10

Estructuras, uniones, manipulaciones de bits y enumeraciones

Objetivos

- Ser capaz de crear y utilizar estructuras, uniones y enumeraciones.
- Ser capaz de pasar estructuras a funciones en llamada por valor y en llamada por referencia.
- Ser capaz de manipular datos con operadores a nivel de bits.
- Ser capaz de crear campos de bits para almacenar datos en forma compacta.

Nunca pude comprender lo que esos malditos puntos significaban.
Winston Churchill

Pero otra vez una unión en desunión;
William Shakespeare

Puedes excluirme.
Samuel Goldwyn

La misma caritativa y vieja mentira
Repetida conforme los años pasan
Siempre con el mismo éxito—
"¡Realmente no has cambiado nada!"
Margaret Fishback

Sinopsis

10.1 introducción

Las *estructuras* son colecciones de variables relacionadas —a veces denominadas *agregados*— bajo un nombre. Las estructuras pueden contener variables de muchos tipos diferentes de datos —a diferencia de los arreglos, que contienen únicamente elementos de un mismo tipo de datos. Generalmente las estructuras se utilizan para definir registros a almacenarse en archivos (vea el capítulo 11, "Procesamiento de archivos"). Los apuntadores y las estructuras facilitan la formación de estructuras de datos de mayor complejidad, como son listas enlazadas, colas de espera, pilas y árboles (vea el capítulo 12, "Estructuras de datos").

10.2 Definiciones de estructuras

Las estructuras son *tipos de datos derivados* —están construidas utilizando objetos de otros tipos. Considere la siguiente definición de estructura:

```
struct card {
   char *face;
   char *suit;
};
```

La palabra reservada **struct** presenta la definición de estructura. El identificador **card** es el *rótulo de la estructura*. El rótulo de la estructura da nombre a la definición de la misma, y se utiliza con la palabra reservada **struct** para declarar variables del *tipo estructura*. En este ejemplo, el tipo estructura es **struct card** . Las variables declaradas dentro de las llaves de la definición de estructura son los *miembros* de la estructura. Los miembros de la misma estructura

deben tener nombres únicos, pero dos estructuras diferentes pueden contener miembros con el mismo nombre sin entrar en conflicto (pronto veremos por qué). Cada definición de estructura debe terminar con un punto y coma.

Error común de programación 10.1

Olvidar el punto y coma que da por terminada una definición de estructura.

La definición de **struct card** contiene dos miembros del tipo **char * face** y **suit**. Los miembros de la estructura pueden ser variables de los tipos de datos básicos (es decir, **int**, **float**, etcétera), o agregados, como son los arreglos y otras estructuras. Como ya vimos en el capítulo 6, cada elemento de un arreglo debe ser del mismo tipo. Los miembros de una estructura, sin embargo, pueden ser de una variedad de tipos de datos. Por ejemplo, un **struct employee** pudiera contener miembros de cadenas de caracteres correspondientes a los nombres y apellidos, un miembro **int**, para la edad del empleado, un miembro **char**, que contenga **'M'** o bien **'F'** para el sexo del empleado, un miembro **float** para el salario horario del empleado, y así sucesivamente. Una estructura no puede contener una instancia de sí misma. Por ejemplo, una variable del tipo **struct card** no puede ser declarada dentro de la definición correspondiente a **struct card**. Sin embargo, pudiera ser incluido un apuntador a **estruct card**. Una estructura, que contenga un miembro que es un apuntador al mismo tipo de estructura, se conoce como una *estructura autorreferenciada*. Las estructuras autorreferenciadas se utilizan en el capítulo 12 para construir varios tipos de estructuras de datos enlazadas.

La anterior definición de estructura no reserva ningún espacio en memoria, más bien genera un nuevo tipo de datos, que se utiliza para declarar variables. Las variables de estructura se declaran como se declaran las variables de otros tipos. La declaración

```
struct card a, deck[52], *c;
```

declara **a** ser una variable del tipo **struct card**, declara **deck** como un arreglo con 52 elementos del tipo **struct card**, y declara **c** como un apuntador a **struct card**. Las variables de un tipo dado de estructura, pudieran también ser declaradas colocando una lista, separada por comas, de los nombres de las variables, entre la llave de cierre de la definición de la estructura y el punto y coma que termina la definición de la misma. Por ejemplo, la declaración anterior podía haberse incorporado en la definición de estructura **struct card** como sigue:

```
struct card {
      char *face;
      char *suit;
} a, deck[52], *c;
```

El nombre del rótulo de la estructura es opcional. Si la definición de una estructura no contiene un nombre de rótulo de estructura, las variables de ese tipo de estructura pueden únicamente ser declaradas dentro de la definición de estructura —y no en una declaración por separado.

Práctica sana de programación 10.1

Al crear un tipo de estructura proporcione un nombre de rótulo de estructura. El nombre de rótulo de estructura es conveniente más adelante en el programa para la declaración de nuevas variables de este tipo de estructura.

Práctica sana de programación 10.2

Seleccionar un nombre de rótulo de estructura significativo ayuda a autodocumentar el programa.

Las únicas operaciones válidas que pueden ejecutarse sobre estructuras son: asignar variables de estructura a variables de estructura del mismo tipo, tomando la dirección (**&**) de una variable de estructura, obteniendo acceso a los miembros de una variable de estructura (vea la Sección 10.4), y utilizando el operador **sizeof**, a fin de determinar el tamaño de la variable de estructura.

Error común de programación 10.2

Asignar una estructura de un tipo a una estructura de un tipo distinto.

Las estructuras no pueden compararse entre sí, porque los miembros de las estructuras no están necesariamente almacenados en bytes de memoria consecutivos. Algunas veces en una estructura existen "huecos" porque las computadoras pudieran almacenar tipos de datos específicos en ciertos límites de memoria, como son límites de media palabra, de palabra o de dobles palabras. Una palabra es una unidad estándar de memoria, utilizada para almacenar datos en una computadora — 2 o 4 bytes, normalmente. Considere la siguiente definición de estructura, en la cual se declaran **sample1** y **sample2**, del tipo **struct example**:

```
struct example {
      char c;
      int i;
} sample1, sample2;
```

Una computadora con palabras de 2 bytes pudiera requerir que cada uno de los miembros de **struct example** fuesen alineados en un límite de palabras, es decir, al principio de una palabra (esto depende de la máquina). En la Figura 10.1 se muestra una alineación de almacenamiento para una variable del tipo **struct example**, que ha sido asignado al carácter **'a'**, y el entero **97** (se muestran las representaciones de los valores en bits). Si los miembros se almacenan empezando en los límites de palabras, aparece un hueco de 1 byte (byte **1** en la figura) en el almacenamiento para variables del tipo **struct example**. El valor en el hueco de un byte se queda sin definir. Si los valores de miembros de **sample1** y **sample2** son de hecho iguales, la comparación de las estructuras no será necesariamente igual, porque los huecos no definidos de un byte probablemente no contendrán valores idénticos.

Error común de programación 10.3

Es un error de sintaxis comparar estructuras, debido a diferentes requisitos de alineación en los diferentes sistemas.

Sugerencia de portabilidad 10.1

Dado que depende de la máquina el tamaño de los elementos de datos de un tipo particular, y debido a que las consideraciones de alineación de almacenamiento también son dependientes de la máquina, entonces también lo será la representación de una estructura.

Fig. 10.1 Una posible alineación de almacenamiento para una variable del tipo **struct example** mostrando en la memoria un área no definida.

10.3 Cómo inicializar estructuras

Las estructuras pueden ser inicializadas mediante listas de inicialización como con los arreglos. Para inicializar una estructura, escriba en la declaración de la estructura, a continuación del nombre de la variable, un signo igual, con inicializadores encerrados entre llaves y separados por comas. Por ejemplo, la declaración

```
struct card a = ("Three", "Hearts");
```

crea la variable **a** del tipo **struct card** (como fue definida anteriormente) e incializa el miembro **face** a **"Three"**, y el miembro **suit** a **"Hearts"**. Si en la lista aparecen menos inicializadores que en la estructura, los miembros restantes automáticamente quedarán inicializados a **0** (o **NULL** si el miembro es un apuntador). Las variables de estructura declarados por fuera de una definición de función (es decir, en forma externa) se inicializan a **0** o **NULL** si en la declaración externa no se inicializan en forma explícita. Las variables de estructura también pueden ser inicializadas en enunciados de asignación, asignándoles una variable de estructura del mismo tipo, o asignando valores a los miembros individuales de la misma.

10.4 Cómo tener acceso a los miembros de estructuras

Para tener acceso a miembros de estructuras se utilizan dos operadores: el *operador de miembro de estructura* (.) —también conocido como *operador punto*— y el *operador de apuntador de estructura* (->) —también conocido como el *operador de flecha*. El operador de miembro de estructura tiene acceso a un miembro de estructura mediante el nombre de la variable de estructura. Por ejemplo, para imprimir el miembro **suit** de la estructura **a** correspondiente a la declaración anterior, utilice el enunciado

```
printf ("%s", a.suit);
```

El operador de apuntador de estructura —que consiste de un signo menos (-) y de un signo mayor que (>), sin espacios intermedios— tiene acceso a un miembro de estructura vía un apuntador a la estructura. Suponga que el apuntador **aPtr** se ha declarado para apuntar a **struct card**, y que la dirección de la estructura **a** ha sido asignada a **aPtr**. Para imprimir el miembro **suit** de la estructura **a** utilizando el apuntador **aPtr**, utilice el enunciado

```
printf ("%s", aPtr->suit);
```

La expresión **aPtr->suit** es equivalente **(*aPtr).suit** que desreferencía el apuntador y tiene acceso al miembro **suit** utilizando el operador de miembro de estructura. Se requiere aquí de los paréntesis, porque el operador de miembro de estructura (.) tiene una precedencia mayor que el operador de desreferenciación de apuntador (*). El operador de apuntador de estructura y el operador de miembro de estructura, junto con los paréntesis y los corchetes ([]) utilizados para los subíndices de arreglos, tienen la precedencia de operadores más alta y se asocian de izquierda a derecha.

