7

Apuntadores

Objetivos

- Ser capaz de utilizar apuntadores.
- Ser capaz de utilizar apuntadores para pasar argumentos a las funciones en llamada por referencia.
- Comprender las relaciones íntimas entre apuntadores, arreglos y cadenas.
- Comprender la utilización de apuntadores a funciones.
- Ser capaz de declarar y utilizar arreglos de cadenas.

Se nos asignan direcciones para ocultar nuestra ubicación.
Saki (H.H. Munro)

Averigüe las instrucciones por medio de la falta de indicaciones.
William Shakespeare
Hamlet

Muchas cosas, con plena referencia
A nuestra complacencia, podrían funcionar en forma inversa.
William Shakespeare
King Henry V

¡Encontrará que resulta excelente el verificar siempre sus referencias, señor!
Dr. Routh

No es posible confiar en un código que no haya usted creado por sí mismo.
(En especial códigos que provengan de empresas que emplean a personas como yo).
Ken Thompson
1983 Turing Award Lecture Association for Computing Machinery, Inc.

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 • Sección especial: cómo construir su propia computadora.

7.1 Introducción

En este capítulo, analizamos una de las características más poderosas del lenguaje de programación en C, el *apuntador*. Los apuntadores son las capacidades más difíciles de dominar en C. Los apuntadores le permiten a los programas simular llamadas por referencia, crear y manipular estructuras de datos, es decir, estructuras de datos que pueden crecer o encogerse, como son listas enlazadas, colas de espera, pilas y árboles. Este capítulo explica conceptos básicos de los apuntadores. En el capítulo 10 se examina el uso de apuntadores junto con las estructuras. En el capítulo 12 se presentan las técnicas de administración dinámica de la memoria y se presentan ejemplos de creación y uso de estructuras dinámicas de datos.

7.2 Declaraciones e inicialización de variables de apuntadores

Los apuntadores son variables que contienen direcciones de memoria como sus valores. Por lo regular una variable contiene directamente un valor específico. Un apuntador, por otra parte, contiene la dirección de una variable que contiene un valor específico. En este sentido, un nombre de variable se refiere *directamente* a un valor y un apuntador se refiere *indirectamente* a un valor (figura 7.1). El referirse a un valor a través de un apuntador se conoce como *indirección.*

Los apuntadores, como cualquier otra variable, deben ser declarados antes de que puedan ser utilizados. La declaración

```
int *countPtr, count;
```

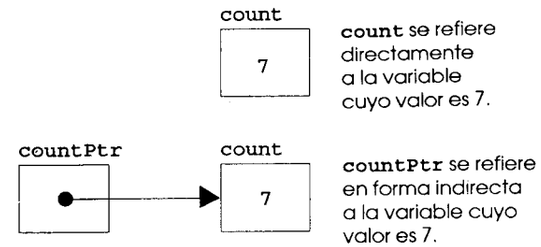



Fig. 7.1 Referenciación directa e indirecta de una variable.

declara la variable `countPtr` siendo del tipo `int*` (es decir, un apuntador a un valor entero) y se lee, "`countPtr` es un apuntador a `int`", o bien "`countPtr` apunta a un objeto del tipo entero". También, la variable `count` se declara como un entero, no un apuntador a un entero. El `*` sólo se aplica a `countPtr` en la declaración. Cuando el `*` se utiliza de esta forma en una declaración, indica que la variable que se está declarando es un apuntador. Los apuntadores pueden ser declarados para apuntar a objetos de cualquier tipo de datos.

Error común de programación 7.1

El operador de indirección `*` no se distribuye a todos los nombres de variables de una declaración. Cada apuntador debe de ser declarado con el `*` prefijo al nombre.

Práctica sana de programación 7.1

En un nombre de variable de apuntador incluya las letras `ptr` para que quede claro que estas variables son apuntadores y deben ser manejadas de forma apropiada.

Los apuntadores deben ser inicializados cuando son declarados o en un enunciado de asignación. Un apuntador puede ser inicializado a `0`, `NULL`, o a una dirección. Un apuntador con el valor `NULL` apunta a nada. `NULL` es una constante simbólica, definida en el archivo de cabecera `<stdio.h>` (y en varios otros archivos de cabecera). Inicializar un apuntador a `0` es equivalente a inicializar un apuntador a `NULL`, pero es preferible `NULL`. Cuando se asigna `0`, primero se convierte a un apuntador del tipo apropiado. El valor `0` es el único valor entero que puede ser directamente asignado a una variable de apuntador. En la sección 7.3 se analiza cómo asignar la dirección de una variable a un apuntador.

Práctica sana de programación 7.2

Inicialice los apuntadores para evitar resultados inesperados.

7.3 Operadores de apuntador

El `&`, u *operador de dirección,* es un operador unario que regresa la dirección de su operando. Por ejemplo, suponiendo las declaraciones

```
int y = 5;
int *yPtr;
```

el enunciado

```
yPtr = &y;
```

asigna la dirección de la variable **y** a la variable de apuntador **yPtr**. La variable **yPtr** se dice entonces que "apunta a" **y**. La figura 7.2 muestra una representación esquemática de la memoria, después de que se ejecuta la asignación anterior.

La figura 7.3 muestra la representación del apuntador en memoria, suponiendo que la variable entera **y** se almacena en la posición **600000**, y la variable de apuntador **yPtr** se almacena en la posición **500000**. El operando del operador de dirección debe ser una variable; el operador de dirección no puede ser aplicado a constantes, a expresiones, o a variables declaradas con la clase de almacenamiento **register**.

El operador *****, conocido comúnmente como el *operador de indirección* o de *desreferencia*, regresa el valor del objeto hacia el cual su operando apunta (es decir, un apuntador). Por ejemplo, el enunciado

```
printf("%d", *yPtr);
```

imprime el valor de la variable **y**, es decir 5. Utilizar a ***** de esta forma se conoce como *desreferenciar a un apuntador*.

Error común de programación 7.2

Desreferenciar un apuntador que no haya sido correctamente inicializado, o que no haya sido asignado para apuntar a una posición específica en memoria. Esto podría causar un error fatal en tiempo de ejecución, o podría modificar de forma accidental datos importantes y permitir que el programa se ejecute hasta su terminación proporcionando resultados incorrectos.

El programa en la figura 7.4 demuestra los operadores de apuntador. La especificación de conversión de **printf** **%p** extrae la localización de memoria en forma de entero hexadecimal (vea el Apéndice E, Sistemas numéricos, para mayor información sobre enteros hexadecimales).

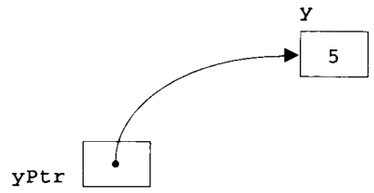


Fig. 7.2 Representación gráfica de un apuntador apuntando a una variable entera en memoria.

Fig. 7.3 Representación en memoria de **y** y **yPtr**.

Note que la dirección de **a** y el valor de **aPtr** son idénticos en la salida, confirmando así que de hecho la dirección de **a** ha sido asignada a la variable de apuntador **aPtr**. Los operadores **&** y ***** son complementos el uno del otro —cuando se aplican ambos de manera consecutiva a **aPtr**, en cualquier orden, el mismo resultado será impreso. La gráfica en la figura 7.5, muestra la precedencia y asociatividad de los operadores presentados hasta este momento.

7.4 Cómo llamar funciones por referencia

Existen dos formas de pasar argumentos a un función llamada por valor y llamada por referencia. En C todas las llamadas de función son llamadas por valor. Como vimos en el capítulo 5, se puede utilizar **return** para regresar un valor de una función llamada hacia el llamador (o para regresar el control de una función llamada sin regresar un valor). Muchas funciones requieren la capacidad de modificar una o más variables del llamador, o de pasar un apuntador a un objeto de datos grande, para evitar la sobrecarga de pasar el objeto en llamada por valor (lo que, naturalmente,

```
/* Using the & and * operators */
#include <stdio.h>

main()
{
   int a;        /* a is an integer */
   int *aPtr;    /* aPtr is a pointer to an integer */

   a = 7;
   aPtr = &a;    /* aPtr set to address of a */

   printf("The address of a is %p\n"
          "The value of aPtr is %p\n\n", &a, aPtr);

   printf("The value of a is %d\n"
          "The value of *aPtr is %d\n\n", a, *aPtr);

   printf("Proving that * and & are complements of "
          "each other.\n&*aPtr = %p\n*&aPtr = %p\n",
          &*aPtr, *&aPtr);
   return 0;
}
```

```
The address of a is FFF4
The value of aPtr is FFF4

The value of a is 7
The value of *aPtr is 7

Proving that * and & are complements of each other.
&*aPtr = FFF4
*&aPtr = FFF4
```

Fig. 7.4 Los operadores de apuntador **&** y *****.

Operadores								Asociatividad	Tipo
()	[]							de izquierda a derecha	oculto
+	-	++	--	!	*	&	(type)	de izquierda a derecha	unario
*	/	%						de izquierda a derecha	multiplicativo
+	-							de izquierda a derecha	aditivo
<	<=	>	>=					de izquierda a derecha	relación
==	!=							de izquierda a derecha	igualdad
&&								de izquierda a derecha	y lógico
||								de izquierda a derecha	o lógico
?:								de izquierda a derecha	condicional
=	+=	-=	*=	/=	%=			de izquierda a derecha	asignación
,								de izquierda a derecha	coma

Fig. 7.5 Precedencia de operadores.

requiere el hacer una copia del objeto). Para estos fines, C proporciona las capacidades de simulación de llamadas por referencia.

En C, los programadores utilizan apuntadores y el operador de indirección para simular llamadas por referencia. Cuando se llama a una función con argumentos que deban ser modificados, se pasan las direcciones de los argumentos. Esto se lleva a cabo normalmente aplicando el operador de dirección (**&**), a la variable cuyo valor deberá ser modificado. Como se vió en el capítulo 6, los arreglos no son pasados mediante el operador **&** porque C pasa de forma automática la posición inicial en memoria del arreglo (el nombre de un arreglo es equivalente a **&arrayName[0]**). Cuando se pasa a una función la dirección de una variable, el operador de indirección (*****), puede ser utilizado en la función para modificar el valor de esa posición en la memoria del llamador.

Los programas de las figuras 7.6 y 7.7 presentan dos versiones de una función que eleva un entero al cubo —**cubeByValue** y **cubeByReference**. El programa de la figura 7.6 pasa la variable **number** a la función **cubeByValue**, utilizando llamada por valor. La función **cubeByValue** eleva al cubo su argumento y pasa su nuevo valor de regreso a **main**, utilizando el enunciado **return**. El nuevo valor se asigna al **number** en **main**.

El programa de la figura 7.7 pasa la variable **number** utilizando llamada por referencia —se pasa la dirección de **number**— a la función **cubeByReference**. La función **cubeByReference** toma un apuntador a **int** conocido como **nPtr** como argumento. La función desreferencia el apuntador y eleva al cubo el valor hacia el cual apunta **nPtr**. Esto cambia el valor de **number** dentro de **main**. Las figuras 7.8 y 7.9 analizan de forma gráfica los programas de las figuras 7.6 y 7.7, respectivamente.

Error común de programación 7.3

No desreferenciar un apuntador, cuando es necesario hacerlo para obtener el valor hacia el cual el apuntador señala.


```
/* Cube a variable using call by value */
#include <stdio.h>

int cubeByValue(int);

main()
{
   int number = 5;

   printf("The original value of number is %d\n", number);
   number = cubeByValue(number);
   printf("The new value of number is %d\n", number);
   return 0;
}

int cubeByValue(int n)
{
   return n * n * n;   /* cube local variable n */
}
```

```
The original value of number is 5
The new value of number is 125
```

Fig. 7.6 Elevación al cubo de una variable, utilizando llamada por valor.

```
/* Cube a variable using call by reference */
#include <stdio.h>

void cubeByReference(int *);

main()
{
   int number = 5;

   printf("The original value of number is %d\n", number);
   cubeByReference(&number);
   printf("The new value of number is %d\n", number);
   return 0;
}

void cubeByReference(int *nPtr)
{
   *nPtr = *nPtr * *nPtr * *nPtr;  /* cube number in main */
}
```

```
The original value of number is 5
The new value of number is 125
```

Fig. 7.7 Elevación al cubo de una variable, utilizando llamada por referencia.

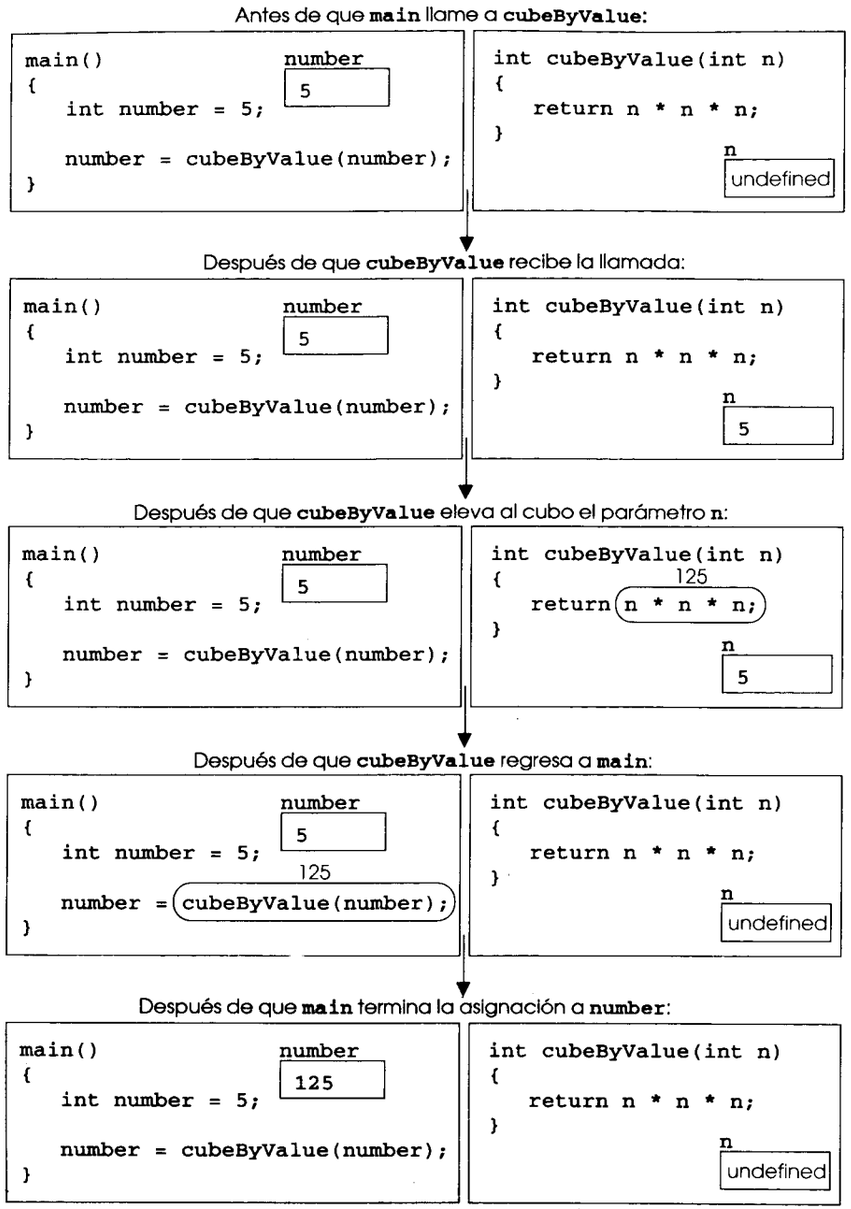


Fig. 7.8 Análisis de una llamada por valor típica.

Una función que recibe una dirección como argumento debe definir un parámetro de un apuntador para recibir la dirección. Por ejemplo, el encabezado para la función **cubeByReference** es

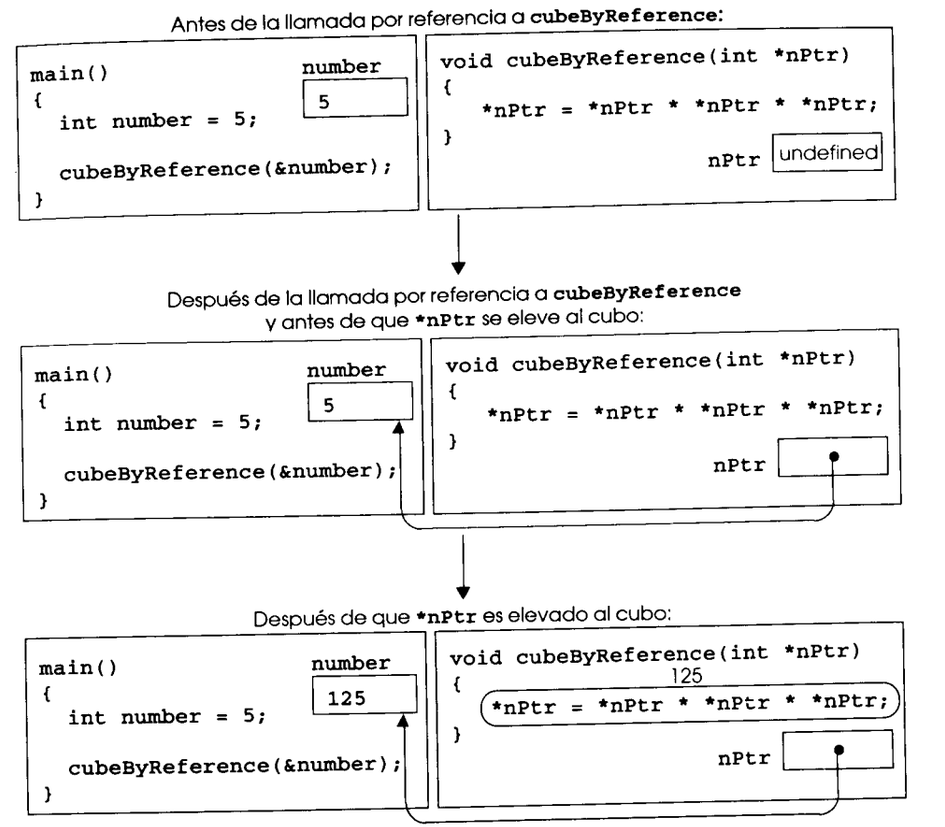


Fig. 7.9 Análisis de una llamada por referencia típica.

```
void cubeByReference(int *nPtr)
```

El encabezado especifica que **cubeByReference** recibe la dirección de una variable entera como argumento, almacena la dirección local en **nPtr**, y no regresa un valor.

