);

   return input;
}

main()
{
   PhoneNumber phone;

   cout << "Enter a phone number in the "
        << "form (123) 456-7890:\n";
   cin >> phone;
   cout << "The phone number entered was:\n"
        << phone << endl;

   return 0;
}
```

```
Enter a phone number in the form (123) 456-7890:
(800) 555-1212
The phone number entered was:
(800) 555-1212
```

Fig. 21.30 Operadores de inserción y de extracción de flujo definidos por usuario.


```
cin.tie(cout);
```

Liga a **cout** (un **ostream**) con **cin** (un **istream**). De hecho, esta llamada en particular resulta redundante, porque C++ lleva a cabo esta operación en forma automática para crear el entorno estándar de entrada/salida. El usuario podría, sin embargo, ligar en forma explícita otros pares **istream/ostream**. A fin de desligar un flujo de entrada, **inputStream**, de un flujo de salida, utilice la llamada

```
inputStream.tie(0);
```

Resumen

- Las operaciones de entrada/salida se ejecutan de una forma que depende o es sensible al tipo de los datos.
- Las entradas y salidas en C++ ocurren en flujos de bytes. Un flujo es sólo una secuencia de bytes.
- Los mecanismos de entrada/salida del sistema mueven bytes de dispositivos a la memoria y viceversa de una forma eficiente y confiable.
- C++ proporciona capacidades de entrada/salida de "bajo nivel" y de "alto nivel". Las capacidades de entrada/salida de bajo nivel especifica que cierto número de bytes deberán ser transferidos del dispositivo a la memoria o de la memoria al dispositivo. Las entradas/salidas de alto nivel se ejecutan con bytes agrupadas en unidades significativas como son enteros, puntos flotantes, caracteres, cadenas y tipos definidos por usuario.
- C++ proporciona operaciones de entradas/salidas tanto con como sin formato. Las entradas y salidas sin formato son rápidas, pero procesan datos en bruto difíciles de utilizar por las personas. Las entradas/salidas con formato procesan los datos en unidades significativas, pero requieren de tiempo de proceso adicional que puede tener un impacto negativo en transferencias de datos a altos volúmenes.
- La mayor parte de los programas de C++ incluyen un archivo de cabecera **iostream.h** que contiene la información básica requerida para todas las operaciones de entradas/salidas de flujo.
- El encabezado **iomanip.h** contiene información para entradas/salidas con formato mediante manipuladores de flujo parametrizados.
- El encabezado **fstream.h** contiene información para operaciones de procesamiento de archivos.
- El encabezado **strstream.h** contiene información para dar formato en memoria.
- El encabezado **stdiostream.h** contiene información para aquellos programas que mezclan los estilos de entradas/salidas de C con las de C++.
- La clase **istream** acepta operaciones de entrada de flujo.
- La clase **ostream** acepta operaciones de salida de flujo.
- La clase **iostream** acepta tanto operaciones de entrada como operaciones de salida de flujo.
- Las clases **istream** y **ostream** son ambas derivadas mediante herencia simple a partir de la clase base **ios**.
- La clase **iostream** es derivada mediante herencia múltiple, tanto de la clase **istream** como de la clase **ostream**.

- Se hace la homonimia del operador de desplazamiento a la izquierda (<<), para designar salida de flujo y se le conoce como el operador de inserción de flujo.
- Se hace la homonimia del operador de desplazamiento a la derecha (>>) para designar entrada de flujo y se le conoce como el operador de extracción de flujo.
- El objeto **cin** de la clase **istream** está ligado al dispositivo de entrada estándar, que por lo general, es el teclado.
- El objeto **cout** de la clase **ostream** está ligado al dispositivo de salida estándar, que por lo regular es la pantalla de exhibición.
- El objeto **cerr** de la clase **ostream** está ligado con el dispositivo de error estándar. Las salidas a **cerr** no pasan por búfer; cada inserción a **cerr** aparece de inmediato.
- El manipulador de flujo **endl** emite un carácter de nueva línea y vacía el búfer de salida.
- El compilador de C++ determina automáticamente los tipos de datos para la entrada y la salida.
- Por omisión las direcciones se exhiben en formato hexadecimal.
- Para imprimir la dirección de una variable del tipo **char***, convierta explícitamente (cast) el apuntador a **void***.
- La función miembro **put** extrae un carácter. Las llamadas a **put** pueden ser concatenadas.
- La salida de flujo se lleva a cabo mediante el operador de extracción de flujo >>. Este operador en forma automática pasa por alto caracteres de espacio en blanco existentes en el flujo de entrada.
- El operador >> devuelve falso cuando en un flujo encuentra la señal de fin de archivo.
- La extracción de flujo hace que se active **failbit** cuando se trate de una entrada incorrecta, y que se active **badbit** si la operación falla.
- Una serie de valores pueden ser introducidos mediante la operación de extracción de flujo en un encabezado de ciclo **while**. La extracción devolverá falso (cero) cuando se encuentre fin de archivo.
- La función miembro **get**, sin argumentos, introduce un carácter y devuelve el carácter; si encuentra en el flujo la señal de fin de archivo, se devolverá **EOF**.
- La función miembro **get**, con un argumento del tipo **char**, introduce un carácter. Se devolverá falso al encontrar el fin de archivo; de lo contrario será devuelto el objeto **istream** para el cual se está invocando la función miembro **get**.
- La función miembro **get**, con tres argumentos, un arreglo de caracteres, un límite de tamaño y un delimitador (con el valor de nueva línea por omisión) —lee caracteres del flujo de entrada hasta un máximo de límite - 1 caracteres y termina, o termina cuando se lea el delimitador. La cadena de entrada se terminará incluyendo un carácter nulo. El delimitador no se coloca en el arreglo de caracteres, sino que se conserva dentro del flujo de entrada.
- La función miembro **getline** opera en forma similar a la función miembro **get** con tres argumentos. La función **getline** elimina el delimitador del flujo de entrada.
- La función miembro **ignore** pasa por alto el número específico de caracteres (su valor por omisión es un carácter) dentro del flujo de entrada; termina si encuentra el delimitador especificado (el delimitador por omisión es **EOF**).
- La función miembro **putback** coloca el carácter previamente obtenido por un **get** de un flujo de regreso en dicho flujo.

- La función miembro **peek** devuelve el siguiente carácter de un flujo de entrada, pero sin eliminar dicho carácter de un flujo.
- C++ ofrece entrada/salida de tipo seguro. Si se procesan datos inesperados por los operadores << y >>, se definirán varias banderas de error, que el usuario podrá probar para determinar si una operación de entrada/salida ha tenido éxito o ha fallado.
- Las entradas/salidas sin formato se llevan a cabo con las funciones miembro **read** y **write**. Estas introducen o extraen un cierto número de bytes, hacia o de la memoria, empezando en una dirección designada de memoria. Se introducen o se extraen como bytes en bruto sin formato.
- La función miembro **gcount** devuelve el número de caracteres introducidos mediante la operación anterior **read** sobre dicho flujo.
- La función miembro **read** introduce un número especificado de caracteres en un arreglo de caracteres. Si se han leído menos que el número especificado de caracteres, se activa **failbit**.
- Para modificar la base numérica en la cual se extraen enteros, utilice el manipulador **hex** para definir la base a hexadecimal (de base 16) o bien **oct** para definir la base a octal (de base 8). Utilice el manipulador **dec** para restaurar la base a decimal. La base se conservará sin modificación en tanto no sea cambiada en forma explícita.
- El manipulador de flujo parametrizado **setbase** también define la base de la salida de enteros. **setbase** toma un argumento entero de **10**, **8**, ó **16** para definir la base.
- La precisión de punto flotante puede ser controlada utilizando o el manipulador de flujo **setprecision** o la función miembro **precision**. Ambos definen la precisión para todas las operaciones de salida subsecuentes hasta la siguiente llamada de ajuste de precisión. La función miembro **precision**, sin argumentos, devuelve el valor de precisión actual. Una precisión de **0** define el valor de precisión por omisión (**6**).
- Los manipuladores parametrizados requieren la inclusión del archivo de cabecera **iomanip.h**.
- La función miembro **width** define el ancho de campo y devuelve el ancho anterior. Los valores que sean menores que el campo serán llenados con caracteres de relleno. El ajuste de ancho de campo se aplica sólo para la siguiente inserción o extracción; después el ancho de campo se define en forma implícita a **0** (los valores subsecuentes serán extraídos del tamaño que necesiten tener). Los valores mayores que el tamaño del campo son impresos por completo. La función **width** sin argumentos, devuelve el ajuste actual de ancho. El manipulador **setw** también define el ancho.
- Para las entradas, el manipulador de flujo **setw** establece un tamaño máximo de cadena; si se introduce una cadena más grande, la línea más grande se divide en piezas no mayores que el tamaño designado.
- Los usuarios pueden crear sus propios manipuladores de flujo.
- Las funciones miembro **setf**, **unsetf**, y **flags** controlan los ajustes de bandera.
- La bandera **skipws** indica que en un flujo de entrada el operador >> deberá pasar por alto los espacios en blanco. El manipulador de flujo **ws** también pasa por alto los espacios en blanco a la izquierda en un flujo de entrada.
- Las banderas de formato se definen como una enumeración en la clase **ios**.
- Las banderas de formato están controladas por las funciones miembro **flags**, y **setf**, pero muchos programadores de C++ prefieren utilizar manipuladores de flujo. La operación OR a

nivel de bits, |, puede ser utilizada para combinar varias opciones en un valor único `long`. Llamar la función miembro `flags` para un flujo y especificar estas opciones con la función OR define las opciones de dicho flujo y devuelve un valor `long` que contiene las opciones precedentes. A menudo, este valor es guardado, por lo que `flags` puede ser llamada con este valor guardado, a fin de restaurar las opciones de flujo anteriores.

- La función `flags` debe definir un valor que represente los ajustes totales de todas las banderas. La función `setf` con un argumento, por otra parte, hace la operación a nivel de bit "OR" correspondiente a las banderas especificadas con los ajustes existentes de bandera, para formar un nuevo estado de formato.
- La bandera `showpoint` se define para obligar a un número de punto flotante a que sea extraído con un punto decimal y una cantidad de dígitos significativos según esté definido en la precisión.
- Las banderas `left` y `right` hacen que los campos queden justificados a la izquierda con caracteres de relleno a la derecha, o a la derecha con caracteres de relleno a la izquierda, respectivamente.
- La bandera `internal` indica que el signo de un número (o la base cuando se trate de la bandera `ios::showbase`) deberá quedar justificado a la izquierda dentro de un campo, la magnitud justificada a la derecha, y llenar los espacios intermedios con un carácter de relleno.
- `ios::adjustfield` contiene las banderas `left`, `right`, e `internal`.
- La función miembro `fill` define el carácter de relleno a utilizar con los campos ajustados `left`, `right`, e `internal` (un espacio en blanco es el valor por omisión); el carácter de relleno anterior es devuelto. El manipulador de flujo `setfill` también define el carácter de relleno.
- El miembro estático `ios::basefield` incluye los bits `oct`, `hex`, y `dec` a fin de especificar que los enteros deben ser tratados como valores octal, hexadecimal y decimales, respectivamente. Si no está activo ninguno de estos bits, la salida de enteros por omisión se hace en decimal; las extracciones de flujo procesan los datos en la misma forma en la cual fueron suministrados.
- Defina la bandera `showbase` para obligar la base de la extracción de un valor integral.
- El miembro de datos estático `ios::floatfield` contiene las banderas `scientific` y `fixed`. Defina la bandera `scientific` para extraer un número en punto flotante en formato científico. Defina la bandera `fixed` para extraer un número de punto flotante con la precisión especificada por la función miembro `precision`.
- La llamada `cout.setf(0, ios::floatfield)` restaura el formato por omisión para la exhibición de los números de punto flotante.
- Active la bandera `uppercase` para obligar a la extracción de una `X` o `E` mayúsculas con los enteros hexadecimales o en la notación científica de valores de punto flotante, respectivamente. Cuando esté activa, la bandera `ios::uppercase` hará que todas las letras en un valor hexadecimal aparezcan en mayúsculas.
- La función miembro `flags` sin argumento, devuelve el valor `long` de los ajustes actuales de las banderas de formato. La función miembro `flags` con el argumento `long` define las banderas de formato especificadas por el argumento y devuelve los ajustes anteriores de bandera.
- La función miembro `setf` define las banderas de formato en su argumento y devuelve los ajustes de banderas anteriores como un valor `long`.

- La función miembro `setf` con dos argumentos `long` —un bit `long` y un campo de bits `long`— elimina los bits en el campo de bits, y a continuación ajusta el bit del primer argumento.
- La función miembro `unsetf` restaura las banderas designadas y devuelve el valor de las banderas antes de su restauración.
- El manipulador de flujo parametrizado `setiosflags` ejecuta las mismas funciones que la función miembro `flags`.
- El manipulador de flujo parametrizado `resetiosflags` ejecuta las mismas funciones que la función miembro `unsetf`.
- El estado de un flujo puede ser probado mediante bits en la clase `ios`.
- El `iofbit` es activado para un flujo de entrada cuando se encuentra un fin de archivo durante una operación de entrada. La función miembro `eof` se utiliza para determinar si `eofbit` está activo.
- El `failbit` se activa para un flujo cuando en dicho flujo ocurrió un error de formato, pero sin pérdida de caracteres. La función miembro `fail` determina si una operación de flujo ha fallado; por lo general, es posible recuperarse de dichos errores.
- `badbit` se activa en un flujo cuando ha ocurrido un error cuyo resultado es pérdida de datos. La función miembro `bad` determina si un operación de flujo ha fallado. Estas fallas serias, por lo regular no son recuperables.
- La función miembro `good` devuelve verdadero si las funciones `bad`, `fail` y `eof` todas ellas devuelven falso. Las operaciones de entrada/salida únicamente deberían ser llevadas a cabo sobre flujos "buenos".
- La función miembro `rdstate` devuelve el estado de error del flujo.
- La función miembro `clear` se utiliza por lo regular para regresar el estado del flujo a "bueno", de tal forma que se pueda proceder con entradas y salidas sobre dicho flujo.
- El usuario puede hacer la homonimia de los operadores de inserción y de extracción de flujo para llevar a cabo entradas/salidas para tipos definidos por usuario.
- El operador de extracción de flujo homónimo toma como argumento una referencia a `istream` y una referencia a un tipo definido por usuario, y devuelve una referencia a `istream`.
- El operador de inserción de flujo homónimo toma como referencias una referencia a `ostream` y una referencia a un tipo definido por usuario, y devuelve una referencia `ostream`.
- A menudo, se declaran funciones homónimas `operator` como funciones `friend` a una clase; esto permite el acceso a miembros de clase no públicos.
- C++ proporciona la función miembro `tie` para sincronizar operaciones `istream` y `ostream` para asegurarse que las salidas aparezcan antes de entradas subsecuentes.

Terminología

función miembro `bad`
`badbit`
`cerr`
`cin`
función miembro `clear`
`clog`
`cout`
manipulador de carácter de flujo `dec`
lcarácter de llenado por omisión (espacio en blanco)
precisión por omisión

fin de archivo
endl
función miembro **eof**
eofbit
extensibilidad
función miembro **fail**
failbit
ancho de campo
carácter de llenado
función miembro **fill**
función miembro **flags**
función miembro **flush**
manipulador de flujo **flush**
banderas de formato
estados de formato
entrada/salida con formato
clase **fstream**
función miembro **gcount**
función miembro **get**
función miembro **getline**
función miembro **good**
manipulador de flujo **hex**
clase **ifstream**
función miembro **ignore**
formato en núcleo
formato en memoria
archivo de cabecera estándar **<iomanip.h>**
clase **ios**
ios::adjustfield
ios::basefield
ios::fixed
ios::floatfield
ios:internal
ios::scientific
ios::showbase
ios::showpoint
ios::showpos
clase iostream
clase istream
0 a la derecha (octal)
0x ó **0X** a la izquierda (hexadecimal)
left
justificado a la izquierda
manipulador de flujo oct
clase ofstream
función miembro **operator!**
función miembro operator void *
clase ostream
relleno
manipulador de flujo parametrizado
función miembro peek
función miembro precision
flujo predeterminado
función miembro put
función miembro **putback**
función miembro **rdstate**
función miembro **read**
manipulador de flujo resetiosflags
justificado a la derecha
manipulador de flujo setbase
función miembro setf
manipulador de flujo setfill
manipulador de flujo setiosflags
manipulador de flujo **setprecision**
manipulador de flujo setw
skipws
bibliotecas de clases de flujo
operador de extracción de flujo (**>>**)
entrada de flujo
operador de inserción de flujo (<<)
manipulador **stream**
salida de flujo
función miembro tie
entrada/salida de tipo seguro
entrada/salida sin formato
función miembro **unsetf**
uppercase
flujos definidos por usuario
caracteres de espacio en blanco
width
función miembro **write**
función miembro **ws**

Errores comunes de programación

21.1 Intentar leer de un **ostream** (o de cualquier otro flujo de sólo salida).
21.2 Intentar escribir a un **istream** (o a cualquier otro flujo de sólo entrada).
21.3 Omitir paréntesis para obligar a una correcta precedencia al utilizar los operadores de inserción de flujo **<<** o de extracción de flujo **>>** que tienen una relativamente alta precedencia.
21.4 Cuando no se de un campo lo suficiente amplio para manejar salidas, dichas salidas se imprimirán del ancho que requieran, posiblemente causando salidas erróneas o difíciles de leer.

Prácticas sanas de programación

21.1 En programas C++, utilice exclusivamente la forma de entradas/salidas de C++, a pesar del hecho que para los programadores C++ esté disponible el estilo de entradas/salidas de C.
21.2 Al extraer expresiones, colóquelas entre paréntesis, para evitar problemas de precedencia de operadores entre los operadores de la expresión y el operador <<.

Sugerencia de rendimiento

21.1 Utilice entradas/salidas sin formato para un rendimiento máximo en procesos de alto volumen de archivos.

Observaciones de ingeniería de software

21.1 El estilo de entradas/salidas de C++ es de tipo seguro.
21.2 C++ permite un tratamiento común de entradas/salidas de tipos predefinidos y de tipos definidos por usuario. Este tipo de estado común facilita el desarrollo de software en general y de la reutilización de software en particular.
21.3 Se pueden añadir a C++ nuevas capacidades de entrada/salida para tipos definidos por usuario, sin modificar las declaraciones o los miembros de datos privados, ya sea para la clase **ostream** o para la clase **istream**. Esto fomenta la extensibilidad del lenguaje de programación C++ —lo que es uno de los aspectos más atractivos de C++.

Ejercicios de autoevaluación

21.1 Llene cada uno de los siguientes espacios vacíos:
a) Las funciones de operador de flujo homónimas deben ser definidas como funciones _____ de una clase.
b) Los bits de formato de justificación que se pueden definir son _____, _____, y _____.
c) En C++ la entrada/salida ocurre como un _____ de bytes.
d) Los manipuladores de flujo parametrizados _____, y _____ se pueden utilizar para activar y desactivar banderas de estado de formato.
e) La mayor parte de los programas de C++ deben incluir el archivo de cabecera _____ que contiene información básica requerida para todas las operaciones de flujo de entrada/salida.
f) Las funciones miembro _____, y _____ pueden ser utilizadas para activar y reactivar banderas de estado de formato.
g) El archivo de cabecera _____ contiene información para llevar a cabo el formato "en memoria".
h) Al utilizar los manipuladores parametrizados, se debe incluir en el programa el archivo de cabecera _____.
i) El archivo de cabecera _____ contiene información para llevar a cabo procesamiento de archivos controlados por usuario.
j) El manipulador de flujo _____ inserta un carácter de nueva línea en el flujo de salida y vacía el flujo de salida.
k) El archivo de cabecera _____ contiene información para aquellos programas que mezclan entradas y salidas en estilo C y en estilo C++.
l) La función miembro **ostream** _____ se utiliza para llevar a cabo salidas sin formato.
m) Las operaciones de entrada son soportadas por la clase _____.
n) Las salidas al flujo de error estándar son dirigidas ya sea por el objeto de flujo _____ o _____.
o) Las operaciones de salida están soportadas por la clase _____.
p) El símbolo para el operador de inserción de flujo es _____.

q) Los cuatro objetos que corresponden a los dispositivos estándar en el sistema incluyen _____, _____, _____, y _____.
r) El símbolo para operador de extracción de flujo es _____.
s) Los manipuladores de flujo _____, _____, y _____ se utilizan para especificar que los enteros deben ser mostrados en formatos octal, hexadecimal y decimal, respectivamente.
t) La precisión por omisión para mostrar valores en punto flotante es _____.
u) Cuando está activa, la bandera _____ hace que los números positivos numéricos aparezcan con un signo más.

21.2 Indique si lo que sigue es verdadero o falso. Si la respuesta es falsa, explique por qué.
a) La función miembro de flujo `flags()` con un argumento long, define a su argumento a la variable de estado `flags` y devuelve su valor anterior.
b) El operador de inserción de flujo << y el operador de extracción de flujo >> son homónimos para manejar todos los tipos de datos estándar —incluyendo cadenas y direcciones de memoria, así como todos los tipos de datos definidos por usuario.
c) La función miembro de flujo `flags()` sin argumentos, restaura todos los bits de bandera en la variable de estado `flags`.
d) Se puede hacer la homonimia del operador de extracción de flujo >> con una función operador que toma como argumentos una referencia `istream` y una referencia a un tipo definido por usuario, y devuelve una referencia `istream`.
e) El manipulador de flujo `ws` pasa por alto el espacio en blanco a la izquierda de un flujo de entrada.
f) Se puede hacer la homonimia del operador de inserción de flujo << con una función operador que toma como referencia una referencia a `istream` y una referencia a un tipo definido por usuario y devuelve una referencia `istream`.
g) La entrada con el operador de extracción de flujo >> siempre pasará por alto los caracteres de espacio en blanco del flujo de entrada.
h) Las características de entrada y de salida son parte de C++.
i) La función miembro de flujo `rdstate()` devuelve el estado del flujo actual.
j) El flujo `cout` por lo general, está conectado a la pantalla de exhibición.
k) La función miembro de flujo `good()` devuelve verdadero si las funciones miembro `bad()`, `fail()` y `eof()` todas ellas devuelven falso.
l) El flujo `cin` por lo regular está conectado a la pantalla de exhibición.
m) Si ocurre un error no recuperable durante una operación de flujo, la función miembro `bad()` devolverá verdadero.
n) La salida a `cerr` no tiene búfer y la salida a `clog` sí lo tiene.
o) Cuando está activa la bandera `ios::showpoint`, los valores en punto flotante están forzados a imprimirse con los seis dígitos de precisión por omisión —siempre y cuando el valor de la precisión no haya sido modificado, en cuyo caso los valores de punto flotante se imprimirán con la precisión especificada.
p) La función miembro `ostream` de nombre `put` extrae el número especificado de caracteres.
q) Los manipuladores de flujo `dec`, `oct` y `hex` solamente afectan la siguiente operación de salida de enteros.
r) Al ser extraído, por omisión las direcciones de memoria se exhiben como enteros `long`.

21.3 Escriba un solo enunciado que ejecute la tarea indicada para cada uno de los siguientes.
a) Extraiga la cadena `"Enter your name: "`.
b) Active una bandera para que el exponente en notación científica y las letras en los valores hexadecimales se impriman en mayúsculas.
c) Extraiga la dirección de la variable `string` del tipo `char *`.
d) Active una bandera, de tal forma que los valores de punto flotante se impriman en notación científica.

e) Extraiga la dirección de la variable `integerPtr` del tipo `int *`.
f) Active una bandera, de tal forma que cuando valores enteros sean extraídos se exhiban la base entera para valores octal y hexadecimal.
g) Extraiga el valor al cual apunta `floatPtr` del tipo `float *`.
h) Utilice una función miembro de flujo para definir el carácter de llenado como `'*'` para imprimir en anchos de campo mayores que los valores siendo extraídos. Escriba un enunciado por separado para ello mediante un manipulador de flujo.
i) Extraiga los caracteres `'O'` y `'K'` en un enunciado mediante la función `put` de `ostream`.
j) Obtenga el siguiente carácter en el flujo de entrada sin extraerlo del mismo flujo.
k) Introduzca un solo carácter en la variable `c` de tipo `char` utilizando la función miembro `istream get` de dos formas distintas.
l) Introduzca y descarte los siguientes seis caracteres del flujo de entrada.
m) Utilice la función miembro `read` de `istream` para introducir 50 caracteres al arreglo `line` del tipo `char`.
n) Lea 10 caracteres en el arreglo de caracteres `name`. Si se encuentra el delimitador `'.'` detenga la lectura de caracteres. No retire el delimitador del flujo de entrada. Escriba otro enunciado que lleve a cabo esa tarea y elimine el delimitador del flujo de entrada.
o) Utilice la función miembro `gcount` de `istream` para determinar el número de caracteres introducidos en el arreglo de caracteres `line` por la última llamada a la función miembro `read` de `istream` y extraiga dicho número de caracteres utilizando la función miembro `write` de `ostream`.
p) Escriba enunciados por separado para vaciar el flujo de salida utilizando una función miembro y un manipulador de flujo.
q) Extraiga los siguientes valores: `124`, `18.376`, `'Z'`, `1000000`, y `"String"`.
r) Imprima el ajuste de precisión actual utilizando una función miembro.
s) Introduzca un valor entero en la variable `months` de `int` y un valor de punto flotante en la variable `percentageRate` de `float`.
t) Imprima `1.92`, `1.925`, y `1.9258` con `3` dígitos de precisión utilizando un manipulador.
u) Imprima el entero `100` en base octal, hexadecimal y decimal utilizando manipuladores de flujo.
v) Imprima el entero `100` en base decimal, octal y hexadecimal utilizando un solo manipulador de flujo para modificar la base.
w) Imprima `1234` justificado a la derecha en un campo de `10` dígitos.
x) Lea los caracteres al arreglo de caracteres `line` hasta que se encuentre con el carácter `'z'` hasta un límite de `20` caracteres (incluyendo el carácter de terminación null). No extraiga el carácter del delimitador del flujo.
y) Utilice las variables enteras `x` e `y` para especificar el ancho de campo y la precisión utilizada para mostrar el valor `87.4573` de tipo `double` y exhiban dicho valor.

21.4 Identifique el error en cada uno de los enunciados siguientes y explique cómo corregirlo.
a) `cout << "Value of x <= y is: " << x <= y;`
b) El siguiente enunciado debe de imprimir el valor entero de `'c'`
```
cout << 'c';
```
c) `cout << ""A string in quotes"";`

21.5 Muestre la salida para cada uno de los siguientes:
a)
```
cout << "12345\n";
cout.width(5);
cout.fill('*');
cout << 123 << '\n' << 123;
```
b) `cout << setw(10) << setfill('$') << 10000;`
c) `cout << setw(8) << setprecision(3) << 1024.987654;`