Práctica sana de programación 10.3

Evite utilizar los mismos nombres para miembros de estructura de distintos tipos. Ello es permitido, pero podría causar confusión.

Práctica sana de programación 10.4

No deje espacios alrededor de los operadores -> y . ya que ayuda a enfatizar que las expresiones en las cuales los operadores están contenidos son esencialmente nombres individuales de variables.

Error común de programación 10.4

Insertar un espacio entre el signo de - y el signo de > del operador de apuntador de estructura, (o insertar espacios entre los componentes de cualquier otro operador múltiple de teclado, a excepción de ?:).

Error común de programación 10.5

Intentar referirse a un miembro de una estructura utilizando únicamente el nombre de dicho miembro.

Error común de programación 10.6

No utilizar paréntesis al referirse a un miembro de estructura utilizando un apuntador y el operador de miembro de estructura (por ejemplo `*aPtr.suit`, es un error de sintaxis).

El programa de la figura 10.2 pone de manifiesto el uso de los operadores de miembro de estructura y de apuntador de estructura. Mediante el uso del operador de miembro de estructura, los miembros de la estructura **a** son asignados los valores **"Ace"** y **"Spades"** respectivamente. Al apuntador **aPtr** se le asigna la dirección de la estructura **a**. Un enunciado **printf** imprime los miembros de la variable de estructura **a**, utilizando el operador de miembro de estructura con el nombre de variable **a**, el operador de apuntador de estructura con el apuntador **aPtr**, y el operador de miembro de estructura con el apuntador desreferenciado **aPtr**.

```
/* Using the structure member and
   structure pointer operators */
#include <stdio.h>

struct card {
   char *face;
   char *suit;
};

main()
{
   struct card a;
   struct card *aPtr;

   a.face = "Ace";
   a.suit = "Spades";
   aPtr = &a;
   printf("%s%s%s\n%s%s%s\n%s%s%s\n",
          a.face, " of ", a.suit,
          aPtr->face, " of ", aPtr->suit,
          (*aPtr).face, " of ", (*aPtr).suit);
   return 0;
}
```

```
Ace of Spades
Ace of Spades
Ace of Spades
```

Fig. 10.2 Cómo utilizar el operador de miembro de estructura y el operador de apuntador de estructura.

10.5 Cómo utilizar estructuras con funciones

Las estructuras pueden ser pasadas a funciones pasando miembros de estructura individuales, pasando toda la estructura, o pasando un apuntador a una estructura. Cuando se pasan estructuras o miembros individuales de estructura a una función, se pasan en llamada por valor. Por lo tanto, los miembros de la estructura de un llamador no podrán ser modificadas por la función llamada.

Para pasar una estructura en llamada por referencia, pase la dirección de la variable de estructura. Los arreglos de estructura —como todos los demás otros arreglos— son automáticamente pasados en llamada por referencia.

En el capítulo 6, indicamos que un arreglo podía ser pasado en llamada por valor mediante el uso de una estructura. Para pasar un arreglo en llamada por valor, origine una estructurza con el arreglo como un miembro. Dado que las estructuras se pasan en llamada por valor, el arreglo será pasado en llamada por valor.

Error común de programación 10.7

Suponer que las estructuras, como los arreglos, se pasan automáticamente en llamada por referencia, e intentar modificar los valores de estructura del llamador en la función llamada.

Sugerencia de rendimiento 10.1

Es más eficaz pasar estructuras en llamada por referencia que pasar estructuras en llamada por valor (ya que esto último requiere que toda la estructura se copie).

10.6 Typedef

La palabra reservada **typedef** proporciona un mecanismo para la creación de sinónimos (o alias) para tipos de datos anteriormente definidos. Los nombres de los tipos de estructura se definen a menudo utilizando **typedef**, a fin de crear nombres de tipo más breves. Por ejemplo, el enunciado

```
typedef struct card Card;
```

define el nuevo nombre de tipo **Card** como un sinónimo para el tipo **struct card**. Los programadores en C utilizan a menudo **typedef** para definir un tipo de estructura de tal forma que un rótulo de estructura no sea requerido. Por ejemplo, la definición siguiente

```
typedef struct {
   char *face;
   char *suit;
} Card;
```

crea el tipo de estructura **Card**, sin necesidad de un enunciado por separado **typedef**.

Práctica sana de programación 10.5

*Ponga los nombres **typedef** en mayúsculas, para enfatizar que esos nombres son sinónimos de otros nombres de tipo.*

Card puede ahora ser utilizado para declarar variables del tipo **struct card**. La declaración

```
Card deck[52];
```

declara un arreglo de 52 estructuras **Card** (es decir, variables del tipo **struct card**). Al crear un nuevo nombre utilizando **typedef** no se crea un nuevo tipo; **typedef** simplemente crea un

nuevo nombre de tipo, que puede ser utilizado como un seudónimo para un nombre de tipo existente. Un nombre significativo auxilia a autodocumentar el programa. Por ejemplo, cuando leemos la declaración anterior, sabemos que "**deck** es un arreglo de 52 **Cards**"

typedef se utiliza a menudo para crear seudónimos para los tipos de datos básicos. Por ejemplo, un programa que requiera de enteros de 4 bytes, pudiera utilizar el tipo **int** en un sistema y el tipo **long** en otro. Los programas diseñados para portabilidad, a menudo utilizan **typedef** para crear un alias o seudónimo para los enteros de 4 bytes como sería **Integer**. Una vez dentro del programa el alias **Integer** puede ser modificado, para hacer que el programa funcione en ambos sistemas.

Sugerencia de portabilidad 10.2

*Utilice **typedef** para ayudar a hacer más portátil un programa.*

10.7 Ejemplo: simulación de barajar y distribuir cartas de alto rendimiento

El programa en la figura 10.3 se basa en la simulación de barajar y distribuir cartas analizado en el capítulo 7. El programa representa el mazo de cartas o de naipes como un arreglo de estructuras. El programa utiliza algoritmos de alto rendimiento para barajar y distribuir. La salida del programa de alto rendimiento para barajar y distribuir se muestra en la figura 10.4.

En el programa, la función **fillDeck** inicializa el arreglo **Card** en orden desde Ace hasta King de cada uno de los palos. El arreglo **Card** se pasa a la función **shuffle**, donde se pone en operación el algoritmo de alto rendimiento de barajar. La función **shuffle** toma como argumento un arreglo de 52 estructuras **Card**. La función cicla a través de las 52 cartas (subíndices de arreglo 0 a 51) mediante una estructura **for**. Para cada una de las cartas, es tomado al azar un número entre 0 y 51. A continuación, en el arreglo son intercambiadas la estructura actual **Card** y la estructura seleccionada al azar **Card**. En una sola pasada de todo el arreglo se llevan a cabo un total de 52 intercambios, y ¡el arreglo de estructuras **Card** queda barajado! Este algoritmo no puede sufrir por posposición indefinida, como sufría el algoritmo de barajar presentado en el capítulo 7. Dado que en el arreglo las estructuras **Card** fueron intercambiadas en su lugar, el algoritmo de distribución de alto rendimiento puesto en marcha en la función **deal** requerirá de únicamente una pasada del arreglo para distribuir las cartas barajadas.

Error común de programación 10.8

Olvidar incluir el subíndice de arreglo al referirse a estructuras individuales de un arreglo de estructuras.

10.8 Uniones

Una *unión* es un tipo de datos derivado —como lo es una estructura— cuyos miembros comparten el mismo espacio de almacenamiento. Para distintas situaciones en un programa, algunas variables pudieran no ser de importancia, pero otras variables lo son —por lo que una unión comparte el espacio, en vez de desperdiciar almacenamiento en variables que no están siendo utilizadas. Los miembros de una unión pueden ser de cualquier tipo. El número de bytes utilizados para almacenar una unión, deben ser por lo menos suficientes para contener el miembro más grande. En la mayor parte de los casos, las uniones contienen dos o más tipos de datos. Unicamente un miembro y, por lo tanto, únicamente un tipo de datos, puede ser referenciado en un momento dado. Es responsabilidad del programador asegurarse que en una unión los datos están referenciados con el tipo de dato apropiado.


```
/* The card shuffling and dealing program using structures */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

struct card {
   char *face;
   char *suit;
};

typedef struct card Card;

void fillDeck(Card *, char *[], char *[]);
void shuffle(Card *);
void deal(Card *);

main()
{
   Card deck[52];
   char *face[] = {"Ace", "Deuce", "Three", "Four", "Five",
                   "Six", "Seven", "Eight", "Nine", "Ten",
                   "Jack", "Queen", "King"};
   char *suit[] = {"Hearts", "Diamonds", "Clubs", "Spades"};

   srand(time(NULL));

   fillDeck(deck, face, suit);
   shuffle(deck);
   deal(deck);
   return 0;
}

void fillDeck(Card *wDeck, char *wFace[], char *wSuit[])
{
   int i;

   for (i = 0; i <= 51; i++) {
      wDeck[i].face = wFace[i % 13];
      wDeck[i].suit = wSuit[i / 13];
   }
}

void shuffle(Card *wDeck)
{
   int i, j;
   Card temp;

   for (i = 0; i <= 51; i++) {
      j = rand() % 52;
      temp = wDeck[i];
      wDeck[i] = wDeck[j];
      wDeck[j] = temp;
   }
}
```

```
void deal(Card *wDeck)
{
   int i;

   for (i = 0; i < 52; i++)
      printf("%5s of %-8s%c", wDeck[i].face, wDeck[i].suit,
             (i + 1) % 2 ? '\t' : '\n');
}
```

Fig. 10.3 Simulación de barajar y distribuir cartas de alto rendimiento (parte 2 de 2).