El prototipo de función para **cubeByReference** contiene **int *** entre paréntesis. Como en el caso de otros tipos de variables, en los prototipos de función no es necesario incluir los nombres de los apuntadores. Los nombres, que se incluyan para fines de documentación, serán ignorados por el compilador de C.

En el encabezado de función y en el prototipo para una función que espera un arreglo de un subíndice como argumento, puede ser utilizada la notación de apuntador en la lista de parámetros de **cubeByReference**. El compilador no diferencia entre una función que recibe un apuntador y una función que recibe un arreglo de un solo subíndice. Esto, naturalmente, significa que la función debe "saber" cuando está recibiendo un arreglo o una variable para la cual deberá llevar a cabo una llamada por referencia. Cuando el compilador encuentre un parámetro de función correspondiente a un arreglo de un subíndice de la forma **int b []**, el compilador convertirá el parámetro a la notación de apuntador **int *b**. Ambas formas son intercambiables.

Práctica sana de programación 7.3

Utilice llamadas por valor para pasar argumentos a una función, a menos de que en forma explícita el llamador requiera que la función llamada modifique el valor de la variable del argumento en el entorno de llamador. Esto es otro ejemplo del principio del mínimo privilegio.

7.5 Cómo utilizar el calificador Const con apuntadores

El *calificador* **const** permite al programador informarle al compilador que el valor de una variable particular no deberá ser modificado. En las primeras versiones de C no existía el calificador **const**; fue añadido al lenguaje por el comité de ANSI C.

Observación de ingeniería de software 7.1

El calificador **const** *puede ser utilizado para forzar el principio del mínimo privilegio. La utilización del principio de mínimo privilegio para diseñar con propiedad el software reduce en forma importante el tiempo de depuración y efectos inadecuados colaterales, y hace un programa más fácil de modificar y mantener.*

Tip de portabilidad 7.1

A pesar de que en ANSI C **const** *está bien definido, algunos sistemas no lo tienen incorporado.*

A través de los años, una gran base de código heredado quedó escrita en las primeras versiones de C, que no utilizan a **const** porque no estaba disponible. Por esta razón, existen grandes oportunidades de mejoría en la ingeniería de software del código existente de C. También, muchos programadores, que en la actualidad utilizan ANSI C, en sus programas no utilizan **const**, porque empezaron a programar usando las primeras versiones de C. Estos programadores están perdiendo muchas oportunidades de buena ingeniería de software.

Existen seis posibilidades para el uso (o el no uso) de **const** con parámetros de función, dos —al pasar parámetros en llamadas por valor y cuatro al pasar parámetros con llamadas por referencia. ¿Cómo escogerá usted una de las seis posibilidades? Deje que el principio del mínimo privilegio sea su guía. Siempre dele a una función suficiente acceso a los datos en sus parámetros para llevar a cabo su tarea especificada, pero no más.

En el capítulo 5, explicamos que todas las llamadas en C son llamadas por valor, en la llamada de función se efectúa una copia del argumento y se pasa a la función. Si en la función la copia se modifica, el valor original en el llamador se mantiene sin cambio. En muchos casos, el valor pasado a la función es modificado para que ésta pueda llevar a cabo su tarea. Sin embargo, en algunas ocasiones, el valor no deberá ser alterado en la función llamada, aun cuando la función llamada manipule una copia del valor original.

Considere una función que toma como argumentos un arreglo de un subíndice y su tamaño e imprime el arreglo. Una función como ésta deberá ciclar a través del arreglo y extraer en forma individual cada elemento del arreglo. El tamaño del arreglo se utiliza en el cuerpo de la función para determinar el subíndice alto del mismo, de tal forma que el ciclo pueda terminar cuándo se termine la impresión. El tamaño del arreglo no se modifica en el cuerpo de la función.

Observación de ingeniería de software 7.2

Si un valor no se modifica (o no debería modificarse) en el cuerpo de una función al cual es pasado, el valor deberá declararse **const**, *para asegurarse que no se modifica de forma accidental.*

Si se hace un intento de modificar un valor declarado **const**, el compilador lo detectará y emitirá ya sea una advertencia, o un error, dependiendo del compilador particular.

Observación de ingeniería de software 7.3

En una función llamadora sólo un valor puede ser alterado cuando se utiliza llamada por valor. Este valor debe ser asignado a partir del valor de regreso de la función. Para modificar varios valores en una función llamadora, debe utilizarse llamada por referencia.

Práctica sana de programación 7.4

Antes de utilizar una función, verifique el prototipo de función correspondiente a esta función, a fin de determinar si la función es capaz de modificar los valores que se le pasan.

Error común de programación 7.4

No estar consciente que una función está esperando apuntadores como argumentos por llamadas por referencia, y está pasando argumentos en llamadas por valor. Algunos compiladores toman los valores, suponiendo que son apuntadores y desreferencian los valores como apuntadores. En tiempo de ejecución, a menudo se generan violaciones de acceso a la memoria o fallas de segmentación. Otros compiladores detectan falta de coincidencia en tipo entre argumentos y parámetros, y generan mensajes de error.

Existen cuatro formas para pasar un apuntador a una función: un apuntador no constante a datos no constantes, un apuntador constante a datos no constantes, un apuntador no constante a datos constantes, y un apuntador constante a datos constantes. Cada una de las cuatro combinaciones proporciona un nivel distinto de privilegios de acceso.

El nivel más alto de acceso de datos se consigue mediante un apuntador no constante a datos no constantes. En este caso, los datos pueden ser modificados a través de un apuntador desreferenciado, y el apuntador puede ser modificado para señalar a otros elementos de datos. Una declaración para un apuntador no constante a datos no constantes no incluye **const**. Tal apuntador pudiera ser utilizado para recibir una cadena como argumento a una función que utiliza aritmética de apuntador para procesar (y posiblemente para modificar) cada carácter dentro de la cadena. La función **convertToUppercase** de la figura 7.10 declara como su argumento un apuntador no constante a datos no constantes, llamado **s (char*s)**. La función procesa la cadena **s**, un carácter a la vez, utilizando aritmética de apuntador. Si un carácter está en el rango **a** a **z**, se convierte a su letra en mayúsculas correspondiente, **A** a **Z** utilizando un cálculo basado en su código ASCII; de lo contrario es pasado por alto, y es procesado el siguiente carácter en la cadena. Note que todas las letras mayúsculas en el conjunto de caracteres ASCII tienen valores enteros que son equivalentes en valores ASCII a sus letras correspondientes en minúsculas menos 32 (vea la tabla de valores de caracteres ASCII en el Apéndice D). En el capítulo 8, presentaremos la función **toupper** de la biblioteca estándar de C para la conversión de letras a mayúsculas.

Un apuntador no constante a datos constantes es un apuntador que puede ser modificado para apuntar a cualquier elemento de datos del tipo apropiado, pero no pueden ser modificados los datos hacia los cuales apunta. Tal apuntador pudiera ser utilizado para recibir un argumento de arreglo a una función, que procesaría cada elemento del arreglo, sin modificar los datos. Por ejemplo, la función **printCharacters** de la figura 7.11 declara los parámetros s del tipo **const char***. La declaración se lee de derecha a izquierda de la forma "**s** es un apuntador a una constante de carácter". El cuerpo de la función utiliza una estructura **for**, para extraer cada carácter de la cadena, hasta que encuentre el carácter NULL. Después de haber impreso cada carácter, el apuntador s es incrementado, para que apunte al siguiente carácter dentro de la cadena.

```
/* Converting lowercase letters to uppercase letters */
/* using a non-constant pointer to non-constant data */
#include <stdio.h>

void convertToUppercase(char *);

main()
{
   char string[] = "characters";

   printf("The string before conversion is: %s\n", string);
   convertToUppercase(string);
   printf("The string after conversion is:  %s\n", string);
   return 0;
}

void convertToUppercase(char *s)
{
   while (*s != '\0') {

      if (*s >= 'a' && *s <= 'z')
         *s -= 32;  /* convert to ASCII uppercase letter */

      ++s;  /* increment s to point to the next character */
   }
}
```

```
The string before conversion is: characters
The string after conversion is: CHARACTERS
```

Fig. 7.10 Cómo convertir una cadena a mayúsculas, utilizando un apuntador no constante a datos no constantes.

En la figura 7.12 se demuestran los mensajes de error producidos por el compilador Borland C++ al intentar compilar una función que recibe un apuntador no constante a datos constantes, y la función utiliza el apuntador a fin de modificar datos.

Como sabemos, los arreglos son conjuntos de tipos de datos, que almacenan bajo un nombre muchos elementos de datos relacionados del mismo tipo. En el capítulo 10, analizaremos otra forma de tipo de conjunto de datos llamada una *estructura* (llamado a veces en otros lenguajes un *registro*). Una estructura es capaz de almacenar muchos elementos de datos relacionados, de distintos tipos de datos, bajo un nombre (por ejemplo, almacenar información sobre cada uno de los empleados de una empresa). Cuando se llama una función con un arreglo como argumento, el arreglo se pasa de forma automática a la función en llamada por referencia. Sin embargo, las estructuras son siempre pasadas en llamada por valor —se pasa una copia de toda la estructura. Esto requiere de sobrecarga en tiempo de ejecución, para efectuar una copia de cada elemento de dato en la estructura y almacenarla en la pila de llamada de la función, de la computadora. Cuando los datos de la estructura deban ser pasados a una función, podemos utilizar apuntadores hacia datos constantes, para conseguir el rendimiento de una llamada por referencia y la protección de una llamada por valor. Cuando se pasa un apuntador a una estructura, sólo debe de ser efectuada una copia de la dirección en donde está almacenada la estructura. En una máquina con direcciones de 4 bytes, se efectúa una copia de 4 bytes de memoria, en vez de una copia de la estructura, de posiblemente cientos o miles de bytes.


```
/* Printing a string one character at a time using */
/* a non-constant pointer to constant data         */
#include <stdio.h>

void printCharacters(const char *);

main()
{
   char string[] = "print characters of a string";

   printf("The string is:\n");
   printCharacters(string);
   putchar('\n');
   return 0;
}

void printCharacters(const char *s)
{
   for ( ; *s != '\0'; s++)   /* no initialization */
      putchar(*s);
}
```

```
The string is:
print characters of a string
```

Fig. 7.11 Cómo imprimir una cadena, un carácter a la vez, utilizando un apuntador no constante a datos constantes.

Sugrencia de rendimiento 7.1

Pase grandes objetos como son estructuras utilizando apuntadores a datos constantes para obtener los beneficios de rendimiento de llamadas por referencia y la seguridad de llamadas por valor.

Utilizar de esta forma apuntadores a datos constantes es un ejemplo de *cambiar tiempo por espacio*. Si la memoria es reducida y la eficiencia de ejecución es una preocupación importante, deberán utilizarse apuntadores. Si la memoria es abundante y la eficiencia no es una preocupación importante, los datos deberán ser pasados en llamada por valor, para forzar el principio del mínimo privilegio. Recuerde que algunos sistemas no manejan bien **const**, por lo que la llamada por valor sigue siendo la mejor forma de evitar que los datos sean modificados.

Un apuntador constante a datos no constantes es un apuntador que siempre apunta a la misma posición de memoria, y los datos en esa posición pueden ser modificados a través del apuntador. Esta es la forma por omisión de un nombre de arreglo. Un nombre de arreglo es un apuntador constante al inicio de dicho arreglo. Todos los datos en el arreglo son accesibles y modificados, utilizando el nombre del arreglo y los subíndices del mismo. Un apuntador constante a datos no constantes puede ser utilizado para recibir un arreglo como argumento de una función, que tiene acceso a elementos de arreglo utilizando sólo notaciones de subíndices del arreglo. Los apuntadores declarados **const** deben ser inicializados al ser declarados (si el apuntador es un parámetro

```
/* Attempting to modify data through a    */
/* non-constant pointer to constant data */
#include <stdio.h>

void f(const int *);

main()
{
   int y;

   f(&y);      /* f attempts illegal modification */
   return 0;
}

void f(const int *x)
{
   *x = 100;  /* cannot modify a const object */
}
```

```
Compiling FIG7_12.C:
Error FIG7_12.C 17: Cannot modify a const object
Warning FIG7_12.C 18: Parameter 'x' is never used
```

Fig. 7.12 Intento de modificación de datos a través de un apuntador no constante a datos constantes.

de función, será inicializado con un apuntador que se pasa a la función). El programa de la figura 7.13 intenta modificar un apuntador constante. El apuntador **ptr** se declara ser del tipo **int * const**. La declaración se lee de derecha a izquierda como "**ptr** es un apuntador constante a un entero". El apuntador se inicializa con la dirección de la variable entera **x**. El programa intenta asignar la dirección de **y** a **ptr**, pero se genera un mensaje de error.

El privilegio de mínimo acceso se concede mediante un apuntador constante a datos constantes. Un apuntador de este tipo siempre apunta a la misma posición de memoria, y los datos en esa posición de memoria no pueden ser modificados. Esta es la forma en que debería pasarse un arreglo a una función que sólo ve al arreglo utilizando notación de subíndices del mismo y no modifica dicho arreglo. El programa de la figura 7.14 declara a la variable de apuntador **ptr** ser del tipo **const int * const**. Esta declaración se lee de derecha a izquierda como "**ptr** es un apuntador constante a un entero constante". La figura muestra los mensajes de error generados cuando se intenta modificar los datos a los cuales apunta **ptr**, y cuando se intenta modificar la dirección almacenada en la variable del apuntador.

7.6 Ordenación de tipo burbuja utilizando llamadas por referencia

Modifiquemos el programa de ordenación de tipo burbuja de la figura 6.15 para utilizar dos funciones **bubbleSort** y **swap**. La función **bubbleSort** ejecuta la ordenación del arreglo. Llama a la función **swap** para intercambiar los elementos del arreglo **array[j]** y **array[j +1]** (vea la figura 7.15). Recuerde que C obliga al ocultamiento de la información entre funciones,


```
/* Attempting to modify a constant pointer to */
/* non-constant data                          */
#include <stdio.h>

main()
{
   int x, y;
   int * const ptr = &x;

   ptr = &y;
   return 0;
}
```

```
Compiling FIG7_13.C:
Error FIG7_13.C 10 Cannot modify a const object
Warning FIG7_13.C 12: 'ptr' is assigned a value that is
   never used
Warning FIG7_13.C 12: 'y' is declared but never used
```

Fig. 7.13 Intento de modificación de un apuntador constante a datos no constantes.

```
/* Attempting to modify a constant pointer to */
/* constant data                              */
#include <stdio.h>

main()
{
   int x = 5, y;
   const int *const ptr = &x;

   *ptr = 7;
   ptr = &y;
   return 0;
}
```

```
Compiling FIG7_14.C:
Error FIG7_14.C 10: Cannot modify a const object
Error FIG7_14.C 11 Cannot modify a const object
Warning FIG7_14.C 13: 'ptr' is assigned a value that is
   never used
Warning FIG7_14.C 13: 'y' is declared but never used
```

Fig. 7.14 Intento de modificación de un apuntador constante a datos constantes.

por lo que **swap** no tiene acceso a los elementos individuales del arreglo en **bubbleSort**. Dado que **bubbleSort** *desea* que **swap** tenga acceso a los elementos del arreglo a ser intercambiados, **bubbleSort** pasa cada uno de estos elementos por llamada por referencia a **swap** la dirección

de cada elemento del arreglo se pasa en forma explícita. Aunque arreglos completos de manera automática se pasan en llamada por referencia, los elementos individuales del arreglo son escalares y, por lo regular, son pasados en llamadas por valor. Por lo tanto, **bubbleSort** utiliza el operador de dirección (**&**) en cada uno de los elementos del arreglo de la llamada de **swap**, como sigue

```
swap(&array[j], &array[j + 1]);
```

para que ocurra la llamada por referencia. La función **swap** recibe **&array[j]** en la variable de apuntador **element1Ptr**. Aun cuando **swap** debido al ocultamiento de información no tiene permitido saber el nombre de **array[j]**, **swap** puede utilizar ***element1Ptr** como un sinónimo de **array[j]**. Por lo tanto, cuando **swap** hace referencia a ***element1Ptr**, de hecho está referenciando a **array[j]** de **bubbleSort**. Similarmente, cuando **swap** hace referencia a ***element2Ptr**, de hecho está referenciando a **array[j + 1]** de **bubbleSort**. Aun cuando **swap** no se le tiene permitido decir

```
temp = array[j];
array[j] = array[j + 1];
array[j + 1] = temp;
```

precisamente el mismo efecto se consigue mediante

```
temp = ;*element1Ptr;
*element1Ptr = *element2Ptr;
*element2Ptr = temp;
```

en la función **swap** de la figura 7.15.