```
d) cout << setiosflags (ios::showbase) << oct << 99
   << '\n' << hex 99;
e) cout << 100000 << '\n'
   << setiosflags (ios::showpos) << 100000;
f) cout << setw(10) << setprecision(2) <<
   << setiosflags(ios::scientific) << 444.93738;
```

Respuestas a los ejercicios de autoevaluación

21.1 a) `friend`. b) `ios::left`, `ios::right`, y `ios::internal`. c) flujos. d) `setiosflags`, `resetiosflags`. e) `iostream.h`. f) `setf`, `unsetf`. g) `strstream.h`. h) `iomanip.h`. i) `fstream.h`. j) `endl`. k) `stdiostream.h`. l) `write`. m) `istream`. n) `cerr` o `clog`. o) `ostream`. p) `<<`. q) `cin`, `cout`, `cerr`, y `clog`. r) `>>`. s) `oct`, `hex`, `dec`. t) seis dígitos de precisión. u) `ios::showpos`.

21.2 a) Verdadero.

b) Falso. No se hace la homonimia de los operadores de extracción y de inserción de flujo para todos los tipos definidos por usuario. El programa de una clase debe proporcionar en forma específica las funciones de operador homónimas para hacer la homonimia de los operadores de flujo para uso con cada uno de los tipos definidos por usuario.

c) Falso. La función miembro de flujo `flags()` sin argumento, sólo devuelve el valor actual de la variable de estado `flags`.

d) Verdadero.

e) Verdadero.

f) Falso. Para hacer la homonimia del operador de inserción de flujo `<<`, la función de operador homónima debe tomar como argumentos una referencia `ostream` y una referencia a un tipo definido por usuario, y devolver una referencia `ostream`.

g) Verdadero. A menos de que `ios::skipws` esté desactivado.

h) Falso. Las características de entrada y de salida de C++ se proporcionan como parte de la biblioteca estándar de C++. El lenguaje de C++ no contiene capacidades para el procesamiento de entradas, salidas o de archivos.

i) Verdadero.

j) Verdadero.

k) Verdadero.

l) Falso. El flujo `cin` está conectado a la entrada estándar de la computadora, que por lo regular es el teclado.

m) Verdadero.

n) Verdadero.

o) Verdadero.

p) Falso. La función miembro `put` de `ostream` extrae su único argumento de carácter.

q) Falso. Los manipuladores de flujo `dec`, `oct` y `hex` establecen el estado de formato de salida para enteros a la base especificada, hasta que ésta sea otra vez específicamente modificada o el programa se termine.

r) Falso. Por omisión las direcciones de memoria se exhiben en formato hexadecimal. Para mostrar las direcciones como enteros `long`, la dirección deberá convertirse explícitamente (cast) a un valor `long`.

21.3 a)
```
cout << "Enter your name: ";
```
b)
```
cout.setf(ios::uppercase);
```
c)
```
cout << long(string);
```
d)
```
cout.setf(ios::scientific, ios::floatfield);
```
e)
```
cout << integerPtr;
```
f)
```
cout << setiosflags(ios::showbase);
```


g)
```
cout << *floatPtr;
```
h)
```
cout.fill('*');
    cout << setfill('*');
```
i)
```
cout.put('0').put('K');
```
j)
```
cin.peek();
```
k)
```
c = cin.get();
cin.get(c);
```
l)
```
cin.ignore(6);
```
m)
```
cin.read(line,50);
```
n)
```
cin.get(name, 10, '.');
    cin.getline(name, 10, '.');
```
o)
```
cout.write(line, cin.gcount());
```
p)
```
cout.flush();
    cout << flush;
```
q)
```
cout << 124 << 18.376 << 'Z' << 1000000 << "String";
```
r)
```
cout.precision();
```
s)
```
cin >> months >> percentageRate;
```
t)
```
cout << setprecision(3) << 1.92 << '\t'
    << 1.925 << '\t' << 1.9258;
```
u)
```
cout << oct << 100 << hex << 100 << dec << 100;
```
v)
```
cout << 100 << setbase(8) << 100 << setbase(16) << 100;
```
w)
```
cout << setw(10) << 1234;
```
x)
```
cin.get(line, 20, 'z');
```
y)
```
cout << setw(x) << setprecision(y) << 87.4573;
```

21.4 a) Error: es mayor la precedencia del operador `<<` que la del operador `<=`, lo que hace que se evalúe en forma incorrecta el enunciado y causará que se emita un error por el compilador.
Corrección: para corregir el enunciado, añada paréntesis encerrando la expresión `x <= y`. Este problema ocurrirá con cualquier expresión que utilice operadores de una precedencia menor que el operador `<<`, si la expresión no se encierra entre paréntesis.

b) Error: en C++, los caracteres no se tratan como pequeños enteros como es el caso en C.
Corrección: imprimir el valor numérico de un carácter en el conjunto de caracteres de la computadora, el carácter deberá ser convertido explícitamente (cast) a un valor entero como en el siguiente:

```
cout << int('c');
```

c) Error: los caracteres entre comillas no pueden ser impresos como una cadena, a menos de que se utilice una secuencia de escape.
Corrección: imprima la cadena en alguna de las siguientes formas:

```
cout << '"' << "A string in quotes" << '"';
    cout << "\"A string in quotes\"";
```

21.5 a)
```
12345
**123
123
```
b)
```
$$$$$10000
```
c)
```
1024.988
```
d)
```
0143
0x63
```
e)
```
100000
+100000
```
f)
```
4.45e+02
```

Ejercicios

21.6 Escriba un enunciado para cada uno de los siguientes:

a) Imprima el entero `40000` justificado a la izquierda en un campo de `15` dígitos.
b) Lea una cadena a la variable de arreglo de caracteres `state`.
c) Imprima `200` con y sin signo.
d) Imprima `100` en forma hexadecimal precedido por `0x`.
e) Lea caracteres al arreglo `s` hasta que se encuentre con el carácter `'p'` hasta un límite de 10 caracteres (incluyendo el carácter nulo de terminación). Extraiga el delimitador del flujo de entrada y descártelo.
f) Imprima `1.234` en un campo de `9` dígitos con ceros a la izquierda.
g) Lea un carácter de la forma `"characters"` de la entrada estándar. Almacene la cadena en el arreglo de caracteres `s`. Elimine las comillas del flujo de entrada. Lea un máximo de 50 caracteres (incluyendo el carácter nulo de terminación).

21.7 Escriba un programa para probar los valores enteros de entrada en formato decimal, octal y hexadecimal. Extraiga cada entero leído por el programa en los tres formatos. Pruebe el programa con los siguientes datos de entrada: 10, 010, 0x10.

21.8 Escriba un programa que imprima los valores de apuntador utilizando conversiones explícitas (cast) para todos los tipos de datos enteros. ¿Cuáles son los que imprimen valores raros? ¿Cuáles causan errores?

21.9 Escriba un programa para probar los resultados de imprimir el valor entero `12345` y el valor de punto flotante `1.2345` en campos de varios tamaños. ¿Qué ocurre cuando los valores se imprimen en campos que contienen menos dígitos que los valores?.

21.10 Escriba un programa que imprima el valor `100.453627` redondeado al siguiente dígito, décimo, centésimo, milésimo y diezmilésimo.

21.11 Escriba un programa que introduzca una cadena desde el teclado y que determine la longitud de dicha cadena. Imprima la cadena utilizando como ancho de campo el doble de su longitud.

21.12 Escriba un programa que convierta temperaturas enteras Fahrenheit desde `0` hasta `212` grados a temperaturas Celsius en punto flotante con `3` dígitos de precisión. Utilice la fórmula.

```
celsius = 5.0/9.0 * (fahrenheit - 32);
```

para efectuar el cálculo. La salida deberá ser impresa en dos columnas justificadas a la derecha, y las temperaturas Celsius deberán estar precedidas por un signo, tanto para valores positivos como negativos.

21.13 En algunos lenguajes de programación, las cadenas se introducen encerradas ya sea entre comillas sencillas o dobles. Escriba un programa que lea tres cadenas `suzy`, `"suzy"`, y `'suzy'`. Las comillas sencillas y dobles ¿son ignoradas o son leídas como parte de la cadena?

21.14 En la figura 21.30 los operadores de extracción y de inserción de flujo son homónimos para la entrada y salida de objetos `PhoneNumber`. Vuelva a escribir el operador de extracción de flujo para llevar a cabo las siguientes tareas de verificación de error a la entrada. Note que la función `operator>>` necesitará recodificarse totalmente.

a) Introduzca en un arreglo la totalidad del número telefónico. Pruebe que han sido introducidos todos los números telefónicos (debe haber un total de 14 caracteres para un número telefónico de la forma `(800) 555-1212`). Use la función miembro de flujo `clear` para definir el bit `ios::fail` para una entrada incorrecta.
b) En un número telefónico el código de área y la centralilla no empiezan con `0` o con `1`. Pruebe el primer dígito de las porciones del código de área y de centralilla del número telefónico, para estar seguro que ninguno empieza con `0` o con `1`. Utilice la función miembro de flujo `clear` para definir el bit `ios::fail` en caso de entrada incorrecta.

c) El dígito intermedio de un código de área es siempre `0` ó `1`. Pruebe el dígito intermedio de esta porción buscando un valor de `0` ó `1`. Utilice la función miembro de flujo `clear` para definir el bit `ios::fail` en caso de entrada incorrecta. Si ninguna de las anteriores operaciones resulta en la activación del bit `ios::fail` debido a entrada incorrecta, copie las tres porciones del número telefónico en los miembros `areaCode`, `exchange`, y `line` del objeto `PhoneNumber`. En el programa principal, si el bit `ios::fail` ha sido activado en la entrada, haga que el programa imprima un mensaje de error y termine antes de imprimir el número telefónico.

21.15 Escriba un programa que lleve a cabo cada uno de los siguientes:

a) Crear una clase definida por usuario de nombre `point` que contenga los miembros de dato enteros privados `xCoordinate` y `yCoordinate`, y declarar las funciones de operador homónimas de extracción y de inserción de flujo como `friend`s de la clase.
b) Defina las funciones de operador de inserción y de extracción de flujo. La función de operador de extracción de flujo debe determinar si los datos introducidos son válidos, de lo contrario, deberá activar `ios::failbit` para indicar entrada incorrecta. Una vez que haya ocurrido el error de entrada el operador de inserción de flujo no deberá ser capaz de mostrar el punto.
c) Escriba una función `main` que pruebe la entrada y la salida de la clase definida por usuario `point` utilizando los operadores de inserción y de extracción de flujo homónimos.

21.16 Escriba un programa que lleve a cabo cada uno de los siguientes:

a) Crear la clase definida por usuario `complex`, que contenga los miembros de dato enteros privados `real` e `imaginary`, y que declare las funciones de operador de extracción y de inserción de flujo homónimas como `friend`s de la clase.
b) Defina las funciones de operador o de extracción y de inserción de flujo. La función de operador de extracción de flujo deberá determinar si los datos introducidos son datos válidos, y de lo contrario, deberá activar `ios::failbit` para indicar entrada incorrecta. La entrada deberá ser de la forma

```
3 + 8i
```

Los valores pueden ser negativos o positivos, y es posible que alguno de los dos valores no sea proporcionado. Si uno de ellos no es proporcionado, el miembro de datos apropiado deberá ser definido como `0`. Si ha ocurrido un error de entrada el operador de inserción de flujo no debe ser capaz de mostrar el punto. El formato de salida debe ser idéntico al formato de entrada que se muestra arriba. Para los valores imaginarios negativos, deberá imprimirse un signo menos en vez de un signo más.
c) Escriba una función `main` que pruebe la entrada y la salida de la clase definida por usuario `complex` utilizando los operadores de inserción y de extracción de flujo homónimos.

21.17 Escriba un programa que utilice una estructura `for` para imprimir una tabla de valores ASCII para los caracteres en el conjunto de caracteres ASCII desde `33` hasta `126`. El programa deberá imprimir el valor decimal, el valor octal, el hexadecimal y el valor de carácter de cada uno de los caracteres. Utilice los manipuladores de flujo `dec`, `oct` y `hex` para imprimir los valores enteros.

21.18 Escriba un programa para demostrar que las funciones miembro istream de nombre `getline` y `get` de tres argumentos cada una de ellas termina la cadena de entrada con un carácter nulo de terminación de cadena. También demuestre que `get` deja el carácter delimitador en el flujo de entrada, en tanto que `getline` extrae el carácter delimitador y lo descarta . ¿Qué ocurre con los caracteres no leídos dentro del flujo?

21.19 Escriba un programa que cree el manipulador definido por usuario `skipwhite` para pasar por alto los caracteres de espacio en blanco a la izquierda en el flujo de entrada. El manipulador deberá utilizar la función `isspace` de la biblioteca `ctype.h` para probar si el carácter es un carácter de espacio en blanco. Cada carácter deberá ser introducido utilizando la función miembro `get` de `istream`. Cuando se encuentre

con un carácter distinto a un espacio en blanco, el manipulador **skipwhite** termina su trabajo colocando el carácter de regreso en el flujo de entrada y devolviendo una referencia **istream**.

Pruebe el manipulador definido por usuario creando una función **main** en el cual quede la bandera **ios::skipws** desactivada de tal forma que el operador de extracción de flujo no se salte en forma automática los espacios en blanco. A continuación pruebe el manipulador sobre el flujo de entrada introduciendo un carácter precedido por un espacio en blanco como entrada. Imprima el carácter que fue introducido a fin de confirmar que el carácter de espacio en blanco no fue introducido.

Bibliografía

(A193) Allison, C., "Code Capsules: A C++ Date Class, Parte I" *The C Users Journal,* Vol 11, No. 2, Febrero 1993, pp. 123-131.

(An92) Anderson, A. E., y W. J. Heinze, *C++ Programming and Fundamental Concepts,* Englewood Cliffs, NJ: Prentice Hall, 1992.

(Ba93) Bar-David, T., *Object-Oriented Design for C++*, Englewood Cliffs, NJ: Prentice Hall, 1993.

(Be93) Berard, E. V., Essays on Object-Oriented Software Engineering: Volumen I, Englewood Cliffs, NJ: Prentice Hall, 1993.

(Br91) Borland, *Borland C++ Programmer's Guide*, Parte No. 14MN-TCP04, Scotts Valley, CA: Borland International, Inc., 1991.

(Br91a) Borland, *Borland C++ Getting Started*, Parte No. 14MN-TCP02, Scotts Valley, CA: Borland International, Inc., 1991.

(Br91b) Borland, *Borland C++ 3.0 Programmers Guide*, Scotts Valley, CA: Borland International, Inc., 1991.

(By93) Byron, D., "The Case for Object Technology Standards," *CASE Trends,* Septiembre de 1993, pp. 22-26.

(Co93) Computerworld, "The CW Guide to Object-Oriented Programming" *Computerworld*, Junio 14, 1993, pp. 107-130.

(De90) Deitel, H. M., *Operating Systems*, Second Edition, Reading, MA: Addison-Wesley, 1990.

(El90) Ellis, M: A. y B. Stroustrup, *The Annotated C++ Reference Manual*, Reading, MA: Addison-Wesley, 1990.

(Fl93) Flaming, B., *Practical Data Structures in C++*, John Wiley & Sons, 1993.

(Ha93) Hagan, T., "C++ Class Libraries for GUIs," *Open Systems Today*, Febrero 15, 1993, pp. 54, 58.

(Ja93) Jacobson, I., "Is Object Technology Software's Industrial Platform?" *IEEE Software Magazine*, Vol. 10, No. 1, Enero de 1993, pp. 24-30.

(Ko93) Kozaczynski, W., y A. Kuntzmann-Combelles, "What It Takes to Make OO Work," *IEEE Software Magazine*, Vol.10, No.1, Enero 1993, pp. 20-23.

(Li91) Lippman, S. B., *C++ Primer* (Second Edition). Reading, MA: Addison-Wesley Publishing Company, 1991.

(Lo93) Lorenz, M., *Object-Oriented Software Development: A Practical Guide*, Englewood Cliffs, NJ: Prentice Hall, 1993.

(Lu92) Lucas, P. J., *The C++ Programmer's Handbook*, Englewood Cliffs, NJ: Prentice Hall, 1992.

(Ma93) Martin, J., *Principles of Object-Oriented Analysis and Design*, Englewood Cliffs, NJ: Prentice Hall, 1993.

(Me93) Matsche, J.J., "Object-oriented programming in Standard C," *Object Magazine*, Vol.2, No.5, Enero/Febrero 1993, pp. 71-74.

(Mi91) Microsoft, *Microsoft C/C++ Class Libraries Reference* (Versión 7.0), Redmond, WA: Microsoft Corporation, 1991.

(Mi91a) Microsoft, *Microsoft C/C++ C++ Language Reference* (Versión 7.0), Redmond, WA: Microsoft Corporation, 1991.

(Mi91b) Microsoft, *Microsoft C/C++ C++ Tutorial* (Versión 7.0), Redmond, WA: Microsoft Corporation, 1991.

(Pi93) Pittman, M., "Lessons Learned in Managing Object-Oriented Development," *IEEE Software Magazine*, Vol.10, No.1, Enero 1993, pp 43-53.

(Pr93) Prieto-Diaz, R., "Status Report: Software Reusability," *IEEE Software*, Vol.10, No.3, Mayo 1993, pp. 61-66.

(Ra92) Ranade, J., y S. Zamir, C++ *Primer for C Programmers,* New York, NY McGraw-Hill, Inc., 1992.

(Re91) Reed, D.R., "Moving from C to C++," *Object Magazine,* Vol.1, No.3, Septiembre/Octubre, 1991, pp. 46-60.

(Rl93) Rettig, M., G.Simmons, y J. Thompson, "Extended Objects," *Communications of the ACM,* Vol.36, No.8, Agosto 1993, pp. 19-24.

(Sa93) Saks, D., "Inheritance, Parte 2," *The C Users Journal,* Mayo 1993, pp.81-89.

(Sh91) Shiffman, H., "C++ Object-Oriented Extensions to C," *SunWorld,* Vol.4, No.5, Mayo de 1991, pp.63-70.

(Sk93) Skelly, C., "Pointer Power in C and C++, Parte 1," *The C Users Journal,* Vol.11, No.2, Febrero 1993, pp. 93-98.

(Sn93) Snyder, A., "The Essence of Objects: Concepts and Terms," *IEEE Software Magazine,* Vol.10, No.1, Enero 1993, pp. 31-42.

(St91) Stroustrup, B., *The C++ Programming Language* (Second Edition), Reading, MA: Addison-Wesley Series in Computer Science, 1991.

(St93) Stroustrup, B., "Why Consider Language Extensions?": Maintaining a Delicate Balance," *C++ Report,* Septiembre 1993, pp. 44-51.

(Vo93) Voss, G., "Objects and Messages," *Windows Tech Journal,* Febrero de 1993, pp. 15-16.

(Wi93) Wiebel, M., y S. Halladay, "Using OOP Techniques Instead of *switch* in C++," Vol.10, No.10, *The C Users Journal,* Octubre de 1992, pp. 105-106,

(Wl93) Wilde, N., P. Matthews, y R Huitt, "Maintaining Object-Oriented Software" *IEEE Software Magazine,* Vol.10, No.1, Enero 1993, pp. 75-80.

(Wt93) Wilt, N., "Templates in C++," *The C Users Journal,* Mayo de 1993, pp. 33-51.

Apéndice A *

Sintaxis de C

En la notación sintáctica utilizada, las categorías sintácticas (no terminales) se indican en *cursivas* y las palabras literales y los miembros de conjuntos de caracteres (terminales) en **negrillas**. Dos puntos después de una terminal introducen su definición. Las definiciones alternas son enlistadas en líneas por separado, excepto cuando estén antecedidas por las palabras "una de". Un símbolo opcional es presentado mediante el subíndice "opt", de tal forma que

{*expresión*$_{opt}$ }

indica una expresión opcional encerrada entre llaves.

Resumen de sintaxis de lenguaje

A.1 Gramática lexicográfica

A.1.1 Componentes léxicos (tokens)

Componente léxico:
- *palabra reservada*
- *identificador*
- *constante*
- *cadena literal*
- *operador*
- *puntuaciones*

componente léxico de preprocesamiento:
- *nombre de encabezado*
- *identificador*
- *número de preprocesador*
- *constante de carácter*
- *cadena literal*
- *operador*

* Certificación de autorización: Este material ha sido condensado y adaptado partiendo de American National Standard for International —Systems Programming Language— C, ANSI/ISO 9899:1990. Se pueden adquirir copias de esta norma de American National Standards Institute en 11 West 42nd. Street, New York, NY 10036.

puntuaciones
cada carácter distinto a un espacio en blanco que no pueda ser uno de los arriba citados

A.1.2 Palabras reservadas

palabra reservada: una de

`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

A.1.3 Identificadores

identificador:
- *no dígito*
- *identificador no dígito*
- *identificador dígito*

no dígito: uno de:

`_ a b c d e f g h i j k l m`
`n o p q r s t u v w x y z`
`A B C D E F G H I J K L M`
`N O P Q R S T U V W X Y Z`

dígito: uno de
`0 1 2 3 4 5 6 7 8 9`

A.1.4 Constantes

constante:
- *constante flotante*
- *constante entero*
- *constante de enumeración*
- *constante de carácter*

constante flotante:
- *constante fraccionaria sufijo*$_{opt}$ *flotante parte*$_{opt}$ *exponencial*
- *secuencia de dígitos sufijo*$_{opt}$ *flotante parte exponencial*

constante fraccionaria:
- *secuencia*$_{opt}$ *de dígitos* . *secuencia de dígitos*
- *secuencia de dígitos*.

parte exponencial:
- *signo*$_{opt}$ **e** *secuencia de dígitos*
- *signo*$_{opt}$ **E** *secuencia*$_{opt}$ *de dígitos*

signo: uno de
`+ -`

secuencia de dígitos
    dígito
    secuencia de dígitos dígito

sufijo flotante: uno de
    `f l F L`

constante entera
    constante decimal sufijo$_{opt}$ *entero*
    constante octal sufijo$_{opt}$ *entero*
    constante hexadecimal sufijo$_{opt}$ *entero*

constante decimal:
    dígito no cero
    constante decimal dígito

constante octal:
    `0`
    constante octal dígito octal

constante hexadecimal:
    dígito *hexadecimal* `0x`
    dígito *hexadecimal* `0X`
    constante hexadecimal dígito hexadecimal

dígito no cero: uno de
    `1 2 3 4 5 6 7 8 9`

dígito octal: uno de
    `0 1 2 3 4 5 6 7`

dígito hexadecimal: uno de
    `0 1 2 3 4 5 6 7 8 9`
    `a b c d e f`
    `A B C D E F`

sufijo entero:
    sufijo unsignado sufijo$_{opt}$ *long*
    sufijo long sufijo$_{opt}$ *unsignado*

sufijo unsignado: uno de
    `u U`

sufijo long: uno de
    `l L`

constante de enumeración:
    identificador

constante de carácter:
    'secuencia c-char'
    'secuencia c-char' `L`

secuencia c-char:
    c-char
    secuencia c-char c-char

c-char:
    cualquier miembro del conjunto de caracteres fuente, a excepción de la comilla sencilla `'`, la diagonal invertida \, o el carácter de nueva línea
    secuencia de escape

secuencia de escape:
    secuencia de escape simple
    secuencia de escape octal
    *secuencia de escape hexadecim*al

secuencia de escape simple: una de
    `\'  \"  \?  \\`
    `\a  \b  \f  \n  \r  \t  \v`

secuencia de escape octal:
    dígito octal \
    dígito octal dígito octal \
    dígito octal dígito octal dígito octal \

secuencia de escape hexadecimal:
    dígito hexadecimal `\x`
    secuencia de escape hexadecimal dígito hexadecimal

A.1.5 Cadenas literales

cadena literal:
    "secuencia$_{opt}$ *s-char"*
    "secuencia$_{opt}$ *s-char"* `L`

secuencia s-char:
    s-char
    secuencia s-char s-char

s-char:
    cualquier miembro del conjunto de caracteres fuente, a excepción de las dobles comillas `"`, la diagonal invertida \ o el carácter de nueva línea
    secuencia de escape

A.1.6 Operadores

operador: uno de