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

Fig. 10.4 Salida de la simulación de barajar y distribuir cartas de alto rendimiento.

Error común de programación 10.9

Es un error lógico referenciar con el tipo equivocado, datos en una unión almacenados con un tipo distinto.

Sugerencia de portabilidad 10.3

Si en una unión los datos se almacenan como de un tipo y se referencían como de otro tipo, los resultados serán dependientes de la instalación.

Una unión se declara con la palabra reservada **union** en el mismo formato que una estructura. La declaración **union**

```
union number{
   int x;
   float y;
};
```

indica que **number** es un tipo **union** con miembros **int x** y **float y**. En un programa normalmente la definición de unión antecede a **main**, por lo que ésta puede ser utilizada para declarar variables en todas las funciones del programa.

Observación de ingeniería de software 10.1

Al igual que en una declaración `struct`*, una declaración* `union` *simplemente crea un tipo nuevo. Colocar una declaración* `union` *o* `struct` *fuera de cualquier función no crea una variable global.*

Las operaciones que pueden ser ejecutadas en una unión son: asignar una unión a otra unión del mismo tipo, tomar la dirección (**&**) de una unión, y tener acceso a los miembros de una unión utilizando el operador de miembro de estructura y el operador de apuntador de estructura. Las uniones no pueden ser comparadas entre sí, por las mismas razones que no pueden compararse las estructuras.

En una declaración, una unión puede ser inicializada únicamente con un valor del mismo tipo que el primer miembro de la unión. Por ejemplo, en la unión anterior, la declaración

```
union number value = {10};
```

es una inicialización válida de la variable de unión **value**, porque la unión está inicializada con un **int**, pero la siguiente declaración no sería válida:

```
union number value = {1.43};
```

Error común de programación 10.10

Es un error de sintaxis comparar uniones, debido a los diferentes requisitos de alineación en varios sistemas.

Error común de programación 10.11

Inicialización de una uión en una declaración con un valor cuyo tipo es distinto al del primer miembro de la unión.

Sugerencia de portabilidad 10.4

La cantidad de almacenamiento requerido para almacenar una unión es dependiente de la instalación.

Sugerencia de portabilidad 10.5

Quizá no sea fácil portar algunas uniones a otros sistemas de computación. El que una unión sea portable o no depende a menudo de los requerimientos de alineación de almacenamiento para los tipos de miembro de unión de información en un sistema particular.

Sugerencia de rendimiento 10.2

Las uniones ahorran almacenamiento.

El programa de la figura 10.5 utiliza la variable **value** del tipo **union number**, para desplegar el valor almacenado en la unión, tanto como un **int** como como un **float**. La salida del programa depende de la instalación. La salida del programa muestra que la representación interna de un valor **float** puede resultar bastante distinta de la representación de **int**.

10.9 Operadores a nivel de bits

En las computadoras en forma interna todos los datos se representan como secuencias de bits. Cada bit puede asumir un valor de **0** o un valor de **1**. En la mayor parte de los sistemas, una secuencia de 8 bits forma un byte —la unidad estándar de almacenamiento para una variable del tipo **char**. Otros tipos de datos son almacenados en números de bits más grandes. Los operadores

```
/* An example of a union */
#include <stdio.h>

union number {
    int x;
    float y;
};

main()
{
    union number value;

    value.x = 100;
    printf("%s\n%s\n%s%d\n%s%f\n\n",
           "Put a value in the integer member",
           "and print both members.",
           "int:   ", value.x,
           "float: ", value.y);

    value.y = 100.0;
    printf("%s\n%s\n%s%d\n%s%f\n",
           "Put a value in the floating member",
           "and print both members.",
           "int:   ", value.x,
           "float: ", value.y);
   return 0;
}
```

```
Put  value in the integer member
and print both members.
int:   100
float: 0.000000

Put a value in the floating member
and print both members.
int:   17096
float: 100.000000
```

Fig. 10.5 Cómo imprimir el valor de una unión en ambos tipos de datos de miembro.

a nivel de bits se utilizan para manipular los bits de operandos integrales (**char**, **short**, **int** y **long**; tanto **signed** como **unsigned**). Los enteros no signados (**unsigned**) son utilizados normalmente con los operadores a nivel de bits.

Sugerencia de portabilidad 10.6

Las manipulaciones de datos a nivel de bits son dependientes de la máquina.

Advierta que los análisis de los operadores a nivel de bits de esta sección, muestran las representaciones binarias de los operandos enteros. Para una explicación detallada del sistema numérico binario (también conocido como de base 2) vea el Apéndice E, "Sistemas numéricos". También, los programas de las Secciones 10.9 y 10.10 fueron probados en un Macintosh de Apple, utilizando Think C y en una PC compatible, utilizando Borland C++. Ambos sistemas utilizan enteros de 16 bits (2 bytes). Dada la naturaleza de dependencia de la máquina de las manipulaciones a nivel de bits, estos programas pudieran no funcionar en su sistema.

Los operadores a nivel de bits son: *AND a nivel de bits (&), OR inclusivo a nivel de bits (|), OR exclusivo a nivel de bits (^), desplazamiento a la izquierda (<<), desplazamiento a la derecha (>>)*, y *complemento (~)*. El AND a nivel de bits, el OR inclusivo a nivel de bits y el OR exclusivo a nivel de bits son operadores que comparan sus dos operandos bit por bit. El operador AND a nivel de bits establece en el resultado cada bit a 1, si el bit correspondiente en ambos operandos es 1. El operador OR inclusivo a nivel de bits, establece en el resultado cada bit a 1, si el bit correspondiente en cada o en (ambos) operandos es 1. El operador OR exclusivo a nivel de bits establece en el resultado cada bit a 1 si el bit correspondiente en exactamente un operando es 1. El operador de desplazamiento a la izquierda desplaza los bits de su operando izquierdo hacia la izquierda por el número de bits especificado en su operando derecho. El operador de desplazamiento a la derecha desplaza los bits de su operando izquierdo hacia la derecha en el número de bits especificado por su operando derecho. El operador de complemento a nivel de bits define en el resultado todos los bits **0** en su operando a 1, y define todos los bits **1** a **0** en el resultado. En

Operador	Descripción
& AND a nivel de bits	Los bits en el resultado se establecen a **1** si los bits correspondientes en ambos operandos son ambos **1**.
| OR inclusivo a nivel de bits	Los bits en el resultado se establecen a **1** si por lo menos uno de los bits correspondientes en los dos operandos es **1**.
^ OR exclusivo a nivel de bits	Los bits en el resultado se definen a **1** si exactamente uno de los bits correspondientes en los dos operandos es **1**.
<< desplazamiento a la izquierda	Desplaza los bits del primer operando hacia la izquierda en el número de bits especificado por el segundo operando; rellena a partir de la derecha con bits **0**.
>> desplazamiento a la derecha	Desplaza los bits del primer operando hacia la derecha en el número de bits especificado por el segundo operando; el método de rellenar a partir de la izquierda depende de la máquina.
~ complemento a uno	Todos los bits **0** se definen a **1** y todos los bits **1** se definen a cero.

Fig. 10.6 Los operadores a nivel de bits.

los ejemplos que siguen aparecen análisis detallados de cada operador a nivel de bits. Los operadores a nivel de bits se resumen en la figura 10.6.

Al utilizar los operadores a nivel de bits, es útil imprimir los valores en su representación binaria, para ilustrar los efectos precisos de estos operadores. El programa de la figura 10.7 imprime un entero **unsigned** en su representación binaria en grupos de ocho bits cada uno. La función **displayBits** utiliza el operador AND a nivel de bits para combinar la variable **value** con la variable **displayMask**. A menudo, el operador AND a nivel de bits se utiliza con un operando conocido como una *máscara* —un valor entero con bits específicos establecidos a 1. Las máscaras se utilizan para ocultar algunos bits en un valor, mientras otros bits se seleccionan. En la función **displayBits**, la variable de máscara **displayMask** es asignada el valor **1 << 15 (10000000 00000000)**. El operador de desplazamiento a la izquierda desplaza el valor

```
/* Printing an unsigned integer in bits */
#include <stdio.h>

main()
{
   unsigned x;
   void displayBits(unsigned);

   printf("Enter an unsigned integer: ");
   scanf("%u", &x);
   displayBits(x);
   return 0;
}

void displayBits(unsigned value)
{
   unsigned c, displayMask = 1 << 15;

   printf("%7u = ", value);

   for (c = 1; c <= 16; c++) {
      putchar(value & displayMask ? '1' : '0');
      value <<= 1;

      if (c % 8 == 0)
         putchar(' ');
   }

   putchar('\n');
}
```

```
Enter an unsigned integer: 65000
  65000 = 11111101 11101000
```

Fig. 10.7 Cómo imprimir un entero no signado en bits.

1 de la posición inferior (más a la derecha) hacia el bit de orden superior (más a la izquierda) en **displayMask**, y rellena con bits 0 a partir de la derecha. El enunciado

```
putchar(value & displayMask ? '1' : '0');
```

determina si deberá de imprimirse un 1 o un 0 para el bit actual más a la izquierda de la variable **value**. Suponga que la variable **value** contiene **65000 (11111101 11101000)**. Cuando se combinan **value** y **displayMask** utilizando **&**, todos los bits, a excepción del bit de orden superior, en la variable **value**, son "enmascarados" (ocultos) porque cualquier bit "manipulado por AND" con 0 da como resultado 0. Si el bit más a la izquierda es 1, **value & displayMask** se evalúa a 1, y 1 se imprime —de lo contrario se imprime 0. La variable **value** es después desplazada un bit a la izquierda, mediante la expresión **value << = 1** (este es equivalente a **value = value << = 1**). Estos pasos se repiten para cada uno de los bits en la variable **unsigned value**. En la figura 10.8 se resumen los resultados de combinar dos bits con el operador AND a nivel de bits.