Deben de hacerse notar ciertas características de la función **bubbleSort**. El encabezado de función declara **array** como **int *array** en vez de **int array[]**, para indicar que **bubbleSort** recibe como argumento un arreglo de un subíndice (otra vez, estas notaciones son intercambiables). El parámetro **size** se declara **const**, a fin de obligar al principio del mínimo privilegio. Aunque el parámetro **size** recibe una copia de un valor en **main**, y modificar dicha copia no puede cambiar el valor en **main**, para llevar a cabo su tarea **bubbleSort** no necesita modificar **size**. Durante la ejecución de **bubbleSort** el tamaño del arreglo se conserva fijo. Por lo tanto, **size** se declara **const**, para asegurarse de que no se modifique. Si durante el proceso de ordenación se llegara a modificar el tamaño del arreglo, sería posible que el algoritmo de ordenamiento no se ejecutase de forma correcta.

El prototipo para la función **swap** se incluye en el cuerpo de la función **bubbleSort**, porque es la única función que llama a **swap**. El colocar el prototipo en **bubbleSort** restringe llamadas directas a **swap** únicamente a las que **bubbleSort** ejecuta. Otras funciones que intenten llamar a **swap** no tienen acceso a un prototipo de función apropiada, por lo que el compilador genera en forma automática una. Esto por lo regular resulta un prototipo que no coincide con el encabezado de función (y genera un error de compilación), porque el compilador supone **int** para el tipo de regreso, y para los tipos de parámetros.

Observación de ingeniería de software 7.4

Colocar prototipos de función en las definiciones de otras funciones obliga al principio del mínimo privilegio al restringir las llamadas correctas de función sólo a aquellas funciones en las cuales dichos prototipos aparecen.


```
/* This program puts values into an array, sorts
   the values into ascending order, and prints the
   resulting array */
#include <stdio.h>
#define SIZE 10

void bubbleSort(int *, const int);

main()
{
   int i, a[SIZE] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};

   printf("Data items in original order\n");

   for (i = 0; i <= SIZE - 1; i++)
      printf("%4d", a[i]);

   bubbleSort(a, SIZE);          /* sort the array */
   printf("\nData items in ascending order\n");

   for (i = 0; i <= SIZE - 1; i++)
      printf("%4d", a[i]);

   printf("\n");
   return 0;
}

void bubbleSort(int *array, const int size)
{
   int pass, j;
   void swap(int *, int *);

   for (pass = 1; pass <= size - 1; pass++)

      for (j = 0; j <= size - 2; j++)

         if (array[j] > array[j + 1])
            swap(&array[j], &array[j + 1]);
}

void swap(int *element1Ptr, int *element2Ptr)
{
   int temp;

   temp = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = temp;
}
```

```
Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

Fig. 7.15 Ordenación de tipo burbuja con llamada por referencia.

Note que la función **bubbleSort** recibe como parámetro el tamaño del arreglo. La función debe saber el tamaño del arreglo, para poder clasificarlo. Cuando se pasa un arreglo a una función, la dirección en memoria del primer elemento del arreglo es recibido por la función. La dirección no proporciona ninguna información a la función en relación con el número de elementos del arreglo. Por lo tanto, el programador debe proporcionar a la función el tamaño de dicho arreglo.

En el programa, la función **bubbleSort** recibió de forma explícita el tamaño del arreglo. Existen dos ventajas principales en este enfoque —la reutilización del software y una adecuada ingeniería de software. Al definir la función de tal forma que reciba el tamaño del arreglo en forma de argumento, permitimos que se utilice la función en cualquier programa, que ordene arreglos enteros de un subíndice, así como que los arreglos puedan ser de cualquier tamaño.

Observación de ingeniería de software 7.5

Al pasar un arreglo a una función, pase también su tamaño. Esto ayuda a generalizar la función. Las funciones generales son a menudo reutilizables.

Podríamos haber almacenado el tamaño del arreglo en una variable global, accesible a todo el programa. Esto hubiera sido más eficiente, porque una copia del tamaño no se hubiera hecho para pasársela a la función. Sin embargo, otros programas que requieren de una capacidad de clasificación de arreglos enteros, pudieran no tener la misma variable global y, por lo tanto, la función no podría ser utilizada en dichos programas.

Observación de ingeniería de software 7.6

Las variables globales violan el principio del mínimo privilegio y son un ejemplo de ingeniería de software pobre.

Sugerencia de rendimiento 7.2

Pasar el tamaño de un arreglo a una función ocupa tiempo y espacio de pilas adicional, porque debe ejecutarse una copia del tamaño para ser pasada a la función. Las variables globales, sin embargo, no requieren de tiempo adicional o de espacio, porque son accesibles directamente por cualquier función.

El tamaño del arreglo pudiera haberse programado en directo dentro de la función. Esto restringe el uso de la función, a un arreglo de un tamaño específico, y reduce en forma importante su reutilización. Sólo programas que procesen arreglos enteros de un subíndice, del tamaño específico codificado dentro de la función, podrían utilizar esta función.

C proporciona el *operador unario* especial **sizeof** para determinar el tamaño de un arreglo en bytes (o de cualquier otro tipo de datos) durante la compilación de un programa. Cuando se aplica al nombre de un arreglo, como en la figura 7.16, el operador **sizeof** regresa como un entero el número total de bytes del arreglo. Note que las variables del tipo **float** están almacenadas por lo regular en 4 bytes de memoria, y **array** está declarado que tiene 20 elementos. Por lo tanto, existe en **array** un total de 80 bytes.

El número de elementos de un arreglo también puede ser determinado en tiempo de compilación. Por ejemplo, considere la siguiente declaración de arreglo:

```
double real[22];
```



```
/* sizeof operator when used on an array name */
/* returns the number of bytes in the array   */
#include <stdio.h>

main()
{
   float array[20];

   printf("The number of bytes in the array is %d\n",
          sizeof(array));

   return 0;
}
```

```
The number of bytes in the array is 80
```

Fig. 7.16 El operador **sizeof** cuando se aplica a un nombre de arreglo, regresa el número de bytes en el mismo.

Las variables **double** están normalmente almacenadas en 8 bytes de memoria. Por lo tanto, el arreglo **real** contiene un total de 176 bytes. Para determinar el número de elementos en el arreglo, puede utilizarse la siguiente expresión:

```
sizeof(real) / sizeof(double)
```

La expresión determina el número de bytes en el arreglo **real**, y divide dicho valor por el número de bytes utilizados en memoria para almacenar un valor **double**.

El programa de la figura 7.17 calcula el número de bytes utilizados para almacenar cada uno de los tipos de datos estándar, en una PC compatible.

Sugerencia de portabilidad 7.2

El número de bytes utilizados para almacenar un tipo particular de datos, pudiera variar de sistema a sistema. Al escribir programas que dependan de tamaños de tipos de datos, y que se ejecutarán en diversos sistemas de computación, utilice **sizeof** *para determinar el número de bytes utilizados para almacenar los tipos de datos.*

El operador **sizeof** puede ser aplicado a cualquier nombre de variable, tipo o constante. Al ser aplicado a un nombre de variable (que no sea un nombre de arreglo),o a una constante, será regresado el número de bytes utilizados para almacenar el tipo específico de variable o de constante. Note que son requeridos los paréntesis utilizados junto con **sizeof**, si el nombre del tipo se proporciona como su operandó. Si se omiten los paréntesis ocurrirá un error de sintaxis. No son requeridos los paréntesis si como su operando se proporciona el nombre de la variable.

7.7 Expresiones y aritmética de apuntadores

Los apuntadores son operandos válidos en expresiones aritméticas, en expresiones de asignación y en expresiones de comparación. Sin embargo, no todos los operadores, normalmente utilizados en estas expresiones son válidos, en conjunción con las variables de apuntador. En esta sección se describen los operadores que pueden tener apuntadores como operandos, y como se utilizan dichos operadores.

```
/* Demonstrating the sizeof operator */
#include <stdio.h>

main()
{
   printf("        sizeof(char) = %d\n"
          "       sizeof(short) = %d\n"
          "         sizeof(int) = %d\n"
          "        sizeof(long) = %d\n"
          "       sizeof(float) = %d\n"
          "      sizeof(double) = %d\n"
          "sizeof(long double) = %d\n",
          sizeof(char), sizeof(short), sizeof(int),
          sizeof(long), sizeof(float), sizeof(double),
          sizeof(long double));
   return 0;
}
```

```
sizeof(char)        = 1
sizeof(short)       = 2
sizeof(int)         = 2
sizeof(long)        = 4
sizeof(float)       = 4
sizeof(double)      = 8
sizeof(long double) = 10
```

Fig. 7.17 Cómo utilizar el operador **sizeof** para determinar los tamaños de tipo de datos estándar.

Con apuntadores se pueden ejecutar un conjunto limitado de operaciones aritméticas. Un apuntador puede ser incrementado (++) o decrementado (—), se puede añadir un entero a un apuntador (+ o +=), un entero puede ser restado de un apuntador (- o -=), o un apuntador puede ser sustraído o restado de otro.

Suponga que ha sido declarado el arreglo **int v[10]** y su primer elemento está en memoria en la posición **3000**. Suponga que el apuntador **vPtr** ha sido inicializado para apuntar a **v[0]**, es decir, el valor de **vPtr** es **3000**. En la figura 7.18 se diagrama esta situación para una máquina con enteros de 4 bytes. Note que **vPtr** puede ser inicializado para apuntar al arreglo **v** con cualquiera de los enunciados

```
vPtr = v;
vPtr = &v[0];
```

Sugerencia de portabilidad 7.3

La mayor parte de las computadoras de hoy día tienen enteros de 2 o de 4 bytes. Algunas de las máquinas más modernas utilizan enteros de 8 bytes. Dado que los resultados de la aritmética de apuntadores depende del tamaño de los objetos a los cuales el apuntador señala, la aritmética de los apuntadores depende de la máquina.

En aritmética convencional, la adición **3000 + 2** da como resultado el valor **3002**. Por lo regular, este no es el caso en la aritmética de apuntadores. Cuando se añade o se resta un entero de un apuntador, el apuntador no se incrementa o decrementa sólo por el valor de dicho entero,

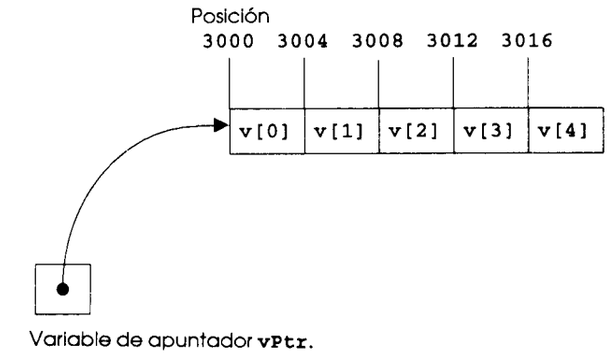


Fig. 7.18 El arreglo **v** y una variable de apuntador **vPtr**, que señala a **v**.

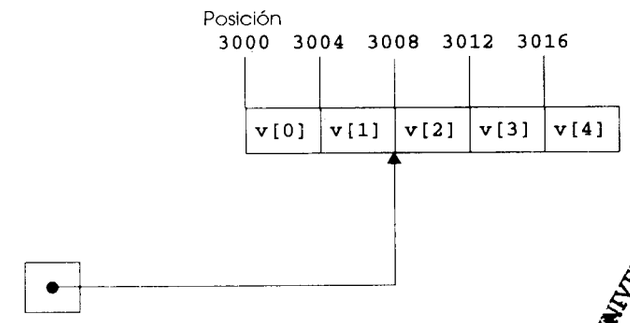
sino por el entero, multiplicado por el tamaño del objeto al cual el apuntador se refiere. El número de bytes depende del tipo de datos del objeto. Por ejemplo, el enunciado

```
vPtr += 2;
```

produciría **3008** (**3000 + 2 * 4**), suponiendo un entero almacenado en 4 bytes de memoria. En el arreglo **v**, **vPtr**, ahora señalaría a **v[2]** (figura 7.19). Si un entero se almacena en 2 bytes de memoria, entonces el cálculo anterior resultaría en una posición de memoria **3004** (**3000 + 2 * 2**). Si el arreglo fuera de un tipo de datos diferente, el enunciado precedente incrementaría el apuntador por dos veces el número de bytes que toma para almacenar un objeto de dicho tipo de datos. Al ejecutar aritmética de apuntadores en un arreglo de caracteres, los resultados serán consistentes con la aritmética normal, porque cada carácter tiene una longitud de un byte.

Si **vPtr** ha sido incrementado a **3016**, lo que señala a **v[4]**, el enunciado

```
vPtr - = 4;
```



definiría a **vPtr** de vuelta a **3000** lo que es decir al principio del arreglo. Si un apuntador está siendo incrementado o decrementado por uno, pueden ser utilizados los operadores de incremento (++)y de decremento (–). Cualquiera de los enunciados

```
++vPtr;
vPtr++;
```

incrementan el apuntador, para que apunte a la siguiente posición dentro del arreglo. Cualquiera de los enunciados

```
--vPtr;
vPtr--;
```

decrementan el apuntador, para que apunte al elemento anterior del arreglo.

Las variables de apuntador pueden ser restadas una de otra. Por ejemplo, si **vPtr** contiene la posición **3000**, y **v2Ptr** contiene la dirección **3008**, el enunciado

```
x = v2Ptr - vPtr;
```

asignaría a **x** el número de los elementos del arreglo de **vPtr** hasta **v2Ptr**, en este caso, **2**. La aritmética de apuntadores no tiene significado, a menos de que se ejecute en un arreglo. No podemos suponer que dos variables del mismo tipo estén almacenados de manera contigua en memoria, a menos de que sean elementos adyacentes de un arreglo.

Error común de programación 7.5

Utilizar aritmética de apuntadores en un apuntador que no se refiere a un arreglo de valores.

Error común de programación 7.6

Restar o comparar dos apuntadores que no se refieren al mismo arreglo.

Error común de programación 7.7

Salirse de cualquiera de los dos extremos de un arreglo al utilizar aritmética de apuntadores.

Un apuntador puede ser asignado a otro apuntador, si ambos son del mismo tipo. De lo contrario, deberá utilizarse un operador cast para convertir el apuntador a la derecha de la asignación al tipo de apuntador de la izquierda de la asignación. La excepción a esta regla es el apuntador a **void** (es decir, **void ***) que es un apuntador genérico, que puede representar cualquier tipo de apuntador. Todos los tipos de apuntador pueden ser asignados a un apuntador **void** y un apuntador **void** puede ser asignado a un apuntador de cualquier tipo. En ambos casos no se requiere de una operación cast.

Un apuntador a **void** no puede ser desreferenciado. Por ejemplo, el compilador sabe que un apuntador a **int** se refiere a 4 bytes, en memoria en una máquina con enteros de 4 bytes, pero un apuntador a **void** sólo contiene una posición de memoria para un tipo de datos desconocido —el número preciso de bytes a los que se refiere el apuntador no es conocido por el compilador. Para un apuntador en particular, el compilador debe saber el tipo de datos, para determinar el número de bytes a desreferenciar. En el caso de un apuntador **void**, el número de bytes no puede ser determinado partiendo del tipo.

Error común de programación 7.8

Se generará un error de sintaxis asignar un apuntador de un tipo a un apuntador de otro tipo, si ninguno de los dos es del tipo **void ***.

Error común de programación 7.9

Desreferenciar un apuntador **void ***.

Los apuntadores pueden ser comparados mediante operadores de igualdad y relacionales, pero dichas comparaciones no tendrán sentido, a menos que los apuntadores señalen a miembros del mismo arreglo. Las comparaciones de apuntadores comparan las direcciones almacenadas en los mismos. Una comparación de dos apuntadores que señalen al mismo arreglo podría mostrar, por ejemplo, que un apuntador señale a un elemento de numeración más alta en el arreglo que el otro. Un uso común de una comparación de apuntadores es determinar si un apuntador es **NULL**.

7.8 Relaciones entre apuntadores y arreglos

Los arreglos y los apuntadores en C están relacionados en forma íntima y pueden ser utilizados *casi* en forma indistinta. Un nombre de arreglo puede ser considerado como un apuntador constante. Los apuntadores pueden ser utilizados para hacer cualquier operación que involucre subíndices de arreglos.

Sugerencia de rendimiento 7.3

Durante la compilación la notación de subíndices de arreglo se convierte a notación de apuntador, por lo que escribir expresiones de subíndices de arreglo con notación de apuntadores, puede ahorrar tiempo de compilación.

Práctica sana de programación 7.5

Al manipular arreglos utilice notación de arreglos en vez de notación de apuntadores. Aunque el programa pudiera tomar más tiempo para su compilación, es probable que sea mucho más claro.

Suponga que han sido declarados el arreglo entero **b[5]** y la variable de apuntador entera **bPtr**. Dado que el nombre del arreglo (sin subíndice) es un apuntador al primer elemento del arreglo, podemos definir **bPtr** igual a la dirección del primer elemento en el arreglo **b**, mediante el enunciado