```
[ ]   ( )   .     ->
++    --    &     *     +     -     ~     !     sizeof
/     %     <<    >>    <     >     <=    >=    ==    !=    ^     |     &&    ||
?     :
=     *=    /=    %=    +=    -=    <<=   >>=   &=    ^=    |=
,     #     ##
```

A.1.7 Puntuaciones

puntuaciones: uno de

```
[ ]   ( )   { }   *   ,   :   =     ...   #
```

A.1.8 Nombres de encabezados

nombre de encabezado:
<h-char-sequence>
"secuencia q-char"

secuencia h-char:
h-char
secuencia h-char h-char

h-char:
cualquier miembro del conjunto de caracteres fuente, a excepción del carácter de nueva línea, y de >

secuencia q-char:
q-char
secuencia q-char q-char

q-char:
cualquier miembro del conjunto de caracteres fuente, a excepción del carácter de nueva línea, y de "

A.1.9 Números de preprocesador

número de preprocesador:
dígito
dígito .
dígito de número de preprocesador
no dígito de número de preprocesador
signo e *de número de preprocesador*
signo E *de número de preprocesador*
. de número de preprocesador

A.2. Gramática estructural de frases

A.2.1 Expresiones

expresión primaria:
identificador
constante
cadena literal
(*expresión*)

expresión posfija:
expresión primaria
expresión posfija [*expresión*]
expresión posfija (*lista*$_{opt}$ *de expresión de argumentos*)
expresión posfija identificador .
expresión posfija identificador ->
expresión posfija ++
expresión posfija --

lista de expresión de argumentos:
expresión de asignación
lista de expresión de argumentos , *expresión de asignación*

expresión unaria:
expresión posfija
expresión unaria ++
expresión unaria --
expresión de operador unario de conversión explícita (cast)
expresión unaria `sizeof`
`sizeof` (*nombre de tipo*)

operador unario: uno de
`&` `*` `+` `-` `~` `!`

expresión de conversión explícita (cast):
expresión unaria
expresión de conversión explícita (cast) (*nombre de tipo*)

expresión multiplicativa:
expresión de conversión explícita (cast)
expresión multiplicativa expresión de conversión explícita (cast) *
expresión multiplicativa expresión de conversión explícita (cast) /
expresión multiplicativa expresión de conversión explícita (cast) %

expresión aditiva:
expresión multiplicativa
expresión aditiva expresión multiplicativa +
expresión aditiva expresión multiplicativa -

expresión de desplazamiento:
expresión aditiva
expresión de desplazamiento expresión aditiva <<
expresión de desplazamiento expresión aditiva >>

expresión relacional:
expresión de desplazamiento
expresión de desplazamiento expresión relacional <
expresión de desplazamiento expresión relacional >

expresión relacional expresión de desplazamiento <=
expresión relacional expresión de desplazamiento >=

expresión de igualdad:
expresión relacional
expresión de igualdad expresión relacional ==
expresión de igualdad expresión relacional !=

expresión AND:
expresión de igualdad
expresión de igualdad &

expresión OR exclusivo:
expresión AND
expresión OR exclusivo expresión AND ^

expresión OR inclusivo:
expresión OR exclusivo
expresión OR inclusivo expresión OR exclusivo |

expresión AND lógica:
expresión OR inclusivo
expresión AND lógica expresión OR inclusivo &&

expresión OR lógica:
expresión AND lógica
expresión OR lógica expresión AND lógica ||

expresión condicional:
expresión OR lógica
expresión OR lógica expresión ? *expresión condicional* :

expresión de asignación:
expresión condicional
expresión de asignación unaria expresión de asignación operador

operador de asignación: uno de
= *= /= %= += -= <<= >>= &= ^= |=

expresión:
expresión condicional
expresión expresión condicional ,

expresión constante:
expresión condicional

A.2.2 Declaraciones

declaración:
declaración de especificadores de inicialización $lista_{opt}$;

declaración de especificadores:
declaración de especificadores de clase de almacenamiento $especificadores_{opt}$
declaración de especificadores de tipo $especificadores_{opt}$
declaración de calificadores de tipo de $especificadores_{opt}$

lista de declarador de inicialización:
declarador de inicialización
lista de declarador de inicialización declarador de inicialización ,

declarador Init:
declarador
declarador = *inicializador*

especificador de clase de almacenamiento:
typedef
extern
static
auto
register

especificador de tipo:
void
char
short
int
long
float
double
signed
unsigned
especificador de estructuras o uniones
especificador de enumeraciones
nombre typedef

especificador de estructura o unión:
identificador de estructura o unión$_{opt}$ {*lista de declaración de estructura*}
identificador de estructura o unión

estructura o unión:
struct
union

lista de declaración de estructura:
declaración de estructura
lista de declaración de estructura declaración de estructura

declaración de estructura:
lista de calificadores de especificación lista de declaración de estructura ;

lista de calificadores de especificación:
especificador de especificadores de tipo lista de calificadores$_{opt}$
especificador de calificadores de tipo lista de calificadores$_{opt}$

lista de declarador de estructura:
declarador de estructura
lista de declarador de estructura declarador de estructura ,

declarador de estructura:
declarador
declarador$_{opt}$: *expresión constante*

especificador de enumeraciones:
identificador$_{opt}$ **enum** { *lista de enumerador* }
identificador **enum**

lista de enumerador:
enumerador
lista de enumerador enumerador ,

enumerador:
constante de enumeración
constante de enumeración expresión constante =

calificador de tipo:
const
volatile

declarador:
apuntador$_{opt}$ declarador directo

declarador directo:
identificador
(*declarador*)
declarador directo [*expresión$_{opt}$constante*]
declarador directo (*lista de parámetros de tipo*)
declarador directo (*lista$_{opt}$ de identificadores*)

apuntador:
lista$_{opt}$ *de calificadores de tipo* *
*lista$_{opt}$ de calificadores de tipo * apuntador*

lista de calificadores de tipo
calificadores de tipo
lista de calificadores de tipo calificador de tipo

lista de tipo de parámetros:
lista de parámetros
lista de parámetros , ...

lista de parámetros:
declaración de parámetros
lista de parámetros , *declaración de parámetros*

declaración de parámetros:
especificador de declaración declarador
especificador de declaración declarador abstracto$_{opt}$

lista de identificadores:
identificador
lista de identificadores , *identificador*

nombre de tipo:
lista de calificador de especificador declarador$_{opt}$ abstracto

declarador abstracto:
apuntador
apuntador$_{opt}$ declarador abstracto directo

declarador abstracto directo:
(declarador abstracto)
declarador abstracto directo$_{opt}$ [*expresión$_{opt}$constante*]
declarador abstracto directo$_{opt}$ (*lista de tipo de parámetros$_{opt}$*)

nombre typedef:
identificador

inicializador:
expresión de asignación
{ *lista de inicializador* }
{ *lista de inicializador* , }

lista de inicializador:
inicializador
lista de inicializador , *inicializador*

A.2.3 Enunciados

enunciado:
enunciado etiquetado
enunciado compuesto
enunciado de expresión
enunciado de selección
enunciado de iteración
enunciado de salto

enunciado etiquetado:
identificador : *enunciado*
expresión constante **case** *expresión* :
enunciado: **default**

enunciado compuesto:
`{` *lista*$_{opt}$ *de declaración lista*$_{opt}$ *de enunciado* `}`

lista de declaración:
declaración
declaración lista de declaración

lista de enunciado:
enunciado
enunciado lista de enunciado

enunciado de expresión:
expresión$_{opt}$ `;`

enunciado de selección:
enunciado `(` *expresión* `)` `if`
enunciado `(` *expresión* `)` `if` *enunciado* `else`
enunciado `(` *expresión* `)` `switch`

enunciado de iteración:
enunciado `(` *expresión* `)` `while`
enunciado `do` `(` *expresión* `)` `while` `;`
enunciado `(`*expresión*$_{opt}$ `;` *expresión*$_{opt}$ `;` *expresión*$_{opt}$`)` `for`

enunciado de salto:
identificador `goto` `;`
`continue` `;`
`break` `;`
expresión$_{opt}$ `return` `;`

A.2.4 Definiciones externas

unidad de traducción:
declaración externa
unidad de traducción declaración externa

declaración externa:
definición de función
declaración

definición de función:
declarador de especificadores$_{opt}$ *de declaración enunciado compuesto de lista*$_{opt}$ *de declaración*

A.3 Directrices de preprocesador

archivo de preprocesador:
grupo$_{opt}$

grupo:
parte de grupo
parte de grupo grupo

parte de grupo:
componentes léxicos$_{opt}$ *de preprocesador nueva línea*
sección if
línea de control

sección if:
grupo if grupos$_{opt}$ *elif grupo*$_{opt}$ *línea endif*

grupo if:
`#` `if` *expresión constante grupo*$_{opt}$ *nueva línea*
`#` `ifdef` *identificador grupo*$_{opt}$ *nueva línea*
`#` `ifndef` *identificador grupo*$_{opt}$ *nueva línea*

grupos elif:
grupos elif
grupos elif grupo elif

grupo elif:
`#` `elif` *expresión constante grupo*$_{opt}$ *nueva línea*

grupo else:
`#` `else` *grupo*$_{opt}$ *nueva línea*

línea endif:
`#` `endif` *nueva línea*

línea de control:
`#` `include` *componentes léxicos nueva línea*
`#` `define` *lista de remplazo identificador nueva línea*
`#` `define` *identificador paréntesis izquierdo (lparen) lista*$_{opt}$ *identificador* `)` *lista de remplazo nueva línea*
`#` `undef` *identificador nueva línea*
`#` `line` *componentes léxicos nueva línea*
`#` `error` *componentes léxicos*$_{opt}$ *nueva línea*
`#` `pragma` *componentes léxicos*$_{opt}$ *nueva línea*
`#` *nueva línea*

lparen:
el carácter de paréntesis izquierdo sin espacio en blanco previo

lista de remplazo:
componentes léxicos$_{opt}$

componentes léxicos:
componente léxico de preprocesador
componentes léxicos de preprocesador componente léxico

nueva línea:
el carácter de nueva línea

Apéndice B*

Biblioteca estándar

B.1 Errores <errno.h>

EDOM

ERANGE

Estas se expanden a expresiones constantes integrales con valores precisos no cero, utilizables para uso en directrices de preprocesador `#if`.

`errno`

Un valor de tipo `int` que es definido por varias funciones de biblioteca como un número positivo de error. Al arranque del programa el valor `errno` es cero, pero jamás es definido como cero por ninguna función de biblioteca. Un programa que utilice `errno` para verificación de errores debe definirlo a cero antes de una llamada a una función de biblioteca, y leerlo antes de una subsiguiente llamada a una función de biblioteca. Una función de biblioteca puede guardar el valor de `errno` al entrar y después definirlo como cero, siempre y cuando el valor original sea restaurado, si justo antes del regreso, el valor de `errno` siga siendo cero. El valor de `errno` puede ser definido por una llamada de función de biblioteca como un valor no cero, exista o no exista error, siempre que el uso de `errno` no haya sido documentado en la descripción de función en el estándar.

B.2 Definiciones comunes <stddef.h>

NULL

Una constante de apuntador nula definida por la puesta en práctica.

`offsetof` (*tipo*, *designador de miembro*)

Se expande a una expresión constante íntegra de tipo `size_t`, el valor de la cual es el desplazamiento en bytes al miembro de estructura (especificado por el *designador de miembro*) a partir del principio de su tipo de estructura (designado por *tipo*). El *designador de miembro* debe ser de tal forma que, dado

```
static tipo t;
```

* Reconocimiento de autorización: Este material ha sido condensado y adaptado a partir del American National Standard for Information Systems —Programming Language— C, ANSI/ISO 9899:1990. Se pueden adquirir copias de esta norma de la American National Standard Institute en 11 West 42nd Street, New York, NY 10036.

entonces, la expresión `&` `(t.` *designador de miembro*) se evalúa a una constante de dirección. (Si el miembro especificado es un campo de bit, el comportamiento queda indefinido).

`ptrdiff_t`

El tipo entero con signo del resultado de la sustracción de dos apuntadores.

`size_t`

El tipo entero sin signo del resultado del operador `sizeof`.

`wchar_t`

Un tipo entero cuyo rango de valores puede representar códigos diferentes para todos los miembros del conjunto extendido de caracteres especificado entre los escenarios soportados; el carácter nulo tendrá el valor de código cero y cada miembro del conjunto básico de caracteres tendrá un valor de código igual a su valor cuando se use como un carácter único en una constante de caracteres íntegra.

B.3 Diagnósticos <assert.h>

`void assert(int expression);`

La macro `assert` efectúa diagnósticos dentro de programas. Cuando se ejecuta, si `expression` es falsa, la macro `assert` escribe información relativa a la llamada particular que falló (incluyendo el texto del argumento, el nombre del archivo fuente, y el número de línea fuente —éstos últimos son respectivamente los valores de los macros de preprocesador `__FILE__` y `__LINE__`) en el archivo de error estándar en formato definido por la puesta en práctica. El mensaje escrito pudiera aparecer de la forma

```
Assertion failed: expression, file xyz, line nnn
```

La macro `assert` a continuación llama a la función `abort`. Si la directiva de preprocesador

```
#define NDEBUG
```

aparece en el archivo fuente donde `assert.h` esté incluido, cualquier verificación sobre el archivo será ignorada.

B.4 Manejo de caracteres <ctype.h>

Las funciones en esta sección devuelven no cero (verdadero) si, y sólo si, el valor del argumento c está conforme con el incluido en la descripción de la función.

`int isalnum(int c);`

Prueba buscando cualquier carácter para el cual `isalpha` o `isdigit` es verdadero.

`int isalpha(int c);`

Prueba buscando cualquier carácter para el cual `isupper` o `islower` es verdadero.

`int iscntrl(int c),`

Prueba la existencia de cualquier carácter de control.

`int isdigit(int c);`

Prueba buscando cualquier carácter de dígito decimal.

`int isgraph(int c);`

Busca cualquier carácter de impresión, excepto el espacio (`' '`).

`int islower(int c);`

Busca cualquier carácter que sea una letra minúscula.

`int isprint(int c);`

Busca cualquier carácter de impresión incluyendo el espacio (`' '`).

`int ispunct(int c);`

Prueba buscando cualquier carácter de impresión que no sea ni espacio (`' '`) ni ningún carácter para el cual **isalnum** resulte verdadero.

int isspace(int c);

Prueba buscando cualquier carácter que sea un carácter estándar de espacio en blanco. Los caracteres estándar de espacio en blanco son: espacio (`' '`), alimentación de forma (`'\f'`), nueva línea (`'\n'`), retorno de carro (`'\r'`), tabulador horizontal (`'\t'`) y tabulador vertical (`'\v'`).

in isupper(int c);

Busca cualquier carácter que esté en mayúsculas.

in isxdigit(int c);

Busca cualquier carácter en dígitos hexadecimales.

int tolower(int c);

Convierte una letra mayúscula en la letra minúscula correspondiente. Si el argumento es un carácter para el cual **isupper** resulta verdadero y existe un carácter correspondiente para el cual **islower** resulta verdadero, la función **tolower** devuelve el carácter correspondiente; de lo contrario, el argumento se conserva sin modificación.

int toupper(int c);

Convierte una letra minúscula a la letra mayúscula correspondiente. Si el argumento a un carácter para el cual **islower** resulta verdadero y existe un carácter correspondiente para el cual **isupper** es verdadero, la función **toupper** devuelve el carácter correspondiente; de lo contrario, el argumento se regresa sin modificación.

B.5 Localización <locale.h>

LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME

Estas se expanden a expresiones constantes íntegras con valores precisos, utilizables como primer argumento de la función **setlocale**.

NULL

Una constante de apuntador nula definida por la puesta en práctica.

struct lconv

Contiene miembros relacionados con el formato de valores numéricos. La estructura deberá contener por lo menos los siguientes miembros en cualquier orden. En el escenario "C" los miembros tendrán los valores especificados en los comentarios.

```
char *decimal_point;        /* "." */
char *thousands-sep;        /* "" */
char *grouping;             /* "" */
char *int_curr_symbol;      /* "" */
char *currency_symbol;      /* "" */
char *mon_decimal_point;    /* "" */
char *mon_thousands_sep;    /* "" */
char *mon_grouping;         /* "" */
char *positive_sign;        /* "" */
char *negative_sign;        /* "" */
char int_frac_digits;       /* CHAR_MAX */
char frac_digits;           /* CHAR_MAX */
```



```
char p_cs_precedes;         /* CHAR_MAX */
char p_sep_by_space;        /* CHAR_MAX */
char n_cs_precedes;         /* CHAR_MAX */
char n_sep_by_space;        /* CHAR_MAX */
char p_sign_posn;           /* CHAR_MAX */
char n_sign_posn;           /* CHAR_MAX */
```

char *setlocale(int category, const char *locale);

La función **setlocale** selecciona la porción apropiada del escenario del programa, tal y como se especifica por los argumentos **category** y **locale**. La función **setlocale** puede ser utilizada para modificar o consultar el escenario actual total del programa o partes del mismo. El valor **LC_ALL** para **category** nombra todo el escenario del programa: los otros valores para **category** nombran sólo una porción del escenario del programa. **LC_COLLATE** afecta el comportamiento de las funciones **strcoll** y **strxfrm**. **LC_CTYPE** afecta el comportamiento de las funciones de manejo de caracteres y de las funciones multibyte. **LC_MONETARY** afecta la información de formato de moneda devuelta por la función **localeconv**. **LC_NUMERIC** afecta el carácter de punto decimal para las funciones de entrada/salida con formato, para las funciones de conversión de cadenas, y para la información de formato no monetario devuelta por **localeconv**. **LC_TIME** afecta el comportamiento de **strftime**.

Un valor de **"C"** para el **locale** especifica el entorno mínimo para traducción en C; un valor de **""** para **locale** especifica el entorno nativo definido por la puesta en práctica. Se pueden pasar a **setlocale** otras cadenas definidas por la puesta en práctica. Al arranque del programa, se ejecuta el equivalente de

