5

Funciones

Objetivos

- Comprender cómo construir programas en forma modular partiendo de pequeñas partes conocidas como funciones.
- Presentar las funciones matemáticas comunes disponibles en la biblioteca estándar de C.
- Ser capaz de crear funciones nuevas.
- Comprender los mecanismos utilizados para pasar información de función a función.
- Introducir técnicas de simulación utilizando generación de números aleatorios.
- Comprender cómo escribir y utilizar funciones que se llamen a sí mismas.

La forma sigue siempre a la función.
Louis Henri Sullivan

E pluribus unum. (uno formado de muchos).
Virgilo

¡Oh! llama al ayer, pídele al tiempo que vuelva.
William Shakespeare
Richard II

Llámame Ismael.
Herman Melville
Moby Dick

Cuando me llames así, sonríe.
Owen Wister

Sinopsis

5.1 Introducción
5.2 Módulos de programa en C
5.3 Funciones matemáticas de biblioteca
5.4 Funciones
5.5 Definiciones de funciones
5.6 Prototipos de funciones
5.7 Archivos de cabecera
5.8 Cómo llamar funciones: llamada por valor y llamada por referencia
5.9 Generación de números aleatorios
5.10 Ejemplo: un juego de azar
5.11 Clases de almacenamiento
5.12 Reglas de alcance
5.13 Recursión
5.14 Ejemplo utilizando recursión: la serie Fibonacci
5.15 Recursión en comparación con iteración

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

5.1 Introducción

La mayor parte de los programas de cómputo que resuelven problemas de la vida real, son mucho mayores que los programas presentados en los primeros pocos capítulos. La experiencia ha mostrado que la mejor forma de desarrollar y mantener un programa grande es construirlo a partir de piezas menores o *módulos*, siendo cada una de ellas más fácil de manipular que el programa original. Esta técnica se conoce como *divide y vencerás*. Este capítulo describe aquellas características del lenguaje C que facilitan el diseño, implantación, operación y mantenimiento de programas grandes.

5.2 Módulos de programa en C

En C los módulos se llaman *funciones*. Por lo general en C, los programas se escriben combinando nuevas funciones que el programador escribe, con funciones "preempacadas", disponibles en la *biblioteca estándar de C*. En este capítulo analizarémos ambos tipos de funciones. La biblioteca estándar de C contiene una amplia colección de funciones para llevar a cabo cálculos matemáticos

comunes, manipulaciones con cadenas, manipulaciones con caracteres, entrada/salida, y muchas otras operaciones útiles. Esto facilita la tarea del programador, porque estas funciones proporcionan muchas de las capacidades que los programadores requieren.

Práctica sana de programación 5.1

Familiarícese con la amplia colección de funciones de la biblioteca estándar ANSI C.

Observación de ingeniería de software 5.1

Evite reinventar la rueda. Siempre que sea posible, utilice funciones estándar de biblioteca ANSI C, en vez de escribir nuevas funciones. Esto reduce el tiempo de desarrollo del programa.

Sugerencia de portabilidad 5.1

Utilizar funciones de la biblioteca estándar ANSI C auxilia a que los programas sean más portátiles.

Aunque técnicamente las funciones estándar de biblioteca no forman parte del lenguaje C, de forma invariable son proporcionadas junto con los sistemas ANSI C. Las funciones **printf**, **scanf**, y **pow**, que hemos utilizado en capítulos anteriores, son funciones estándar de biblioteca.

El programador puede escribir funciones para definir tareas específicas, que puedan ser utilizadas en muchos puntos de un programa. Estas a veces se conocen como *funciones definidas por el programador*. Los enunciados que definen la función se escriben sólo una vez, quedando los enunciados ocultos de otras funciones.

Las funciones se *invocan* mediante una *llamada de función*. La llamada de función especifica el nombre de la misma y proporciona información (en forma de *argumentos*), que la función llamada necesita a fin de llevar a cabo su tarea designada. Una analogía común de lo anterior es la administración de tipo jerárquico. Un jefe (la *función que llama* o *el llamador*) le pide al trabajador (la *función llamada*) que ejecute una tarea y cuando ésta haya sido efectuada, informe. Por ejemplo, una función jefe, que desea mostrar información en pantalla, llama la función trabajador **printf** para que ejecute esa tarea, y a continuación **printf** despliega la información y cuando ha terminado su tarea informa, o *regresa* a la función llamadora. La función jefe no sabe cómo la función trabajador ejecuta sus tareas designadas. El trabajador pudiera tener que llamar a otras funciones trabajador, y el jefe no se dará cuenta de ello. Pronto veremos cómo este "ocultamiento" de los detalles de puesta en práctica promueve una buena ingeniería de software. En la figura 5.1 se muestra la función **main**, comunicándose con varias funciones trabajador, de una forma jerárquica. Advierta que **worker1** funciona como función jefe hacia **worker4** y **worker5**. Las relaciones entre las funciones pudieran ser distintas en estructura jerárquica a la que se muestra en esta figura.