```
bPtr = b;
```

Este enunciado es equivalente a tomar la dirección del primer elemento del arreglo, como sigue

```
bPtr = &b[0];
```

Alternativamente el elemento del arreglo **b[3]** puede ser referenciado con la expresión de apuntador

```
*(bPtr + 3)
```

El **3** en la expresión arriba citada es el *desplazamiento* del apuntador. Cuando el apuntador apunta al principio de un arreglo, el desplazamiento indica qué elemento del arreglo debe ser referenciado, y el valor de desplazamiento es idéntico al subíndice del arreglo. La notación anterior se conoce como *notación apuntador/desplazamiento*. Son necesarios los paréntesis porque la precedencia de * es más alta que la de +. Sin los paréntesis, la expresión arriba citada sumaría **3** al valor de la expresión ***bPtr** (es decir, se añadiría **3** a **b[0]**, suponiendo que **bPtr** apunta al principio del arreglo). Al igual que el elemento del arreglo puede ser referenciado con una expresión de apuntador, la dirección

```
&b[3]
```

puede ser escrita con la expresión de apuntador

```
bPtr + 3
```

El arreglo mismo puede ser tratado como un apuntador, y utilizado en aritmética de apuntador. Por ejemplo, la expresión

```
*(b + 3)
```

también se refiere al elemento del arreglo `b[3]`. En general, todas las expresiones de arreglos con subíndice pueden ser escritas mediante un apuntador y un desplazamiento. En este caso se utilizó notación apuntador/desplazamiento junto con el nombre del arreglo como un apuntador. Note que el enunciado anterior no cambia de forma alguna el nombre del arreglo; `b` aún apunta al primer elemento del arreglo.

Los apuntadores pueden tener subíndices exactamente de la misma forma que los arreglos. Por ejemplo, la expresión

```
bPtr[1]
```

se refiere al elemento del arreglo `b[1]`. Esto se conoce como *notación apuntador/subíndice.*

Recuerde que el nombre de un arreglo es, en esencia, un apuntador constante; siempre apunta al principio del arreglo. Por lo tanto, la expresión

```
b += 3
```

resulta inválida, porque intenta modificar el valor del nombre de un arreglo con aritmética de apuntador.

Error común de programación 7.10

Es un error de sintaxis intentar modificar un nombre de arreglo con aritmética de apuntador.

El programa de la figura 7.20 utiliza los cuatro métodos analizados aquí para referirse a los elementos del arreglo —arreglos con subíndices, apuntador/desplazamiento con el nombre del arreglo como un apuntador, subíndice de apuntador, y apuntador/desplazamiento con un apuntador para imprimir los cuatro elementos del arreglo entero `b`.

Para ilustrar más aún la intercambiabilidad de arreglos y apuntadores, veamos las dos funciones de copia de cadenas —`copy1` y `copy2`— del programa de la figura 7.21. Ambas funciones copian una cadena (posiblemente un arreglo de caracteres) en un arreglo de caracteres. Después de una comparación de los prototipos de función de `copy1` y `copy2`, las funciones parecen idénticas. Llevan a cabo la misma tarea; sin embargo, se ponen en operación de forma distinta.

La función `copy1` utiliza notación de subíndices de arreglo para copiar la cadena en `s2` al arreglo de caracteres `s1`. La función declara una variable de contador entera `i`, para usarla como subíndice del arreglo. El encabezado de estructura `for` ejecuta toda la operación de copia —su cuerpo es el enunciado vacío. El encabezado especifica que `i` se inicializa a cero y se incrementa en uno en cada iteración del ciclo. La condición en la estructura `for`, `s1[i] = s2[i]`, ejecuta la operación de copia, carácter por carácter, desde `s2` a `s1`. Cuando se encuentra el carácter null en `s2`, se asigna a `s1`, y el ciclo termina porque el valor entero del carácter null es cero (falso). Recuerde que el valor de un enunciado de asignación es el valor asignado al argumento izquierdo.


```
/* Using subscripting and pointer notations with arrays */
#include <stdio.h>

main()
{
   int i, offset, b[] = {10, 20, 30, 40};
   int *bPtr = b;   /* set bPtr to point to array b */

   printf("Array b printed with:\n"
          "Array subscript notation\n");

   for (i = 0; i <= 3; i++)
      printf("b[%d] = %d\n", i, b[i]);

   printf("\nPointer/offset notation where \n"
          "the pointer is the array name\n");

   for (offset = 0; offset <= 3; offset++)
      printf("*(b + %d) = %d\n", offset, *(b + offset));

   printf("\nPointer subscript notation\n");

   for (i = 0; i <= 3; i++)
      printf("bPtr[%d] = %d\n", i, bPtr[i]);

   printf("\nPointer/offset notation\n");

   for (offset = 0; offset <= 3; offset++)
      printf("*(bPtr + %d) = %d\n", offset, *(bPtr + offset));

   return 0;
}
```

Fig. 7.20 Cómo usar los cuatro métodos de referenciar los elementos de arreglo (parte 1 de 2).

La función `copy2` utiliza apuntadores y aritmética de apuntadores para copiar la cadena de `s2` al arreglo de caracteres `s1`. Otra vez, el encabezado de estructura `for` ejecuta toda la operación de copia. El encabezado no incluye ninguna inicialización de variables. Como en la función `copy1`, la condición (`*s1 = *s2`) ejecuta la operación de copia. El apuntador `s2` se desreferencia y el carácter resultante se asigna al apuntador desreferenciado `s1`. Después de la asignación en la condición, los apuntadores se incrementan para señalar al siguiente elemento del arreglo `s1`, y el siguiente carácter de la cadena `s2`, respectivamente. Cuando en `s2` se encuentra el carácter null, se le asigna al apuntador desreferenciado `s1` y el ciclo se termina.

Note que el primer argumento, tanto de `copy1` como de `copy2` debe ser un arreglo lo suficiente grande para contener la cadena del segundo argumento. De lo contrario, podría ocurrir un error cuando se intente escribir en una posición de memoria que no forme parte del arreglo. También, note que el segundo parámetro de cada función se declara como `const char *` (una cadena constante). En ambas funciones, el segundo argumento se copia en el primer argumento —los caracteres se leen a partir de él uno por uno, pero los caracteres nunca se modifican. Por lo tanto, se declara el segundo parámetro para señalar a un valor constante, y que se cumpla el

```
Array b printed with:
Arry subscript notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

Pointer/offset notion where
the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

Pointer subscript notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

Pointer/offset notion
*(bPtr + 9) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40
```

Fig. 7.20 Cómo usar los cuatro métodos de referenciar los elementos de arreglo (parte 2 de 2).

principio del mínimo privilegio. Ninguna de las funciones requiere de capacidad de modificar el segundo argumento, por lo que ninguna de ellas la tiene.

7.9 Arreglos de apuntadores

Los arreglos pueden contener apuntadores. Un uso común para una estructura de datos como ésta, es formar un arreglo de cadenas, conocida como un *arreglo de cadenas*. Cada entrada en el arreglo es una cadena, pero en C una cadena es esencial un apuntador a su primer carácter. Por lo que en un arreglo de cadenas cada entrada es de hecho un apuntador al primer carácter de una cadena. Veamos la declaración del arreglo de cadenas **suit**, que pudiera ser útil para representar un mazo de naipes.

```
char *suit[4] = {"Hearts", "Diamonds", "Clubs", "Spades"};
```

La porción **suit[4]** de la declaración indica un arreglo de 4 elementos. La porción **char*** de la declaración indica que cada elemento del arreglo **suit** es del tipo "apuntador a **char**". Los cuatro valores a colocarse en el arreglo son **"Hearts"**, **"Diamonds"**, **"Clubs"** y **"Spades"**. Cada una de estas está almacenada en memoria como una cadena de caracteres terminada por NULL, de una longitud de un carácter más largo que el número de caracteres entre las comillas. Las cuatro cadenas son de 7, 9, 6 y 7 caracteres de longitud, respectivamente. Aunque pareciera como si estas cadenas están colocadas en el arreglo **suit**, de hecho en el arreglo sólo están almacenados los apuntadores (figura 7.22).


```
/* Copying a string using array notation
   and pointer notation */
#include <stdio.h>

void copy1(char *, const char *);
void copy2(char *, const char *);

main()
{
   char string1[10], *string2 = "Hello",
        string3[10], string4[] = "Good Bye";

   copy1(string1, string2);
   printf("string1 = %s\n", string1);

   copy2(string3, string4);
   printf("string3 = %s\n", string3);
   return 0;
}

/* copy s2 to s1 using array notation */
void copy1(char *s1, const char *s2)
{
   int i;

   for (i = 0; s1[i] = s2[i]; i++)
      ;   /* do nothing in body */
}

/* copy s2 to s1 using pointer notation */
void copy2(char *s1, const char *s2)
{
   for ( ; *s1 = *s2; s1++, s2++)
      ;   /* do nothing in body */
}
```

```
string1 = Hello
string3 = Good Bye
```

Fig. 7.21 Como copiar una cadena utilizando notación de arreglo y notación de apuntador.

Cada apuntador señala al primer carácter de su cadena correspondiente. Por lo tanto, aunque el arreglo **suit** es de tamaño fijo, permite el acceso a cadenas de caracteres de cualquier longitud. Esta flexibilidad es un ejemplo de las capacidades poderosas de estructuración de datos de C.

Los palos de la baraja podrían haber sido colocados en un arreglo doble, en el cual cada renglón podría representar un palo, y cada columna representaría una de las letras del nombre de dicho palo. Una estructura de datos como ésta tendría que tener un número fijo de columnas por renglón, y dicho número debería ser igual de grande que la cadena más larga. Por lo tanto, se desperdiciaría gran cantidad de memoria, cuando se tuviera que almacenar un gran número de

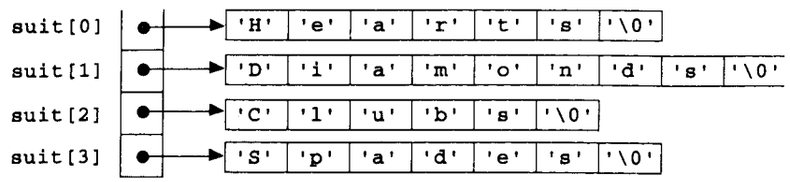


Fig. 7.22 Un ejemplo gráfico del arreglo **suit**.

cadenas, con la mayoría de ellas más cortas que la más larga. En la siguiente sección usamos arreglos de cadenas para representar un mazo de naipes.

7.10 Estudio de caso: simulación de barajar y repartir naipes

En esta sección, utilizamos la generación de números aleatorios para desarrollar un programa de simulación de barajar y repartir naipes. Este programa podrá ser después utilizado como base para formar programas, que jueguen juegos específicos de naipes. A fin de revelar algunos problemas sutiles de rendimiento, con intención hemos utilizado algoritmos subóptimos de barajar y de repartir. En los ejercicios y en el capítulo 10, desarrollaremos algoritmos más eficaces.

Utilizando el enfoque de refinación descendente, paso a paso, desarrollamos un programa que barajará un mazo de 52 naipes, y a continuación repartirá cada uno de dichos 52 naipes. El enfoque descendente es en particular útil para atacar problemas más grandes y más complejos que los que hemos visto en capítulos anteriores.

Utilizamos un arreglo de doble subíndice de 4 por 13 de nombre **deck** para representar el mazo de naipes (figura 7.23). Los renglones corresponden a los palos —el renglón 0 corresponde a los corazones, el renglón 1 a los diamantes, el renglón 2 a los tréboles y el renglón 3 a las espadas. Las columnas corresponden a los valores nominales de los naipes —las columnas del 0 al 9 corresponden a los valores del as al diez respectivamente, y las columnas 10 a la 12 corresponden al jack, reina y rey. Cargaremos el arreglo de cadenas **suit**, con cadenas de caracteres que representen los cuatro palos, y el arreglo de cadenas **face**, con cadenas de caracteres que representen los trece valores nominales.

Este mazo simulado de naipes puede ser barajado como sigue. Primero el arreglo **deck** se iguala a cero. A continuación, se selecciona aleatoriamente un **renglón** (0-3) y una **columna** (0-12). El número 1 se inserta en el elemento del arreglo **deck[row] [column]** para indicar que este naipe será el primero que se repartirá del mazo barajado. Este proceso continúa con los números 2, 3, ..., 52 que se insertarán al azar en el arreglo **deck**, para indicar cuáles son los naipes que se le colocarán segundo, tercero, ..., y cincuenta y dos en el mazo barajado. Conforme el arreglo **deck** se empiece a llenar con números de naipes, es posible que un naipe quede seleccionado dos veces, es decir **deck[row] [column]** al ser seleccionado resulte en no cero. Esta selección se ignora simplemente y se vuelve a repetir la selección de otros **rows** y **columns** en forma aleatoria, hasta que se encuentre un naipe no seleccionado. Eventualmente los números 1 hasta el 52 ocuparán los 52 renglones del arreglo **deck**. Llegado a este punto, el mazo de naipes ha sido totalmente barajado.

Si los naipes que ya han sido barajados se seleccionasen repetidamente al azar, este algoritmo de barajar se ejecutaría en forma indefinida. Este fenómeno se conoce como *posposición indefinida*. En los ejercicios analizaremos un mejor algoritmo de barajar, que elimina la posibilidad de posposición indefinida.

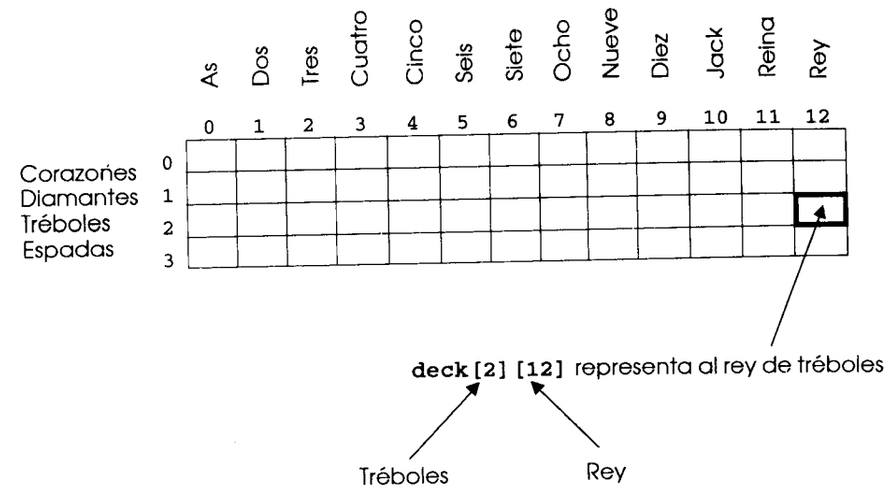


Fig. 7.23 Representación de arreglo de doble subíndice de un mazo de naipes.

Sugerencia de rendimiento 7.4

Algunas veces un algoritmo que aparece de forma "natural" puede contener problemas sutiles de rendimiento, como es la posposición indefinida. Busque algoritmos que eviten la posposición indefinida.

Para repartir el primer naipe, buscaremos en el arreglo a **deck[row] [column] = 1**. Esto se lleva a cabo con una estructura anidada **for**, que varía **row** desde 0 hasta 3 y a **column** desde 0 hasta 12. ¿A qué naipe corresponde esta posición dentro del arreglo? El arreglo **suit** ha sido precargado con los cuatro palos, por lo que para obtener el palo, imprimimos la cadena de caracteres **suit[row]**. Similarmente, para obtener el valor nominal del naipe, imprimimos la cadena de caracteres **face[column]**. También imprimimos la cadena de caracteres **" of "**. Al imprimir esta información en el orden adecuado, nos permite imprimir cada naipe de la forma **"King of Clubs"**, **"Ace of Diamonds"**, y así en lo sucesivo.

Sigamos con el proceso de refinación descendente paso a paso. Lo general es simplemente

Shuffle and deal 52 cards

Nuestro primer refinamiento da como resultado:

Initialize the suit array
Initialize the face array
Initialize the deck array
Shuffle the deck
Deal 52 cards

"Shuffle the deck" pudiera expandirse, como sigue:

For each of the 52 cards
  Place card number in randomly selected unoccupied slot of deck

"Deal 52 cards" puede expandirse como sigue:

For each of the 52 cards
Find card number in deck array print face and suit of card

La incorporación de estas expansiones da como resultado nuestro segundo refinamiento completo:

Initialize the suit array
Initialize the face array
Initialize the deck array
For each of the 52 cards
Place card number in randomly selected unoccupied slot of deck

For each of the 52 cards
Find card number in deck array and print face and suit of card

"Place card number in randomly selected unoccupied slot of deck" se puede expandir como sigue:

Choose slot of deck randomly
While chosen slot of deck has been previously chosen
Choose slot of deck randomly
Place card number in chosen slot of deck

"Find card number in deck array and print face and suit of card" puede ser expandido como sigue:

For each slot of the deck array
If slot contains card number
Print the face and suit of the card

Al incorporar estas expansiones da como resultado nuestro tercer refinamiento:

Initialize the suit array
Initialize the face array
Initialize the deck array

For each of the 52 cards
Choose slot of deck randomly
While slot of deck has been previously chosen
Choose slot of deck randomly
Place card number in chosen slot of deck

For each of the 52 cards
For each slot of deck array
If slot contains desired card number
Print the face and suit of the card

Esto completa el proceso de refinamiento. Note que este programa es más eficiente si se combinan las porciones de barajar y de repartir del algoritmo, de tal forma que cada naipe sea repartido conforme se coloca en el mazo. Hemos decidido programar estas operaciones por separado, ya que por lo regular los naipes se reparten después de que han sido barajados (y no mientras se barajan).

El programa de barajar y repartir naipes se muestra en la figura 7.24 y una ejecución de muestra aparece en la figura 7.25. Note en las llamadas a **printf** el uso del especificador de conversión **%s** para imprimir cadenas de caracteres. El argumento correspondiente en la llamada **printf** debe ser un apuntador a **char** (o un arreglo **char**). En la función **deal**, la especificación de formato **"%5s of%-8s"** imprime una cadena de caracteres, justificado a la derecha, en un campo de cinco caracteres, seguido por **" of "** y una cadena de caracteres, justificado a la izquierda, en un campo de ocho caracteres. El signo menos en **%-8s** significa que la cadena está justificada a la izquierda, en un campo de ancho 8.