```
setlocale(LC_ALL, "C");
```

Si un apuntador a una cadena se da para **locale** y dicha selección puede ser aceptada, la función **setlocale** devuelve un apuntador a la cadena asociada con la **category** especificada para el nuevo escenario. Si la selección no puede ser aceptada, la función **setlocale** devuelve un apuntador nulo y el escenario del programa no es modificado.

Un apuntador nulo para **locale** hace que el función **setlocale** devuelva un apuntador a la cadena asociada con la **category** correspondiente al escenario actual del programa; el escenario del programa no se modifica.

El apuntador a la cadena devuelta por la función **setlocale** es tal que una llamada subsecuente a ese valor de cadena y a su categoría asociada restaurará dicha parte de la escenario del programa. La cadena a la cual se señala será modificada por el programa, pero podría ser sobrescrita por una llamada subsecuente a la función **setlocale**.

struct lconv *localeconv(void);

La función **localeconv** define los componentes de un objeto con tipo **struct lconv** con valores apropiados para el formato de cantidades numéricas (de moneda y otras) de acuerdo con las reglas del escenario actual.

Los miembros de la estructura con el tipo **char *** son apuntadores a cadenas, cualquiera de los cuales (a excepción de **decimal_point**) puede señalar a **""**, para indicar que el valor no está disponible en el escenario actual o tiene una longitud cero. Los miembros con tipo **char** son números no negativos, cualquiera de los cuales puede ser **CHAR_MAX** para indicar que el valor no está disponible en el escenario actual.

Los miembros incluyen lo siguiente:

char *decimal_point

El carácter de punto decimal utilizado para darle formato a cantidades no monetarias.

char *thousands_sep

El carácter utilizado para separar grupos de dígitos antes del carácter de punto decimal en cantidades no monetarias con formato.

char *grouping
Una cadena cuyos elementos indican el tamaño de cada grupo de dígitos en cantidades no monetarias con formato.

char *int_curr_symbol
El símbolo de moneda internacional aplicable al escenario actual. Los tres primeros caracteres contienen el símbolo de moneda internacional alfabético de acuerdo con lo especificado en ISO 4217:1987. El cuarto carácter (que precede al carácter nulo) es el carácter utilizado para separar el símbolo internacional de moneda de la cantidad monetaria.

char *currency_symbol
El símbolo local de moneda aplicable al escenario actual.

char *mon_decimal_point
El punto decimal utilizado para darle formato a cantidades monetarias.

char *mon_thousands_sep
El separador para grupos de dígitos antes del punto decimal en cantidades monetarias con formato.

char *mon_grouping
Una cadena cuyos elementos indican el tamaño de cada grupo de dígitos en cantidades monetarias con formato.

char *positive_sign
Una cadena utilizada para indicar una cantidad monetaria con formato con valores no negativos.

char *negative_sign
La cadena que se utiliza para indicar una cantidad monetaria con formato de valor negativo.

char int_frac_digits
El número de dígitos fraccionarios (aquellos que aparecen después del punto decimal) al mostrarse en una cantidad monetaria internacional con formato.

char frac_digits
El número de dígitos fraccionarios (los que aparecen después del punto decimal) a mostrarse en una cantidad monetaria con formato.

char p_cs_precedes
Se define como 1 ó como 0 el **currency_symbol** respectivamente antecederá o seguirá al valor en una cantidad monetaria no negativa con formato.

char p_sep_by_space
Definido como 1 ó como 0 el **currency_symbol** respectivamente está o no está separado mediante un espacio de la cantidad monetaria no negativa con formato.

char n_cs_precedes
Definido a 1 ó como 0 el **currency_symbol** respectivamente antecede o sigue al valor en una cantidad monetaria negativa con formato.

char n_sep_by_space
Definido a 1 ó como 0 el **currency_symbol** respectivamente está o no está separado mediante un espacio del valor en una cantidad monetaria negativa con formato.

char p_sign_posn
Definido a un valor indicado en la posición del **positive_sign** para una cantidad monetaria no negativa con formato.

char n_sign_posn
Definido a un valor indicado en la posición del **negative_sign** en cantidad monetaria negativa con formato.

Los elementos de **grouping** y de **mon_grouping** se interpretan de acuerdo con lo siguiente:

CHAR_MAX No se ejecutará más agrupamiento

0 El elemento precedente debe ser utilizado de forma repetida para el resto de los dígitos.

other El valor entero es el número de dígitos que comprenden el grupo actual. Se examinará el siguiente elemento para determinar el tamaño del siguiente grupo de dígitos antes del grupo actual.

Los valores de **p_sign_posn** y de **n_sign_posn** se interpretan de acuerdo con lo siguiente.

0 Paréntesis encerrando la cantidad y **currency_symbol**.

1 La cadena de signos antecede a la cantidad y **currency_symbol**.

2 La cadena de signos sigue a la cantidad y **currency_symbol**.

3 La cadena de signos precede de inmediato antes de **currency_symbol**.

4 La cadena de signos sigue de inmediato a **currency_symbol**.

La función **localeconv** devuelve un apuntador al objeto rellenado. La estructura a la cual se señala mediante el valor de regreso no deberá ser modificado por el programa, pero puede ser sobreescrito mediante una llamada subsecuente a la función **localeconv**. En adición, llamadas a la función **setlocale** con las categorías **LC_ALL, LC_MONETARY,** o bien **LC_NUMERIC** pueden sobreescribir el contenido de la estructura.

B.6 Matemáticas <math.h>

HUGE_VAL
Una constante simbólica que representa una expresión positiva **double**.

double acos(double x);
Calcula el valor principal del arco cuyo coseno es **x**. Para argumentos que no estén en el rango [-1, +1] ocurrirá un error de dominio. La función **acos** devuelve el arco del coseno en el rango de [0, π] radianes.

double asin(double x);
Calcula el valor principal del arco cuyo seno es **x**. Para argumentos que no estén en el rango [-1, +1] ocurrirá un error de dominio. La función **asin** devuelve el seno arco en el rango de [-π/2, +π/2] radianes.

double atan(double x);
Calcula el valor principal del arco cuya tangente es **x**. La función **atan** devuelve el arco tangente en el rango de [-π/2, +π/2] radianes.

double atan2(double y, double x);
La función **atan2** calcula el valor principal del arco tangente **y/x**, utilizando los signos de ambos argumentos para determinar el cuadrante del valor devuelto. Puede ocurrir un error de dominio si ambos argumentos son cero. La función **atan2** devuelve el arco tangente de **y/x**, en el rango [-π, +π] radianes.

double cos(double x);
Calcula el coseno de **x** (medido en radianes).

double sin(double x);
Calcula el seno de **x** (medido en radianes).

`double tan(double x);`
Devuelve la tangente de `x` (medida en radianes).

`double cosh(double x);`
Calcula el coseno hiperbólico de `x`. Puede ocurrir un error de rango si la magnitud de `x` es demasiado grande.

`double sinh(double x);`
Calcula el seno hiperbólico de `x`. Ocurre un error de rango si la magnitud de `x` es demasiado grande.

`double tanh(double x);`
La función `tanh` calcula la tangente hiperbólica de `x`.

`double exp(double x);`
Calcula la función exponencial de `x`. Ocurre un error de rango si la magnitud de `x` es muy extensa.

`double frexp(double value, int *exp);`
Divide el número de punto flotante en una fracción normalizada y una potencia entera de 2. Almacena el entero en el objeto `int` apuntado por `exp`. La función `frexp` devuelve el valor `x`, tal que `x` es un `double` con magnitud en el intervalo [1/2, 1] o cero, y `value` es igual a `x` multiplicado por 2 elevado a la potencia `*exp`. Si `value` es igual a cero, ambas partes del resultado son cero.

`double ldexp(double x, int exp);`
Multiplica un número de punto flotante por una potencia entera de 2. Puede ocurrir un error de rango. La función `ldexp` devuelve el valor de `x` multiplicado por 2 elevado a la potencia `exp`.

`double log(double x);`
Calcula el logaritmo natural de `x`. Ocurre un error de dominio si el argumento es negativo. Pudiera ocurrir un error de rango si el argumento es cero.

`double log10(double x);`
Calcula el logaritmo en base diez de `x`. Ocurre un error de dominio si el argumento es negativo. Pudiera ocurrir un error de rango si el argumento es cero.

`double modf(double value, double *iptr);`
Divide el argumento `value` en partes enteras y fraccionarias, cada una de las cuales tiene el mismo signo que el argumento. Almacena en la parte entera como un `double` en el objeto al cual apunta `iptr`. La función `modf` devuelve la parte fraccionaria signada de `value`.

`double pow(double x, double y);`
Calcula `x` elevado a la potencia `y`. Ocurre un error de dominio si `x` es negativo e `y` no es un valor entero. Ocurre un error de dominio si el resultado no puede ser representado cuando `x` es cero e `y` es menor o igual a cero. Puede ocurrir un error de rango.

`double sqrt (double x);`
Calcula la raíz cuadrada no negativa de `x`. Ocurre un error de dominio si el argumento es negativo.

`double ceil(double x);`
Calcula el valor integral más pequeño no menor que `x`.

`double fabs(double x);`
Calcula el valor absoluto del número de punto flotante `x`.

`double floor(double x);`
Calcula el valor integral más grande no mayor que `x`.

`double fmod(double x, double y);`
Calcula el residuo en punto flotante de `x` dividido entre `y`.

B.7 Saltos no locales `<setjmp.h>`

`jmp_buf`
Un tipo de arreglo adecuado para contener la información necesaria para restaurar un entorno llamador.

`int setjmp(jmp_buf env);`
Guarda su entorno llamador en el argumento `jmp_buf` para uso posterior por la función `longjmp`.

Si el regreso es debido a una invocación directa, la macro `setjmp` devuelve el valor cero. Si el regreso es debido a una llamada de la función `longjmp`, la macro `setjmp` devuelve un valor no cero.

Una invocación a la macro `setjmp` debe de aparecer sólo en alguno de los siguientes contextos:

- en la totalidad de la expresión de control de un enunciado de selección o de iteración;
- un operando de un operador relacional o de igualdad siendo el otro operando una expresión constante entera, con la expresión resultante siendo la expresión de control total de un enunciado de selección o de iteración;
- el operando del operador unario `!` con la expresión resultante siendo la expresión de control total de un enunciado de selección o de iteración; o bien
- la expresión total de un enunciado de expresión.

`void longjmp(jmp_buf env, int val);`
Devuelve el entorno guardado por la invocación más reciente de la macro `setjmp` en la misma invocación del programa, con el argumento correspondiente `jmp_buf`. Si dicha invocación no ha ocurrido o si en el ínterin la función que contiene la invocación de la macro `setjmp` ha terminado su ejecución, el comportamiento queda indefinido.

Todos los objetos accesibles tienen valores correspondientes al momento en que fue llamado `longjmp`, excepto que los valores de objeto de persistencia automática, que son locales a la función que contiene la invocación de la macro `setjmp` correspondiente que no son del tipo volátil y que han sido modificados entre la invocación `setjmp` y la invocación `longjmp`, quedan indeterminados.

Como pasa por alto los mecanismos usuales de llamada de función y de regreso, `longjmp` ejecutará de forma correcta en el contexto de interrupciones, señales y cualquiera de sus funciones asociadas. Sin embargo, si la función `longjmp` es invocada a partir de un manejador de señales anidado (esto es, desde una función invocada como resultado de una señal iniciada durante el manejo de otra señal), el comportamiento queda indefinido.

Después de que `longjmp` se haya completado, la ejecución del programa continuará como si la invocación correspondiente de la macro `setjmp` hubiera devuelto el valor especificado por `val`. La función `longjmp` no puede hacer que la macro `setjmp` devuelva el valor 0; si `val` es 0, la macro `setjmp` devuelve el valor 1.

B.8 Manejo de señales `<signal.h>`

`sig_atomic_t`
El tipo entero de un objeto al cual se puede tener acceso como entidad atómica, inclusive en presencia de interrupciones asincrónicas.

`SIG_DFL`
`SIG_ERR`
`SIG_IGN`
Estas se expanden a expresiones constantes con valores precisos que tienen tipos compatibles con el segundo argumento y con el valor de regreso a la función `signal`, y cuyos valores se comparan en forma desigual con la dirección de cualquier función declarable; en lo que sigue, cada uno de los cuales se expande a una expresión constante entera positiva que es el número de señal para la condición especifica:

`SIGABRT`	terminación anormal, como la que es iniciada por la función `abort`
`SIGFPE`	una operación aritmética errónea, como división por cero o una operación que resulte en desbordamiento
`SIGILL`	detección de una imagen de función inválida, como sería una instrucción ilegal
`SIGINT`	recepción de una señal de atención interactiva
`SIGSEGV`	un acceso inválido a almacenamiento
`SIGTERM`	una solicitud de terminación enviada al programa.

Una puesta en práctica no necesita generar ninguna de estas señales, excepto como resultado de llamadas explícitas a la función `raise`.

```
void (*signal(int sig, void (*func)(int)))(int);
```

Escoge una de tres formas en la cual será subsecuentemente manejado el número de señal `sig`. Si el valor de `func` es `SIG_DEF`, ocurrirá el manejo por omisión de la señal. Si el valor de `func` es `SIG_IGN`, la señal será ignorada. De lo contrario, cuando dicha señal ocurra `func` apuntará a la función a llamarse. Dicha función es un *manejador de señal.*

Cuando ocurre una señal, si `func` apunta a una función, primero se ejecuta el equivalente de `signal (sig, SIG_DFL)` o se ejecuta un bloqueo de la señal, definido por la puesta en práctica. (Si el valor de `sig` es `SIGILL`, la restauración de `SIG_DFL` dependerá de la puesta en práctica). A continuación se ejecuta el equivalente de `(*func) (sig)` La función `func` puede terminar ejecutando un enunciado `return` o llamando a la función `abort`, `exit` o `longjmp`. Si `func` ejecuta un enunciado `return` y el valor de `sig` era `SIGFPE` o cualquier otro valor definido por la puesta en práctica correspondiente a una excepción de cómputo, el comportamiento queda indefinido. De lo contrario, el programa continuará en su ejecución desde el punto en que fue interrumpido.

Si la señal ocurre de forma distinta que como resultado a la llamada de la función `abort` o `raise`, el comportamiento queda indefinido si el manejador de señal llama a cualquier función de la biblioteca estándar distinta a la función `signal` misma (con un primer argumento del número de señal correspondiente a la señal que causó la invocación del manejador) o se refiere a cualquier otro objeto con persistencia estática, distinto que mediante la asignación de un valor a una variable de persistencia estática del tipo `volatile sig_atomic_t`. Además, si dicha llamada a una función `signal` resulta en un regreso `SIG_ERR`, el valor de `errno` queda indeterminado.

Al arranque del programa, el equivalente de

```
signal(sig, SIG_IGN);
```

puede ser ejecutado para algunas señales seleccionadas de una forma definida por la puesta en práctica; el equivalente de

```
signal(sig, SIG_DFL);
```

es ejecutado para todas las demás señales definidas por la puesta en práctica.
Si la solicitud puede ser aceptada, la función `signal` devuelve el valor de `func` para la llamada más reciente a `signal` para la señal especificada `sig`. De lo contrario, se devuelve un valor de `SIG_ERR` y en `errno` queda almacenado un valor positivo.

```
int raise(int sig);
```

La función `raise` envía la señal `sig` al programa en ejecución. Si tiene éxito la función `raise` devuelve cero, y no cero si no lo tiene.

B.9 Argumentos variables <stdarg.h>

`va_list`

Un tipo adecuado para contener información necesaria para las macros `va_start`, `va_arg`, y `va_end`. Si se desea tener acceso a los distintos argumentos, la función llamada declarará un objeto

(llamada en esta sección `ap`) con el tipo `va_list`. El objeto `ap` puede ser pasado como argumento a otra función; si dicha función invoca la macro `va_arg` con el parámetro `ap`, el valor de `ap` en la función llamada es determinada y será pasada a la macro `va_end` antes de cualquier otra referencia `ap`.

```
void va_start(va_list ap, parmN);
```

Será invocada antes de cualquier acceso a argumentos sin nombre. La macro `va_start` inicializa `apuntador` para uso subsecuente por las macros `va_arg` y `va_end`. El parámetro *parmN* es el identificador del parámetro más a la derecha de la lista de parámetros variables en la definición de función (una justo antes de la coma , ...). Si se declara el parámetro *parmN* con la clase de almacenamiento `register`, con un tipo de función o de arreglo, o con un tipo que no sea compatible con el tipo que resulte después de la aplicación de las promociones de argumento por omisión, el comportamiento quedará indefinido.

tipo `va_arg(va_list, tipo);`

Se expande a una expresión que tiene el tipo y el valor del siguiente argumento dentro de la llamada. El parámetro `ap` será el mismo que el de `va_list ap` inicializado por `va_start`. Cada invocación de `va_arg` modifica `ap` de tal forma, que los valores de argumentos sucesivos sean regresados en orden. El parámetro *type* es el nombre de tipo definido de tal forma, que el tipo de un apuntador a un objeto que tiene el tipo especificado se pueda obtener mediante la colocación postfija de un * a *type*. Si no existe un argumento siguiente o si *type* no es compatible con el tipo del siguiente argumento (como promovido de acuerdo con las promociones de argumento por omisión), el comportamiento queda indefinido. La primera invocación de la macro `va_arg` después de la macro `va_start` devuelve el valor del argumento después del especificado por *parmN*. Invocaciones subsiguientes devuelven en sucesión los valores de los argumentos siguientes.

```
void va_end(va_list ap);
```

Facilita el regreso normal de una función cuya lista de argumentos variables fue referida mediante la expansión de `va_start` que inicializó `va_list ap`. La macro `va_end` puede modificar de tal forma `ap`, que ya no sea utilizable (sin una invocación intermedia de `va_start`). Si no existe una invocación correspondiente de la macro `va_start` o si la macro `va_end` no se invoca antes del regreso, el comportamiento queda indefinido.

B.10 Entrada/salida <stdio.h>

`_IOFBF`
`_IOLBF`
`_IONBF`

Expresiones constantes enteras con valores distintos, adecuadas para uso como el tercer argumento a la función `setvbuf`.

`BUFSIZ`

Una expresión constante entera, que representa el tamaño del búfer utilizado por la función `setvbuf`.

`EOF`

Una expresión constante entera negativa, devuelta por varias funciones a fin de indicar fin de archivo, esto es, no más entradas a partir de un flujo.

`FILE`

Un tipo de objeto capaz de registrar toda la información necesaria para controlar un flujo, incluye el indicador de posición de archivo, un apuntador a su búfer asociado (si es que existe alguno), un indicador de error que registre si ha ocurrido algún error de escritura/lectura y un indicador de fin de archivo que registra si se ha llegado al fin del archivo.

FILENAME_MAX
Una expresión constante entera que tiene el tamaño necesario para un arreglo de **char** lo suficiente extensa para contener la cadena de nombre de archivo más larga posible que la puesta en práctica garantiza es posible abrir.

FOPEN_MAX
Una expresión constante entera que es el número mínimo de archivos que la puesta en práctica garantiza se abrirán en forma simultánea.

fpos_t
Un tipo de objeto capaz de registrar toda la información necesaria para especificar todas las posiciones únicas dentro de un archivo.

L_tmpnam
Una expresión constante integral que es del tamaño necesario para un arreglo **char** lo suficiente extensa para contener una cadena de nombre de archivo temporal generado por una función **tmpnam**.

NULL
Una constante de apuntador nulo definido por la puesta en práctica.

SEEK_CUR
SEEK_END
SEEK_SET
Expresiones constantes integrales con valores precisos, adecuadas para uso como el tercer argumento de la función **fseek**.

size_t
El tipo entero sin signo del resultado del operador **sizeof**.

stderr
Expresión del tipo "apuntador a **FILE**" que señala al objeto **FILE** asociado con el flujo estándar de error.

stdin
Expresión del tipo "apuntador a **FILE**" que señala al objeto **FILE** asociado con el flujo de entrada estándar.

stdout
Expresión del tipo "apuntador a **FILE**" que señala al objeto **FILE** asociado con el flujo de salida estándar.

TMP_MAX
Una expresión constante entera que es el número mínimo de nombres únicos de archivo que se generarán mediante la función **tmpnam**. El valor de la macro **TMP_MAX** será de por lo menos 25.

int remove(const char *filename);
Hace que el archivo cuyo nombre es la cadena a la cual señala **filename** ya no sea accesible para dicho nombre. Cualquier intento subsiguiente para abrir dicho archivo utilizando este nombre fallará, a menos que sea creado de nuevo. Si el archivo está abierto, el comportamiento de la función **remove** queda definido por la puesta en práctica. La función **remove** devuelve cero si la operación tiene éxito, y no cero si falla.

int rename(const char *old, const char *new);
Hace que el archivo cuyo nombre es la cadena a la cual señala **old** que de ahí en adelante se conozca por el nombre dado por la cadena a la cual señala **new**. El archivo de nombre **old** ya no es accesible utilizando ese nombre. Si antes de la llamada a la función **rename** existe un archivo llamado mediante la cadena a la cual señala **new**, el comportamiento queda definido por la puesta en práctica. La función **rename** devuelve cero si la operación tiene éxito, no cero si falla, en cuyo caso si el archivo ya existía aún se llamará con su nombre original.

FILE *tmpfile(void);
Genera un archivo temporal binario que será eliminado de forma automática cuando se cierre o al final del programa. Si el programa termina en forma anormal, si este programa temporal abierta se elimina o no, depende de la puesta en práctica. El archivo se abre para actualización mediante el modo "**wb+**". La función **tmpfile** devuelve un apuntador al flujo del archivo que ha sido creado. Si el archivo no puede ser creado, la función **tmpfile** devuelve un apuntador nulo.

char *tmpnam(char *s)
La función **tmpnam** genera una cadena que es un nombre válido de archivo y que no es el mismo que el nombre de un archivo existente. La función **tmpnam** genera una cadena distinta cada vez que es llamada, hasta **TMP_MAX** veces. Si es llamada más de **TMP_MAX** veces, el comportamiento queda definido por la puesta en práctica.

Si el argumento es un apuntador nulo, la función **tmpnam** deja su resultado en un objeto estático interno y devuelve un apuntador a dicho objeto. Llamadas subsecuentes a la función **tmpnam** pueden modificar el mismo objeto. Si el argumento no es un apuntador nulo, se supone que debe señalar a un arreglo de por lo menos **L_tmpnam char**; la función **tmpnam** escribe su resultado en dicho arreglo y devuelve el argumento como su valor.

int fclose(FILE *stream);
La función **fclose** hace que el flujo al cual señala **stream** sea vaciado y el archivo asociado se cierre. Cualquier dato en archivo temporal o búfer aún no escrito para el flujo es entregado al entorno huésped para ser escrito al archivo; cualesquiera datos en archivos temporales no leídos serán descartados. El flujo queda desasociado del archivo. Si el búfer asociado fue asignado en forma automática, queda desasignado. La función **fclose** devuelve cero si el flujo fue cerrado de forma exitosa o bien **EOF** si se detectaron algunos errores.

int fflush(FILE *stream);
Si **stream** señala a un flujo de salida o a uno actualizado en el cual la operación más reciente no ha sido introducida, la función **fflush** hace que todos los datos aún no escritos para dicho flujo sean entregados al entorno huésped o se escriban al archivo; de lo contrario, el comportamiento queda indefinido.

Si **stream** es un apuntador nulo, la función **fflush** lleva a cabo su acción de vaciado sobre todos los flujo para los cuales el comportamiento se ha definido arriba. Si ocurre algún error de escritura la función **fflush** devuelve **EOF**, de lo contrario devuelve cero.

FILE *fopen(const char *filename, const char *mode);
La función **fopen** abre el archivo cuyo nombre es la cadena a la cual señala **filename** y asocia un flujo al mismo. El argumento **mode** señala una cadena que empieza con alguna de las secuencias siguientes:

r	abre archivo de texto para lectura.
w	trunca a cero longitud o crea un archivo de texto para escritura.
a	agrega; abre o crea un archivo de texto para escritura al fin de archivo.
rb	abre archivo binario para lectura.
wb	trunca a longitud cero o crea archivo binario para escritura.
ab	agrega; abre o crea un archivo binario para escritura al fin de archivo.
r+	abre archivo de texto para actualizar (lectura y escritura).
w+	trunca a longitud cero o crea un archivo de texto para actualizar.

a+	agrega; abre o crea un archivo de texto para actualizar, escribiendo al fin de archivo.
r+b o rb+	abre archivo binario para actualizar (lectura y escritura).
w+b o wb+	trunca a longitud cero o crea un archivo binario para actualizar.
a+b o ab+	agrega; abre o crea un archivo binario para actualizar, escribiendo al fin de archivo.

Abrir un archivo en modo de lectura (**'r'** como el primer carácter en el argumento **mode**) fallará si el archivo no existe o no puede ser leído. Abrir el archivo con el modo de agregar (**'a'** como el primer carácter en el argumento **mode**) hace que todas las escrituras subsiguientes al archivo se obliguen a lo que en ese momento sea el fin del archivo, independiente de llamadas a la función **fseek**. En algunas puestas en práctica, la apertura de un archivo binario en modo de agregar (**'b'** como segundo y tercer caracteres en la lista anterior de los valores de argumento **mode**) puede colocar en forma inicial el indicador de posición de archivo para el flujo más allá de los últimos datos escritos, debido al relleno de carácter nulo.

Cuando se abre un archivo con el modo de actualizar (**'+'** como segundo y tercer carácter en la lista anterior de valores de argumento **mode**), se puede llevar a cabo tanto entrada como salida sobre el flujo asociado. Sin embargo, la salida no puede ser seguida en forma directa por entrada, sin una llamada intermedia a la función **fflush** o a la función de posicionamiento de archivo (**fseek**, **fsetpos** o **rewind**), y la entrada no puede ser de manera directa seguida por salida sin una llamada intermedia a una función de posicionamiento de archivo, a menos de que la operación de entrada se encuentre con fin de archivo. La apertura (o creación) de un archivo de texto en el modo de actualizar, puede quizás abrir (o crear) un flujo binario en algunas puestas en práctica particulares.

Una vez abierto, un flujo está por completo equipado con memoria temporal (búfer) si y sólo si se puede determinar que no se refiere a un dispositivo interactivo. Los indicadores de error y de fin de archivo para el flujo estarán desactivados. La función **fopen** devuelve un apuntador al objeto que controla el flujo. Si la operación de apertura falla, **fopen** devuelve un apuntador nulo.