Error común de programación 10.12

Usar el operador lógico AND (&&), en lugar del operador AND a nivel de bits (&), y viceversa.

El programa de la figura 10.9 demuestra el uso del operador AND a nivel de bits, del operador OR inclusivo a nivel de bits, del operador OR exclusivo a nivel de bits, y del operador de complemento a nivel de bits. El programa utiliza la función **display Bits** para imprimir los valores enteros **unsigned**. La salida se muestra en la figura 10.10.

En la figura 10.9, la variable entera **mask** es asignada al valor **1 (00000000 00000001)**, y a la variable **number1** se le asigna el valor **65535 (11111111 11111111)**. Cuando se combinan **mask** y **number1** utilizando el operador AND a nivel de bits (**&**) en la expresión **number1 & mask**, el resultado es **00000000 00000001**. Todos los bits, salvo el bit de orden inferior en la variable **number1** quedan "enmascarados" (ocultos), mediante la operación con el operador "AND" con la variable **mask**.

El operador OR inclusivo a nivel de bits se utiliza para definir en un operando bits específicos a 1. En la figura 10.9, la variable **number1** es asignada **15 (00000000 00001111)**, y la variable **setBits** es asignada **241 (00000000 11110001)**. Cuando se combinan **number1** y **setBits**, utilizando el operador OR a nivel de bits en la expresión **number1 | setBits**, el resultado es **255 (00000000 11111111)**. En la figura 10.11 se resumen los resultados de combinar dos bits con el operador OR inclusivo a nivel de bits.

Error común de programación 10.13

Usar el operador OR lógico (||), en lugar del operador OR a nivel de bits (|), y viceversa.

Bit 1	Bit 2	Bit 1 & Bit 2
0	0	0
1	0	0
0	1	0
1	1	1

Fig. 10.8 Resultados de combinar dos bits mediante el operador AND a nivel de bits **&**.

```
/* Using the bitwise AND, bitwise inclusive OR, bitwise
   exclusive OR, and bitwise complement operators */

#include <stdio.h>

void displayBits(unsigned);

main()
{
   unsigned number1, number2, mask, setBits;

   number1 = 65535;
   mask = 1;
   printf("The result of combining the following\n");
   displayBits(number1);
   displayBits(mask);
   printf("using the bitwise AND operator & is\n");
   displayBits(number1 & mask);

   number1 = 15;
   setBits = 241;
   printf("\nThe result of combining the following\n");
   displayBits(number1);
   displayBits(setBits);
   printf("using the bitwise inclusive OR operator | is\n");
   displayBits(number1 | setBits);

   number1 = 139;
   number2 = 199;
   printf("\nThe result of combining the following\n");
   displayBits(number1);
   displayBits(number2);
   printf("using the bitwise exclusive OR operator ^ is\n");
   displayBits(number1 ^ number2);

   number1 = 21845;
   printf("\nThe one's complement of\n");
   displayBits(number1);
   printf("is\n");
   displayBits(~number1);

   return 0;
}
```

Fig. 10.9 Cómo utilizar el AND a nivel de bits, el OR inclusivo a nivel de bits, el OR exclusivo a nivel de bits, y el operador de complemento a nivel de bits (parte 1 de 2).

El operador OR exclusivo a nivel de bits (^) define cada bit en el resultado a 1, si *exactamente* uno de los bits, correspondiente en sus dos operandos, es 1. En la figura 10.9, las variables **number1** y **number2** se les asigna a los valores `139 (00000000 10001011)` y `199 (00000000 11000111)` respectivamente. Cuando se combinan estas variables con el operador OR exclusivo, en la expresión **number1 ^ number2**, el resultado es `00000000 01001100`. En la figura 10.12 se resumen los resultados de combinar dos bits utilizando el operador OR exclusivo a nivel de bits.


```
void displayBits(unsigned value)
{
   unsigned c, displayMask = 1 << 15;

   printf("%7u = ", value);

   for (c = 1; c <= 16; c++) {
      putchar(value & displayMask ? '1' : '0');
      value <<= 1;

      if (c % 8 == 0)
         putchar(' ');
   }

   putchar('\n');
}
```

Fig. 10.9 Cómo utilizar el AND a nivel de bits, OR inclusivo a nivel de bits, el OR exclusivo a nivel de bits, y el operador de complemento a nivel de bits (parte 2 de 2).

```
The result of combining the following
  65535 = 11111111 11111111
      1 = 00000000 00000001
using the bitwise AND operator & is
      1 = 00000000 00000001

The result of combining the following
     15 = 00000000 00001111
    241 = 00000000 11110001
using the bitwise inclusive OR opertor | is
    255 = 00000000 11111111

The result of combining the following
    139 = 00000000 10001011
    199 = 00000000 11000111
using the bitwise exclusive OR operator ^ is
     76 = 00000000 01001100

The one's complement of
  21845 = 01010101 01010101
is
  43690 = 10101010 10101010
```

Fig. 10.10 Salida correspondiente al programa de la figura 10.9.

El operador de complemento *a nivel de bits* (~) define todos los bits `1` existentes en su operando a `0` en el resultado y define todos los bits `0` a `1` en el resultado —o de otra forma conocido como "tomar el *complemento a uno* del valor". En la figura 10.9 la variable **number1** es asignada al valor `21845 (01010101 01010101)`. Cuando se evalúa la expresión **~number1** el resultado es `(10101010 10101010)`.

Bit 1	Bit 2	Bit 1 | Bit 2
0	0	0
1	0	1
0	1	1
1	1	1

Fig. 10.11 Resultados de combinar dos bits mediante el operador OR inclusivo a nivel de bits |.

Bit 1	Bit 2	Bit 1 ^ Bit 2
0	0	0
1	0	1
0	1	1
1	1	0

Fig. 10.12 Resultados de combinar dos bits mediante el operador OR exclusivo a nivel de bits ^.

El programa de la figura 10.13 demuestra el operador de desplazamiento a la izquierda (<<) así como el operador de desplazamiento a la derecha (>>). La función **displayBits** se utiliza para imprimir los valores enteros **unsigned**.

El operador de desplazamiento a la izquierda (<<) desplaza los bits de su operando izquierdo hacia la izquierda, en el número de bits especificados en su operando derecho. Los bits desalojados a la derecha serán remplazados con 0s; los 1s que se desplazan hacia la izquierda se pierden. En el programa de la figura 10.13, la variable **number1** es asignada al valor **960 (00000011 11000000)**. El resultado de desplazar a la izquierda la variable **number1** 8 bits en la expresión **number1 << 8** es **49152 (11000000 00000000)**.

El operador de desplazamiento a la derecha (>>) desplaza los bits de su operando izquierda hacia la derecha, en el número de bits especificado por su operador derecho. Ejecutar un desplazamiento a la derecha en un entero **unsigned** hace que los bits desalojados a la izquierda sean remplazados por 0s; los 1s desplazados hacia la derecha se pierden. En el programa de la figura 10.13, el resultado de desplazar hacia la derecha **number1** en la expresión **number1>> 8** es **3 (00000000 00000011)**.

Error común de programación 10.14

El resultado de desplazar un valor queda indefinido si el operando derecho es negativo o si el operando derecho es más grande que el número de bits en el cual se almacena el operando izquierdo.

Sugerencia de portabilidad 10.7

El desplazamiento a la derecha es dependiente de la máquina. Desplazar a la derecha un entero signado, en algunas máquinas llena los bits desalojados con ceros y en otras con 1s.

Cada operador a nivel de bits (a excepción del operador de complemento a nivel de bits) tiene un operador de asignación correspondiente. Estos *operadores de asignación* a nivel de bits se muestran en la figura 10.14, y se utilizan de forma similar a los operadores de asignación aritméticos, presentados en el capítulo 3.


```
/* Using the bitwise shift operators */
#include <stdio.h>

void displayBits(unsigned);

main()
{
   unsigned number1 = 960;

   printf("\nThe result of left shifting\n");
   displayBits(number1);
   printf("8 bit positions using the ");
   printf("left shift operator << is\n");
   displayBits(number1 << 8);

   printf("\nThe result of right shifting\n");
   displayBits(number1);
   printf("8 bit positions using the ");
   printf("right shift operator >> is\n");
   displayBits(number1 >> 8);
   return 0;
}

void displayBits(unsigned value)
{
   unsigned c, displayMask = 1 << 15;

   printf("%7u = ", value);

   for (c = 1; c <= 16; c++) {
      putchar(value & displayMask ? '1' : '0');
      value <<= 1;

      if (c % 8 == 0)
         putchar(' ');
   }

   putchar('\n');
}
```

```
The result of left shifting
    960 = 00000011 11000000
8 bit positions using the left shift opertor << is
  49152 = 11000000 00000000

The result of right shifting
    960 = 00000011 11000000
8 bit positions using the right shift operator >> is
      3 = 00000000 00000011
```

Fig. 10.13 Cómo utilizar los operadores de desplazamiento a nivel de bits.

Operadores de asignación a nivel de bits	
`&=`	Operador de asignación AND a nivel de bits.
`|=`	Operador de asignación OR inclusivo a nivel de bits.
`^=`	Operador de asignación OR exclusivo a nivel de bits.
`<<=`	Operador de asignación de desplazamiento a la izquierda.
`>>=`	Operador de asignación de desplazamiento a la derecha.

Fig. 10.14 Los operadores de asignación a nivel de bits.

En la figura 10.15 se muestra la precedencia y asociatividad de los varios operadores presentados hasta este punto en el texto. Se muestran de arriba hacia abajo, en orden decreciente de precedencia.