```
/* Card dealing program */

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

void shuffle(int [][13]);
void deal(const int [][13], const char *[], const char *[]);

main()
{
   char *suit[4] = {"Hearts", "Diamonds", "Clubs", "Spades"};
   char *face[13] = {"Ace", "Deuce", "Three", "Four",
                     "Five", "Six", "Seven", "Eight",
                     "Nine", "Ten", "Jack", "Queen", "King"};
   int deck[4][13] = {0};

   srand(time(NULL));

   shuffle(deck);
   deal(deck, face, suit);

   return 0;
}

void shuffle(int wDeck[][13])
{
   int card, row, column;

   for(card = 1; card <= 52; card++) {
      row = rand() % 4;
      column = rand() % 13;

      while(wDeck[row][column] != 0) {
         row = rand() % 4;
         column = rand() % 13;
      }

      wDeck[row][column] = card;
   }
}
```

Fig. 7.24 Programa de repartición de naipes (parte 1 de 2).

```
void deal(const int wDeck[][13], const char *wFace[],
          const char *wSuit[])
{
   int card, row, column;

   for (card = 1; card <= 52; card++)

      for (row = 0; row <= 3; row++)

         for (column = 0; column <= 12; column++)

            if (wDeck[row][column] == card)
               printf("%5s of %-8s%c",
                      wFace[column], wSuit[row],
                      card % 2 == 0 ? '\n' : '\t');
}
```

Fig. 7.24 Programa de repartición de naipes (parte 2 de 2).

```
  Six of Clubs        Seven of Dimonds
  Ace of Spades         Ace of Diamonds
  Ace of Hearts       Queen of Diamonds
Queen of Clubs        Seven of Hearts
  Ten of Hearts       Deuce of Clubs
  Ten of Spades       Three of Spades
  Ten of Diamonds      Four of Spades
 Four of Diamonds       Ten of Clubs
  Six of Diamonds       Six of Spades
Eight of Hearts       Three of Diamonds
 Nine of Hearts       Three of Hearts
Deuce of Spades         Six of Hearts
 Five of Clubs        Eight of Clubs
Deuce of Diamonds     Eight of Spades
 Five of Spades        King of Clubs
 King of Diamonds      Jack of Spades
Deuce of Hearts      Queen of Hearts
  Ace of Clubs        King of Spades
Three of Clubs        King of Hearts
 Nine of Clubs         Nine of Spades
 Four of Hearts      Queen of Spades
Eight of Diamonds     Nine of Diamonds
 Jack of Diamonds     Seven of Clubs
 Five of Hearts        Five of Diamonds
 Four of Clubs         Jack of Hearts
 Jack of Clubs       Seven of Spades
```

Fig. 7.25 Ejecución de muestra del programa de repartición de naipes.

Existe una debilidad en el algoritmo de distribución. Una vez que se encuentra una coincidencia, aún si se encuentra en el primer intento, las dos estructuras internas **for** continúan su búsqueda de los elementos restantes de **deck**, buscando coincidencia. En los ejercicios y en el estudio de caso del capítulo 10, corregiremos esta deficiencia.

7.11 Apuntadores a funciones

Un apuntador a una función contiene la dirección de la función en memoria. En el capítulo 6, vimos que el nombre de un arreglo es en realidad la dirección de memoria del primer elemento de dicho arreglo. Similarmente, el nombre de una función es realmente la dirección inicial en memoria del código que ejecuta la tarea de dicha función. Los apuntadores a las funciones pueden ser pasados a las funciones, regresado de las funciones, almacenados en arreglos, y asignados a otros apuntadores de función.

Para ilustrar el uso de apuntadores a funciones, hemos modificado el programa de clasificación de tipo burbuja de la figura 7.15 para formar el programa de la figura 7.26. Nuestro nuevo programa consiste de **main**, y de las funciones **bubble**, **swap**, **ascending**, y **descending**. La función **bubbleSort** recibe un apuntador a una función ya sea a la función **ascending** o a la función **descending** como un argumento, en adición a un arreglo entero y el tamaño del arreglo. El programa le solicita al usuario que escoja si el arreglo deberá de ser ordenado en orden ascendente o en orden descendente. Si el usuario escribe 1, un apuntador a la función **ascending** se pasa a la función **bubble**, haciendo que el arreglo sea ordenado en orden ascendente. Si el usuario introduce 2, un apuntador a la función **descending** se pasa a la función **bubble**, causando que el arreglo sea ordenado en orden descendente. La salida del programa se muestra en la figura 7.27.

El parámetro siguiente aparece en el encabezado de función correspondiente **a bubble**:

```
int (*compare)(int, int)
```

Esto le indica a **bubble** que espere un parámetro, que es un apuntador a una función, que recibe dos parámetros enteros y regresa un resultado entero. Dado que * tiene una precedencia inferior que los paréntesis que encierran a los parámetros de función, se requieren paréntesis alrededor de ***compare**. Si no se hubieran incluido los paréntesis, la declaración habría sido

```
int *compare(int, int)
```

lo que declara una función que recibe dos enteros como parámetros, y que regresa un apuntador como un entero.

El parámetro correspondiente a la función prototipo de **bubble** es

```
int (*)(int, int)
```

Note que sólo se han incluído tipos, pero para fines de documentación el programador puede incluir nombres, mismos que serán ignorados por el compilador.

La función pasada a **bubble** es llamada en un enunciado **if**, como sigue

```
if ((*compare)(work[count], work[count + 1]))
```

Igual que un apuntador a una variable es desreferenciado para poder tener acceso al valor de la variable, un apuntador a una función es desreferenciado, para utilizar la función.

La llamada a la función podría haber sido efectuada sin desreferenciar el apuntador, como en

```
if (compare(work[count], work[count + 1]))
```

```
/* Multipurpose sorting program using function pointers */
#include <stdio.h>
#define SIZE 10

void bubble(int *, const int, int (*)(int, int));
int ascending(const int, const int);
int descending(const int, const int);

main()
{
   int a[SIZE] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};
   int counter, order;

   printf("Enter 1 to sort in ascending order,\n");
   printf("Enter 2 to sort in descending order: ");
   scanf("%d", &order);

   printf("\nData items in original order\n");

   for (counter = 0; counter <= SIZE - 1; counter++)
      printf("%4d", a[counter]);

   if (order == 1) {
      bubble(a, SIZE, ascending);
      printf("\nData items in ascending order\n");
   }
   else {
      bubble(a, SIZE, descending);
      printf("\nData items in descending order\n");
   }

   for (counter = 0; counter <= SIZE - 1; counter++)
      printf("%4d", a[counter]);

   printf("\n");

   return 0;
}

void bubble(int *work, const int size, int (*compare)(int, int))
{
   int pass, count;
   void swap(int *, int *);

   for (pass = 1; pass <= size - 1; pass++)

      for (count = 0; count <= size - 2; count++)

         if ((*compare)(work[count], work[count + 1]))
            swap(&work[count], &work[count + 1]);
}
```

Fig. 7.26 Programa de ordenación de uso múltiple, utilizando apuntadores de función (parte 1 de 2).


```
void swap(int *element1Ptr, int *element2Ptr)
{
   int temp;

   temp = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = temp;
}

int ascending(const int a, const int b)
{
   return b < a;
}

int descending(const int a, const int b)
{
   return b > a;
}
```

Fig. 7.26 Programa de ordenación de uso múltiple, utilizando apuntadores de función (parte 2 de 2).

mismo que usa el apuntador directamente como nombre de función. Para llamar una función a través de un apuntador preferimos el primer método, porque de forma explícita ilustra que **compare** es un apuntador a una función, mismo que está desreferenciado para llamar la función. El segundo método de llamar una función a través de un apuntador, lo hace aparecer como si **compare** fuera en realidad una función. Esto pudiera resultar confuso a un usuario del programa que desease ver la definición de la función **compare**, encontrándose que nunca se define en el archivo.

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in descending order
  89  68  45  37  12  10   8   6   4   2
```

Fig. 7.27 Despliegues del programa de ordenación tipo burbuja de la figura 7.26.

Un uso común de los apuntadores de función ocurre en los denominados sistemas manejados por menú. Al usuario se le pide que seleccione una opción de un menú (posiblemente de 1 a 5). Cada opçión está servida por una función diferente. En un arreglo de apuntador a funciones se almacenan apuntadores a cada función. La selección del usuario se utiliza en el arreglo como un subíndice, y el apuntador en el arreglo se utiliza para llamar a la función.

El programa de la figura 7.28 da un ejemplo genérico de la mecánica de la declaración y uso de un arreglo de apuntadores a funciones. Se definen tres funciones —**function1**, **function2** y **function3**— donde cada una de ellas toma un argumento entero y no regresa nada. Los apuntadores a estas tres funciones se almacenan en un arreglo **f**, que se declara como sigue:

```
void *f[3])(int) = {function1, function2, function3};
```

La declaración se lee empezando en el conjunto de paréntesis más a la izquierda, "**f** es un arreglo de tres apuntadores a funciones que toman como argumento un **int** y que regresan **void**". El arreglo se inicializa con los nombres de las tres funciones. Cuando el usuario introduce un valor entre 0 y 2, el valor se utiliza como subíndice en el arreglo de apuntadores a funciones. La llamada de función se efectúa como sigue:

```
(*f[choice])(choice);
```

En la llamada, **f[choice]** selecciona el apuntador en la posición **choice** del arreglo. El apuntador es desreferenciado para llamar la función, y **choice** es pasado como el argumento a la función. Cada función imprime su valor de argumento y su nombre de función, para indicar que la función ha sido correctamente llamada. En los ejercicios, usted desarrollará un sistema manejado por menú.

Resumen

- Los apuntadores son variables que contienen como sus valores direcciones de otras variables.
- Los apuntadores deben ser declarados, antes de que puedan ser usados.
- La declaración

```
int *ptr;
```

- declara a **ptr** como un apuntador a un objeto del tipo **int**, y se lee, "**ptr** es un apuntador a **int**". El * como se utiliza aquí en una declaración, indica que la variable es un apuntador.
- Existen tres valores que pueden ser utilizados para inicializar un apuntador; **0**, **NULL**, o una dirección. Es lo mismo inicializar un apuntador a **0** e inicializar el mismo apuntador a **NULL**.
- El único entero que puede ser asignado a un apuntador es 0.
- El operador **&** (de dirección) regresa la dirección de su operando.
- El operando del operador de dirección debe ser una variable; el operador de dirección no puede ser aplicado a constantes, a expresiones, o a variables declaradas con la clase de almacenamiento **register**.
- El operador *, conocido como operador de indirección o de desreferenciación, regresa el valor del objeto al cual apunta su operando en memoria. Esto se llama desreferenciar el apuntador.


```
/* Demonstrating an array of pointers to functions */
#include <stdio.h>

void function1(int);
void function2(int);
void function3(int);

main()
{
   void (*f[3])(int) = {function1, function2, function3};
   int choice;

   printf("Enter a number between 0 and 2, 3 to end: ");
   scanf("%d", &choice);

   while (choice >= 0 && choice < 3) {
      (*f[choice])(choice);
      printf("Enter a number between 0 and 2, 3 to end: ");
      scanf("%d", &choice);
   }

   printf("You entered 3 to end\n");
   return 0;
}

void function1(int a)
{
   printf("You entered %d so function1 was called\n\n", a);
}

void function2(int b)
{
   printf("You entered %d so function2 was called\n\n", b);
}

void function3(int c)
{
   printf("You entered %d so function3 was called\n\n", c);
}
```

```
Enter a number between 0 and 2, 3 to end: 0
You entered 0 so function1 was called

Enter a number between 0 and 2, 3 to end: 1
You entered 1 so function2 was called

Enter a number between 0 and 2, 3 to end: 2
You entered 2 so function3 was called

Enter a number between 0 and 2, 3 to end: 3
You entered 3 to end
```

Fig. 7.28 Cómo demostrar un arreglo de apuntadores a funciones.

- Al llamar una función con un argumento, que el llamador desea que la función llamada modifique, se pasa la dirección del argumento. A continuación la función llamada utiliza el operador de indirección (*) para modificar el valor del argumento de la función llamadora.
- Una función que recibe una dirección como un argumento, debe incluir un apuntador como su parámetro formal correspondiente.
- En los prototipos de función no es necesario incluir los nombres de los apuntadores; sólo es necesario incluir los tipos de apuntador. Los nombres de apuntador pueden ser incluídos por razones de documentación, pero el compilador los ignorará.
- El calificador **const** permite al programador informarle al compilador que el valor de una variable particular no debe ser modificado.
- Si se intenta modificar un valor declarado **const**, el compilador lo detectará y emitirá una advertencia o un error, dependiendo del compilador particular.
- Existen cuatro formas para pasar un apuntador a una función: un apuntador no constante a datos no constantes, un apuntador constante a datos no constantes, un apuntador no constante a datos constantes y un apuntador constante a datos constantes.
- Los arreglos son pasados por referencia en forma automática, porque el valor del nombre del arreglo es la dirección del mismo.
- Para pasar un elemento de un arreglo en llamada por referencia, deberá ser pasada la dirección del elemento específico del arreglo.
- C proporciona el operador unario especial **sizeof**, para determinar el tamaño en bytes de un arreglo (o de cualquier otro tipo de datos) durante la compilación del programa.
- Al aplicarse al nombre de un arreglo, el operador **sizeof** regresa el número total de bytes en el arreglo, como un entero.
- El operador **sizeof** puede ser aplicado a cualquier nombre de variable, tipo o constante.
- Las operaciones aritméticas que pueden ser ejecutadas sobre apuntadores son incrementar (++) un apuntador, decrementar (–) un apuntador, sumar (+ o +=) un apuntador y un entero, restar (- o -=) un apuntador y un entero y restar un apuntador de otro.
- Cuando un entero se añade o se resta de un apuntador, el apuntador se incrementa o se decrementa por dicho entero, multiplicado por el tamaño del objeto al cual se apunta.
- Las operaciones aritméticas de apuntador deberían ser ejecutadas sólo en porciones contiguas de memoria, como existen en un arreglo. Todos los elementos de un arreglo están almacenados en memoria de forma contigua.
- Al ejecutar aritmética de apuntador en un arreglo de caracteres, los resultados son como en aritmética normal, dado que cada carácter está almacenado en un byte de memoria.
- Los apuntadores pueden ser asignados de uno a otro, si ambos apuntadores son del mismo tipo. De lo contrario, deberá de utilizarse una conversión (cambio de tipo). La excepción a lo anterior es un apuntador a **void**, que es un tipo de apuntador genérico que puede contener apuntadores de cualquier tipo. A los apuntadores a **void** se les pueden asignar apuntadores de otros tipos y pueden ser asignados a apuntadores de otros tipos sin necesidad de una conversión.
- Un apuntador **void** no puede ser desreferenciado.
- Los apuntadores pueden ser comparados utilizando los operadores de igualdad y relacionales. Las comparaciones de apuntadores por lo regular son significativas sólo si los apuntadores apuntan a miembros del mismo arreglo.

- Los apuntadores pueden tener subíndices exactamente como los nombres de los arreglos.
- Un nombre de arreglo sin subíndice es un apuntador al primer elemento del arreglo.
- En notación apuntador/desplazamiento, el desplazamiento es el mismo que un subíndice de arreglo.
- Todas las expresiones de arreglos con subíndice pueden ser escritas con un apuntador y un desplazamiento, utilizando ya sea el nombre del arreglo como un apuntador, o un apuntador por separado, que apunte al arreglo.
- Un nombre de arreglo es un apuntador constante, que siempre apunta a la misma posición en memoria. Los nombres de arreglo no pueden ser modificados, como pueden ser modificados los apuntadores convencionales.
- Es posible tener arreglos de apuntadores.
- Es posible tener apuntadores a funciones.
- Un apuntador a una función es la dirección donde reside el código de la función.
- Los apuntadores a las funciones pueden ser pasados a funciones, regresados de funciones, almacenados en arreglos y asignados a otros apuntadores.
- Un uso común de apuntadores de función es en los sistemas conocidos como manejados por menú.