```
FILE *freopen(const char *filename, const char *mode,
              FILE *stream);
```

La función **freopen** abre el archivo cuyo nombre es la cadena a la cual señala **filename** y asocia el flujo al cual apunta **stream**. El argumento **mode** se utiliza de la misma forma que en la función **fopen**.

La función **freopen** intenta primero cerrar cualquier archivo que esté asociado con el flujo especificado. Se ignorará cualquier falla para cerrar el archivo con éxito. Los indicadores de error y de fin de archivo serán desactivados. Si falla la operación de apertura la función **freopen** devuelve un apuntador nulo. De lo contrario, **freopen** devuelve el valor de **stream**.

```
void setbuf(FILE *stream, char *buf);
```

La función **setbuf** es equivalente a la función **setvbuf** que se invoca con los valores **_IOFBF** para **mode** y **BUFSIZ** para **size**, o (**buf** es un apuntador nulo) con el valor **_IONBF** para **mode**. La función **setbuf** no devuelve valor.

```
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

La función **setvbuf** puede ser utilizada sólo cuando el flujo al cual señala **stream** ha sido asociado con un archivo abierto y antes de cualquier otra operación se haya ejecutado sobre el flujo. El argumento **mode** determina como se harán los búfers de **stream**, como sigue: **_IOFBF** hará que existan búfers completos para entradas/salidas; **_IOLBF** creará búfer de línea para entrada/salida; **_IONBF** hará que las entradas y salidas no tengan búfer. Si **buf** no es un apuntador nulo, el arreglo al cual señala podrá ser utilizado en vez de un búfer asignado por la función **setvbuf**. El argumento **size** especifica el tamaño del arreglo. Los contenidos del arreglo en cualquier momento quedan indeterminados. La función **setvbuf** devuelve cero si tiene éxito o algún valor no cero si se ha dado un valor inválido para **mode** o si la solicitud no puede ser aceptada.


```
int fprintf(FILE *stream, const char *format, ...);
```

La función **fprintf** escribe salida al flujo al cual señala **stream**, bajo control de la cadena al cual señala **format** que define como los argumentos subsecuentes son convertidos para la salida. Si no hay suficientes argumentos para el formato, el comportamiento queda indefinido. Si el formato se termina aunque queden argumentos, los argumentos en exceso se evalúan (como siempre), pero serán ignorados. La función **fprintf** devuelve cuando se encuentra con el fin de la cadena de formato. Vea el capítulo 9, "Entradas/salidas con formato", para una descripción detallada de las especificaciones de conversión de salida. La función **fprintf** devuelve el número de caracteres transmitidos, o un valor negativo si ha ocurrido un error de salida.

```
int fscanf(FILE *stream, const char *format, ...);
```

La función **fscanf** lee la entrada desde el flujo al cual señala **stream**, bajo control de la cadena a la cual apunta **format** que define las secuencias de entrada admisibles y cómo deben de ser convertidas para asignación, utilizando los argumentos subsecuentes como apuntadores a los objetos que han de recibir la entrada convertida. Si para el formato existen argumentos insuficientes, el comportamiento queda indefinido. Si el formato se termina aunque sobren argumentos, los argumentos excedentes se evalúan (como siempre), pero por lo demás se ignorarán. Vea el capítulo 9, "Entradas/salidas con formato", para una descripción detallada de las especificaciones de conversión de entrada.

La función **fscanf** devuelve el valor de la macro **EOF** si ocurre una falla de entrada antes de cualquier conversión. De lo contrario, la función **fscanf** devuelve el número de elementos de entrada asignados, mismos que pueden resultar menos de los proveídos inclusive cero, en caso de una falla temprana de coincidencias.

```
int printf(const char *format, ...);
```

La función **printf** es equivalente a **fprintf** con la interposición del argumento **stdout** antes de los argumentos a **printf**. La función **printf** devuelve el número de caracteres transmitidos o un valor negativo si ocurrió un error de salida.

```
int scanf(const char *format, ...);
```

La función **scanf** es equivalente a **fscanf** con la interposición del argumento **stdin** antes de los argumentos a **scanf**. Si ocurre una falla de entrada antes de cualquier conversión la función **scanf** devuelve el valor de la macro **EOF**. De lo contrario, **scanf** devuelve el número de elementos de entrada asignados, mismos que pueden ser menos de los proveídos o inclusive cero, en el caso de una falla temprana de coincidencia.

```
int sprintf(char *s, const char *format, ...);
```

La función **sprintf** es equivalente a **fprintf**, excepto que el argumento **s** especifica un arreglo al cual deberá de ser escrita la salida generada, en vez de a un flujo. Se escribe un carácter nulo al final de los caracteres escritos; no se cuenta como una parte de la suma devuelta. El comportamiento de copia entre objetos que se superponen queda indefinido. La función **sprintf** devuelve el número de caracteres escritos por el arreglo, sin contar el carácter nulo de terminación.

```
int sscanf(const char *s, const char *format, ...);
```

La función **sscanf** es equivalente a **fscanf**, excepto que el argumento **s** especifica una cadena a partir de la cual se obtendrá la entrada, más bien que a partir de un flujo. El llegar al final de la cadena es equivalente a encontrar fin de archivo para la función **fscanf**. Si la copia ocurre entre objetos que se superponen, el comportamiento queda indefinido.

La función **sscanf** devuelve el valor de la macro **EOF** si ocurre una falla de entrada antes de cualquier conversión. Lo contrario, la función **sscanf** devuelve el número de elementos de entrada asignados, mismos que pueden ser menos de los proveídos inclusive cero, en el caso de una falla temprana de coincidencia.

```
int vfprintf(FILE *stream, const char *format, va_list arg);
```

La función **vfprintf** es equivalente a **fprintf**, con la lista de argumentos variables remplazados por **arg**, que se incializa mediante la macro **va_start** (y mediante posibles llamadas subsecuentes

va_arg). La función vfprintf no invoca a la macro va_end. La función vfprintf devuelve el número de caracteres transmitidos o un valor negativo si ocurrió un error de salida.

```
int vprintf(const char *format, va_list arg);
```

La función vprintf es equivalente a printf, con la lista de argumentos variables remplazadas por arg, que debe haber sido inicializada por la macro va_start (y llamadas subsecuentes posibles va_arg). La función vprintf no invoca a la macro va_end. La función vprintf devuelve el número de caracteres transmitidos o un valor negativo si ocurrió un error de salida.

```
int vsprintf(char *s, const char *format, va_list arg);
```

La función vsprintf es equivalente a sprintf, con la lista de argumentos variables remplazadas por arg, misma que deberá haber sido inicializada por la macro va_start (y subsecuentes posibles llamadas va_arg). La función vsprintf no invoca a la macro va_end. Si la copia se lleva a cabo entre objetos que se superponen, el comportamiento queda indefinido. La función vsprintf devuelve el número de caracteres escritos en el arreglo, sin contar el carácter nulo de terminación.

```
int fgetc(FILE *stream);
```

La función fgetc obtiene el carácter siguiente (si está presente) como un unsigned char convertido a un int, del flujo de entrada al cual señala stream y avanza el indicador de posición de archivo asociado correspondiente al flujo (si está definido). La función fgetc devuelve el siguiente carácter del flujo de entrada al cual señala stream. Si el flujo está al fin de archivo, se activa el indicador de fin de archivo para el flujo y fgetc devuelve EOF. Si ocurre un error de lectura, se activa el indicador de error para el flujo y fgetc devuelve EOF.

```
char *fgets(char *s, int n, FILE *stream);
```

La función fgets lee por lo menos uno menos que el número de caracteres definido por n del flujo al cual señala stream al arreglo al cual señala s. No se leen caracteres adicionales después del carácter de nueva línea (mismos que se conserva) o después de fin de archivo. Se escribe un carácter nulo inmediatamente después del último carácter leído en el arreglo.

Si tiene éxito la función fgets devuelve s. Si se encuentra con un fin de archivo y no se han leído caracteres al arreglo, el contenido del arreglo se conserva sin modificación y se devuelve un apuntador nulo. Si durante la operación ocurre un error de lectura, el contenido del arreglo queda indeterminado y se devuelve un apuntador nulo.

```
int fputc(int c, FILE *stream);
```

La función fputc escribe el carácter definido por c (convertido a unsigned char) al flujo de salida al cual apunta stream, en la posición indicada por el indicador de posición de archivo asociado para el flujo (si está definido), y adelanta en forma apropiada dicho indicador. Si el archivo no puede aceptar solicitudes de posicionamiento o si el flujo fue abierto en modo de agregar, el carácter será agregado al flujo de salida. La función fputc devuelve el carácter escrito. Si ocurre un error de escritura, se activará el indicador de error para el flujo y fputc devolverá EOF.

```
int fputs(const char *s, FILE *stream);
```

La función fputs escribe la cadena a la cual señala s en el flujo al cual señala stream. No es escrito el carácter nulo de terminación. Si ocurre un error de escritura la función fputs devuelve EOF; de lo contrario devolverá un valor no negativo.

```
int getc(FILE *stream);
```

La función getc es equivalente a fgetc, excepto que cuando es puesta en práctica como una macro, pudiera evaluar más de una vez stream —el argumento deberá ser una expresión sin efectos colaterales.

La función getc devuelve el carácter siguiente del flujo de entrada al cual señala stream. Si el flujo está al fin de archivo, se activará el indicador de fin de archivo para el flujo y getc devuelve EOF. Si ocurre un error de lectura, el indicador de error para el flujo se activará y getc devolverá EOF.


```
int getchar(void);
```

La función getchar es equivalente a getc, con el argumento stdin. La función getchar devuelve el siguiente carácter del flujo de entrada al cual señala stdin. Si el flujo está a fin de archivo, se activará el indicador de fin de archivo para el flujo y getchar devolverá EOF. Si ocurre un error de lectura, se activará el indicador de error para el flujo y getchar devolverá EOF.

```
char *gets(char *s);
```

La función gets lee caracteres del flujo de entrada al cual señala stdin, al arreglo apuntado por s, hasta que se encuentra fin de archivo o se lee un carácter de nueva línea. Cualquier carácter de nueva línea será descartado o si no se escribirá un carácter nulo después del último carácter leído al arreglo. La función gets devuelve s si tiene éxito. Si se encuentra un fin de archivo y no se han leído caracteres al arreglo, el contenido del arreglo se conservará sin modificación y se devolverá un apuntador nulo. Si durante la operación ocurre un error de lectura, el contenido del arreglo queda indeterminado y se devuelve un apuntador nulo.

```
int putc(int c, FILE *stream);
```

La función putc es equivalente a fputc, excepto que cuando se pone en práctica como una macro, pudiera evaluar stream más de una vez, por lo que el argumento no debería nunca ser una expresión con efectos colaterales. La función putc devuelve el carácter escrito. Si ocurre un error de escritura, se activa el indicador de error correspondiente al flujo y putc devuelve EOF.

```
int putchar(int c);
```

La función putchar es equivalente a putc con stdout como segundo argumento. La función putchar devuelve el carácter escrito. Si ocurre un error de escritura, se activa el indicador de error para el flujo y putchar devuelve EOF.

```
int puts(const char *s);
```

La función puts escribe la cadena a la cual señala s en el flujo al cual señala stdout, y agrega un carácter de nueva línea a la salida. El carácter nulo de terminación no es escrito. La función puts devuelve EOF si ocurre un error de escritura; de lo contrario, devuelve un valor no negativo.

```
int ungetc(int c, FILE *stream);
```

La función ungetc coloca el carácter especificado por c (convertido a un unsigned char) de regreso en el flujo de entrada al cual señala stream. Los caracteres devueltos serán regresados en lecturas subsecuentes de dicho flujo en orden inverso a su entrada. Una llamada intermedia exitosa (con el flujo al cual apunta stream) a un función de posicionamiento de archivo (fseek, fsetpos, o rewind) descartará todos los caracteres vueltos a colocar para el flujo. El almacenamiento externo correspondiente al flujo se mantiene sin modificación.

Un carácter de los regresados queda garantizado. Si una función ungetc es llamada demasiadas veces sobre el mismo flujo sin una operación intermedia de posicionamiento de lectura o de archivo sobre dicho flujo, la operación pudiera fallar. Si el valor de c se iguala al de la macro EOF, la operación fallará y el flujo de entrada queda sin modificar.

Una llamada exitosa a la función ungetc desactiva el indicador de fin de archivo correspondiente al flujo. El valor del indicador de posición de archivo para el flujo después de leer o descartar todos los caracteres introducidos y regresados será el mismo que era antes de que los caracteres fueran retornados. Para un flujo de texto, el valor de este indicador de posición de archivo después de una llamada a la función ungetc queda sin especificar en tanto todos los caracteres devueltos al flujo sean leídos o descartados. En el caso de un flujo binario, su indicador de posición de archivo queda determinado mediante cada llamada exitosa a la función ungetc; si su valor antes de la llamada era cero, quedará después de la llamada indeterminado. La función ungetc devuelve el carácter devuelto al flujo después de la conversión o si no en caso de fallo de la operación EOF.

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

La función fread lee, al arreglo al cual señala ptr, hasta nmemb elementos cuyos tamaños están definidos por size, del flujo al cual señala stream. El indicador de posición de archivo para el flujo

(si está definido) se avanza en el número de caracteres con éxito leídos. Si ocurre algún error, queda indeterminado el valor resultante del indicador de posición de archivo para el flujo. Si se lee un elemento en forma parcial, su valor también queda indeterminado.

La función **fread** devuelve el número de elementos leídos, mismos que pudieran ser menos de **nmemb** si se encuentra con un error de lectura o con el fin de archivo. Si **size** o **nmemb** es cero, **fread** devuelve cero y tanto el contenido del arreglo como el estado del flujo se conservan sin modificar.

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
              FILE *stream);
```

La función **fwrite** escribe, desde el arreglo al cual señala **ptr** hasta **nmemb** elementos cuyo tamaño está definido por **size**, al flujo al cual señala **stream**. El indicador de posición de archivo para el flujo (si está definido) se avanza en el número de carácter escritos con éxito. Si ocurre un error, el valor resultante de la posición de archivo para el flujo queda indeterminado. La función **fwrite** devuelve el número de elementos escritos, mismo que sólo en el caso de que se encuentre un error de escritura será menor que **nmemb**.

```
int fgetpos(FILE *stream, fpos_t *pos);
```

La función **fgetpos** almacena el valor actual del indicador de posición de archivo para el flujo al cual señala **stream** en el objeto al cual apunta **pos**. El valor almacenado contiene información no especificada utilizable por la función **fsetpos** para recolocar el flujo a su posición en el momento de la llamada de la función **fgetpos**. En caso de éxito, la función **fgetpos** devuelve cero; en caso de falla, la función **fgetpos** devuelve no cero y almacena un valor positivo definido según la puesta en práctica en **errno**.

```
int fseek(FILE *stream, long int offset, int whence);
```

La función **fseek** define el indicador de posición de archivo para el flujo al cual señala **stream**. En el caso de un flujo binario, la nueva posición, medida en caracteres a partir del principio del archivo, se obtiene añadiendo **offset** a la posición especificada por **whence**. La posición especificada es el principio del archivo en el caso de que **whence** sea **SEEK_SET**, el valor actual del indicador de posición de archivo si **whence** es **SEEK_CUR**, o fin de archivo si es **SEEK_END**. Un flujo binario no necesariamente debe aceptar en forma significativa llamadas **fseek** con un valor **whence** de **SEEK_END**. En el caso de un flujo de texto, u **offset** deberá de ser cero, o un valor devuelto por una llamada anterior a la función **ftell** sobre el mismo flujo y **whence** deberá ser **SEEK_SET**.

Una llamada con éxito a la función **fseek** desactiva el indicador de fin de archivo para el flujo y deshace cualquier efecto de la función **ungetc** sobre el mismo archivo. Después de una llamada a **fseek**, la siguiente operación en un flujo de actualizar puede ser entrada o salida. La función **fseek** devuelve no cero sólo en el caso de una solicitud que no puede ser satisfecha.

```
int fsetpos(FILE *stream, const fpos_t *pos);
```

La función **fsetpos** define el indicador de posición de archivo para el flujo al cual señala **stream** de acuerdo al valor del objeto al cual señala **pos**, mismo que debe de ser un valor obtenido de una llamada anterior a la función **fgetpos** sobre el mismo flujo. Una llamada con éxito a la función **fsetpos** desactiva el indicador de fin de archivo para el flujo y deshace cualquier efecto de la función **ungetc** del mismo flujo. Después de una llamada a **fsetpos**, la siguiente operación sobre un flujo de actualizar puede ser introducida o extraída. Si tiene éxito, la función **fsetpos** devuelve cero; en caso de falla, la función **fsetpos** devuelve no cero y almacena un valor positivo definido según la puesta en práctica en **errno**.

```
long int ftell(FILE *stream);
```

La función **ftell** obtiene el valor actual del indicador de posición de archivo del flujo al cual señala **stream**. En el caso de un archivo binario, el valor es el número de caracteres a partir del principio del archivo. En el caso de un archivo de texto, el indicador de posición de archivo contiene información no especifica, utilizable por la función **fseek** para devolver el indicador de posición de archivo para el flujo a su posición en el momento de la llamada **ftell**; la diferencia entre estos dos valores devueltos no es necesariamente una medida significativa del número de caracteres escritos o leídos. En caso de éxito, una función **ftell** devuelve el valor actual del indicador de posición de archivo para el flujo. En caso de falla, la función **ftell** devuelve **-1L** y almacena un valor positivo definido por la puesta en práctica en **errno**.


```
void rewind(FILE *stream);
```

La función **rewind** define el indicador de posición de archivo para el flujo al cual señala **stream** al principio del mismo. Es equivalente a

```
(void) fseek(stream, 0L, SEEK_SET)
```

excepto que el indicador de error del flujo también es desactivado.

```
void clearerr(FILE *stream);
```

La función **clearerr** desactiva los indicadores de fin de archivo y de error para el flujo al cual señala **stream**.

```
int feof(FILE *stream);
```

La función **feof** prueba el indicador de fin de archivo en el flujo al cual señala **stream**. La función **feof** devuelve no cero si y sólo si el indicador de fin de archivo está activo para **stream**.

```
int ferror(FILE *stream);
```

La función **ferror** prueba el indicador de error para el flujo al cual señala **stream**. La función **ferror** devuelve no cero si y sólo si el indicador de error está activo para **stream**

```
void perror(const char *s);
```

La función **perror** traduce el número de error en la expresión entera **errno** en un mensaje de error. Escribe una secuencia de caracteres al flujo de error estándar, de tal forma que: primero (si **s** no es un apuntador nulo y el carácter al cual señala **s** no es un carácter nulo), escribe la cadena a la cual señala **s** seguida por (:) y un espacio; a continuación escribe un mensaje de error apropiado seguido por un carácter de nueva línea. El contenido de las cadenas de mensaje de error son las mismas que devuelve la función **strerror** mediante el argumento **errno**, que están definidas por la puesta en práctica.

B.11 Utilerías generales <stdlib.h>

EXIT_FAILURE
EXIT_SUCCESS

Expresiones enteras que pueden ser utilizadas como argumentos de la función **exit** para devolver estados de terminación con o sin éxito, al entorno huésped respectivamente.

MB_CUR_MAX

Una expresión entera positiva cuyo valor es el número máximo de bytes en un carácter de multibyte para el conjunto de caracteres extendido definido por el escenario actual (categoría **LC_CTYPE**), y cuyo valor nunca es mayor de **MB_LEN_MAX**.

NULL

Una constante de apuntador nulo definida por la puesta en práctica.

RAND_MAX

Una expresión constante de intervalo, cuyo valor es el valor máximo devuelto por la función **rand**. El valor de la macro **RAND_MAX** deberá ser por lo menos 32767.

div_t

Un tipo de estructura que es el tipo de valor devuelto por la función **div**.

ldiv_t

Un tipo de estructura que es el tipo de valor devuelto por la función **ldiv**.

size_t
El tipo entero no signado del resultado del operador **sizeof**.

wchar_t
Un tipo entero cuyo rango de valores puede representar códigos precisos para todos los miembros del conjunto de caracteres extendido más especifico entre escenarios soportados; el carácter nulo deberá tener el valor de código cero y cada miembro del conjunto de caracteres básico deberá tener un valor de código igual a su valor cuando se utilice como un carácter solo en una constante de caracteres entera.

```
double atof(const char *nptr);
```
Convierte la porción inicial de la cadena a la cual señala **nptr** a representación **double**. La función **atof** devuelve el valor convertido.

```
int atoi(const char *nptr);
```
Convierte la porción inicial de la cadena a la cual señala **nptr** a representación **int**. La función **atoi** devuelve el valor convertido.

```
long int atol(const char *nptr);
```
Convierte la porción inicial de la cadena a la cual señala **nptr** a representación **long**. La función **atol** devuelve el valor convertido.

```
double strtod(const char *nptr, char **endptr);
```
Convierte la porción inicial de la cadena a la cual señala **nptr** a representación **double**. Primero, divide en tres partes la cadena de entrada: una secuencia inicial, posiblemente vacía, de caracteres en blanco (tal y como se define en la función **isspace**), una secuencia sujeto, parecida a una constante de punto flotante; y una cadena final de uno o más caracteres no reconocidos, que incluyen el carácter nulo de terminación de la cadena de entrada. A continuación, intenta convertir la secuencia sujeto a un número de punto flotante y devuelve el resultado.

La forma expandida de la secuencia sujeto es un signo más o un signo menos opcional, a continuación una secuencia de dígitos no vacía, que de forma opcional contenga un carácter de punto decimal, a continuación una parte exponencial opcional, pero sin sufijo flotante. La secuencia sujeto se define como la subsecuencia inicial más extensa de la cadena de entrada, empezando con el primer carácter de espacio no en blanco, esto es de la forma esperada. La secuencia sujeto no contiene ningún carácter, si la cadena de entrada está vacía o está solo formada por espacios en blanco o si el carácter de no espacio en blanco, que está en primer término, es distinto de un signo, de un dígito, o de un carácter de punto decimal.

Si la secuencia sujeto tiene la forma esperada, la secuencia de caracteres que se inicia con el primer dígito o con el carácter de punto decimal (lo que ocurra primero) se interpreta como una constante flotante, excepto que el carácter de punto decimal se utilizará en lugar de un periodo, y que si no aparecen ni una parte exponencial ni un carácter de punto decimal, se supondrá un punto decimal que deberá seguir al último dígito en la cadena. Si la secuencia sujeto empieza con un signo menos, el valor que resulte de la conversión se hace negativa. Un apuntador a la cadena final se almacena en el objeto al cual señala **endptr**, siempre y cuando **endptr** no sea un apuntador nulo.

Si la secuencia sujeto está vacía o no tiene la forma esperada, no se llevará a cabo la conversión; el valor **nptr** se almacenará en el objeto al cual señala **endptr**, siempre y cuando que **endptr** no sea un apuntador nulo.

La función **strtod** devuelve el valor convertido si es que hubiera alguno. Si no se ha ejecutado conversión, devuelve cero. Si el valor correcto queda fuera del rango de los valores representables, se devuelve más o menos **HUGE_VAL** (según el signo del valor), y se almacena el valor de la macro **ERANGE** en **errno**. Si el valor correcto pudiera generar desbordamiento, se devuelve cero y el valor de la macro **ERANGE** queda almacenado en **errno**.