5.3 Funciones matemáticas de biblioteca

Las funciones matemáticas de biblioteca le permiten al programador ejecutar ciertos cálculos matemáticos comunes. Para iniciarnos en el concepto de las funciones utilizamos aquí varias funciones matemáticas de biblioteca. Más adelante en el libro, analizaremos muchas de las demás funciones de la biblioteca estándar C. Una lista completa de las funciones estándar de biblioteca de C aparece en el Apéndice B.

Las funciones se utilizan normalmente en un programa, escribiendo el nombre de la función, seguido por un paréntesis izquierdo y a continuación por el *argumento* (o una lista de argumentos separados por comas), de la función seguida por un paréntesis derecho. Por ejemplo, un programador que desea calcular e imprimir la raíz cuadrada de **900.0** pudiera escribir

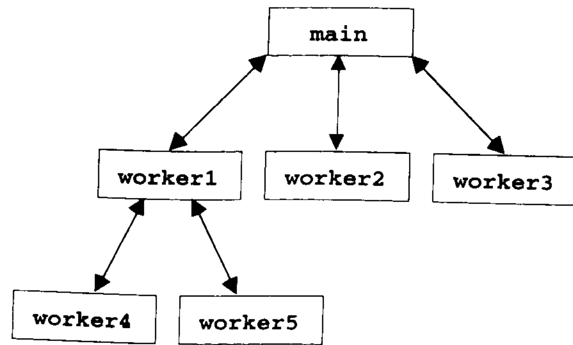


Fig. 5.1 Relación jerárquica función jefe/función trabajadora.

```
printf("%.2f", sqrt(900.0));
```

Cuando este enunciado se ejecuta, es llamada la función matemática de biblioteca **sqrt**, a fin de que calcule la raíz cuadrada del número contenido entre paréntesis (900.0). El número 900.0 es el argumento de la función **sqrt**. El enunciado anterior imprimiría 30.00. La función **sqrt** toma un argumento del tipo **double** y regresa un resultado del tipo **double**. Todas las funciones de la biblioteca de matemáticas devuelven el tipo de datos **double**.

Práctica sana de programación 5.2

Incluya el archivo de cabecera de matemáticas utilizando la directriz de preprocesador **#include <math.h>** cuando esté utilizando funciones de la biblioteca de matemáticas.

Error común de programación 5.1

Olvidar incluir el archivo de cabecera matemático, al usar las funciones matemáticas de biblioteca, puede causar resultados extraños.

Los argumentos de las funciones pueden ser constantes, variables o expresiones. Si **c1 = 13.0**, **d = 3.0**, y **f = 4.0**, entonces el enunciado

```
printf("%.2f", sqrt(c1 + d * f));
```

calcula e imprime la raíz cuadrada de **13.0 + 3.0 * 4.0 = 25.0**, es decir **5.00**.

Algunas funciones matemáticas de biblioteca de C se resumen en la figura 5.2. En la figura, las variables **x** y **y** son del tipo **double**.

5.4 Funciones

Las funciones permiten al programador modularizar un programa. Todas las variables declaradas en las definiciones de función son *variables locales* —son conocidas sólo en la función en la cual están definidas. La mayor parte de las funciones tienen una lista de *parámetros*. Los parámetros proporcionan la forma de comunicar información entre funciones. Los parámetros de una función también son variables locales.