Terminología

como sumar un apuntador y un entero
operador de dirección (&)
arreglo de apuntadores
arreglo de cadenas
llamada por referencia
llamada por valor
apuntador de carácter
const
apuntador constante
apuntador constante a datos constantes
apuntador constante a datos no constantes
decrementar un apuntador
desreferenciar un apuntador
operador de desreferenciar (*)
referencia directa a una variable
asignación dinámica de memoria
apuntador de función
incrementar un apuntador
posposición indefinida
indirección
operador de indirección (*)
referencia indirecta a una variable
cómo inicializar apuntadores
lista enlazada
apuntador no constante a datos constantes
apuntador no constante a datos no constantes
apuntador **NULL**
desplazamiento
apuntador
aritmética de apuntador
asignación de apuntador
comparación de apuntador
expresión de apuntador
indexación de apuntador
notación apuntador/desplazamiento
subscripción de apuntador
apuntador a una función
apuntador a **void** (**void ***)
tipos de apuntador
principio de mínimo privilegio
llamada por referencia simulada
sizeof
arreglo de cadenas
resta de un entero de un apuntador
resta de dos apuntadores
refinación descendente paso a paso
void * (apuntador a **void**)

Errores comunes de programación

7.1 El operador de indirección * no se distribuye a todos los nombres de variables de una declaración. Cada apuntador debe de ser declarado con el * prefijo al nombre.

7.2 Desreferenciar un apuntador que no haya sido correctamente inicializado, o que no haya sido asignado para apuntar a una posición específica en memoria. Esto podría causar un error fatal en tiempo de ejecución, o podría modificar de forma accidental datos importantes y permitir que el programa se ejecute hasta su terminación proporcionando resultados incorrectos.

7.3 No desreferenciar un apuntador, cuando es necesario hacerlo para obtener el valor hacia el cual el apuntador señala.

7.4 No estar consciente que una función está esperando apuntadores como argumentos en llamadas por referencia, y está pasando argumentos en llamadas por valor. Algunos compiladores toman los valores, suponiendo que son apuntadores y desreferencian los valores como apuntadores. En tiempo de ejecución, a menudo se generan violaciones de acceso a la memoria o fallas de segmentación. Otros compiladores detectan falta de coincidencia en tipo entre argumentos y parámetros, que generan mensajes de error.

7.5 Utilizar aritmética de apuntadores en un apuntador que no se refiere a un arreglo de valores.

7.6 Restar o comparar dos apuntadores que no se refieren al mismo arreglo.

7.7 Salirse de cualquiera de los dos extremos de un arreglo al utilizar aritmética de apuntadores.

7.8 Se generará un error de sintaxis asignar un apuntador de un tipo a un apuntador de otro tipo, si ninguno de los dos es del tipo `void *`.

7.9 Desreferenciar un apuntador `void *`.

7.10 Es un error de sintaxis intentar modificar un nombre de arreglo con aritmética de apuntador.

Prácticas sanas de programación

7.1 En un nombre de variable de apuntador incluya las letras `ptr` para que quede claro que estas variables son apuntadores y deben ser manejadas con propiedad.

7.2 Inicialice los apuntadores para evitar resultados inesperados.

7.3 Utilice llamadas por valor para pasar argumentos a una función, a menos de que en forma explícita el llamador requiera que la función llamada modifique el valor de la variable del argumento en el entorno de llamador. Esto es otro ejemplo del principio del mínimo privilegio. Algunas personas prefieren llamadas por referencia por razones de rendimiento ya que el copiado de valores de término medio se le elude.

7.4 Antes de utilizar una función, verifique el prototipo de función correspondiente a esta función, a fin de determinar si la función es capaz de modificar los valores que se le pasan.

7.5 Al manipular arreglos utilice notación de arreglos en vez de notación de apuntadores. Aunque el programa pudiera tomar más tiempo para su compilación, probablemente será mucho más claro.

Sugerencias de rendimiento

7.1 Pase grandes objetos como son estructuras utilizando apuntadores a datos constantes para obtener los beneficios de rendimiento de llamadas por referencia y la seguridad de llamadas por valor.

7.2 Pasar el tamaño de un arreglo a una función ocupa tiempo y espacio de pilas adicional, porque debe ejecutarse una copia del tamaño para ser pasada a la función. Las variables globales, sin embargo, no requieren de tiempo adicional o de espacio, porque son accesibles de forma directa por cualquier función.

7.3 Durante la compilación la notación de subíndices de arreglo se convierte a notación de apuntador, por lo que escribir expresiones de subíndices de arreglo con notación de apuntadores, puede ahorrar tiempo de compilación.

7.4 Algunas veces un algoritmo que aparece de forma "natural" puede contener problemas sutiles de rendimiento, como es la posposición indefinida. Busque algoritmos que eviten la posposición indefinida.

Sugerencias de portabilidad

7.1 A pesar de que en ANSI C `const` está bien definido, algunos sistemas no lo tienen incorporado.

7.2 El número de bytes utilizados para almacenar un tipo particular de datos, pudiera variar de sistema a sistema. Al escribir programas que dependan de tamaños de tipos de datos, y que se ejecutarán en diversos sistemas de computación, utilice `sizeof` para determinar el número de bytes utilizados para almacenar los tipos de datos.

7.3 La mayor parte de las computadoras de hoy día tienen enteros de 2 o de 4 bytes. Algunas de las máquinas más modernas utilizan enteros de 8 bytes. Dado que los resultados de la aritmética de apuntadores depende del tamaño de los objetos a los cuales el apuntador señala, la aritmética de los apuntadores depende de la máquina.

Observaciones de ingeniería de software

7.1 El calificador `const` puede ser utilizado para forzar el principio del mínimo privilegio. La utilización del principio de mínimo privilegio para diseñar apropiadamente el software reduce en forma importante el tiempo de depuración y efectos inadecuados colaterales, y hace un programa más fácil de modificar y mantener.

7.2 Si un valor no se modifica (o no debería modificarse) en el cuerpo de una función al cual es pasado, el valor deberá declararse `const`, para asegurarse que no se modifica accidentalmente.

7.3 En una función llamadora sólo un valor puede ser alterado cuando se utiliza llamada por valor. Este valor debe ser asignado a partir del valor de regreso de la función. Para modificar varios valores en una función llamadora, debe utilizarse llamada por referencia.

7.4 Colocar prototipos de función en las definiciones de otras funciones obliga al principio del mínimo privilegio al restringir las llamadas correctas de función sólo a aquellas funciones en las cuales dichos prototipos aparecen.

7.5 Al pasar un arreglo a una función, pase también su tamaño. Esto ayuda a generalizar la función. Las funciones generales son a menudo reutilizables.

7.6 Las variables globales violan el principio del mínimo privilegio y son un ejemplo de ingeniería de software pobre.

Ejercicios de autoevaluación

7.1 Conteste cada uno de los siguientes:

a) Un apuntador es una variable que contiene como su valor la _____ de otra variable.

b) Los tres valores que se pueden utilizar para inicializar un apuntador son _____, _____, o una _____.

c) El único entero que puede ser asignado a un apuntador es _____.

7.2 Indique si lo siguiente es cierto o falso. Si la respuesta es falso, explique por qué.

a) El operador de dirección `&` puede sólo aplicarse a constantes, a expresiones y a variables declaradas con la clase de almacenamiento `register`.

b) Un apuntador que se declara ser `void` puede ser desreferenciado.

c) Los apuntadores de diferentes tipos no pueden ser asignados uno al otro, sin una operación de conversión.

7.3 Conteste cada una de las siguientes. Suponga que números de una precisión de punto flotante están almacenados en 4 bytes, y que la dirección inicial del arreglo está en memoria en la posición 1002500. Cada parte del ejercicio deberá de utilizar los resultados de las partes anteriores, donde sea apropiado.

a) Declare un arreglo del tipo `float`, llamado `numbers` con 10 elementos, e inicialice los elementos a los valores `0.0, 1.1, 2.2, ....9.9`. Suponga que la constante simbólica `SIZE` ha sido definida como `10`.

b) Declare un apuntador `nPtr`, que apunte a un objeto de tipo `float`.

c) Imprima los elementos del arreglo **numbers**, utilizando notación de subíndice de arreglo. Utilice una estructura **for**, y suponga que se ha declarado la variable de control entera **i**. Imprima cada número con una precisión de una posición a la derecha del punto decimal.
d) Proporcione dos enunciados por separado, que asignen la dirección inicial del arreglo **numbers** a la variable de apuntador **nPtr**.
e) Imprima los elementos del arreglo **numbers**, utilizando la notación apuntador/desplazamiento, con el apuntador **nPtr**.
f) Imprima los elementos del arreglo **numbers**, utilizando notación apuntador/desplazamiento con el nombre del arreglo como el apuntador.
g) Imprima los elementos del arreglo **numbers** mediante subíndices del apuntador **nPtr**.
h) Refiérase al elemento 4 del arreglo **numbers**, utilizando la notación de subíndice de arreglo, la notación de apuntador/desplazamiento y utilizando como el apuntador el nombre del arreglo, la notación de subíndice de apuntador con **nPtr**, y la notación de apuntador/desplazamiento con **nPtr**.
i) Suponiendo que **nPtr** apunta al principio del arreglo **numbers**, ¿cuál es la dirección referenciada por **nPtr + 8**? ¿Cuál es el valor almacenado en esa posición?
j) Suponiendo que **nPtr** apunta a **numbers[5]**, qué dirección es referenciada por **nPtr -= 4**. ¿Cuál es el valor almacenado en esta posición?

7.4 Para cada uno de los siguientes, escriba un enunciado que ejecute la tarea indicada. Suponga que se han declarado las variables de punto flotante **number1** y **number2**, y que **number1** ha sido inicializado a **7.3**.

a) Declare la variable **fPtr** que sea un apuntador a un objeto del tipo **float**.
b) Asigne la dirección de la variable **number1** a una variable de apuntador **fPtr**.
c) Imprima el valor del objeto señalado hacia por **fPtr**.
d) Asigne el valor del objeto al que se señala con **fPtr** a la variable **number2**.
e) Imprima el valor de **number2**.
f) Imprima la dirección de **number1**. Utilice el especificador de conversión **%p**.
g) Imprima la dirección almacenada en **fPtr**. Utilice el especificador de conversión **%p**. ¿Es el valor impreso el mismo al de la dirección de **number1**?

7.5 Haga cada uno de lo siguiente.

a) Escriba el encabezado de función de una función llamada **exchange** que toma como parámetros dos apuntadores a números de punto flotante **x** e **y**, y no regresa un valor.
b) Escriba el prototipo de función para la función en la parte (a).
c) Escriba el encabezado de función para la función llamada **evaluate**, que regresa un entero y que toma como parámetros el entero **x** y un apuntador a la función **poly**. La función **poly** toma un parámetro entero y regresa un entero.
d) Escriba el prototipo de función para la función de la parte (c).

7.6 Encuentre el error en cada uno de los segmentos de programas siguientes. Suponga

```
int *zPtr; /* zPtr will reference array z */
int *aPtr = NULL;
void *sPtr = NULL;
int number, i;
int z[5] = {1, 2, 3, 4, 5}

sPtr = z;
```

a) `++zptr;`
b)
```
/* use pointer to get first value of array */
number = zPtr;
```
c)
```
/* assign array element 2 (the value 3) to number */
number = *zPtr[2];
```


d)
```
/* print entire array z */
for (i = 0; i  5; i++)
printf("%d ", zPtr[i]);
```
e)
```
/* assign the value pointed to by sPtr number */
number = *sPtr;
```
f) `++z;`

Respuestas a los ejercicios de autoevaluación

7.1 a) dirección. b) `0`, `NULL`, una dirección. c) `0`

7.2 a) Falso. El operador de dirección pude ser sólo aplicado a variables y no puede aplicarse a variables declaradas con la clase de almacenamiento **register**.
b) Falso. Un apuntador **void** no puede ser desreferenciado porque no existe manera de saber con exactitud cuantas bytes de memoria deberán ser desreferenciadas.
c) Falso. Apuntadores del tipo **void** pueden ser asignados como apuntadores de otros tipos, y apuntadores del tipo **void** pueden ser asignados a apuntadores de otros tipos.

7.3 a)
```
float numbers[SIZE] = {0.0, 1.1, 2.2, 3.3, 4.4, 5.5,
                       6.6, 7.7, 8.8, 9.9};
```
b) `float *nPtr;`
c)
```
for (i = 0; i <= 0; i  SIZE - 1; i++)
 printf("%.1f ", numbers[i]);
```
d)
```
nPtr = numbers;
nPtr = &numbers[0];
```
e)
```
for (i = 0; i <= SIZE - 1; i++)
  printf("%.1f ", *(nPtr + i));
```
f)
```
for (i = 0; i <= SIZE - 1; i++)
printf("%.1f ", *(numbers + i));
```
g)
```
for (i = 0; i <= SIZE - 1; i++)
 printf("%.1f ", nPtr[i]);
```
h)
```
numbers[4]
*(numbers + 4)
nPtr[4]
*(nPtr + 4)
```
i) La dirección es `1002500 + 8 * 4 = 1002532`. El valor es `8.8`.
j) La dirección de `numbers[5]` es `1002500 + 5 * 4 = 1002520`.
La dirección de `nPtr -= 4` es `1002520 - 4 * 4 = 1002504`.
El valor en dicha posición es `1.1`.

7.4 a)
```
float *fPtr;
```
b)
```
fPtr = &number1;
```
c)
```
printf("The value of *fPtr is %f\n", *fPtr);
```
d)
```
number2 = *fPtr;
```
e)
```
printf("The value of number2 is %f\n", number2);
```
f)
```
printf("The address of number1 is %p\n", &number1);
```
g)
```
printf("The address stored in fptr is %p\n", fPtr);
```
Sí, el valor es el mismo.

7.5 a)
```
void exchange (float*x, float*y)
```
b)
```
voyd exchange (float *,float *)
```
c)
```
int evaluate(int x, int (*poly)(int))
```
d)
```
int evaluate(int, int (*)(int));
```

7.6 a) Error: `zPtr` no ha sido inicializado.
Corrección: inicialice `zPtr` con `zPtr = z;`
b) Error: El apuntador no está desreferenciado.
Corrección: modifique el enunciado a `number = *zPtr;`
c) Error: `zPtr[2]` no es un apuntador y no deberá ser desreferenciado.
Corrección: modifique `*zPtr[2]` a `zPtr[2]`.
d) Error: referirse a un elemento de arreglo exterior a los límites del mismo, utilizando subíndices de apuntador.
Corrección: cambie el valor final de la variable de control en la estructura `for` a `4`.
e) Error: Desreferenciando un apuntador `void`.
Corrección: a fin de desreferenciar el apuntador, primero debe de ser convertido a un apuntador entero. Modifique el enunciado anterior a `number = (int *)sPtr;`
f) Error: Tratar de modificar el nombre de un arreglo utilizando aritmética de apuntador.
Corrección: utilice una variable de apuntador, en vez del nombre del arreglo, para llevar a cabo aritmética de apuntador, o suscriba el nombre del arreglo para referirse a un elemento específico.

Ejercicios

7.7 Conteste cada una de las siguientes:
a) El operador _____ regresa la posición en memoria donde está almacenado su operando.
b) El operador _____ regresa el valor del objeto hacia el cual apunta su operando.
c) Para simular llamada por referencia al pasar una variable no de arreglo a una función, es necesario pasar el _____ de la variable a la función.

7.8 Indique si lo siguiente es verdadero o falso. Si es falso, explique por qué.
a) Dos apuntadores que señalen a diferentes arreglos no pueden ser comparados en forma significativa.
b) Dado que el nombre de un arreglo es un apuntador para el primer elemento del mismo, los nombres de los arreglos pueden ser manipulados de la misma forma que los apuntadores.

7.9 Conteste cada una de los siguientes. Suponga que enteros unsigned están almacenados en 2 bytes, y que las dirección inicial del arreglo es en la posición 1002500 en memoria.
a) Declare un arreglo del tipo `unsigned int` llamado `values` con 5 elementos, e inicie los elementos a los enteros pares del 2 al 10. Suponga la constante simbólica `SIZE` definida como `5`.
b) Declare un apuntador `vPtr` que señale a un objeto del tipo `unsigned int`.
c) Imprima los elementos del arreglo `values` utilizando notación de subíndices de arreglo. Utilice una estructura `for` y suponga que ha sido declarada una variable de control entera `i`.
d) Proporcione dos enunciados separados que asignen la dirección inicial del arreglo `values` a la variable de apuntador `vPtr`.
e) Imprima los elementos del arreglo `values`, utilizando notación apuntador/desplazamiento.
f) Imprima los elementos del arreglo `values`, utilizando notación apuntador/desplazamiento con el nombre del arreglo como el apuntador.
g) Imprima los elementos del arreglo `values`, mediante subíndices del apuntador al arreglo.
h) Refiérase al elemento 5 del arreglo `values` utilizando notación de subíndice de arreglo, notación de apuntador/desplazamiento con el nombre del arreglo como el apuntador, notación de subíndice de apuntador, y notación de apuntador/desplazamiento.
i) ¿Cuál es la dirección referenciada por `vPtr + 3`? ¿Cuál es el valor almacenado en esa posición?
j) Suponiendo que `vPtr` apunta a `values[4]`, ¿cuál es la dirección referenciada por `vPtr -= 4`. Cuál es el valor almacenado en dicha posición?