10.10 Campos de bits

C proporciona la capacidad de especificar o definir el número de bits en el cual se almacena un miembro **unsigned** o **int** de una estructura o de una unión —conocidos como un *campo de bits*. Los campos de bits le permiten una mejor utilización de la memoria, al almacenar datos en el mínimo número de bits requeridos. Los miembros de campos de bits *deben* ser declarados como **int** o **unsigned**.

Operador	Asociatividad	Tipo
`() [] . ->`	de izquierda a derecha	el mas alto
`+ - ++ -- ! (tipo) & * ~ sizeof`	de derecha a izquierda	unario
`* / %`	de izquierda a derecha	multiplicativo
`+ -`	de izquierda a derecha	aditivo
`<< >>`	de izquierda a derecha	de desplazamiento
`< <= > >=`	de izquierda a derecha	relacional
`== !=`	de izquierda a derecha	igualdad
`&`	de izquierda a derecha	AND a nivel de bits
`^`	de izquierda a derecha	negación
`|`	de izquierda a derecha	OR a nivel de bits
`&&`	de izquierda a derecha	AND lógico
`||`	de izquierda a derecha	OR lógico
`?:`	de derecha a izquierda	condicional
`= += -= *= /= %= &= |= ^= <<= >>=`	de derecha a izquierda	asignación
	de izquierda a derecha	coma

Fig. 10.15 Precedencia y asociatividad de operadores.

Sugerencia de rendimiento 10.3

Los campos de bits ayudan a ahorrar almacenamiento.

Considere la siguiente definición de estructura:

```
struct bitCard {
   unsigned face : 4;
   unsigned suit : 2;
   unsigned color : 1;
};
```

La definición contiene tres campos de bits **unsigned** —**face**, **suit**, y **color**— utilizadas para representar una carta de un mazo de 52 cartas. Se declara un campo de bits, haciendo seguir a un nombre de miembro **unsigned** o **int** con un signo de dos puntos (**:**) y una constante entera que representa el *ancho* del campo, es decir, el número de bits en el cual queda almacenado el miembro. La constante que representa el ancho debe ser un entero entre 0 y el número total de bits utilizados para almacenar un **int** en su sistema. Nuestros ejemplos fueron probados en una computadora con enteros de dos bytes (16 bits).

La definición de estructura anterior indica que el miembro **face** está almacenada en 4 bits, el miembro **suit** en 2 bits y el miembro **color** en 1 bit. El número de bits se basa en el rango deseado de valores correspondiente a cada miembro de estructura. El miembro **face** almacena valores entre **0** (Ace) y **12** (Rey) —4 bits pueden almacenar un valor entre 0 y 15. El miembro **suit** almacena valores entre **0** y **3** (**0** = Diamantes, **1** = Corazones, **2** = Tréboles y **3** = Espadas)— 2 bits pueden almacenar un valor entre **0** y **3**. Finalmente, el miembro **color** almacena ya sea **0** (Rojo) o **1** (Negro) —1 bit puede almacenar ya sea **0** ó **1**.

El programa de la figura 10.16 (cuya salida se muestra en la figura 10.17), crea el arreglo **deck**, que contiene 52 estructuras **struct bitCard**. La función **fillDeck** inserta las 52 cartas en el arreglo **deck**, y la función **deal** imprime las 52 cartas. Note que se tiene acceso a los miembros de campos de bits de las estructuras exactamente como cualquier otro miembro de estructura. El miembro **color** se incluye para tener la posibilidad de indicar el color de la carta en un sistema que permita despliegues en color.

Es posible especificar un *campo de bits sin nombre*, en cuyo caso el campo se utiliza en la estructura como un *relleno*. Por ejemplo, la definición de estructura

```
struct example {
   unsigned a : 13;
   unsigned   : 3;
   unsigned b : 4;
};
```

usa como relleno un campo de 3 bits sin nombre —en estos tres bits no se puede almacenar nada. El miembro **b** (en nuestra computadora de palabras de 2 bytes) se almacena en otra unidad de almacenamiento.

Un *campo de bits sin nombre con ancho cero*, se utiliza para alinear el siguiente campo de bits en el límite de la nueva unidad de almacenamiento. Por ejemplo, la definición de estructura

```
struct example {
   unsigned a : 13;
   unsigned   : 0;
   unsigned b : 4;
};
```

utiliza un campo sin nombre de `0` bits para saltarse los bits restantes (tantos como existan) de la unidad de almacenamiento en la cual está almacenado **a**, y alinear **b** con el límite de la siguiente unidad de almacenamiento.

Sugerencia de portabilidad 10.8

Las manipulaciones de campos de bits son dependientes de la máquina. Por ejemplo, algunas computadoras permiten que los campos de bits crucen límites de palabras, en tanto que otras no lo permiten.

Error común de programación 10.15

Intentar tener acceso a bits individuales de un campo de bits como si fueran elementos de un arreglo. Los campos de bits no son "arreglos de bits".

Error común de programación 10.16

Intentar tomar la dirección de un campo de bits (el operador & no puede ser utilizado en conjunción con campos de bits, porque éstos no tienen direcciones).

Sugerencia de rendimiento 10.4

Aunque los campos de bits ahorran espacio, su uso puede hacer que el compilador genere código en lenguaje de máquina de ejecución más lenta. Esto ocurre debido a que tener acceso a sólo porciones de una unidad de almacenamiento direccionable toma más operaciones en lenguaje de máquina. Esto es uno de los muchos ejemplos de los tipos de intercambios espacio-tiempo que ocurren en la ciencia de la computación.

10.11 Constantes de enumeración

C proporciona un tipo final, definido por el usuario, conocido como una *enumeración*. Una enumeración, introducida por la palabra reservada **enum**, es un conjunto de constantes enteras representadas por identificadores. Estas *constantes de enumeración* son, en efecto, constantes simbólicas, cuyos valores pueden ser definidos automáticamente. Los valores de un **enum** se inician con `0`, a menos de que se defina de otra manera, y se incrementan en `1`. Por ejemplo, la enumeración

```
enum months {JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP,
             OCT, NOV, DEC};
```

crea un nuevo tipo en **enum months**, en el cual los identificadores son definidos automáticamente a los enteros `0` a `11`. Para numerar los meses 1 a 12, utilice la enumeración siguiente:

```
enum months {JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG,
             SEP, OCT, NOV, DEC};
```

Dado que el primer valor de la enumeración anterior se define explícitamente en `1`, los valores subsiguientes se incrementan en `1` dando como resultado los valores `1` hasta `12`. Los identificadores en una enumeración deben ser únicos. En una enumeración el valor de cada constante de numeración puede ser establecido explícitamente en la definición, mediante la asignación de un valor al identificador. Varios miembros de una enumeración pueden tener el mismo valor entero. En el programa de la figura 10.18, la variable de enumeración **month** se utiliza en una estructura **for** para imprimir los meses del año del arreglo **monthName**. Note que hemos hecho **monthName[0]** la cadena vacía `""`. Algunos programadores pudieran preferir definir **monthName[0]** a un valor como `***ERROR***` para indicar que ocurrió un error lógico.


```
/* Example using a bit field */

#include <stdio.h>

struct bitCard {
   unsigned face : 4;
   unsigned suit : 2;
   unsigned color : 1;
};

typedef struct bitCard Card;

void fillDeck(Card *);
void deal(Card *);

main()
{
   Card deck[52];

   fillDeck(deck);
   deal(deck);

   return 0;
}

void fillDeck(Card *wDeck)
{
   int i;

   for (i = 0; i <= 51; i++) {
      wDeck[i].face = i % 13;
      wDeck[i].suit = i / 13;
      wDeck[i].color = i / 26;
   }
}

/* Function deal prints the cards in two column format */
/* Column 1 contains cards 0-25 subscripted with k1 */
/* Column 2 contains cards 26-51 subscripted with k2 */

void deal(Card *wDeck)
{
   int k1, k2;

   for (k1 = 0, k2 = k1 + 26; k1 <= 25; k1++, k2++) {
      printf("Card:%3d  Suit:%2d  Color:%2d   ",
             wDeck[k1].face, wDeck[k1].suit, wDeck[k1].color);
      printf("Card:%3d  Suit:%2d  Color:%2d\n",
             wDeck[k2].face, wDeck[k2].suit, wDeck[k2].color);
   }
}
```

Fig. 10.16 Cómo utilizar campos de bits para almacenar un mazo de cartas.

```
Card:  0 Suit: 0 Color: 0 Card:  0 Suit: 2 Color: 1
Card:  1 Suit: 0 Color: 0 Card:  1 Suit: 2 Color: 1
Card:  2 Suit: 0 Color: 0 Card:  2 Suit: 2 Color: 1
Card:  3 Suit: 0 Color: 0 Card:  3 Suit: 2 Color: 1
Card:  4 Suit: 0 Color: 0 Card:  4 Suit: 2 Color: 1
Card:  5 Suit: 0 Color: 0 Card:  5 Suit: 2 Color: 1
Card:  6 Suit: 0 Color: 0 Card:  6 Suit: 2 Color: 1
Card:  7 Suit: 0 Color: 0 Card:  7 Suit: 2 Color: 1
Card:  8 Suit: 0 Color: 0 Card:  8 Suit: 2 Color: 1
Card:  9 Suit: 0 Color: 0 Card:  9 Suit: 2 Color: 1
Card: 10 Suit: 0 Color: 0 Card: 10 Suit: 2 Color: 1
Card: 11 Suit: 0 Color: 0 Card: 11 Suit: 2 Color: 1
Card: 12 Suit: 0 Color: 0 Card: 12 Suit: 2 Color: 1
Card:  0 Suit: 1 Color: 0 Card:  0 Suit: 3 Color: 1
Card:  1 Suit: 1 Color: 0 Card:  1 Suit: 3 Color: 1
Card:  2 Suit: 1 Color: 0 Card:  2 Suit: 3 Color: 1
Card:  3 Suit: 1 Color: 0 Card:  3 Suit: 3 Color: 1
Card:  4 Suit: 1 Color: 0 Card:  4 Suit: 3 Color: 1
Card:  5 Suit: 1 Color: 0 Card:  5 Suit: 3 Color: 1
Card:  6 Suit: 1 Color: 0 Card:  6 Suit: 3 Color: 1
Card:  7 Suit: 1 Color: 0 Card:  7 Suit: 3 Color: 1
Card:  8 Suit: 1 Color: 0 Card:  8 Suit: 3 Color: 1
Card:  9 Suit: 1 Color: 0 Card:  9 Suit: 3 Color: 1
Card: 10 Suit: 1 Color: 0 Card: 10 Suit: 3 Color: 1
Card: 11 Suit: 1 Color: 0 Card: 11 Suit: 3 Color: 1
Card: 12 Suit: 1 Color: 0 Card: 12 Suit: 3 Color: 1
```

Fig. 10.17 Salida del programa de la figura 10.16.