```
long int strtol(const char *nptr, char **endptr, int base);
```
Convierte la porción inicial de la cadena a la cual señala **nptr** a **long int**. Primero, descompone la cadena de entrada en tres partes: una secuencia inicial, posiblemente vacía, de caracteres de espacio en blanco (según queda especificado por la función **isspace**), una secuencia sujeto, que se parece a un entero representado en alguna raíz o base determinada por el valor de **base**, y una cadena final, de uno o más caracteres reconocibles, incluyendo el carácter nulo de terminación de la cadena de entrada. A continuación, intenta convertir la secuencia sujeto a un entero, y devuelve el resultado.

Si el valor de **base** es cero, la forma esperada de la secuencia sujeto es la de una constante entera, precedida en forma opcional de un signo más o menos, pero no incluyendo un sufijo entero. Si el valor de **base** es entre 2 y 36, la forma esperada de la secuencia sujeto es una secuencia de letras y dígitos que representan un entero con la raíz o base especificada por **base**, precedida en forma opcional por un signo más o menos, pero sin incluir un sufijo entero. Las letras desde **a** (o **A**) hasta **z** (o **Z**) reciben los valores 10 a 35; sólo se permiten aquellas letras cuyos valores adscritos sean menores que el de la **base**. Si el valor de la **base** es 16, los caracteres **0x** ó **0X** pudieran opcionalmente preceder a la secuencia de letras y de dígitos, a continuación del signo, si es que está presente.

La secuencia sujeto se define como la subsecuencia inicial más larga de la cadena de entrada, empezando a partir del primer carácter que no sea de espacio en blanco, esto es, de la forma esperada.

La secuencia sujeto no contiene ningún carácter, si la cadena de entrada está vacía o está formada sólo de espacios en blanco, o si el primer carácter distinto a espacio en blanco es diferente a un signo o a una letra o dígito permisible.

Si la secuencia sujeto tiene la forma esperada y el valor **base** es cero, la secuencia de caracteres que empieza a partir del primer dígito se interpreta como una constante entera. Si la secuencia sujeto tiene la forma esperada y el valor de la **base** es entre 2 y 36, se utiliza como base para la conversión, dándole a cada letra su valor como se especifica arriba. Si la secuencia sujeto empieza con un signo menos, se colocará un signo menos al valor resultante de la conversión. Un apuntador a la cadena final queda almacenado en el objeto al cual señala **endptr**, siempre y cuando **endptr** no sea nulo.

Si la secuencia sujeto está vacía o no tiene la forma esperada, la conversión no se llevará a cabo; el valor **nptr** se almacenará en el objeto al cual señala **endptr**, siempre y cuando **endptr** no sea un apuntador nulo.

La función **strtol** devuelve el valor convertido, si es que hubiera alguno. Si no se pudo ejecutar ninguna conversión, se devuelve cero. Si el valor correcto queda fuera del rango de valores representables, se devuelve **LONG_MAX** o **LONG_MIN** (de acuerdo al signo del valor), y el valor de la macro **ERANGE** queda almacenado en **errno**.

```
unsigned long int strtoul(const char *nptr, char **endptr, int base);
```
Convierte la porción inicial de la cadena a la cual señala **nptr** a una representación **unsigned long int**. La función **strtoul** funciona de forma idéntica a la función **strtol**. La función **strtoul** devuelve el valor convertido, si es que hay alguno. Se devuelve cero, si no se pudo llevar a cabo la conversión. Si el valor correcto queda por fuera del rango de los valores representables, se devuelve **ULONG_MAX**, y se almacena el valor de la macro **ERANGE** en **errno**.

```
int rand(void);
```
La función **rand** calcula una secuencia de enteros pseudo aleatorios de rango 0 hasta **RAND_MAX**. La función **rand** devuelve un entero pseudo aleatorio.

```
void srand(unsigned int seed);
```
Utiliza el argumento como semilla para una nueva secuencia de números pseudo aleatorios, a devolverse en subsecuentes llamadas a **rand**. Si entonces es llamada a **srand** con el mismo valor de semilla, la secuencia de los números pseudo aleatorios se repetirá. Si se llama a **rand** antes de cualquier llamada a **srand**, la misma secuencia será generada como si se hubiera llamado primero a **srand** con un valor de semilla de 1. La siguientes funciones definen una puesta en práctica portátil de **rand** y de **srand**.

```
static unsigned long int next = 1;
```

```
int rand(void) /* RAND_MAX assumed to be 32767 */
{
   next = next * 1103515245 + 12345;
   return (unsigned int) (next/65536) % 32768;
}

void srand(unsigned int seed)
{
   next = seed;
}
```

void *calloc(size_t nmemb, size_t size);

Asigna espacio para un arreglo de objetos **nmemb**, cada uno de los cuales tiene un tamaño de **size**. El espacio es inicializado para todos los bits cero. La función **calloc** devuelve al espacio asignado un apuntador nulo o un apuntador.

void free(void *ptr);

Hace que sea desasignado el espacio al cual señala **ptr**, esto es, se deje disponible para otra asignación. Si **ptr** es un apuntador nulo, no ocurrirá ninguna acción. De lo contrario, si el argumento no coincide con un apuntador ya devuelto por las funciones **calloc**, **malloc** o **realloc**, o si el espacio ya ha sido desasignado mediante una llamada a **free** o **realloc**, el comportamiento queda indefinido.

void *malloc(size_t size);

Asigna espacio para un objeto cuyo tamaño queda especificado por **size** y cuyo valor es indeterminado. La función **malloc** devuelve un apuntador nulo o un apuntador al espacio asignado.

void *realloc(void *ptr, size_t size);

Modifica el tamaño del objeto al cual señala **ptr** al tamaño definido por **size**. El contenido del objeto deberá conservarse sin modificar hasta el menor de entre los tamaños nuevo y viejo. Si el nuevo tamaño es mayor, el valor de la porción recién asignada del objeto queda indeterminada. Si **ptr** es un apuntador nulo, la función **realloc** se comporta como la función **malloc** para el tamaño especificado. De lo contrario, si **ptr** no encuentra un apuntador devuelto por las funciones **calloc**. **malloc**, o **realloc**, o si el espacio ha sido ya desasignado mediante una llamada a las funciones **free** o **realloc**, el comportamiento queda indefinido. Si el espacio no puede ser asignado, el objeto al cual apunta **ptr** queda sin modificación. Si **size** es cero y **ptr** no es nulo, el objeto al cual señala queda liberado. La función **realloc** devuelve o un apuntador nulo o un apuntador al espacio asignado posiblemente relocalizado.

void abort(void);

Hace que ocurra una terminación anormal de programa, a menos de que se detecte una señal **SIGABRT** y el manejador de señal no se devuelva. Queda dependiente de la puesta en marcha si se vaciarán los flujos de salida abiertos o si se cerrarán los flujos abiertos y serán eliminados los archivos temporales. Mediante la llamada de función **raise(SIGABRT** se devuelve al entorno huésped una forma dependiente de la puesta en práctica del estado de *terminación no exitoso*). La función **abort** no puede devolver a su llamador.

int atexit(void (*func)(void));

Registra la función a la cual apunta **func**, para que a la terminación normal de un programa ésta sea llamada sin argumentos. La puesta en práctica debe aceptar el registro de por lo menos 32 funciones. La función **atexit** devuelve cero si el registro tiene éxito, y no cero si falla.

void exit(int status);

Hace que ocurra una terminación normal del programa. Si el programa ejecuta más de una llamada a la función **exit**, el comportamiento queda indefinido. Primero, serán llamadas todas las funciones registradas por la función **atexit**, en orden inverso a la de su registro. Cada función será llamada

tantas veces como fue registrada. A continuación, serán vaciados todos los flujos abiertos con datos en búfer no escritos, serán cerrados todos los flujos abiertos, y serán eliminados todos los archivos creados mediante la función **tempfile**.

Por último, el control será devuelto al entorno huésped. Si el valor de **status** es cero o **EXIT_SUCCESS**, será devuelta una forma, definida por la puesta en práctica, del estado de *terminación con éxito*. Si el valor de **status** es **EXIT_FAILURE**, será devuelta una forma, definida por la puesta en práctica, del estado *terminación sin éxito*. De lo contrario, el estado devuelto queda definido por la puesta en práctica. La función **exit** no puede devolver a su llamador.

char *getenv(const char *name);

Busca en una *lista de entorno*, proporcionada por el entorno huésped, una cadena que coincida con la cadena a la cual apunta **name**. El conjunto de nombres de entorno y el método para modificar la lista de entorno, quedan definidas por la puesta en práctica. Devuelve un apuntador a una cadena asociada con el miembro de lista coincidente. La cadena a la cual apunta no será modificada por el programa, pero puede ser sobrescrita por una llamada subsiguiente a la función **getenv**. Si no puede ser encontrado el **name** especificado, devuelve un apuntador nulo.

int system(const char *string);

Pasa la cadena a la cual apunta **string** al entorno huésped para que sea ejecutada por un *procesador de comandos* de una forma definida por la puesta en práctica. Se puede utilizar un apuntador nulo en lugar de **string** para indagar si existe el procesador de comandos. Si el argumento es un apuntador nulo, la función **system** devolverá no cero sólo en el caso de que el procesador de comandos esté disponible. Si el argumento no es un apuntador nulo, la función **system** devolverá un valor definido por la puesta en práctica.

void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));

Busca en un arreglo de objetos **nmemb**, el elemento inicial al cual señala **base**, en busca de un elemento que coincida con el objeto al cual señala **key**. El tamaño de cada elemento del arreglo se especifica mediante **size**. Se llama a la función de comparación a la cual señala **compar**, con dos argumentos que apuntan al objeto **key** y a un elemento del arreglo, en este orden. La función devolverá un entero menor que, igual que o mayor que cero si el objeto **key** se considera, respectivamente, ser menor que, coincidir o ser mayor que el elemento del arreglo. El arreglo debe consistir de: todos los elementos que se comparan como menores que, todos los elementos que se comparan como iguales y todos los elementos que se comparan como mayores que el objeto **key**, en ese orden.

La función **bsearch** devuelve un apuntador a un elemento coincidente del arreglo, o un apuntador nulo, si no encuentra coincidencia. Si dos elementos son comparados como iguales, el elemento coincidente queda indeterminado.

void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));

Ordena un arreglo de **nmemb** objetos. El elemento inicial es el que señala a **base**. El tamaño de cada objeto está especificado por **size**. El contenido del arreglo es clasificado u ordenado en orden ascendente, de acuerdo con la función de comparación a la cual señala **compar**, misma que es llamada con dos argumentos, que apuntan a los objetos bajo comparación. La función devuelve un entero menor que o igual, o mayor que cero si el primer argumento se considera ser respectivamente menor que, igual a o mayor que el segundo. Si los dos elementos que se comparan son iguales, su orden en el arreglo clasificado queda indefinido.

int abs(int j);

Calcula el valor absoluto de un entero **j**. Si el resultado no puede ser representado, el comportamiento queda indefinido. La función **abs** devuelve el valor absoluto.

div_t div(int numer, int denom);

Calcula el cociente y residuo o módulo de la división del numerador **numer** entre el denominador **denom**. Si la división es inexacta, el cociente resultante es el entero de menor magnitud que resulte más cercano al cociente algebraico. Si el resultado no puede ser representado, el comportamiento queda indefinido; de lo contrario, **quot * denom + rem** debe ser igual a **numer**. La función **div** devuelve una estructura del tipo **div_t**, que comprende tanto el cociente como el residuo. La estructura debe contener los miembros siguientes, en cualquier orden:

```
int quot;    /* quotient */
int rem;     /* remainder */
```

```
long int labs(long int j);
```

Similar a la función **abs**, excepto que el argumento y el valor devuelto cada uno de ellos tiene el tipo **long int**.

```
ldiv_t ldiv(long int numer, long int denom);
```

Similar a la función **div**, excepto que el argumento y los miembros de la estructura devuelta (que tiene el tipo **ldiv_t**) todos ellos tienen el tipo **long int**.

```
int mblen(const char *s, size_t n);
```

Si **s** no es un apuntador nulo, la función **mblen** determina el número de bytes contenidos en el carácter multibyte al cual señala **s**. Si **s** es un apuntador nulo, la función **mblen** devuelve un valor no cero o cero, si las codificaciones de caracteres de multibyte, respectivamente tienen o no, codificaciones dependientes del estado. Si **s** no es un apuntador nulo, la función **mblen** devuelve cero (si **s** apunta al carácter nulo) o devuelve el número de bytes contenida en el carácter de multibyte (si los siguientes **n** o menos bytes forman un carácter de multibyte válidos), o devuelve -1 (si no forman un carácter de multibyte válido).

```
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

Si **s** no es un apuntador nulo, la función **mbtowc** determina el número de bytes contenidos en el carácter de multibyte al cual señala **s**. A continuación determina el código para el valor del tipo **wchar_t** que corresponda a dicho carácter de multibyte. (El valor del código que corresponda al carácter nulo es cero). Si el carácter de multibyte es válido y **pwc** no es un apuntador nulo, la función **mbtowc** almacena el código en el objeto al cual apunta **pwc**. Por lo menos n bytes del arreglo al cual apunta **s** serán examinados.

Si **s** es un apuntador nulo, la función **mbtowc** devuelve un valor no cero o cero, si las codificaciones de caracteres de multibyte, en forma respectiva tienen o no tienen codificaciones que dependan del estado. Si **s** no es un apuntador nulo, la función **mbtowc** devuelve cero (si **s** señala al carácter nulo) o devuelve el número de bytes que están contenidas en el carácter de multibyte convertido (si la **n** siguiente o menos bytes forman un carácter de varios bytes válidos), o devuelve -1 (si no forman un carácter de multibyte válido). En ninguno de los casos el valor devuelto será mayor que n o el valor de la macro **MB_CUR_MAX**.

```
int wctomb(char *s, wchar_t wchar);
```

La función **wctomb** determina el número de bytes necesarios para representar un carácter de multibyte correspondiendo al código cuyo valor es **wchar** (incluyendo cualquier modificación en estado de desplazamiento). Almacena la representación de caracteres de multibyte en el objeto de arreglo al cual apunta **s** (si **s** no es un apuntador nulo). Como máximo se almacenan **MB_CUR_MAX** caracteres. Si el valor de **wchar** es cero, la función **wctomb** se deja en el estado de desplazamiento inicial.

Si **s** es un apuntador nulo, la función **wctomb** devuelve un valor no cero o cero, en el caso que las codificaciones de caracteres de multibyte, respectivamente tengan o no tengan codificaciones dependiendo del estado. Si **s** no es un apuntador nulo, la función **wctomb** devuelve -1 si el valor de **wchar** no corresponde a un carácter de multibyte válido o devuelve el número de bytes contenidos en el carácter de multibyte correspondiente al valor de **wchar**. En ningún caso el valor devuelto será mayor que el valor de la macro **MB_CUR_MAX**.


```
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

La función **mbstowcs** convierte una secuencia de caracteres multibyte que empieza en el estado de desplazamiento inicial del arreglo al cual apunta **s** en una secuencia de códigos correspondientes y almacena no más de **n** códigos en el arreglo al cual señala **pwcs**. No se examinarán o convertirán caracteres de multibyte que sigan después de un carácter nulo (mismo que se convierte a un código con valor cero). Cada carácter multibyte se convierte igual que en el caso se una función **mbtowc**, excepto de que no se afecta el estado de desplazamiento de la función **mbtowc**.

No se modificarán más de **n** elementos en el arreglo al cual señala **pwcs**. El comportamiento de copiar entre objetos que se superpongan queda indefinido. Si se encuentra un carácter de multibyte inválido, la función **mbstowcs** devuelve (**size_t**)-1. De no ser así, la función **mbstowcs** devuelve el número de elementos de arreglos modificados, sin incluir el código cero de terminación, si es que hay alguno.

```
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

La función **wcstombs** convierte una secuencia de códigos que corresponden a caracteres de multibyte del arreglo al cual apunta **pwcs** en una secuencia de caracteres de multibyte que empieza en el estado inicial de desplazamiento y almacena estos caracteres de multibyte en el arreglo al cual señala **s**, deteniéndose si un carácter multibyte se excediese del límite total de bytes igual a **n** o si se almacena un carácter nulo. Cada código se convierte igual que en el caso de una llamada a una función **wctomb**, a excepción de que el estado de desplazamiento de la función **wctomb** no se afecta.

No se modificarán más de **n** bytes en el arreglo al cual señala **s**. Si la copia ocurre entre objetos que se superponen, el comportamiento queda indefinido. Si se encuentra con un código que no corresponda con un carácter válido de multibyte, la función **wcstombs** devuelve (**size_t**) -1. De lo contrario, la función **wcstombs** devuelve el número de bytes modificados, sin incluir el carácter nulo de terminación, si existiese.

B.12 Manejo de cadenas <string.h>

NULL

Una constante de apuntador nula, definida por la puesta en práctica.

size_t

El tipo integral no signado resultante del operador **sizeof**.

```
void *memcpy(void *s1, const void *s2, size_t n);
```

La función **memcpy** copia n caracteres del objeto al cual señala **s2** al objeto al cual señala **s1**. Si la copia tiene lugar entre objetos que se superponen, el comportamiento queda indefinido. La función **memcpy** devuelve el valor de **s1**.

```
void *memmove(void *s1, const void *s2, size_t n);
```

La función **memmove** copia **n** caracteres del objeto al cual señala **s2** en el objeto al cual señala **s1**. La copia se lleva a cabo como si los **n** caracteres del objeto al cual señala **s2** se copian primero a un arreglo temporal de **n** caracteres, que no se superponen sobre los objetos al cual señala **s1** y **s2**, y a continuación los **n** caracteres del arreglo temporal se copian al objeto al que señala **s1**. La función **memmove** devuelve el valor de **s1**.

```
char *strcpy(char *s1, const char *s2);
```

La función **strcpy** copia la cadena a la cual señala **s2** (incluyendo el carácter nulo de terminación) al arreglo al cual señala **s1**. Si la copia se lleva a cabo entre objetos que se superponen, el comportamiento queda indefinido. La función **strcpy** devuelve el valor de **s1**.

```
char *strncpy(char *s1, const char *s2, size_t n);
```

La función **strncpy** copia no más de **n** caracteres, (caracteres que sigan a un carácter nulo ya no son copiados) del arreglo al cual señala **s2** al arreglo al cual señala **s1**. Si la copia se lleva a cabo entre objetos que se superponen, el comportamiento queda indefinido. Si el arreglo al cual señala **s2** es una cadena más corta de **n** caracteres, se agregarán caracteres nulos a la copia en el arreglo al cual señala **s1**, hasta que en total se hayan escrito **n** caracteres. La función **strncpy** devuelve el valor de **s1**.

```
char *strcat(char *s1, const char *s2);
```

La función **strcat** agrega una copia de la cadena a la cual señala **s2** (incluye el carácter nulo de terminación) al final de la cadena a la cual señala **s1**. El carácter inicial de **s2** sobreescribe el carácter nulo existente al final de **s1**. Si la copia se lleva a cabo entre objetos que se superponen, el comportamiento queda indefinido. La función **strcat** devuelve el valor de **s1**.

```
char *strncat(char *s1, const char *s2, size_t n);
```

La función **strncat** agrega no más de **n** caracteres (el carácter nulo y los caracteres que le sigan no serán agregados) del arreglo al cual apunta **s2** al final de la cadena a la cual apunta **s1**. El carácter inicial de **s2** sobreescribe el carácter nulo al final de **s1**. Siempre se agregará un carácter nulo de terminación al resultado. Si la copia se lleva a cabo entre objetos que se superponen, el comportamiento queda indefinido. La función **strncat** devuelve el valor de **s1**.

```
int memcmp(const void *s1, const void *s2, size_t n);
```

La función **memcmp** compara los primeros **n** caracteres del objeto al cual señala **s1** con los primeros **n** caracteres del objeto al cual señala **s2**. La función **memcmp** devuelve un entero mayor que, igual a o menor que cero, según si el objeto al cual señala **s1** es mayor que, igual a o menor que el objeto al cual señala **s2**.

```
int strcmp(const char *s1, const char *s2);
```

La función **strcmp** compara la cadena a la cual señala **s1** con la cadena a la cual señala **s2**. La función **strcmp** devuelve un entero mayor que, igual a o menor que cero, según que la cadena a la cual señala **s1** sea mayor que, igual a o menor que la cadena a la cual señala **s2**.

```
int strcoll(const char *s1, const char *s2);
```

La función **strcoll** compara la cadena a la cual señala **s1** con la cadena a la cual señala **s2**, ambas interpretadas como apropiadas a la categoría **LC_COLLATE** del escenario actual. La función **strcoll** devuelve un entero mayor que, igual a o menor que cero, según que la cadena a la cual señala **s1** sea mayor que, igual a o menor que la cadena a la cual señala **s2** cuando ambos están interpretadas como apropiadas para el escenario actual.

```
int strncmp(const char *s1, const char *s2, size_t n);
```

La función **strncmp** compara no más de **n** caracteres (los caracteres que sigan a un carácter nulo no son comparados) del arreglo al cual señala **s1** con el arreglo al cual señala **s2**. La función **strncmp** devuelve un entero mayor que, igual a o menor que cero, según si el arreglo, posiblemente terminado en nulo, al cual señala **s1**, es mayor que, igual a o menor que el arreglo, posiblemente terminado en nulo, al cual señala **s2**.

```
size_t strxfrm(char *s1, const char *s2, size_t n);
```

La función **strxfrm** transforma la cadena a la cual señala **s2** y coloca la cadena resultante en el arreglo al cual señala **s1**. La transformación es tal que si la función **strcmp** se aplica a las dos cadenas transformadas, devuelve un valor mayor que igual a o menor que cero, en concordancia con el resultado de la función **strcoll** aplicable a las dos cadenas originales. No se colocan más de **n** caracteres en el arreglo resultante al cual señala **s1**, incluyendo el carácter nulo de terminación. Si **n** es cero, **s1** es posible que sea un apuntador nulo. Si la copia se lleva acabo entre objetos que se superponen, el comportamiento queda indefinido. La función **strxfrm** devuelve la longitud de la cadena transformada (sin incluir el carácter nulo de terminación). Si el valor es **n** o más, el contenido del arreglo al cual señala **s1** queda indeterminado.


```
void *memchr(const void *s, int c, size_t n);
```

La función **memchr** localiza la primera instancia de **c** (convertida a un **unsigned char**) en los **n** caracteres iniciales (cada uno de ellos interpretados como un **unsigned char**) del objeto al cual señala **s**. La función **memchr** devuelve un apuntador al carácter localizado, o un apuntador nulo si el carácter no existe en el objeto.

```
char *strchr(const char *s, int c );
```

La función **strchr** localiza la primera instancia de **c** (convertido a un **char**) en la cadena a la cual señala **s**. El carácter nulo de terminación se considera como parte de la cadena. La función **strchr** devuelve un apuntador al carácter localizado, o un apuntador nulo si el carácter no está incluido en la cadena.

```
size_t strcspn(const char *s1, const char *s2);
```

La función **strcspn** calcula la longitud del segmento inicial máximo de la cadena a la cual señala **s1** que está formada por completo de caracteres que no sean de la cadena a la cual señala **s2**. La función **strcspn** devuelve la longitud del segmento.

```
char *strpbrk(const char *s1, const char *s2);
```

La función **strpbrk** localiza la primera instancia de la cadena a la cual señala **s1** de cualquier carácter de la cadena a la cual señala **s2**. La función **strpbrk** devuelve un apuntador al carácter, o un apuntador nulo si ningún carácter de **s2** ocurre en **s1**.

```
char *strrchr(const char *s, int c);
```

La función **strrchr** localiza la última instancia de **c** (convertido a **char**) en la cadena a la cual señala **s**. El carácter nulo de terminación se considera como parte de la cadena. La función **strrchr** devuelve un apuntador al carácter, o un apuntador nulo si **c** no está incluido en la cadena.

```
size_t strspn(const char *s1, const char *s2);
```

La función **strspn** calcula la longitud del segmento inicial máximo de la cadena a la cual señala **s1**, que consista en totalidad de caracteres de la cadena a la cual señala **s2**. La función **strspn** devuelve la longitud del segmento.

```
char *strstr(const char *s1, const char *s2);
```

La función **strstr** localiza la primera instancia en la cadena a la cual señala **s1** de la secuencia de caracteres (excluyendo el carácter nulo de terminación) en la cadena a la cual señala **s2**. La función **strstr** devuelve un apuntador a la cadena localizada, o un apuntador nulo si la cadena no se encuentra. Si **s2** señala a una cadena de longitud cero, la función devuelve **s1**.

```
char *strtok(char *s1, const char *s2);
```

Una secuencia de llamadas a función **strtok** divide la cadena a la cual señala **s1** en una secuencia de componentes léxicos, cada uno de los cuales queda delimitado por un carácter de la cadena a la cual señala **s2**. La primera llamada en la secuencia tiene como argumento **s1**, y es seguida por llamadas con un apuntador nulo como primer argumento. La cadena separadora a la cual señala **s2** puede cambiar de llamada a llamada.