7.10 Para cada uno de los siguientes, escriba un solo enunciado que ejecute la tarea indicada. Suponga que han sido declaradas las variables enteras long `value1` y `value2`, y que `value1` ha sido inicializado a `200000`.
a) Declare la variable `lPtr` que sea un apuntador a un objeto del tipo `long`.
b) Asigne la dirección de la variable `value1` a la variable de apuntador `lPtr`.
c) Imprima el valor del objeto señalado por `lPtr`.
d) Asigne el valor del objeto señalado por `lPtr` a la variable `value2`.
e) Imprima el valor de `value2`.
f) Imprima la dirección de `value1`.
g) Imprima la dirección almacenada en `lPtr`. ¿Es el valor impreso el mismo que la dirección de `value1`?

7.11 Lleve a cabo cada uno de lo siguiente.
a) Escriba el encabezado de función para la función `zero` que toma un parámetro de arreglo entero long `bigIntegers` y no regresa un valor.
b) Escriba la función prototipo para la función de la parte (a).
c) Escriba el encabezado de función de la función `add1AndSum` que toma un parámetro de arreglo entero `oneTooSmall` y regresa un entero.
d) Escriba el prototipo de función para la función descrita en la parte (c).

Nota: los ejercicios 7.12 al 7.15 tienen un cierto elemento de reto. Una vez que haya llevado a cabo estos problemas, deberá estar en condiciones de resolver con facilidad los juegos de naipes más populares.

7.12 Modifique el programa de la figura 7.24, de tal forma que la función de distribución de naipes reparta una mano de póker de cinco naipes. A continuación escriba las funciones adicionales siguientes:
a) Determine si la mano contiene un par.
b) Determine si la mano contiene dos pares.
c) Determine si la mano contiene tres de un tipo (es decir, por ejemplo, tres jacks).
d) Determine si la mano contiene cuatro de un tipo (por ejemplo, cuatro ases).
e) Determine si la mano contiene un color (es decir, todos los cinco naipes del mismo palo).
f) Determine si la mano contiene una flor imperial (es decir, cinco naipes de valores nominales consecutivos).

7.13 Utilice las funciones desarrolladas en el ejercicio 7.12, para escribir un programa que distribuya dos manos de póker de 5 naipes, evalúe cada una de las manos, y determine cuál es la mejor mano.

7.14 Modifique el programa desarrollado en el ejercicio 7.13, de tal forma que pueda simular al tallador. La mano de 5 naipes del tallador se distribuye "tapada", de tal manera que el jugador no puede verla. El programa deberá entonces evaluar la mano del tallador, y basado en la calidad de la misma, el tallador deberá de solicitar uno, dos o tres naipes adicionales para remplazar el número correspondiente de naipes innecesarios de la mano original. El programa deberá entonces reevaluar la mano del tallador. (*Advertencia:* ¡Este es un problema difícil!)

7.15 Modifique el programa desarrollado en el ejercicio 7.14, de tal forma de que pueda manejar de forma automática la mano del tallador, pero el jugador pueda decidir qué naipes de su mano remplazar. El programa deberá entonces evaluar ambas manos y determinar quien gana. Ahora utilice este nuevo programa para jugar 20 juegos contra la computadora. ¿Quién gana más juegos, usted o la computadora? Haga que uno de sus amigos juegue 20 juegos contra la computadora. ¿Quién gana más juegos? Basado en los resultados de estos juegos, efectúe modificaciones apropiadas para refinar su programa de jugar al póker (esto también es un problema difícil). Juegue otros 20 juegos. ¿Su programa modificado juega un mejor juego?.

7.16 En el programa de barajar y distribuir naipes de la figura 7.24, intencionalmente utilizamos un algoritmo de barajar ineficiente, que introducía la posibilidad de posposición indefinida. En este problema, usted creará un algoritmo de barajar de alto rendimiento, que elimine la posposición indefinida.

Modifique el programa de la figura 7.24 como sigue. Empiece inicializando el arreglo `deck` como se muestra en la figura 7.29. Modifique la función `shuffle` para ciclar, renglón por renglón y columna por

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	14	15	16	17	18	19	20	21	22	23	24	25	26
2	27	28	29	30	31	32	33	34	35	36	37	38	39
3	40	41	42	43	44	45	46	47	48	49	50	51	52

Fig. 7.29 Arreglo **deck**, no barajado.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	19	40	27	25	36	46	10	34	35	41	18	2	44
1	13	28	14	16	21	30	8	11	31	17	24	7	1
2	12	33	15	42	43	23	45	3	29	32	4	47	26
3	50	38	52	39	48	51	9	5	37	49	22	6	20

Fig. 7.30 Arreglo **deck** de muestra, barajado.

columna, a través del arreglo, tocando cada elemento una vez. Cada elemento deberá ser intercambiado con un elemento elegido al azar del arreglo.

Imprima el arreglo resultante para determinar si el mazo ha sido barajado de manera satisfactoria (como en la figura 7.30, por ejemplo). Posiblemente desee que su programa llame varias veces a la función **shuffle**, para asegurarse que se ha barajado correctamente.

Note que, a pesar que el enfoque en este problema mejora el algoritmo de barajar, el algoritmo de repartir todavía requiere buscar en el arreglo **deck** el naipe 1, a continuación el naipe 2, y en seguida el naipe 3, y así en lo sucesivo. Y lo que es aún peor, aun después de que el algoritmo de distribución localiza y reparte el naipe, el algoritmo sigue buscando a través del resto del mazo. Modifique el programa de la figura 7.24 de tal forma de que una vez que se haya distribuido un naipe, no se hagan intentos posteriores para hacer coincidir este número de naipe, y el programa continúe de inmediato con la distribución del naipe siguiente. En el capítulo 10 desarrollamos un algoritmo de reparto de naipes, que requiere de una operación por cada naipe.

7.17 (*Simulación: La liebre y la tortuga*). En este problema recreará uno de los verdaderos grandes momentos de la historia, es decir, la carrera clásica entre la liebre y la tortuga. Utilizará una generación de números aleatorios para desarrollar una simulación de este evento memorable.

Nuestros contendientes empiezan la carrera en el "cuadro 1" de 70 cuadros. Cada cuadro representa una posición posible a lo largo de la carrera. La línea de meta está en el cuadro 70. El primer contendiente que llegue o que pase el cuadro 70 gana una cubeta de zanahorias y lechugas frescas. El desarrollo de la pista transcurre sobre la ladera de una montaña resbalosa, por lo que ocasionalmente los corredores pierden terreno.

Existe un reloj que hace tic una vez por segundo. Con cada tic del reloj, su programa deberá ajustar la posición de los animales, de acuerdo con las reglas de la página siguiente:

Utilice variables para llevar control de las posiciones de los animales (es decir, los números de posición son del 1 al 70). Inicie cada animal en la posición 1 (es decir, en la "línea de arranque"). Si un animal resbala antes del cuadro 1, regrese el animal de vuelta al cuadro 1.

Genere los porcentajes de la tabla anterior mediante la producción de un entero al azar, i, en el rango $1 \le i \le 10$. Para la tortuga, ejecute un "paso rápido" cuando $1 \le i \le 5$, un "resbalón" cuando $6 \le i \le 7$, o un "paso lento" cuando $8 \le i \le 10$. Utilice una técnica similar para mover a la liebre.

Empiece la carrera imprimiendo

```
BANG !!!!!
AND THEY'RE OFF !!!!!
```


Animal	Tipo de movimiento	Porcentaje del tiempo	Movimiento real
Tortuga	Paso rápido	50%	3 cuadros a la derecha
	Resbalón	20%	6 cuadros a la izquierda
	Paso lento	30%	1 cuadro a la derecha
Liebre	Dormido	20%	Ningún movimiento
	Salto grande	20%	9 cuadros a la derecha
	Resbalón grande	10%	12 cuadros a la izquierda
	Salto pequeño	30%	1 cuadro a la derecha
	Pequeño resbalón	20%	2 cuadros a la izquierda

Entonces, por cada pulso del reloj (es decir, para cada repetición del ciclo) imprima una línea de 70 posiciones, que muestre la letra `T` en la posición de la tortuga, y la letra `H` en la posición de la liebre. Ocasionalmente, ambos contendientes coincidirán en el mismo cuadro. En esta condición, la tortuga muerde a la liebre y su programa deberá de imprimir `OUCH!!!` empezando en esta posición. Todas las posiciones de impresión, distintas a la de `T`, la `H`, o la `OUCH!!!` (en caso de un empate) deberán estar vacías.

Después de que se ha impreso cada línea, pruebe si ambos animales han alcanzado o han pasado el cuadro 70. Si es así, entonces imprima el ganador y dé por terminada la simulación. Si la tortuga gana, imprima `TORTOISE WINS!!! YAY!!!`. Si la liebre gana, imprima `Hare wins. Yuch.` Si ambos animales ganan en el mismo pulso del reloj, quizás desee favorecer a la tortuga (el "más débil"), o quizás desee imprimir `It's a tie`. Si ninguno de los animales gana, otra vez vuelva a ejecutar el ciclo, para simular el siguiente pulso del reloj. Cuando esté listo para ejecutar su programa, reúna un grupo de observadores para presenciar la carrera. ¡Se asombrará del interés que su auditorio mostrará!

Sección especial: cómo construir su propia computadora

En los siguientes varios problemas, nos desviamos temporalmente del mundo de la programación en lenguajes de alto nivel. Le "quitamos la cubierta" a una computadora y estudiamos su estructura interna. Presentamos la programación en lenguaje máquina y escribimos varios programas en lenguaje máquina. Para que esto resulte una experiencia en especial valiosa, luego construimos una computadora (mediante la técnica de la *simulación* basada en software) ¡en la cual usted podrá ejecutar sus programas en lenguaje de máquina!

7.18 (*Programación en lenguaje de máquina*). Procedamos a crear una computadora, que llamaremos Simpletron. Como su nombre implica, es una máquina sencilla, pero como veremos pronto, es también poderosa. Simpletron ejecuta programas, escritos en el único lenguaje que comprende en forma directa, esto es, en el lenguaje de máquina Simpletron, o bien, abreviando, en LMS.

Simpletron contiene un *acumulador* —un "registro especial" en el cual se coloca la información ante Simpletron, mismo que utiliza esta información en cálculos o la examina de varias formas. Toda la información en Simpletron se maneja en términos de *palabras*. Una palabra es un número decimal de cuatro dígitos signados, como `+3364`, `-1293`, `+0007`, `-0001`, etcétera. Simpletron está equipado con una memoria de 100 palabras, y estas palabras están referenciadas por sus números de posición `00`, `01`, ..., `99`.

Antes de ejecutar un programa LMS, debemos de *cargar*, es decir colocar, el programa en memoria. La primera instrucción (o enunciado) de cualquier programa LMS siempre se colocará en la posición `00`.

Cada instrucción escrita en LMS ocupa una palabra de la memoria Simpletron (y, por lo tanto, las instrucciones resultan números decimales de cuatro dígitos signados). Supondremos que el signo de una instrucción LMS será siempre más, pero el signo de una palabra de datos podrá ser o más o menos. Cada posición en la memoria de Simpletron pudiera contener instrucción, o un valor de datos usado por un programa, o un área no utilizada (y, por lo tanto, no definida) de memoria. Los dos primeros dígitos de cada instrucción LMS son el *código de operación*, que define la operación a ejecutarse. Los códigos de operación LMS se resumen en la figura 7.31.

Los últimos dos dígitos de una instrucción LMS son el *operando*, es decir la dirección de la posición de memoria que contiene la palabra a la cual se aplica la operación. Ahora consideremos varios programas simples LMS.

Código de operación	Significado
Operaciones de entrada/salida:	
`#define READ 10`	Lee una palabra de la terminal y la coloca en una posición específica en memoria.
`#define WRITE 11`	Escribe una palabra desde una posición específica en memoria a la terminal.
Operaciones de cargar/almacenar:	
`#define LOAD 20`	Carga una palabra de una posición específica en memoria al acumulador.
`#define STORE 21`	Almacena una palabra del acumulador a una posición específica en memoria.
Operaciones aritméticas:	
`#define ADD 30`	Añade una palabra de una posición específica en memoria a la palabra en el acumulador (deja el resultado en el acumulador).
`#define SUBTRACT 31`	Sustrae una palabra de una posición específica en memoria de la palabra existente en el acumulador (deja el resultado en el acumulador).
`#define DIVIDE 32`	Divide la palabra existente en el acumulador entre una palabra de una posición específica en la memoria (deja el resultado en el acumulador).
`#define MULTIPLY 33`	Multiplica una palabra de una localización específica en la memoria por la palabra en el acumulador (deja el resultado en el acumulador).
Operaciones de transferencia de control:	
`#define BRANCH 40`	Se desvía a una posición específica en memoria.
`#define BRANCHNEG 41`	Se desvía a una posición específica en memoria si el acumulador es negativo.
`#define BRANCHZERO 42`	Se desvía a una posición específica en memoria si el acumulador es cero.
`#define HALT 43`	Se detiene, es decir, el programa ha terminado su tarea.

Fig. 7.31 Códigos de operación del lenguaje de máquina Simpletron (LMS).

Ejemplo 1 Posición	Número	Instrucción
00	+1007	(Read A)
01	+1008	(Read B)
02	+2007	(Load A)
03	+3008	(Add B)
04	+2109	(Store C)
05	+1109	(Write C)
06	+4300	(Halt)
07	+0000	(Variable A)
08	+0000	(Variable B)
09	+0000	(Result C)

Este programa LMS lee dos números del teclado y calcula e imprime su suma. La instrucción **+1007** lee el primer número del teclado y lo coloca en la posición **07** (que ha sido inicializada a cero). Entonces **+1008** lee el siguiente número a la posición **08**. La instrucción *load*, **+2007**, coloca el primer número en el acumulador, y la instrucción *add* **+3008**, suma el segundo número al número en el acumulador. *Todas las instrucciones aritméticas LMS dejan sus resultados en el acumulador.* La instrucción *store* **+2109**, coloca el resultado de regreso en la posición de memoria **09** a partir de la cual la instrucción *write* **+1109**, toma el número y lo imprime (en forma de un número decimal de cuatro dígitos signado). La instrucción *halt* **+4300**, da por terminada la ejecución.

Ejemplo 2 Posición	Número	Instrucción
00	+1009	(Read A)
01	+1010	(Read B)
02	+2009	(Load A)
03	+3110	(Subtract B)
04	+4107	(Branch negative to 07)
05	+1109	(Write A)
06	+4300	(Halt)
07	+1110	(Write B)
08	+4300	(Halt)
09	+0000	(Variable A)
10	+0000	(Variable B)

Este programa LMS lee dos números del teclado, y determina e imprime el valor más grande. Note el uso de la instrucción **+4107** como una transferencia condicional de control, de la misma forma que un enunciado **if** de C. Ahora escriba programas LMS que lleven a cabo cada una de las tareas siguientes.

a) Utilice un ciclo controlado por centinela, para leer 10 números positivos y calcular e imprimir su suma.
b) Use un ciclo controlado por contador para leer 7 números, algunos positivos y algunos negativos, y calcular e imprimir su promedio.
c) Lea una serie de números y determine e imprima el número más grande. El primer número leído indicará cuántos números deberán de procesarse.

7.19 (*Un simulador de computadora*). Al principio pudiera resultar extravagante, pero en este problema construirá su propia computadora. No, no tendrá que soldar componentes. Más bien, utilizará la técnica poderosa de la *simulación basada en software* para crear un *modelo en software* de Simpletron. No quedará decepcionado. Su simuladora Simpletron convertirá la computadora que está utilizando en una Simpletron, y de hecho será capaz de ejecutar, probar y depurar los programas LMS que escribió en el ejercicio 7.18.

Cuando haga funcionar su simuladora Simpletron, deberá empezar imprimiendo:

```
*** Welcome to Simpletron! ***

*** Please enter your program one instruction ***
*** (or data word) at a time. I will type the ***
*** location number and a question mark (?).  ***
*** You then type the word for that location. ***
*** Type the sentinel -99999 to stop entering ***
*** your program. ***
```

Simule la memoria de Simpletron con un arreglo **memory** de un subíndice, que tenga 100 elementos. Ahora suponga que el simulador está funcionando, y examinemos el diálogo, conforme introducimos el programa del ejemplo 2 del ejercicio 7.18:

```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -99999

*** Program loading completed ***
*** Program execution begins  ***
```

Ahora el programa LMS ha sido colocado (es decir cargado) en el arreglo **memory**. A continuación Simpletron ejecutará su programa LMS. La ejecución empieza con la instrucción en la posición **00**, y, al igual que C, continuará en forma secuencial, a menos de que sea dirigido a otra parte del programa, mediante una transferencia de control.

Utilice la variable **accumulator**, para representar el registro de acumulador. Utilice la variable **instructionCounter**, para llevar control de la posición en memoria que contiene la instrucción bajo ejecución. Utilice la variable **operationCode**, para indicar la operación en este momento en ejecución, es decir los dos dígitos a la izquierda de la palabra de instrucción. Utilice la variable **operand**, para indicar la posición en memoria en la cual opera la instrucción actual. Entonces, **operand** son los dígitos más a la derecha de la instrucción en ejecución en ese momento. No ejecute las instrucciones desde la memoria directa. Más bien, transfiera la siguiente instrucción a ejecutarse de la memoria a una variable, llamada

instructionRegister. A continuación "tome" los dos dígitos a la izquierda y colóquelos en **operationCode**, y luego "tome" los dos dígitos a la derecha y colóquelos en **operand**.