Error común de programación 10.17

Es un error de sintaxis asignar un valor a una constante de numeración después de haber sido definida.

Práctica sana de programación 10.6

Utilice sólo letras mayúsculas en los nombres de las constantes de numeración. Esto hace que estas constantes destaquen en un programa y le recuerdan al programador que las constantes de numeración no son variables.

Resumen

- Las estructuras son colecciones de variables relacionadas, algunas veces conocidas como agregados, bajo un solo nombre.
- Las estructuras pueden contener variables de varios tipos de datos.
- La palabra reservada **`struct`** empieza toda definición de estructura. Dentro de las llaves de la definición de estructura, están las declaraciones de los miembros de la estructura.
- Los miembros de la misma estructura deben de tener nombres únicos.


```
/* Using an enumeration type */
#include <stdio.h>

enum months {JAN = 1, FEB, MAR, APR, MAY, JUN,
             JUL, AUG, SEP, OCT, NOV, DEC};

main()
{
   enum months month;
   char *monthName[] = {"", "January", "February", "March",
                        "April", "May", "June", "July",
                        "August", "September", "October",
                        "November", "December"};

   for (month = JAN; month <= DEC; month++)
      printf("%2d%11s\n", month, monthName[month]);

   return 0;
}
```

```
 1    January
 2   February
 3      March
 4      April
 5        May
 6       June
 7       July
 8     August
 9  September
10    October
11   November
12   December
```


Fig. 10.18 Cómo utilizar una enumeración.

- Una definición de estructura crea un nuevo tipo de datos que puede ser utilizado para declarar variables.
- Existen dos métodos para declarar variables de estructura. El primer método es declarar las variables en una declaración, como se hace con las variables de otros tipos de datos, utilizando **`struct tagName`** como el tipo. El segundo método es incluir las variables encerradas en las llaves de la definición de estructura y en el punto y coma que termina la definición de estructura.
- El nombre de rótulo de la estructura es opcional. Si la estructura se define sin un nombre de rótulo, las variables del tipo de datos derivados deben de ser declarados en la definición de estructura, y no se pueden declarar otras variables del nuevo tipo de estructura.
- Una estructura puede ser inicializada con una lista de incialización, siguiendo el nombre de la variable en la declaración de estructura con un signo igual y una lista de inicializadores, separados por comas y encerrados en llaves. Si en la lista existen menos inicializadores que

miembros en la estructura, los miembros restantes serán automáticamente inicializados a cero (o a **NULL** si el miembro es un apuntador).

- Estructuras completas pueden ser asignadas a variables de estructura del mismo tipo.
- Una variable de estructura puede ser inicializada con una variable de estructura del mismo tipo.
- El operador de miembro de estructura se utiliza al tener acceso a un miembro de una estructura vía el nombre de la variable de estructura.
- El operador de apuntador de estructura —creado con un signo de menos (-) y un signo de mayor que (>)— se utiliza al tener acceso a un miembro de una estructura vía un apuntador a la estructura.
- Las estructuras y los miembros individuales de las estructuras se pasan a las funciones en llamada por valor.
- Para pasar a una estructura llamada por referencia, pase la dirección de la variable de estructura.
- Un arreglo de estructura se pasa automáticamente en llamada por referencia.
- Para pasar un arreglo en llamada por valor, cree una estructura con el arreglo como un miembro.
- Crear un nuevo nombre utilizando **typedef** no crea un nuevo tipo; crea un nombre que es un seudónimo del tipo anteriormente definido.
- Una unión es un tipo de datos derivado, cuyos miembros comparten el mismo espacio de almacenamiento. Los miembros pueden ser de cualquier tipo.
- El almacenamiento reservado para una unión debe ser lo suficientemente grande para almacenar su miembro mayor. En la mayoría de los casos, las uniones contienen dos o más tipos de datos. Solamente un miembro y, por lo tanto, un solo tipo de datos, pueden ser referenciados en un momento dado.
- Una unión se declara con la palabra reservada **union**, en el mismo formato que una estructura.
- Una unión puede ser inicializada únicamente con el valor del tipo de su primer miembro.
- El operador AND a nivel de bits (**&**) toma dos operandos integrales. Un bit en el resultado se define a 1 si los bits correspondientes en cada uno de los operandos son 1.
- Se utilizan máscaras para ocultar algunos bits mientras se conservan otros.
- El operador OR inclusivo a nivel de bits (|) toma dos operandos. Un bit en el resultado se define a 1 si el bit correspondiente en cualquier operando está definido a 1.
- Cada uno de los operadores a nivel de bits (a excepción del operador de complemento a nivel de bits unario) tiene un operador de asignación correspondiente.
- El operador OR exclusivo a nivel de bits (^) toma dos operandos. Un bit en el resultado se define a 1 si exactamente 1 de los bits correspondientes en los dos operandos está definido a 1.
- El operador de desplazamiento de izquierda (<<) desplaza los bits de su operando izquierdo hacia la izquierda en el número de bits especificados por su operando derecho. Los bits desalojados a la derecha se remplazan con 0s.
- El operador de desplazamiento a la derecha (>>) desplaza los bits de su operando izquierdo hacia la derecha en el número de bits especificado en su operando derecho. El ejecutar un desplazamiento a la derecha en un entero no signado hace que los bits desocupados a la izquierda sean remplazados por cero. Los bits desocupados en enteros signados podrían ser remplazados con 0s o con 1s —esto dependerá de la máquina.

- El operador de complemento a nivel de bits (~) toma un operando e invierte sus bits —esto produce el complemento a uno del operando.
- Los campos de bits reducen la utilización del almacenamiento al almacenar datos en el número mínimo de bits requeridos.
- Los miembros de campos de bits deben de ser declarados como **int** o **unsigned.**
- Un campo de bits se declara haciendo seguir un nombre de miembro **unsigned** o **int** con un punto y coma, y con el ancho del campo de bits.
- El ancho del campo de bits debe ser una constante entera, entre 0 y el número total de bits utilizados para almacenar una variable **int** en su sistema.
- Si un campo de bits se especifica sin nombre, el campo se utilizará como relleno en la estructura.
- Un campo de bits sin nombre con ancho **0** se utiliza para alinear el siguiente campo de bits en el límite de la siguiente palabra de máquina.
- Una enumeración, designada con la palabra reservada **enum**, es un conjunto de enteros que se representan mediante identificadores. Los valores de un **enum** se inician con **0**, a menos de que se especifique lo contrario, y son siempre incrementados en **1**.

Terminología

^ operador OR exclusivo a nivel de bits
^= operador de asignación OR exclusivo a nivel de bits
~ operador de complemento a uno
& operador AND a nivel de bits
&= operador de asignación AND a nivel de bits
| operador OR inclusivo a nivel de bits.
|= operador de asignación OR inclusivo a nivel de bits.
<< operador de desplazamiento a la izquierda
<<= operador de asignación de desplazamiento a la izquierda
>> operador de desplazamiento a la derecha
>>= operador de asignación de desplazamiento a la derecha
como tener acceso a miembros de estructuras
agregados
arreglo de estructuras
campo de bits
operador a nivel de bits
complementar
tipo derivado
enumeración
constante de enumeración
inicialización de estructuras
desplazamiento a la izquierda
máscara
enmascarar bits
miembro
nombre de miembro
estructuras anidadas
complemento a uno
relleno
apuntador a una estructura
tipos de datos definidos por el programador
registro
desplazamiento a la derecha
estructura autorreferenciada
desplazar
intercambios espacio-tiempo
struct
asignación de estructura
declaración de estructura
definición de estructura
inicialización de estructura
operador de miembro de estructura (punto) (.)
nombre de estructura
operador de apuntador de estructura (flecha) (->)
etiqueta de estructura
tipo de estructura
nombre de etiqueta
typedef
union
campo de bits sin nombre
ancho de un campo de bits
campo de bits de ancho cero

Errores comunes de programación

10.1 Olvidar el punto y coma que da por terminada una definición de estructura.
10.2 Asignar una estructura de un tipo a una estructura de un tipo distinto.
10.3 Es un error de sintaxis comparar estructuras, debido a diferentes requisitos de alineación en los diferentes sistemas.
10.4 Insertar un espacio entre el signo de - y el signo de > del operador de apuntador de estructura (o insertar espacios entre los componentes de cualquier otro operador múltiple de teclado, a excepción de `?:`).
10.5 Intentar referirse a un miembro de una estructura utilizando únicamente el nombre de dicho miembro.
10.6 No utilizar paréntesis al referirse a un miembro de estructura utilizando un apuntador y el operador de miembro de estructura (por ejemplo `*aptr.suit`, es un error de sintaxis).
10.7 Suponer que las estructuras, como los arreglos, se pasan automáticamente en llamada por referencia, e intentar modificar los valores de estructura del llamador en la función llamada.
10.8 Olvidar incluir el subíndice de arreglo al referirse a estructuras individuales de un arreglo de estructuras.
10.9 Es un error lógico referenciar con el tipo equivocado, datos en una unión almacenados con un tipo distinto.
10.10 Es un error de sintaxis comparar uniones, debido a los diferentes requisitos de alineación en varios sistemas.
10.11 Inicializar una unión en una declaración con un valor cuyo tipo es diferente del tipo del primer miembro de la unión.
10.12 Usar el operador lógico AND (`&&`), en lugar del operador AND a nivel de bits (`&`), y viceversa.
10.13 Usar el operador OR lógico (`||`), en lugar del operador OR a nivel de bits (`|`), y viceversa.
10.14 El resultado de desplazar un valor queda indefinido si el operando derecho es negativo o si el operando derecho es más grande que el número de bits en el cual se almacena el operando izquierdo.
10.15 Intentar tener acceso a bits individuales de un campo de bits como si fueran elementos de un arreglo. Los campos de bits no son "arreglos de bits".
10.16 Intentar tomar la dirección de un campo de bits (el operador `&` no puede ser utilizado en conjunción con campos de bits, porque estos no tienen direcciones).
10.17 Es un error de sintaxis asignar un valor a una constante de numeración después de haber sido definida.