La primera llamada en la secuencia busca en la cadena a la cual señala **s1** el primer carácter que no esté contenido en la cadena separadora actual a la cual señala **s2**. Si dicho carácter no se encuentra, entonces no existen componentes léxicos en la cadena a la cual señala **s1** y la función **strtok** devuelve un apuntador nulo. Si se encuentra dicho carácter, ese es el inicio del primer componente léxico.

La función **strtok** a continuación busca a partir de ahí un carácter que esté contenido en la cadena separadora actual. Si no encuentra dicho carácter, el componente léxico actual se extiende hasta el fin de la cadena a la cual señala **s1**, y las búsquedas subsecuentes para un componente léxico devolverán un apuntador nulo. Si se encuentra dicho carácter, queda sobreescrito por un carácter nulo, lo cual da por terminado el componente léxico actual. La función **strtok** guarda un apuntador al carácter siguiente, a partir del cual se iniciará el siguiente componente léxico.

Cada llamada subsiguiente, con un apuntador nulo como valor de su primer argumento, empieza a buscar a partir del apuntador guardado y se comporta como se describió anteriormente. La puesta en práctica debe comportarse como si ninguna función de biblioteca llama a la función `strtok`. La función `strtok` devuelve un apuntador al primer carácter de un componente léxico, o a un apuntador nulo si tal componente no existe.

`void *memset(void *s, int c, size_t n);`

La función `memset` copia el valor de `c` (convertido a un `unsigned char`) a cada uno de los primeros `n` caracteres en el objeto al cual señala `s`. La función `memset` devuelve el valor de `s`.

`char *strerror(int errnum);`

La función `strerror` relocaliza el número de error en `errnum` a una cadena de mensaje de error. La puesta en práctica deberá comportarse como si ninguna función de biblioteca llama a la función `strerror`. La función `strerror` devuelve un apuntador a la cadena, el contenido de la cual está definido por la puesta en práctica. El arreglo al cual se apunta no debe modificarse por el programa, que pudiera ser sobreescrito debido a una llamada subsiguiente a la función `strerror`.

`size_t strlen(const char *s);`

La función `strlen` computa la longitud de la cadena a la cual señala `s`. La función `strlen` devuelve el número de caracteres que anteceden al carácter nulo de terminación.

B.13 Fecha y hora <time.h>

`CLOCKS_PER_SEC`

El número por segundo del valor devuelto por la función `clock`.

`NULL`

Constante de apuntador nula, definida por la puesta en práctica.

`clock_t`

Un tipo aritmético capaz de representar la fecha.

`time_t`

Un tipo aritmético capaz de representar la hora.

`size_t`

Un tipo entero no signado del resultado del operador `sizeof`.

`struct tm`

Contiene los componentes de una hora de calendario, conocida como *hora desglosada*. La estructura deberá contener por lo menos los siguientes miembros, en cualquier orden. La semántica de los miembros y sus rangos normales se expresan en los comentarios.

```
int tm_sec;     /* seconds after the minute—[0,61] */
int tm_min;     /* minutes after the hour—[0,59] */
int tm_hour;    /* hours since midnight—[0,23] */
int tm_mday;    /* day of the month—[1,31] */
int tm_mon;     /* months since January—[0,11]*/
int tm_year;    /* year since 1900 *
int tm_wday;    /* days since Sunday—[0,6] */
int tm_yday;    /* days since January 1—[0,365] */
int tm_isdst;   /* Daylight Saving Time flag */
```


El valor `tm_isdst` es positivo si está en efecto Daylight Saving Time, es cero si no está en efecto Daylight Saving Time, y negativo si la información no está disponible.

`clock_t clock(void)`

La función `clock` determina la hora de procesador utilizada. La función `clock` devuelve la mejor aproximación de la puesta en práctica de la hora del procesador utilizada por el programa desde el principio de una era, definida por la puesta en práctica, únicamente relacionada con la invocación del programa. Para determinar la hora en segundos, el valor devuelto por la función `clock` deberá ser dividido por el valor de la macro `CLOCKS_PER_SEC`. Si la hora del procesador utilizada no está disponible o su valor no puede ser representado, la función devuelve el valor `(clock_t)-1`.

`double difftime(time_t time1, time_t time0);`

La función `difftime` calcula la diferencia entre dos horas de calendario: `time1 -time0`. La función `difftime` devuelve la diferencia, expresada en segundos, como un `double`.

`time_t mktime(struct tm *timeptr);`

La función `mktime` convierte la hora desglosada, expresada como hora local, en la estructura a la cual señala `timeptr` en un valor de hora de calendario, con la misma codificación que la de los valores devueltos por la función `time`. Serán ignorados los valores originales de los componentes `tm_wday` y `tm_yday` de la estructura, y los valores originales de los demás componentes no quedarán restringidos a los rangos arriba indicados. A la terminación exitosa, se ajustarán apropiadamente los valores de los componentes `tm_wday` y `tm_yday` y los demás componentes se ajustarán para representar la hora de calendario específico, pero con sus valores forzados a los rangos arriba indicados; el valor final de `tm_mday` no se define hasta que estén determinados `tm_mon` y `tm_year`. La función `mktime` devuelve la hora especificada de calendario codificada como un valor del tipo `time_t`. Si la hora de calendario no puede representarse, la función devuelve el valor `(time_t)-1`.

`time_t time(time_t *timer);`

La función `time` determina la hora de calendario actual. La función `time` devuelve la mejor aproximación según la puesta en práctica de la hora de calendario actual. El valor `(time_t)-1` será devuelto si dicha hora de calendario no está disponible. Si `timer` no es un apuntador nulo, el valor devuelto también se asignará al objeto al cual apunta.

`char *asctime(const struct tm *timeptr);`

La función `asctime` convierte la hora desglosada en la estructura a la cual señala `timeptr` en una cadena de la forma

```
Sun Sep 16 01:03:52 1973\n\0
```

La función `asctime` devuelve un apuntador a la cadena.

`char *ctime(const time_t *timer);`

La función `ctime` convierte la hora de calendario al cual señala `timer` a hora local en forma de una cadena. Es equivalente a

```
asctime(localtime(timer))
```

La función `ctime` devuelve el apuntador devuelto por la función `asctime` con dicha hora desglosada como argumento.

`struct tm *gmtime(const time_t *timer);`

La función `gmtime` convierte la hora de calendario al cual señala `timer` en una hora desglosada, expresada en forma de Coordinated Universal Time (UTC) La función `gmtime` devuelve un apuntador a dicho objeto, o un apuntador nulo si UTC no está disponible.

`struct tm *localtime(const time_t *timer);`

La función `localtime` convierte la hora de calendario al cual señala `timer` en una hora desglosada, expresada en forma de hora local. La función `localtime` devuelve un apuntador a dicho objeto.

```
size_t strftime(char *s, size-t maxsize, const char *format, const
                struct tm *timeptr);
```

La función **strftime** coloca caracteres en el arreglo al cual señala **s** controlados por la cadena a la cual señala **format**. La cadena **format** consiste de un cero o más especificadores de conversión y caracteres ordinarios de varios bytes. Todos los caracteres ordinarios (incluyendo el carácter nulo de terminación) serán copiados al arreglo sin modificarse. Si la copia se hace entre objetos que se superponen, el comportamiento quedará indefinido. No se colocarán más caracteres que los especificados en **maxsize** en el arreglo. Cada especificador de conversión será remplazado por los caracteres apropiados tal y como se describe en la lista siguiente. Los caracteres apropiados quedan determinados por la categoría **LC_TIME** del escenario actual y por los valores contenidos en la estructura a la cual señala **timeptr**.

- **%a** es reemplazado por la abreviatura del nombre del día de la semana del escenario.
- **%A** es reemplazado por el nombre completo del día de la semana del escenario.
- **%b** es reemplazado por la abreviatura del nombre del mes del escenario.
- **%B** es reemplazado por el nombre completo del mes del escenario.
- **%c** es reemplazado por la representación apropiada de fecha y de hora del escenario.
- **%d** es reemplazado por el día del mes como número decimal (**01-31**).
- **%H** es reemplazado por la hora (reloj de 24 horas) como número decimal (**00-23**).
- **%I** es reemplazado por la hora (reloj de 12 horas) como número decimal (**01-12**).
- **%j** es reemplazado por el día del año como número decimal (**001-366**).
- **%m** es reemplazado por el mes como número decimal (**01-12**).
- **%M** es reemplazado por el minuto como número decimal (**00-59**).
- **%p** es reemplazado por el equivalente para el escenario de las designaciones AM/PM asociadas con un reloj de 12 horas.
- **%S** es reemplazado como segundos como número decimal (**00-61**).
- **%U** es reemplazado por el número de la semana correspondiente del año (siendo el primer domingo el primer día de la semana 1) como número decimal (**00-53**).
- **%w** es reemplazado por el día de la semana como número decimal (**0-6**), donde domingo es **0**.
- **%W** es reemplazado por el número de la semana correspondiente del año (el primer lunes es el primer día de la semana 1) como número decimal (**00-53**).
- **%x** es reemplazado por la representación apropiada de la fecha, de acuerdo con el escenario.
- **%X** es reemplazado por la representación apropiada de la hora para el escenario.
- **%y** es reemplazado por el año, sin siglos, como un número decimal (**00-99**).
- **%Y** es reemplazado por el año, incluyendo siglos, como un número decimal.
- **%Z** es reemplazado por el nombre o abreviatura de la zona de tiempo, y mediante ningún carácter, si no hay determinable ninguna zona de tiempo.
- **%%** es reemplazado por **%**.

Si el especificador de conversión no es ninguno de los arriba citados, el comportamiento queda indefinido. Si el número total de caracteres resultantes, incluyendo el carácter nulo de terminación no es más de **max-size**, la función **strftime** devuelve el número de carácter colocados en el arreglo al cual señala **s**, sin incluir el carácter nulo de terminación. De lo contrario, se devuelve cero y el contenido del arreglo queda indeterminado.

B.14 Límites de puesta en práctica

<limits.h>

Los siguientes deben ser definidos en magnitud (valor absoluto) como igual a o mayor que los valores siguientes.

#define CHAR_BIT **8**
El número de bits para el objeto más pequeño que no sea un campo de bits (byte).

#define SCHAR_MIN **-127**
El valor mínimo para un objeto del tipo **signed char**.

#define SCHAR_MAX **+127**
El valor máximo para un objeto del tipo **signed char**.

#define UCHAR_MAX **255**
El valor máximo para un objeto del tipo **unsigned char**.

#define CHAR_MIN **0 ó SCHAR_MIN**
El valor mínimo para un objeto del tipo **char**.

#define CHAR_MAX **UCHAR_MAX ó SCHAR_MAX**
El valor máximo para un objeto del tipo **char**.

#define MB_LEN_MAX **1**
El número máximo de bytes de un carácter multibyte, para cualquier escenario soportado.

#define SHRT_MIN **-32767**
El valor mínimo para un objeto del tipo **short int**.

#define SHRT_MAX **+32767**
El valor máximo para un objeto del tipo **short int**.

#define USHRT_MAX **65535**
El valor máximo para un objeto del tipo **unsigned short int**.

#define INT_MIN **-32767**
El valor mínimo para un objeto del tipo **int**.

#define INT_MAX **+32767**
El valor máximo para un objeto del tipo **int**.

#define UINT_MAX **65535**
El valor máximo para un objeto del tipo **unsigned int**

#define LONG_MIN **-2147483647**
El valor mínimo para un objeto del tipo **long int**.

#define LONG_MAX **+2147483647**
El valor máximo para un objeto del tipo **long int**

#define ULONG_MAX **4294967295**
El valor máximo para un objeto del tipo **unsigned long int**

<float.h>

#define FLT_ROUNDS
El modo de redondeo para la suma en punto flotante

- **-1** indeterminable
- **0** hacia cero

1 hacia el más cercano

2 hacia infinito positivo

3 hacia infinito negativo

Lo siguiente debe ser definido en magnitud (valor absoluto) igual a o mayor que a los valores siguientes.

```
#define FLT_RADIX                    2
```

La raíz de la representación exponencial, *b*.

```
#define FLT_MANT_DIG
#define LDBL_MANT_DIG
#define DBL_MANT_DIG
```

El número de base **FLT_RADIX** dígitos en el significando del punto flotante *p*.

```
#define FLT_DIG                      6
#define DBL_DIG                     10
#define _LDB_DIG                    10
```

El número de dígitos decimales *q*, tal que cualquier número de punto flotante con *q* dígitos decimales pueda ser redondeado a un número de punto flotante con *p* raíz *b* dígitos y de regreso otra vez sin modificación a los dígitos decimales *q*.

```
#define FLT_MIN_EXP
#define DBL_MIN_EXP
#define LDBL_MIN_EXP
```

El entero negativo mínimo tal que **FLT_RADIX** elevado a dicha potencia menos 1 es un número de punto flotante normalizado.

```
#define FLT_MIN_10_EXP             -37
#define DBL_MIN_10_EXP             -37
#define LDBL_MIN_10_EXP            -37
```

El entero negativo mínimo tal que 10 elevado a dicha potencia está en el rango de los números de punto flotante normalizados.

```
#define FLT_MAX_EXP
#define DBL_MAX_EXP
#define LDBL_MAX_EXP
```

El entero máximo tal que **FLT_RADIX** elevado a dicha potencia menos 1 es un número finito de punto flotante representable.

```
#define FLT_MIN_10_EXP             +37
#define DBL_MIN_10_EXP             +37
#define LDBL_MIN_10_EXP            +37
```

El entero máximo tal que 10 elevado a esa potencia está en el rango de los números de punto flotante finitos y representables.

Lo siguiente deberá ser definido igual a o mayor que los valores mostrados a continuación.

```
#define FLT_MAX                    1E+37
#define DBL_MAX                    1E+37
#define LDBL_MAX                   1E+37
```

El número de punto flotante finito representable máximo.

Lo siguiente debe ser definido igual a o menor que los valores mostrados a continuación.

```
#define FLT_EPSILON                1E-5
```



```
#define DBL_EPSILON                1E-9
#define LDBL_EPSILON               1E-9
```

La diferencia entre 1.0 y el valor mínimo mayor que 1.0 que sea representable en el tipo dado de punto flotante.

```
#define FLT_MIN                    1E-37
#define DBL_MIN                    1E-37
#define LDBL_MIN                   1E-37
```

El número de punto flotante positivo normalizado mínimo.

Apéndice C

Precedencia y asociatividad de operadores

Operador	Asociatividad
() [] -> .	izquierda a derecha
++ — + - ! ~ (tipo) * & sizeof	derecha a izquierda
* / %	izquierda a derecha
+ -	izquierda a derecha
<< >>	izquierda a derecha
< <= > >=	izquierda a derecha
== !=	izquierda a derecha
&	izquierda a derecha
^	izquierda a derecha
|	izquierda a derecha
&&	izquierda a derecha
||	izquierda a derecha
?:	derecha a izquierda
= += -= *= /= %= &= ^= |= <<= >>=	derecha a izquierda
,	izquierda a derecha

Los operadores aparecen de la parte superior a la inferior en orden decreciente de precedencia.

Apéndice D

Conjunto de caracteres ASCII

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	'	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

Los dígitos a la izquierda de la tabla son los dígitos izquierdos del equivalente decimal del código de caracteres, y los dígitos en la parte superior de la tabla son los dígitos derechos del código de caracteres. Por ejemplo, el código de carácter para 'F' es 70, y el correspondiente para & es 38.

APÉNDICE E

Sistemas numéricos

Objetivos

- Comprender los conceptos básicos de los sistemas numéricos como base, valor posicional y valor simbólico.
- Comprender cómo trabajar con números representados en sistemas numéricos binario, octal, y hexadecimal.
- Ser capaz de reducir los números binarios a números octales o a números hexadecimales.
- Ser capaz de convertir los números octales y hexadecimales a números binarios.
- Ser capaz de convertir en ambos sentidos entre números decimales y sus equivalentes binario, octal y hexadecimal.
- Comprender la aritmética binaria y cómo se representan los números binarios negativos mediante la notación de complemento a dos.

He aquí sólo números ratificados.
William Shakespeare

La naturaleza tiene cierta clase de sistema de coordenadas geométrico aritméticas, porque la naturaleza tiene todo tipo de modelos. Lo que nosotros concebimos de la naturaleza está en los modelos y todos los modelos de la naturaleza son tan bellos. Se me ocurre que el sistema de la naturaleza debe ser una verdadera belleza, porque en química encontramos que las asociaciones se presentan siempre en bellos números enteros —no existen fracciones.
Richard Buckminster Fuller

Sinopsis

Resumen • Terminología • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios.

E.1 Introducción

En este apéndice, presentamos los sistemas numéricos clave utilizados por los programadores de C, en especial cuando están trabajando sobre proyectos de software que requieren una cercana interacción a "nivel máquina". Con el hardware proyectos como éstos incluyen sistemas operativos, software de redes de cómputo, compiladores, sistemas de base de datos y aplicaciones que requieran de alto rendimiento.

Cuando en un programa en C escribimos un entero como 227 o -63, se supone que el número está escrito en *sistema numérico decimal (de base 10)*. En el sistema numérico decimal los dígitos son 0, 1, 2, 3, 4, 5, 6, 7, 8, y 9. El dígito menor es 0 y el mayor es 9 —uno menos que la *base*, que es 10. En forma interna, las computadoras utilizan el *sistema numérico binario (de base 2)*. El sistema numérico binario utiliza únicamente dos dígitos, es decir 0 y 1. Su dígito menor es 0 y su dígito mayor es 1 —uno menos que la base de 2.

Como veremos, los números binarios tienden a ser mucho más largos que sus equivalentes decimales. Los programadores que tienen que trabajar en lenguajes de ensambladores, y en lenguajes de alto nivel como C que permiten a los programadores llegar hasta el "nivel máquina", por lo que otros dos sistemas numéricos se han hecho populares el *sistema numérico octal (de base 8) y el sistema numérico hexadecimal (de base 16)* primordialmente debido a que resultan convenientes para abreviar números binarios.

En el sistema numérico octal, los dígitos tienen un rango del 0 al 7. Dado que tanto el sistema numérico binario como el sistema numérico octal tienen menos dígitos que el sistema numérico decimal, sus dígitos son los mismos que los correspondientes dígitos en decimal.

El sistema numérico hexadecimal presenta un problema, porque requiere de dieciséis dígitos —un dígito menor 0 y un dígito mayor, con un valor equivalente al 15 decimal (uno menos que la base de 16). Por regla convencional, utilizamos las letras A hasta F para representar los dígitos hexadecimales que corresponden a los valores decimales del 10 hasta el 15. Por lo tanto en hexadecimal podemos tener números como 876, que están formados únicamente por dígitos parecidos a los decimales, números como 8A55F, que están formados de dígitos y de letras, y números como FFE, que sólo están formados por letras. Ocasionalmente, un número hexadecimal deletrea una palabra común, como FACE, o bien FEED y esto pudiera parecer extraño a aquellos programadores acostumbrados a trabajar solo con números.

Cada uno de estos sistemas numéricos utilizan *notación posicional* —la posición en la cual se escribe un dígito tiene un *valor posicional diferente*. Por ejemplo, en el número decimal 937 (el 9, el 3 y el 7 se conocen como *valores simbólicos*), decimos que el 7 está escrito en la *posición de las unidades*, el 3 está escrito en la *posición de las decenas*, y el 9 está escrito en la *posición de las centenas*. Note que cada una de estas posiciones es una potencia de la base (base 10), y estas potencias empiezan en 0 y se incrementan en 1 conforme nos movemos hacia la izquierda dentro del número (figura E.3).

Dígito binario	Dígito octal	Dígito decimal	Dígito hexadecimal
0	0	0	0
1	1	1	1
	2	2	2
	3	3	3
	4	4	4
	5	5	5
	6	6	6
	7	7	7
		8	8
		9	9
			A (valor decimal 10)
			B (valor decimal 11)
			C (valor decimal 12)
			D (valor decimal 13)
			E (valor decimal 14)
			F (valor decimal 15)

Fig. E.1 Dígitos de los sistemas numéricos binario, octal, decimal y hexadecimal.

Atributo	Binario	Octal	Decimal	Hexadecimal
Base	2	8	10	16
Dígito menor	0	0	0	0
Dígito mayor	1	7	9	F

Fig. E.2 Comparación de los sistemas numéricos binario, octal, decimal y hexadecimal.

Valores posiciones en el sistema numérico decimal			
Dígito decimal	9	3	7
Nombre de la posición	Centenas	Decenas	Unidades
Valor posicional	100	10	1
Valor posicional como una potencia de la base (10)	10^2	10^1	10^0

Fig. E.3 Valores posicionales en el sistema numérico decimal.

Para números decimales más grandes, las siguientes posiciones a la izquierda serían la *posición de los millares* (10 a la tercera potencia), la *posición de los diez millares* (10 a la cuarta potencia), la *posición de los cien millares* (10 a la quinta potencia), la *posición de los millones* (10 a la sexta potencia), la *posición de los diez millones* (10 a la séptima potencia), y así en lo sucesivo.