Cuando Simpletron empieza a operar, los registros especiales son inicializados como sigue:

```
accumulator          +0000
instructionCounter     00
instructionRegister  +0000
operationCode          00
operand                00
```

Ahora "recorramos" la ejecución de la primera instrucción LMS, **+1009** en la posición de memoria **00**. Esto se conoce como un *ciclo de ejecución de instrucción*.

El **instructionCounter** nos indica la posición de la siguiente instrucción a ejecutarse. Tomamos el contenido de esa posición **memory**, utilizando el enunciado en C

```
istructionRegister = memory[instructionCounter];
```

El código de operación y el operando se extraen del registro de instrucción mediante los enunciados

```
operationCode = instructionRegister / 100;
operand = instructionRegister % 100;
```

Ahora Simpletron debe determinar que el código de operación es de hecho un *read* (a diferencia de un *write*, un *load*, etcétera). Con un **switch** se diferencia entre las doce operaciones de LMS.

En la estructura **switch**, el comportamiento de las varias instrucciones LMS se simula como sigue (dejamos las demás al lector):

read: `scanf("%d", &memory[operand]);`
load: `accumulator = memory[operand];`
add: `accumulator += memory[operand];`
Varias instrucciones de ramificación: analizaremos éstas pronto.
halt:: Esta instrucción imprime el mensaje
`*** Simpletron execution terminated ***`

y a continuación el nombre y el contenido de cada registro, así como el contenido completo de la memoria. Una impresión como ésta, con frecuencia se conoce como un *vaciado de computadora* (y, no, un vaciado de computadora no es un lugar donde va la computadora vieja). Para auxiliarle a programar su función de vaciado, en la figura 7.32 se muestra un formato de muestra de vaciado. Note que después de ejecutar un programa Simpletron un vaciado mostraría los valores reales de las instrucciones y los valores de datos, en el momento en que la ejecución terminó.

Sigamos con la ejecución de la primera instrucción de su programa, es decir, la **+1009** en la posición **00**. Como hemos indicado, el enunciado **switch** simula lo anterior, ejecutando en C el enunciado

```
scanf("%d", &memory[operand]);
```

Antes de que se ejecute **scanf**, deberá aparecer un signo de interrogación (**?**) en pantalla, para solicitarle entradas al usuario. Simpletron espera a que el usuario escriba un valor y a continuación presione la *tecla Return*. El valor a continuación se lee a la posición **09**.

Llegado a ese punto, la simulación de la primera instrucción ha sido terminada. Todo lo que se requiere es preparar Simpletron para que ejecute la siguiente instrucción. Dado que la instrucción que se acaba de terminar no fue una transferencia de control, simplemente necesitamos incrementar el registro de contador de instrucciones, como sigue:

```
++instructionCounter;
```

```
REGISTERS:
accumulator              +0000
instructionCounter         00
instructionRegister      +0000
operationCode              00
operand                    00

MEMORY:
      0     1     2     3     4     5     6     7     8     9
 0 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
10 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
20 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
30 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
30 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
40 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
50 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
60 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
70 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
80 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
90 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
```

Fig. 7.32 Una muestra de vaciado.

Esto completa la ejecución simulada de la primera instrucción. Empieza otra vez todo el proceso, (es decir, el ciclo de ejecución de instrucción), con la obtención de la siguiente instrucción a ejecutarse.

Ahora veamos como se simulan las instrucciones de ramificación —las transferencias de control. Todo lo que necesitamos hacer es ajustar en forma apropiada el valor del contador de instrucciones. Por lo tanto, la instrucción de ramificación incondicional (**40**) se simula dentro de **switch** como

```
instructionCounter = operand;
```

La instrucción condicional "ramifique si el acumulador es cero" se simula como

```
if (accumulator == 0)
   instructionCounter = operand;
```

Llegado a este punto deberá poner en marcha su simulador Simpletron y ejecutar cada uno de los programas LMS escritos en el ejercicio 7.18. Puede embellecer a LMS con características adicionales e incluirlas en su simulador.

Su simulador deberá verificar varios tipos de errores. Durante la fase de carga de programa, por ejemplo, cada número que escriba el usuario en **memory** del Simpletron debe estar en el rango **-9999** hasta **+9999**. Su simulador deberá utilizar un ciclo **while** y probar que cada uno de los números escritos estén en este rango y, de lo contrario, insistir en solicitarle al usuario que vuelva a escribir el número, hasta que éste lo escriba correcto.

Durante la fase de ejecución, su simulador deberá de vigilar varios errores serios, como intentos de división entre cero, intentos de ejecución inválida de códigos de operaciones, desbordamiento del acumulador (es decir, operaciones aritméticas que resulten en valores mayores que **+9999** o menores **-9999**) y otras similares. Estos errores serios son conocidos como *errores fatales*. Cuando se detecta un error fatal, su simulador deberá imprimir un mensaje de error como:

```
*** Attempt to divide by zero ***
*** Simpletron execution abnormally terminated ***
```

y deberá imprimir un vaciado completo de computadora, en el formato ya analizado. Esto ayudará al usuario a localizar el error dentro del programa.

7.20 Modifique el programa de barajar y distribuir naipes de la figura 7.24, de tal forma que las operaciones de barajar y de reparto sean ejecutados por la misma función (**shuffleAndDeal**). La función deberá contener una estructura de ciclo anidado, similar a la función **shuffle**, de la figura 7.24.

7.21 ¿Qué es lo que efectúa este programa?

```
#include <stdio.h>
void mystery1 (char *, const char *);
main()
{
   char string1[80], string2[80];
   printf("Enter two strings: ");
   scanf("%s%s", string1, string2);
   mystery1(string1, string2);
   printf("%s\n", string1);
   return 0;
}

void mystery1(char *s1, const char *s2)
{
   while (*s1 != '0')
      ++s1;
   for ( ; *s1 = *s2; s1++, s2++)
      ; /* empty statement */
}
```

7.22 ¿Qué es lo que ejecuta este programa?

```
#include <stdio.h>

int mystery2(const char *);

main()
{
   char string[80];

   printf("Enter a string: ");
   scanf("%s", string);
   printf("%d\n", mystery2(string));
   return 0;
}

int mystery2(const char *s)
{
   int x = 0;

for ( ; *s != '\0' s++)
   ++x;

return x;
}
```

7.23 Encuentre el error en cada uno de los segmentos de programa siguientes. Si el error puede ser corregido, explique cómo

a)
```
int *number:
printf("%d\n",*number);
```

b)
```
float *realPtr;
long *integerPtr;
integerPtr = realPtr;
```

c)
```
int * x, y;
x = y;
```
d)
```
char s[] = "this is a character array";
int count;
for ( ; *s != '\0'; s++)
   printf("%c ", *s);
```
e)
```
short *numPtr, result;
void *genericPtr = numPtr;
result = *genericPtr + 7;
```
f)
```
float x 0 19.34;
float xPtr = &x;
printf("%f\n", xPtr);
```
g)
```
char *s;
printf("%s\n", s);
```

7.24 (*Quicksort*). En los ejemplos y ejercicios del capítulo 6, se analizaron técnicas de ordenamiento de tipo burbuja, de tipo cubeta y de tipo selección. Introducimos ahora la técnica de ordenación recursiva conocida como Quicksort. El algoritmo básico, para un conjunto de un subíndice de valores, es como sigue:

1) *Paso de particionamiento:* tome el primer elemento del arreglo no ordenado y determine su posición final en el arreglo ordenado. Esto ocurre cuando todos los valores a la izquierda del elemento del arreglo son menores que el elemento, y todos los valores a la derecha del elemento en el arreglo son mayores que el elemento. Tenemos ahora un elemento en su posición correcta y dos subarreglos no ordenados.

2) *Paso recursivo:* Ejecute el paso 1 para cada uno de los subarreglos no ordenados.

Cada vez que en un subarreglo se ejecuta el paso 1, se coloca otro elemento del arreglo ordenado en su posición final, y se crean dos subarreglos no ordenados. Cuando un subarreglo está formado por un solo elemento, deberá ser ordenado, por lo que dicho elemento aparece en su posición final.

El algoritmo básico parece lo suficiente simple, pero ¿cómo determinaremos la posición final del primer elemento de cada subarreglo? Como un ejemplo, considere el siguiente conjunto de valores (el elemento que aparece en negritas es el elemento particionante —será colocado en su posición final del arreglo ordenado):

37 2 6 4 89 8 10 12 68 45

1) Empezando a partir del elemento más a la derecha del arreglo, compare cada uno de los elementos con **37** hasta que se encuentre un elemento menor que **37** y a continuación intercambie **37** con dicho elemento. El primer elemento menor que **37** es 12, por lo que **37** y 12 son intercambiados. El siguiente arreglo queda así:

12 2 6 4 89 8 10 **37** 68 45

El elemento 12 aparece en itálicas, a fin de indicar que ha sido intercambiado con **37**.

2) Empezando a la izquierda del arreglo, pero comenzando con el elemento siguiente a 12, compare cada uno de los elementos con **37** hasta que se encuentra un elemento mayor que **37**, y a continuación intercambie **37** con dicho elemento. El primer elemento mayor que **37** es 89, por lo que se intercambian **37** y 89. El nuevo arreglo resulta:

12 2 6 4 **37** 8 10 *89* 68 45

3) Empezando a partir de la derecha, pero iniciando con el elemento de inmediato anterior a 89, compare cada uno de los elemento con **37**, hasta que se encuentre un elemento menor que **37**, y a continuación intercambie **37** con dicho elemento. El primer elemento menor de **37** es 10, por lo que **37** y 10 resultarán intercambiados. El nuevo arreglo es:

12 2 6 4 *10* 8 **37** 89 68 45

4) Empezando por la izquierda, pero iniciando con el elemento después de 10, compare cada elemento con **37** hasta que se encuentra un elemento mayor de **37** y a continuación intercambie **37** con dicho elemento. Ya no existen más elementos mayores que **37**, por lo que al comparar **37** consigo mismo sabemos que **37** ha sido colocado en su posición final en el arreglo ordenado.

Una vez que la partición ha sido aplicada en el arreglo anterior, tenemos dos subarreglos sin ordenar. El subarreglo con valores menores de 37 contiene 12, 2, 6, 4, 10 y 8. El subarreglo con valores mayores que 37 contiene 89, 68 y 45. La ordenación continúa con ambos subarreglos, particionándose de la misma forma que el arreglo original.

Basados en el análisis anterior, escriba la función recursiva `quicksort` para ordenar un arreglo entero de un subíndice. La función deberá recibir como argumentos un arreglo entero, un subíndice inicial y un subíndice final. La función `partition` deberá ser llamada por `quicksort` para ejecutar el paso de partición.

7.25 (*Atravesar laberintos*). La siguiente cuadrícula de unos y de ceros es un arreglo de doble subíndice, que representa un laberinto.

```
1 1 1 1 1 1 1 1 1 1 1 1
1 0 0 0 1 0 0 0 0 0 0 1
0 0 1 0 1 0 1 1 1 1 0 1
1 1 1 0 1 0 0 0 0 1 0 1
1 0 0 0 0 1 1 1 0 1 0 0
1 1 1 1 0 1 0 1 0 1 0 1
1 0 0 1 0 1 0 1 0 1 0 1
1 1 0 1 0 1 0 1 0 1 0 1
1 0 0 0 0 0 0 0 0 1 0 1
1 1 1 1 1 1 0 1 1 1 0 1
1 0 0 0 0 0 0 1 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1
```

Los unos representan los muros del laberinto, y los ceros representan cuadros en las trayectorias posibles a través del mismo.

Existe un algoritmo simple para caminar a través de un laberinto que garantiza encontrar la salida (suponiendo que una exista). Si no existe salida, usted llegará de regreso a la posición inicial. Coloque su mano derecha sobre el muro a su derecha y empiece a caminar hacia adelante. No retire nunca su mano de la pared. Si el laberinto gira a la derecha, usted sigue el muro a la derecha. Siempre que no retire su mano de la pared, de forma eventual llegará a la salida del laberinto. Pudiera existir una trayectoria más corta de la que haya tomado, pero está garantizado de que saldrá del laberinto.

Escriba la función recursiva `mazeTraverse` para caminar a través del laberinto. La función deberá recibir como argumentos un arreglo de 12 por 12 caracteres, que representa el laberinto, y la posición inicial del laberinto. Conforme `mazeTraverse` intenta localizar la salida del laberinto, deberá colocar el carácter `x` en cada uno de los cuadros de la trayectoria. Después de cada movimiento la función deberá mostrar el laberinto, para que el usuario pueda observar conforme resuelve el laberinto.

7.26 (*Cómo generar laberintos en forma aleatoria*). Escriba una función `mazeGenerator`, que toma como argumento un arreglo de 12 por 12 de doble subíndice, y que produce en forma aleatoria un laberinto. La función también deberá proporcionar las posiciones iniciales y finales del mismo. Pruebe su función `mazeTreverse`, del ejercicio 7.25, utilizando varios laberintos generados al azar.

7.27 (*Laberintos de cualquier tamaño*). Generalice las funciones `mazeTreverse` y `mazeGenerator`, de los ejercicios 7.25 y 7.26, para poder procesar laberintos de cualquier ancho y altura.

7.28 (*Arreglos de apuntadores a funciones*). Vuelva a escribir el programa de la figura 6.22 para utilizar una interfaz manejada por menú. El programa deberá darle 4 opciones al usuario, como sigue:

```
Enter a choice:
  0  Print the array of grades
  1  Find the minimum grade
  2  Find the maximum grade
  3  Print the average on all tests for each student
  4  End program
```

Una restricción para el uso de arreglos de apuntadores a funciones es que todos los apuntadores tienen que ser del mismo tipo. Los apuntadores deben ser a funciones con el mismo tipo de regreso y que reciban argumentos del mismo tipo. Por esta razón, deberán ser modificadas las funciones en la figura 6.22, de tal forma que cada una de ellas pueda regresar el mismo tipo y tomar los mismos parámetros. Modifique las funciones **minimum y maximum** para imprimir el valor mínimo o máximo y para que regresen nada. En el caso de la opción 3, modifique la función **average** de la figura 6.22 para que se extraiga el promedio de cada alumno (y no de un alumno en particular). La función **average** debe regresar nada y tomar los mismos parámetros que **printArray**, **minimum** y **maximum**. Almacene los apuntadores a las cuatro funciones en el arreglo **processGrades** y utilice la elección que efectúe el usuario como subíndice del arreglo para llamar a cada una de las funciones.

7.29 (*Modificaciones al simulador Simpletron*). En el ejercicio 7.19, usted escribió una simulación en software de una computadora que ejecuta programas escritos en lenguaje de máquina Simpletron (LMS). En este ejercicio, proponemos varias modificaciones y mejoras al simulador Simpletron. En los ejercicios 12.26 y 12.27, proponemos la construcción de un compilador, que convierta programas escritos en un lenguaje de programación de alto nivel (una variación de BASIC) al lenguaje de máquina Simpletron. Algunas de las modificaciones y mejoras siguientes pudieran ser requeridas para ejecutar los programas producidos por el compilador.

a) Extienda la memoria del simulador Simpletron para que contenga 1000 posiciones de memoria y permitir que Simpletron maneje programas más grandes.
b) Permita que el simulador ejecute cálculos de módulos. Esto requiere de una instrucción adicional en lenguaje de máquina Simpletron.
c) Permita que el simulador ejecute cálculos de exponenciación. Esto requiere de una instrucción adicional en lenguaje de máquina Simpletron.
d) Modifique el simulador para que pueda utilizar valores hexadecimales en vez de valores enteros, para representar instrucciones en lenguaje de máquina Simpletron.
e) Modifique el simulador para que permita la salida de nueva línea. Esto requiere de una instrucción adicional en lenguaje de máquina Simpletron.
f) Modifique el simulador para que pueda procesar valores en punto flotante en adición a valores enteros.
g) Modifique el simulador para manejar entradas de cadenas. *Sugerencia*: cada palabra Simpletron puede ser dividida en dos grupos, cada uno de ellos conteniendo un entero de dos dígitos. Cada entero de dos dígitos puede representar el equivalente decimal ASCII de un carácter. Añada una instrucción en lenguaje máquina que introduzca una cadena y almacene el principio de la cadena en una posición específica de la memoria Simpletron. La primera mitad de la palabra en dicha posición será una cuenta del número de caracteres dentro de la cadena (es decir, la longitud de la cadena). Cada mitad de palabra siguiente contiene un carácter ASCII, expresado en forma de dos dígitos decimales. La instrucción en lenguaje de máquina convierte cada carácter en su equivalente ASCII y lo asigna a dicha media palabra.
h) Modifique el simulador para manejar la salida de cadenas almacenadas en el formato diseñado en la parte (g). *Sugerencia*: añada una instrucción en lenguaje de máquina que imprima una cadena, empezando en cierta posición de memoria Simpletron. La primera parte 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 siguiente contiene el carácter ASCII, expresado como dos

dígitos decimales. La instrucción en lenguaje de máquina verifica la longitud e imprime la cadena, traduciendo cada dos números en su carácter equivalente.

7.30 ¿Qué hace este programa?

```
#include <stdio.h>

int mystery3(const char *, const char *);

main()
{
   char string1[80], string2[80];

   printf("Enter two strings: ");
   scanf("%s%s", string1, string2);
   printf("The result is %d\n", mystery3(string1, string2));

   return 0;
}

int mystery3(const char *s1, const char *s2)
{
   for ( ; *s1 != '\0' && *s2 != '\0'; s1++, s2++)

      if (*s1 != *s2)
         return 0;

   return 1;
}
```