Prácticas sanas de programación

10.1 Al crear un tipo de estructura proporcione un nombre de rótulo de estructura. El nombre de rótulo de estructura es conveniente más adelante en el programa para la declaración de nuevas variables de este tipo de estructura.
10.2 Seleccionar un nombre de rótulo de estructura significativo ayuda a autodocumentar el programa.
10.3 Evite utilizar los mismos nombres para miembros de estructura de distintos tipos. Ello es permitido, pero podría causar confusión.
10.4 No deje espacios alrededor de los operadores `->` y `.` ya que ayuda a enfatizar que las expresiones en las cuales los operadores están contenidos son esencialmente nombres individuales de variables.
10.5 Ponga los nombres `typedef` en mayúsculas, para enfatizar que esos nombres son sinónimos de otros nombres de tipo.
10.6 Utilice sólo letras mayúsculas en los nombres de las constantes de numeración. Esto hace que estas constantes destaquen en un programa y le recuerdan al programador que las constantes de numeración no son variables.

Sugerencia de portabilidad

10.1 Dado que depende de la máquina el tamaño de los elementos de datos de un tipo particular, y debido a que las consideraciones de alineación de almacenamiento también son dependientes de la máquina, entonces también lo será la representación de una estructura.
10.2 Utilice `typedef` para ayudar a hacer más portátil un programa.
10.3 Si en una unión los datos se almacenan como de un tipo y se referencían como de otro tipo, los resultados serán dependientes de la instalación.
10.4 La cantidad de almacenamiento requerido para almacenar una unión es dependiente de la instalación.
10.5 En algunas uniones no pueden aplicarse fácilmente otros sistemas de computación. Si una unión es portable o no, a menudo depende de la alineación de almacenamiento requerida para los tipos de datos de miembros de unión en un sistema dado.
10.6 Las manipulaciones de datos a nivel de bits son dependientes de la máquina.
10.7 El desplazamiento a la derecha es dependiente de la máquina. Desplazar a la derecha un entero signado, en algunas máquinas llena los bits desalojados con ceros y en otras con `1s`.
10.8 Las manipulaciones de campos de bits son dependientes de la máquina. Por ejemplo, algunas computadoras permiten que los campos de bits crucen límites de palabras, en tanto que otras no lo permiten.

Sugerencia de rendimiento

10.1 Es más eficaz pasar estructuras en llamada por referencia que pasar estructuras en llamada por valor (ya que esto último requiere que toda la estructura se copie).
10.2 Las uniones ahorran almacenamiento.
10.3 Los campos de bits ayudan a ahorrar almacenamiento.
10.4 Aunque los campos de bits ahorran espacio, su uso puede hacer que el compilador genere código en lenguaje de máquina de ejecución más lenta. Esto ocurre debido a que tener acceso a sólo porciones de una unidad de almacenamiento direccionable toma más operaciones en lenguaje de máquina. Esto es uno de los muchos ejemplos de los tipos de intercambios espacio-tiempo que ocurren en la ciencia de la computación.

Observación de ingeniería de software

10.1 Al igual que en una declaración `struct`, una declaración `union` simplemente crea un tipo nuevo. Colocar una declaración `union` o `struct` fuera de cualquier función no crea una variable global.

Ejercicios de autoevaluación

10.1 Llene los espacios en blanco con cada uno de los siguientes:
a) Una _____ es una colección de variables relacionadas bajo un nombre.
b) Una _____ es una colección de variables bajo un nombre en la cual las variables comparten el mismo almacenamiento.
c) Los bits en el resultado de una expresión utilizando el operador _____ se definen a 1 si los bits correspondientes en cada operando están establecidos en 1. De lo contrario, los bits se definen a cero.
d) Las variables declaradas en una definición de estructura se conocen como sus _____.
e) Los bits en el resultado de una expresión utilizando el operador _____ se definen a 1, si por lo menos uno de los bits correspondientes en cualquiera de los operandos está definido a 1. De lo contrario los bits se establecen a cero.
f) La palabra reservada _____ introduce una declaración de estructura.

g) La palabra reservada _____ se utiliza para crear un seudónimo de un tipo de datos previamente definido.
h) Los bits en el resultado de una expresión utilizando el operador _____ se definen a 1, si exactamente uno de los bits correspondientes en cada uno de los operandos está definido a uno. De lo contrario, los bits se definen a cero.
i) El operador AND a nivel de bits `&`, se utiliza a menudo para _____ a los bits, esto es para seleccionar ciertos bits a partir de una cadena de bits, en tanto que se ponen a cero los demás.
j) La palabra reservada _____ se utiliza para introducir una definición de unión.
k) El nombre de la estructura se conoce como el _____ de la estructura.
l) Se tiene acceso a un miembro de estructura ya sea con el operador _____ o con el operador _____.
m) Los operadores _____ y _____ se utilizan para desplazar los bits de un valor hacia la izquierda o hacia la derecha, respectivamente.
n) Una _____ es un conjunto de enteros representados por identificadores.

10.2 Indique si cada uno de los siguientes es verdadero o falso. Si es falso, explique por qué.
a) Las estructuras pueden contener únicamente un tipo de datos.
b) Dos uniones pueden ser comparadas entre sí para determinar si son iguales.
c) El nombre del rótulo de una estructura es opcional.
d) Los miembros de diferentes estructuras deben tener nombres únicos.
e) La palabra reservada `typedef` se utiliza para definir nuevos tipos de datos.
f) Las estructuras se pasan siempre a las funciones en llamada por referencia.
g) Las estructuras no pueden ser comparadas.

10.3 Escriba un solo enunciado o un conjunto de enunciados para llevar a cabo cada uno de los siguientes:
a) Defina una estructura llamada `part` conteniendo la variable `int partNumber`, y el arreglo `char partName` cuyos valores pudieran ser de hasta 25 caracteres de largo.
b) Defina `Part` como un sinónimo para el tipo `struct part`.
c) Utilice `Part` para declarar la variable `a` que sea de tipo `struct part`, el arreglo `b[10]` que sea del tipo `struct part` y la variable `ptr` que sea del tipo apuntador a `struct part`.
d) Lea un número de parte y un nombre de parte del teclado a los miembros individuales de la variable `a`.
e) Asigne los valores del miembro de la variable `a` al elemento 3 del arreglo `b`.
f) Asigne las direcciones del arreglo `b` a la variable de apuntador `ptr`.
g) Imprima los valores de miembros del elemento 3 del arreglo `b`, utilizando la variable `ptr` y el operador de apuntador de estructura para referirse a los miembros.

10.4 Encuentre el error en cada uno de los siguientes:
a) Suponga que `struct car` ha sido definido conteniendo dos apuntadores al tipo `char`, es decir, `face` y `suit`. También, la variable `c` ha sido declarada del tipo `struct card` y la variable `cPtr` ha sido declarada ser del tipo apuntador a `struct card`. La variable `cPtr` ha sido asignada a la dirección de `c`.

```
printf("%s\n", *cPtr->face);
```

b) Suponga que `struct card` ha sido definida conteniendo dos apuntadores de tipo `char`, es decir `face` y `suit`. También, el arreglo `hearts[13]` ha sido declarado ser del tipo `struct card`. El siguiente enunciado debería imprimir el miembro `face` del elemento 10 del arreglo.

```
printf("%s\n", hearts.face);
```

c)
```
union values {
   char w;
   float x;
   double y;
} v = {1.27};
```


d)
```
struct person {
   char last Name[15]
   char firstName[15]
   int age;
}
```

e) Suponga que `struct person` ha sido definido como en la parte (d) pero con la corrección apropiada.

```
person d;
```

f) Suponga que la variable `p` ha sido declarada como del tipo `struct person` y la variable `c` ha sido declarada del tipo `struct card`.

```
p = c;
```

Respuestas a los ejercicios de autoevaluación

10.1 a) estructura. b) unión. c) AND a nivel de bits (`&`). d) miembros. e) OR inclusivo a nivel de bits (`|`). f) `struct`. g) `typedef`. h) OR exclusivo a nivel de bits (`^`). i) máscara. j) `union`. k) etiqueta. l) miembro de estructura, apuntador de estructura. m) operador de desplazamiento a la izquierda (`<<`), operador de desplazamiento a la derecha (`>>`). n) enumeración.