En el número binario 101, decimos que el 1 más a la derecha está escrito en la *posición de las unidades*, el 0 está escrito en la *posición de los dos*, y el 1 más a la izquierda está escrito en la *posición de los cuatros*. Note que cada una de estas posiciones es una potencia de la base (base 2), y que estas potencias empiezan en 0 y se incrementan por 1 conforme nos movemos a la izquierda en el número (figura E.4).

Para números binarios más grandes, la siguiente posición a la izquierda sería la *posición de los ochos* (2 a la tercera potencia), la *posición de los diez y seis* (2 a la cuarta potencia), la *posición de los treinta y dos* (2 a la quinta potencia), la *posición de los sesenta y cuatro* (2 a la sexta potencia), y así en lo sucesivo.

En el número octal 425, decimos que el 5 está escrito en la *posición de las unidades*, el 2 está escrito en la *posición de los ochos*, y el 4 está escrito en la *posición de los sesenta y cuatros*. Note que cada una de estas posiciones es una potencia de la base (base 8), y que estas potencias empiezan en 0 y se incrementan en 1 conforme vamos hacia la izquierda en el número (figura E. 5).

Para números octales más grandes, las siguientes posiciones a la izquierda serían la *posición de los quinientos doces* (8 a la tercera potencia), la *posición de los cuatro mil noventa y seis* (8 a la cuarta potencia), la *posición de los treinta y dos mil setecientos y sesenta y ochos* (8 a la quinta potencia), y así en lo sucesivo.

En el número hexadecimal 3DA, decimos que la A está escrita en la *posición de las unidades*, la D está escrita en la *posición de los diez y seis*, y el 3 está escrito en la *posición de los doscientos cincuenta y seis*. Note que cada una de estas posiciones es una potencia de la base (base 16) y que estas potencias empiezan en 0 y se incrementan en 1 conforme vamos hacia la izquierda en el número (figura E.6).

Valores posicionales en el sistema numérico binario

Dígito binario	1	0	1
Nombre de la posición	Cuatros	Dos	Unidades
Valor posicional	4	2	1
Valor posicional como una potencia de la base (2)	2^2	2^1	2^0

Fig. E.4 Valores posicionales en el sistema numérico binario.

Valores posicionales en el sistema numérico octal

Dígito octal	4	2	5
Nombre de la posición	Sesenta y cuatros	Ochos	Unidades
Valor posicional	64	8	1
Valor posicional como una potencia de la base (8)	8^2	8^1	8^0

Fig. E.5 Valores posicionales en el sistema numérico octal.

Valores posicionales en el sistema numérico hexadecimal

Dígito hexadecimal	3	D	A
Nombre de la posición	Doscientos cincuenta y seis	Diez y seis	Unidades
Valor posicional	256	16	1
Valor posicional como una potencia de la base (16)	16^2	16^1	16^0

Fig. E.6 Valores posicionales en el sistema numérico hexadecimal.

Para números hexadecimales mayores, la siguientes posiciones a la izquierda serían la *posición cuatro mil noventa y seis* (16 a la tercera potencia), la *posición treinta y dos mil setecientos y sesenta y ochos* (16 a la cuarta potencia), y así sucesivamente.

E.2 Cómo abreviar números binarios como octales y hexadecimales

El uso principal para los números octales y hexadecimales en la computación es para abreviar representaciones binarias largas. En la figura E.7 se destaca el hecho que números binarios largos pueden ser expresados en forma concisa en sistemas numéricos con bases más altas que el sistema numérico binario.

Número decimal	Representación binaria	Representación octal	Representación hexadecimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

Fig. E.7 Equivalentes decimal, binario, octal y hexadecimal.

Una relación particular importante que tienen tanto el sistema numérico octal como el hexadecimal con el sistema binario, es que las bases de los sistemas octal y hexadecimal (8 y 16 respectivamente), son potencias de la base del sistema numérico binario (base 2). Considere el siguiente número binario de 12 dígitos y sus equivalentes octal y hexadecimal. Vea si puede determinar cómo resulta conveniente esta relación para abreviar números binarios a octal y hexadecimal. La respuesta sigue a los números.

Número binario	Equivalente octal	Equivalente hexadecimal
100011010001	4321	8D1

Para ver como los números binarios se convierten con fácilidad a octal, sólo divida el número binario de 12 dígitos en grupos cada uno de ellos de tres bits consecutivos, y escriba dichos grupos sobre los dígitos correspondientes del número octal como sigue

```
100     011     010     001
4       3       2       1
```

Note que el dígito octal que usted ha escrito bajo cada número de estos tres bits corresponde precisamente al equivalente octal del número binario de tres dígitos, tal y como se muestra en la figura E.7.

El mismo tipo de relación se puede observar en la conversión de números de binario a hexadecimal. En particular divida el número binario de 12 dígitos en grupos de cuatro bits consecutivos cada uno y escriba estos grupos sobre los dígitos correspondientes al número hexadecimal como sigue

```
1000    1101    0001
8       D       1
```

Note que el dígito hexadecimal que ha escrito bajo cada grupo de cuatro bits corresponde precisamente al equivalente hexadecimal de dicho número binario de cuatro dígitos, tal y como se muestra en la figura E.7.

E.3 Cómo convertir números octales y hexadecimales a binarios

En la sección anterior, vimos cómo convertir números binarios a sus equivalentes octal y hexadecimal formando grupos de dígitos binarios y sólo reescribiendo estos grupos en sus valores digitales octales o hexadecimales. Este proceso pude ser utilizado en reversa para producir el equivalente binario de un número octal o hexadecimal dado.

Por ejemplo, el número octal 653 se convierte a binario simplemente escribiendo el 6 como su equivalente binario de 3 dígitos 110, el 5 como su equivalente binario de 3 dígitos 101, y el 3 como su equivalente binario de 3 dígitos 011 para formar el número binario de 9 dígitos 110101011.

El número hexadecimal FAD5 se convierte a binario simplemente escribiendo la F como su equivalente binario de 4 dígitos 1111, la A como su equivalente binario de 4 dígitos 1010, la D como su equivalente binario de 4 dígitos 1101, y el 5 como su equivalente binario de 4 dígitos 0101 para formar el número de 16 dígitos 1111101011010101.

E.4 Cómo convertir de binario, octal y hexadecimal a decimal

Dado que estamos acostumbrados a escribir en decimal, a menudo resulta conveniente convertir un número binario, octal o hexadecimal a decimal para darnos cuenta de lo que "realmente" vale ese número. Nuestros diagramas en la sección E.1 expresan los valores posicionales en decimal. Para convertir un número de decimal a otra base, multiplique el equivalente decimal de cada dígito por su valor posicional, y sume dichos productos. Por ejemplo, el número binario 110101 se convierte al decimal 53 tal y como se muestra en la figura E.8.

Cómo convertir un número binario a decimal

Valores posicionales:	32	16	8	4	2	1
Valores simbólicos:	1	1	0	1	0	1
Productos:	1*32=32	1*16=16	0*8=0	1*4=4	0*2=0	1*1=1
Suma:	= 32 + 16 + 0 + 4 + 0 + 1 = 53					

Fig. E.8 Cómo convertir un número binario a decimal.

Para convertir el octal 7614 al decimal 3980, utilizamos la misma técnica, esta vez utilizando los valores posicionales octales apropiados, tal y como se muestra en la figura E.9.

Para convertir el valor hexadecimal AD3B al decimal 44347, utilizamos la misma técnica, esta vez utilizando los valores posicionales hexadecimales apropiados, como se muestra en la figura E.10.

E.5 Cómo convertir de decimal a binario, octal o hexadecimal

Las conversiones de la sección anterior son una consecuencia natural de las reglas convencionales de la notación posicional. La conversión de decimal a binario, octal o hexadecimal también sigue estas reglas convencionales.

Suponga que deseamos convertir el decimal 57 a binario. Empezamos escribiendo los valores posicionales de las columnas derecha a la izquierda hasta que alcanzamos una columna cuyo valor posicional es mayor que el número decimal. No necesitamos esta columna, por lo que la descartamos. Por lo tanto primero escribimos:

Cómo convertir un número octal a decimal

Valores posicionales:	512	64	8	1
Valores simbólicos:	7	6	1	4
Productos:	7*512=3584	6*64=384	1*8=8	4*1=4
Suma:	= 3584 + 384 + 8 + 4 = 3980			

Fig. E.9 Cómo convertir un número octal a decimal.

Cómo convertir un número hexadecimal a decimal

Valores posicionales:	4096	256	16	1
Valores simbólicos:	A	D	3	B
Productos:	A*4096=40960	D*256=3328	3*16=48	B*1=11
Suma:	= 40960 + 3328 + 48 + 11 = 44347			

Fig. E.10 Cómo convertir un número hexadecimal a decimal.

```
Valores posicionales:   64   32   16   8   4   2   1
```

A continuación descartamos la columna con el valor posicional 64 dejando:

```
Valores posicionales:       32   16   8   4   2   1
```

A continuación trabajamos partiendo de la columna más a la izquierda hacia la derecha. Dividimos 57 entre 32 y observamos que el resultado es 1 con un residuo de 25, por lo que en la columna de los 32 escribimos 1. Ahora dividimos 25 entre 16 y observamos que 25 entre 16 da 1 con un residuo de 9 y escribimos 1 en la columna de los 16. Ahora dividimos 9 entre 8 y observamos que 9 entre 8 da 1 con un residuo de 1. Las siguientes dos columnas cada una de ellas produce resultados de cero cuando dividimos 1 entre sus valores posicionales, por lo que escribimos ceros en las columnas 4 y 2. Finalmente 1 dividido entre 1 es 1, por lo que escribimos 1 en la columna de los unos. Esto da como resultado:

```
Valores posicionales:       32   16   8   4   2   1
Valores simbólicos          1    1    1   0   0   1
```

y, por lo tanto, el equivalente del decimal 57 en binario es 111001

Para convertir el decimal 103 a octal, empezamos escribiendo los valores posicionales de las columnas hasta que alcanzamos una columna cuyo valor posicional es mayor que el número decimal. No necesitamos de esa columna, por lo que la descartamos. Por lo tanto primero escribimos:

```
Valores posicionales:   512   64   8   1
```

A continuación descartamos la columna con el valor posicional 512, dando por resultado:

```
Valores posicionales:       64   8   1
```

A continuación trabajamos a partir de la columna más a la izquierda y hacia la derecha. Dividimos 103 entre 64 y observamos que 103 entre 64 da a uno con un residuo de 39, por lo que escribimos 1 en la columna de los 64. A continuación dividimos 39 entre 8 y observamos que el resultado es 4 con un residuo de 7, y escribimos 4 en la columna de los 8. Por último dividimos 7 entre 1 y observamos que el resultado es 7 sin residuo, por lo que escribimos 7 en la columna de los unos. Esto da como resultado:

```
Valores posicionales:       64   8   1
Valores simbólicos          1    4   7
```

y, por lo tanto, el decimal 103 es equivalente al octal 147.

Para convertir el decimal 375 a hexadecimal, empezamos escribiendo los valores posicionales de las columnas hasta que llegamos a una columna cuyo valor posicional es mayor que el número decimal. No necesitamos dicha columna, por lo que la descartamos. Entonces, primero escribimos

```
Valores posicionales:   4096 256   16   1
```

A continuación descartamos la columna con el valor posicional 4096, dando por resultado:

```
Valores posicionales:       256   16   1
```

A continuación trabajamos a partir de la columna más a la izquierda hacia la derecha. Dividimos 375 entre 256 y observamos que el resultado es uno con un residuo de 119, por lo que escribimos 1 en la columna de los 256. Ahora dividimos 119 entre 16 y observamos que el resultado es 7 con un residuo de 7 y escribimos 7 en la columna de los 16. Por último dividimos 7 entre 1 y observamos que existe un resultado de 7 sin residuo, por lo que escribimos 7 en la columna de los unos. Esto da como resultado:


```
Valores posicionales:       256   16   1
Valores simbólicos          1     7    7
```

por lo tanto, el equivalente del decimal 375 en hexadecimal es 177.

E.6 Números binarios negativos: notación de complemento a dos

El análisis en este apéndice ha sido enfocado a números positivos. En esta sección, explicarémos cómo representan las computadoras números negativos mediante la *notación de complemento a dos*. Primero explicaremos cómo se forma el complemento a dos de un número binario, y a continuación mostraremos porqué representa el valor negativo de un número binario dado.

Considere una máquina con enteros de 32 bits. Suponga

```
int value = 13;
```

La representación en 32 bits de `value` es

```
00000000 00000000 00000000 00001101
```

Para formar el negativo de `value`, primero formamos su complemento a uno aplicando el operador de complemento a nivel de bits (~)de C.

```
ones_complement_of_value = ~value;
```

Internamente, `~value` es ahora `value` con cada uno de sus bits invertidos —los unos se han convertido en ceros y los ceros se han convertido en unos, como sigue:

```
value:
00000000 00000000 00000000 00001101

~value (es decir, el complemento a uno de value):
11111111 11111111 11111111 11110010
```

Para formar el complemento a dos de `value`, simplemente añadimos uno al complemento a uno de `value`. Por lo tanto

```
El complemento a dos de value:
11111111 11111111 11111111 11110011
```

Ahora si esto de hecho es igual a -13, deberíamos estar en condiciones de añadirlo al número binario 13 y obtener como resultado 0. Probémoslo:

```
 00000000 00000000 00000000 00001101
+11111111 11111111 11111111 11110011
-----------------------------------
 00000000 00000000 00000000 00000000
```

Se descarta el bit de acarreo que resulta en la columna más a la izquierda y como resultado obtenemos cero. Si a un número le añadimos el complemento a uno del número, el resultado serían todos unos. La clave para obtener un resultado de todos ceros es que el complemento a dos es uno más que el complemento a uno. La adición de uno hace que cada columna añada a cero, con un uno que se acarrea a la izquierda. El acarreo se va trasladando hacia la izquierda hasta que se descarta del bit más a la izquierda, y de ahí el número resultante de todos ceros.

En realidad las computadoras llevan a cabo una substracción como

```
x = a - value;
```

sumando el complemento a dos de **value** a **a** como sigue:

```
x = a + (~value + 1);
```

Suponga que **a** es 27 y **value** es 13, como antes. Si el complemento a dos de **value** es de hecho el valor negativo de **value**, entonces la suma del complemento a dos de **value** a **a** debería producir el resultado 14. Probémoslo:

```
a (es decir, 27)     00000000 00000000 00000000 00011011
+(~value + 1)       +11111111 11111111 11111111 11110011
                    -----------------------------------
                     00000000 00000000 00000000 00001110
```

lo que realmente da igual a 14.

Resumen

- Cuando escribimos un entero como 19 y 227 ó -63 en un programa C, automáticamente se supone que el número está en sistema numérico decimal (en base 10). Los dígitos en sistema numérico decimal son 0, 1, 2, 3, 4, 5, 6, 7, 8, y 9. El dígito menor es cero y el dígito mayor es 9 —uno menos que la base de 10.
- En forma interna, las computadoras utilizan el sistema numérico binario (de base 2). El sistema numérico binario tiene sólo dos dígitos, es decir 0 y 1. Su número menor es 0 y su número mayor es 1 —uno menos que el de la base de 2.
- El sistema numérico octal (de base 8) y el sistema numérico hexadecimal (de base 16) se han hecho populares principalmente debido a que resultan convenientes para abreviar números binarios.
- Los dígitos del sistema numérico octal van desde el 0 al 7.
- El sistema numérico hexadecimal presenta un problema, porque requiere de 16 dígitos —un dígito menor 0 y un dígito mayor con un valor equivalente al 15 decimal (uno menos que la base de 16). Por regla convencional, utilizamos las letras A hasta F para representar los dígitos hexadecimales que corresponden a los valores decimales 10 hasta el 15.
- Cada sistema numérico utiliza notación posicional —cada posición en la cual se escribe un dígito, tiene un valor posicional diferente.
- Una relación particularmente importante que tienen tanto el sistema numérico octal como el hexadecimal con el sistema binario, es que las bases de los sistemas octal y hexadecimal (8 y 16 respectivamente) son potencias de la base del sistema numérico binario (base 2).
- Para convertir un número octal a un número binario, simplemente remplace cada dígito octal por su equivalente binario de tres dígitos.
- Para convertir un número hexadecimal a un número binario, simplemente remplace cada dígito hexadecimal por su equivalente binario de cuatro dígitos.
- Dado que estamos acostumbrados a trabajar en decimal, a menudo resulta conveniente convertir un número binario, octal o hexadecimal a decimal para tener una mejor idea de lo que en verdad vale un número.
- Para convertir un número de decimal desde otra base, multiplique el equivalente decimal de cada dígito por su valor posicional y sume dichos productos.
- Las computadoras representan los números negativos utilizando notación de complemento a dos.

- Para formar el negativo de un valor en binario, primero forme el complemento a uno aplicando el operador de complemento a nivel de bits (~) de C. Esto invierte los bits del valor. Para formar el complemento a dos de dicho valor, sólo añada uno al valor del complemento a uno.

Terminología

base
sistema numérico de base 2
sistema numérico de base 8
sistema numérico de base 10
sistema numérico de base 16
sistema numérico binario
operador de complemento a nivel de bits (~)
conversiones
sistema numérico decimal
dígito
sistema numérico hexadecimal
valor negativo
sistema numérico octal
notación de complemento a uno
notación posicional
valor posicional
valor simbólico
notación de complemento a dos

Ejercicios de autoevaluación

E.1 Las bases de los sistemas numéricos decimal, binario, oct y hexadecimal son _____, _____, _____, y _____ respectivamente.

E.2 En general, las representaciones decimal, octal y hexadecimal de un número binario dado contienen (más/menos) dígitos que los que contiene el número binario.

E.3 (Verdadero/falso). Una razón popular para utilizar el sistema numérico decimal es que forma una notación conveniente para abreviar números binarios simplemente sustituyendo un dígito decimal por cada grupo de cuatro dígitos binarios.

E.4 La representación (octal/hexadecimal/decimal) de un valor binario muy grande es la más concisa (de las alternativas dadas).

E.5 (Verdadero/falso). El dígito mayor en cualquier base es uno más que la base.

E.6 (Verdadero/falso). El dígito menor de cualquier base es uno menos que la base.

E.7 El valor posicional del dígito más a la derecha de cualquier número en binario, octal, decimal o hexadecimal es siempre _____.

E.8 El valor posicional del dígito a la izquierda del dígito más a la derecha de cualquier número en binario, octal, decimal o hexadecimal es siempre igual a _____.

E.9 Complete los valores faltantes en esta tabla de valores posicionales para las cuatro posiciones más a la derecha en cada uno de los sistema numéricos indicados.

decimal	1000	100	10	1
hexadecimal	...	256	...	...
binario	...	...	...	...
octal	512	...	8	...

E.10 Convertir el número binario `110101011000` a octal y a hexadecimal.

E.11 Convertir el número hexadecimal **FACE** a binario.

E.12 Convertir el número octal `7316` a binario.

E.13 Convertir el número hexadecimal `4FEC` a octal. (*Sugerencia:* primero convierta `4FEC` a binario y a continuación convierta el número binario a octal).)

E.14 Convierta el número binario `1101110` a decimal.

E.15 Convierta el número octal `317` a decimal.

E.16 Convierta el número hexadecimal `EFD4` a decimal.

E.17 Convierta el número decimal `177` a binario, a octal, y a hexadecimal.

E.18 Muestre la representación binaria del decimal `417`. Después muestre el complemento a uno de `417` y el complemento a dos.

E.19 ¿Cuál es el resultado cuando el complemento a uno de un número se añade a sí mismo?

Respuestas a los ejercicios de autoevaluación

E.1 `10, 2, 8, 16.`

E.2 Menos.

E.3 Falso.

E.4 Hexadecimal.

E.5 Falso —el dígito mayor en cualquier base es uno menos que la base.

E.6 Falso —el dígito menor de cualquier base es cero.

E.7 1 (la base elevada a la potencia cero).

E.8 La base del sistema numérico.

E.9 Complete los valores faltantes en esta tabla de valores posicionales para las cuatro posiciones más a la derecha en cada uno de los sistemas numéricos indicados.

decimal	1000	100	10	1
hexadecimal	4096	256	16	1
binario	8	4	2	1
octal	512	64	8	1

E.10 Octal `6530`; hexadecimal `D58`.

E.11 Binario `1111 1010 1100 1110`.

E.12 Binario `111 011 001 110`

E.13 Binario `0 100 111 111 101 100`; Octal `47754`

E.14 Decimal `2+4+8+32+64=110`.

E.15 Decimal `7+1*8+3*64=7+8+192=207`

E.16 Decimal `4+13*16+15*256+14*4096=61396`.

E.17 Decimal `177`

a binario:

```
256 128 64 32 16 8 4 2 1
128 64 32 16 8 4 2 1
(1*128)+(0*64)+(1*32)+(1*16)+(0*8)+(0*4)+(0*2)+(1*1)
10110001
```


a octal:

```
512 64 8 1
64 8 1
(2*64)+(6*8)+(1*1)
261
```

a hexadecimal:

```
256 16 1
16 1
(11*16)+(1*1)
(B*16)+(1*1)
B1
```

E.18 Binario:

```
512 256 128 64 32 16 8 4 2 1
256 128 64 32 16 8 4 2 1
(1*256)+(1*128)+(0*64)+(1*32)+(0*16)+(0*8)+(0*4)+(0*2)+
(1*1)
110100001
```

Complemento a uno:`001011110`
Complemento a dos:`001011111`
Verificación: número binario original + su complemento a dos

```
110100001
001011111
---------
000000000
```

E.19 Cero

Ejercicios

E.20 Algunas personas alegan que muchos de nuestros cálculos serían más fáciles en un sistema numérico de base `12` porque `12` es divisible por muchísimos más números que `10` (debido a la base `10`). ¿Cuál es el dígito menor en la base `12`? ¿Cuál debería ser el símbolo más alto para el dígito en la base `12`? ¿Cuáles son los valores posicionales para las cuatro posiciones más a la derecha de cualquier número en un sistema numérico de base `12`?

E.21 ¿Cómo se relaciona el valor simbólico más alto en los sistemas numéricos anteriormente analizados con el valor posicional del primer dígito a la izquierda del dígito más a la derecha de cualquier número en estos sistemas numéricos?

E.22 Complete la tabla siguiente de valores posicionales para las cuatro posiciones más a la derecha en cada uno de los sistemas numéricos indicados:

decimal	1000	100	10	1
base 6	...	...	6	...
base 13	...	169	...	...
base 3	27	...	...	...

E.23 Convierta el binario `100101111010` a octal y a hexadecimal.

E.24 Convierta el hexadecimal a `3A7D` a binario.

E.25 Convierta el hexadecimal a `765F` a octal. (*Sugerencia*: primero convierta `765F` a binario y a continuación convierta dicho número binario a octal).

E.26 Convierta el número binario `1011110` a decimal.

E.27 Convierta el número octal `426` a decimal.

E.28 Convierta el número hexadecimal `FFFF` a decimal.

E.29 Convierta el número decimal `299` a binario, a octal y a hexadecimal.

E.30 Muestre la representación binario del decimal `779`. A continuación muestre el complemento a uno de `779`, así como el complemento a dos.

E.31 ¿Cuál es el resultado cuando el complemento a dos de un número se añade a sí mismo?.

E.32 Muestre el complemento a dos del valor entero `-1` en una máquina con enteros de 32 bits.

INDICE

Indice

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W