Función	Descripción	Ejemplo
sqrt(x)	raíz cuadrada de *x*	**sqrt(900.0)** es **30.0** **sqrt(9.0)** es **3.0**
exp(x)	función exponencial e^x	**exp(1.0)** es **2.718282** **exp(2.0)** es **7.389056**
log(x)	logaritmo natural de *x* (de base *e*)	**log(2.718282)** es **1.0** **log(7.389056)** es **2.0**
Log10(x)	logaritmo de *x* (de base 10)	**log10(1.0)** es **0.0** **log10(10.0)** es **1.0** **log10(100.0)** es **2.0**
fabs(x)	valor absoluto de *x*.	si **x > 0** entonces **fabs(x)** es **x** si **x = 0** entonces **fabs(x)** es **0.0** si **x < 0** entonces **fabs(x)** es **-x**
ceil(x)	redondea a *x* al entero más pequeño que no sea menor que *x*	**ceil(9.2)** es **10.0** **ceil(-9.8)** es **9.0**
floor(x)	redondea a *x* al entero más grande no mayor que *x*	**floor(9.2)** es **9.0** **floor(-9.8)** es **-10.0**
pow(x, y)	*x* elevado a la potencia *y* (x^y)	**pow(2.7)** es **128.0** **pow(9, .5)** es **3.0**
fmod(x, y)	residuo de *x/y* como un número de punto flotante	**fmod(13.657, 2.333)** es **1.992**
sin(x)	seno trigonométrico de *x* (*x* en radianes)	**sin(0.0)** es **0.0**
cos(x)	coseno trigonométrico de *x* (*x* en radianes)	**cos (0.0)** es **1.0**
tan(x)	tangente trigonométrica de *x* (*x* en radianes)	**tan(0.0)** es **0.0**

Fig. 5.2 Uso común de las funciones matemáticas de biblioteca.

Observación de ingeniería de software 5.2

En programas que contengan muchas funciones, **main** *deberá de ser organizada como un grupo de llamadas a funciones que ejecuten la mayor parte del trabajo del programa.*

Existen varios intereses que dan motivo a la "funcionalización" de un programa. El enfoque de divide y vencerás hace que el desarrollo del programa sea más manipulable. Otra motivación es la *reutilización del software* —el uso de funciones existentes, como bloques constructivos, para crear nuevos programas. La reutilización del software es un factor primordial en el concepto de la programación orientada a objetos. Con buena identificación y definición de funciones, los programas pueden ser creados partiendo de funciones estandarizadas, que ejecuten tareas específicas, en vez de ser elaborados usando código personalizado. Esta técnica se conoce como *abstracción*.

Cada vez que escribimos programas incluyendo funciones estándar de biblioteca, como **printf**, **scanf** y **pow**, utilizamos la abstracción. Un tercer motivo es evitar un programa repetición de código. Empaquetar código en forma de función, permite que se ejecute dicho código, desde distintas posiciones en un programa, simplemente llamando dicha función.

Observación de ingeniería de software 5.3

Cada función debería limitarse a ejecutar una tarea sencilla y bien definida, y el nombre de la función debería expresar de forma clara dicha tarea. Ello facilitaría la abstracción y promovería la reutilización del software.

Observación de ingeniería software 5.4

Si no puede elegir un nombre conciso, que exprese lo que la función ejecuta, es probable que su función este intentando ejecutar demasiadas tareas diversas. A menudo es mejor dividir dicha función en varias funciones más pequeñas.

5.5 Definiciones de función

Cada programa que hemos presentado ha consistido de una función llamada **main**, que para llevar a cabo sus tareas ha llamado funciones estándar de biblioteca. Veremos ahora, como los programadores escriben sus propias funciones personalizadas.

Considere un programa que utiliza una función **square** para calcular los cuadrados de los enteros del 1 al 10 (figura 5.3).

```
/* A programmer-defined square function */
#include <stdio.h>

int square(int);   /* function prototype */

main()
{
   int x;

   for (x = 1; x <= 10; x++)
      printf("%d  ", square(x));

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

/* Function definition */
int square(int y)
{
   return y * y;
}
```

```
1  4  9  16  25  36  49  64  81  100
```

Fig. 5.3 Uso de una función definida-programador.

Práctica sana de programación 5.3

Coloque una línea en blanco entre definiciones de función, para separarlas y para mejorar la legibilidad del programa.

La función **square** es *invocada*, o bien *llamada* en **main** dentro del enunciado **printf**

```
printf("%d ", square(x));
```

La función **square** recibe una copia del valor de **x** en el *parámetro* **y**. A continuación **square** calcula **y * y**. El resultado se regresa a la función **printf** en **main** donde se llamó a **square**, y **printf** despliega el resultado. Este proceso se repite diez veces utilizando la estructura de repetición **for**.

La definición de **square** muestra que **square** espera un parámetro entero **y**. La palabra reservada **int**, que precede al nombre de la función indica que **square** regresa o devuelve un resultado entero. El enunciado **return** en **square** pasa el resultado del cálculo de regreso a la función llamadora.

La línea

```
int square(int);
```

es un *prototipo de función*. El **int** dentro del paréntesis informa al compilador que **square** espera recibir del llamador un valor entero. El **int** a la izquierda del nombre de la función **square** le informa al compilador que **square** regresa al llamador un resultado entero. El compilador hace referencia al prototipo de la función para verificar que las llamadas a **square** contengan el tipo de regreso correcto, el número correcto de argumentos, el tipo correcto de argumentos, y que los argumentos están en el orden correcto. En la Sección 5.6 se analizan los prototipos de las funciones en detalle.