10.2 a) Falso. Una estructura puede contener muchos tipos de datos.
b) Falso. Las uniones no pueden ser comparadas, debido a los mismos problemas de alineación asociados con las estructura.
c) Verdadero.
d) Falso. Los miembros de estructuras separadas pueden tener los mismos nombres, pero los miembros de una misma estructura deben tener nombres únicos.
e) Falso. La palabra reservada `typedef` se utiliza para definir nuevos nombres (sinónimos) para tipos de datos definidos previamente.
f) Falso. Las estructuras son siempre pasadas a las funciones en llamada por valor.
g) Verdadero, debido a los problemas de alineación.

10.3 a)
```
struct part {
int partNumber;
char partName[25];
};
```
b) `typedef struct part Part;`
c) `Part a, b[10], *ptr;`
d `scanf("%d%s", &a.partNumber, &a.partName};`
e) `b[3] = a;`
f) `ptr = b;`
g)
```
printf("%d %s\n", (ptr + 3)->partNumber,
(ptr + 3)-partName);
```

10.4 a) Error: los paréntesis que deberían de encerrar a `*cPtr` han sido omitidos, causando que sea incorrecto el orden de evaluación de la expresión.
b) Error: El subíndice del arreglo ha sido omitido. La expresión debería ser `hearts[10].face`.
c) Error: Una unión solamente puede ser inicializada con un valor que tenga el mismo tipo que el primer miembro de la misma.
d) Error: se requiere de un punto y coma para determinar una definición de estructura.
e) Error: la palabra reservada `struct` fue omitida de la declaración de variable.
f) Error: las variables de tipos de estructuras diferentes no pueden ser asignadas unos a los otros.

Ejercicios

10.5 Dé la definición de cada una de las siguientes estructuras y uniones:

a) La estructura `inventory` que contiene el arreglo de caracteres `partName[30]`, en entero `partNumber`, el punto flotante `price`, el entero `stock`, y el entero `reorder`.
b) La unión `data` que contiene `char c, short s, long l, float f` y `double d`.
c) Una estructura llamada `address` que contiene los arreglos de caracteres `streetAddress[25], city[20], state[3]`, y `zipCode[6]`.
d) La estructura `student` que contiene los arreglos `firstName[15]` y `lastName[15]`, y la variable `homeAddress` del tipo `struct address` correspondiente a la parte (c).
e) La estructura `test` que contenga 16 campos de bits con anchos de 1 bit. Los nombres de los campos de bits son las letras `a` a la `p`.

10.6 Dadas las siguientes definiciones de estructuras y las declaraciones de variables,

```
struct customer {
    char lastName[15];
    char firstName[15];
    int customerNumber;

    struct {
       char phoneNumber[11];
       char address[50];
       char city[15];
       char state[3];
       char zipCode[6];
    } personal;

} customerRecord, *customerPtr;

customerPtr = &customerRecord;
```

escriba una expresión por separado que pueda ser utilizada para tener acceso a los miembros de la estructura en cada una de las partes siguientes.

a) El miembro `lastName` de la estructura `customerRecord`.
b) El miembro `lastName` de la estructura a la cual apunta `customerPtr`.
c) El miembro `firstName` de la estructura `customerRecord`.
d) El miembro `firstName` de la estructura a la cual apunta `customerPtr`.
e) El miembro `customerNumber` de la estructura `customerRecord`.
f) El miembro `customerNumber` de la estructura a la cual apunta `customerPtr`.
g) El miembro `phoneNumber` del miembro `personal` de la estructura `customerRecord`.
h) El miembro `phoneNumber` del miembro `personal` de la estructura a la cual apunta `customerPtr`.
i) El miembro `address` del miembro `personal` de la estructura `customerRecord`.
j) El miembro `address` del miembro `personal` de la estructura apuntada por `customerPtr`.
k) El miembro `city` del miembro `personal` de la estructura `customerRecord`.
l) El miembro `city` del miembro `personal` de la estructura a la cual apunta `customerPtr`.
m) El miembro `state` del miembro `personal` de la estructura `customerRecord`.
n) El miembro `state` del miembro `personal` de la estructura a la cual apunta `customerPtr`.
o) El miembro `zipCode` del miembro `personal` de la estructura `customerRecord`.
p) El miembro `zipCode` del miembro `personal` de la estructura a la cual apunta `customerPtr`.

10.7 Modifique el programa de la figura 10.16 para barajar las cartas utilizando un algoritmo de barajar de alto rendimiento (como se muestra en la figura 10.3). Imprima el mazo resultante en un formato de dos columnas, como en la figura 10.4. Anteceda cada carta con su color.

10.8 Crear la unión `integer` con miembros `char c, short s, int i`, y `long l`. Escriba un programa que introduzca el valor del tipo `char, short, int`, y `long`, y que almacene los valores en las variables de unión del tipo `union integer`. Cada variable de unión deberá ser impresa como un `char`, un `short`, un `int` y un `long`. ¿Se imprimen siempre los valores en forma correcta?

10.9 Crear la unión `floatingPoint` con los miembros `float f, double d`, y `long double l`. Escriba un programa que introduzca valor del tipo `float, double` y `long double`, y almacene los valores en variables de unión del tipo `unionfloatingPoint`. Cada variable de unión deberá imprimirse como un `float`, un `double` y un `long double`. ¿Se imprimen siempre los valores correctamente?

10.10 Escriba un programa que desplace una variable entera 4 bits hacia la derecha. El programa deberá imprimir el entero en bits antes y después de la operación de desplazamiento. ¿Su sistema coloca ceros, o bien unos en los bits desalojados?

10.11 Si su computadora utiliza enteros de 4 bytes, modifique el programa de la Figura 10.7, de tal forma que funcione con enteros de 4 bytes.

10.12 El desplazar a la izquierda un entero `unsigned` en 1 bit es equivalente a multiplicar el valor por 2, Escriba la función `power2` que toma dos argumentos enteros `number` y `pow` y calcule

```
number * 2^pow
```

Utilice el operador de desplazamiento para calcular el resultado. El programa deberá imprimir los valores como enteros y como bits.

10.13 El operador de desplazamiento a la izquierda puede ser utilizado para empacar dos valores de caracteres en una variable entera no signada de 2 bytes. Escriba un programa que introduzca dos caracteres del teclado y que los pase a la función `packCharacters`. Para empacar dos caracteres en una variable entera `unsigned`, asigne el primer carácter a la variable `unsigned`, desplace la variable a la izquierda en 8 posiciones de bits, y combine la variable `unsigned` con el segundo carácter utilizando el operador OR inclusivo a nivel de bits. El programa deberá extraer los caracteres en su formato de bits, antes y después de haber sido empacados en el entero `unsigned`, para probar que los caracteres de hecho han sido empacados correctamente en la variable `unsigned`.

10.14 Utilizando el operador de desplazamiento a la derecha, el operador AND a nivel de bits y una máscara, escriba la función `unpackCharacters` que toma el entero `unsigned` del Ejercicio 10.13 y lo desempaca en dos caracteres. Para desempacar dos caracteres de un entero `unsigned` de 2 bytes, combine el entero `unsigned` con la máscara `65280 (11111111 00000000)` y desplace hacia la derecha el resultado en 8 bits. Asigne el valor resultante a una variable `char`. A continuación combine el entero `unsigned` con la máscara `255 (00000000 11111111)`. Asigne el resultado a otra variable `char`. El programa deberá imprimir el entero `unsigned` en bits, antes de ser desempacado, y a continuación imprimir los caracteres en bits para confirmar que fueron desempacados correctamente.

10.15 Si su sistema utiliza enteros de 4 bytes, vuelva a escribir el programa de Ejercicio 10.13 para empacar 4 caracteres.

10.16 Si su sistema utiliza enteros de 4 bytes, vuelva a escribir la función `unpackCharacters` del Ejercicio 10.14 para desempacar 4 caracteres. Crear las máscaras que necesite para desempacar los 4 caracteres desplazando hacia la izquierda el valor 255 en la variable de enmascaramiento en 8 bits 0, 1, 2 o 3 veces, (dependiendo del byte que está desempacando).

10.17 Escriba un programa que invierta el orden de los bits de un valor entero no signado. El programa deberá introducir el valor proveniente del usuario y llamar a la función **reverseBits** para imprimir los bits en orden inverso. Imprima el valor en bits tanto antes como después de la inversión de bits, para confirmar que los bits hayan sido invertidos correctamente.

10.18 Modifique la función **displayBits** de la Figura 10.7, de tal forma que resulte portátil entre sistemas, utilizando enteros de 2 bytes y sistemas de enteros de 4 bytes. *Sugerencia:* utilice el operador **sizeof** para determinar el tamaño de un entero en una máquina en particular.

10.19 El programa siguiente utiliza la función **multiple** para determinar si el entero introducido desde el teclado es un múltiplo de algún entero **x**. Examine la función múltiple y a continuación determine el valor de **x**.

```
/* This program determines if a value is a multiple of X */

#include <stdio.h>

int multiple(int);

main()
{
   int y;

   printf("Enter an integer between 1 and 32000: ");
   scanf("%d", &y);

   if (multiple(y))
      printf("%d is a multiple of X\n", y);
   else
      printf("%d is not a multiple of X\n", y);

   return 0;
}

int multiple(int num)
{
   int i, mask = 1, mult = 1;

   for (i = 1; i <= 10; i++, mask <<= 1)
      if ((num & mask) != 0) {
         mult = 0;
         break;
      }

    return mult;
}
```


10.20 ¿Qué es lo que ejecuta el siguiente programa?

```
#include <stdio.h>

int mystery(unsigned);

main()
{
   unsigned x;

   printf("Enter an integer: ");
   scanf("%u", &x);
   printf("The result is %d\n", mystery(x));
   return 0;
}

int mystery(unsigned bits)
{
   unsigned i, mask = 1 << 15, total = 0;

   for (i = 1; i <= 16; i++, bits <<= 1)
      if ((bits & mask) == mask)
         ++total;

    return total % 2 == 0 ? 1 : 0;;
}
```