El formato de una definición de función es

tipo de valor de regreso-nombre de la función (*lista de parámetros*)
{
    declaraciones

    enunciados
}

El *nombre de la función* es cualquier identificador válido. El *tipo de valor de regreso* es el tipo de los datos del resultado regresado al llamador. El *tipo de valor de regreso* **void** indica que una función no devolverá un valor. Un *tipo de valor de regreso* no especificado será siempre supuesto por el compilador como **int**.

Error común de programación 5.2

Omitir el tipo de valor de regreso en una definición de función causa un error de sintaxis, si el prototipo de función especifica un regreso de tipo distinto a **int**.

Error común de programación 5.3

Olvidar regresar un valor de una función, que se supone debe regresar un valor, puede llevar a errores inesperados. El estándar ANSI indica que el resultado de esta omisión queda indefinido.

Error común de programación 5.4

Regresar un valor de una función, cuyo tipo de regreso se ha declarado como `void`, *causará un error de sintaxis.*

Práctica sana de programación 5.4

Aun cuando un tipo de regreso omitido resulte en `int` *por omisión, declare siempre en forma explícita el tipo de regreso. Sin embargo, por lo regular, se omite el tipo de regreso correspondiente a* `main`.

La *lista de parámetros* consiste en una lista, separada por comas, que contiene las declaraciones de los parámetros recibidas por la función al ser llamada. Si una función no recibe ningún valor, la *lista de parámetros* es **`void`**. Para cada parámetro deberá ser enlistado de forma explícita un tipo, a menos de que el parámetro sea del tipo `int`. Si un tipo no es enlistado, se supondrá como **`int`**.

Error común de programación 5.5

Declarar parámetros de función del mismo tipo como `float x, y` *en vez de* `float x, float y`. *La declaración de parámetros* `float x, y` *convertiría de hecho a* `y` *en un parámetro del tipo* `int`, *porque* `int` *es el valor por omisión.*

Error común de programación 5.6

Es un error de sintaxis colocar un punto y coma después del paréntesis derecho que encierra una lista de parámetros de una definición de función.

Error común de programación 5.7

Volver a definir dentro de la función un parámetro de función como variable local es un error de sintaxis.

Práctica sana de programación 5.5

Incluya en la lista de parámetros el tipo de cada parámetro, inclusive si algún parámetro es del tipo por omisión `int`.

Práctica sana de programación 5.6

Aunque hacerlo no es incorrecto, no utilice los mismos nombres para argumentos pasados a una función y parámetros correspondientes de la definición de función. Con ello ayuda a evitar ambigüedad.

Las *declaraciones*, junto con *los enunciados* dentro de las llaves, forman el *cuerpo de la función*. El cuerpo de la función también se conoce como un *bloque*. Un bloque es un enunciado compuesto que incluye declaraciones. Las variables pueden ser declaradas en cualquier bloque, y los bloques pueden estar anidados. *Bajo ninguna circunstancia puede ser definida una función en el interior de otra función.*

Error común de programación 5.8

Definir una función en el interior de otra función es un error de sintaxis.

Práctica sana de programación 5.7

Seleccionar nombres significativos para funciones y nombres significativos para parámetros, hace que sean más legibles los programas y ayuda a evitar un uso de comentarios excesivo.

Observación de ingeniería de software 5.5

Una función no debería tener una longitud mayor que una página. Aún mejor, una función no debería ser más larga que media página. Las funciones pequeñas promueven la reutilización del software.

Observación de ingeniería de software 5.6

Los programas deberían ser escritos como recopilaciones de pequeñas funciones. Esto haría que los programas fuesen más fáciles de escribir, de depurar, de mantener y de modificar.

Observación de ingeniería de software 5.7

Una función que requiera un gran número de parámetros quizás esté ejecutando demasiadas tareas. Considere dividir esta función en funciones más pequeñas, que ejecuten las tareas por separado. El encabezado de función, si es posible, debería caber en una línea.

Observación de ingeniería de software 5.8

El prototipo de función, el encabezado de función y las llamadas de función deberán todas estar de acuerdo en lo que se refiere al número, tipo y orden de argumentos y de parámetros, así como en el tipo de valor de regreso.

Existen tres formas de regresar el control al punto desde el cual se invocó a una función. Si la función no regresa un resultado, el control sólo se devuelve cuando se llega a la llave derecha que termina la función, o al ejecutar el enunciado

```
return;
```

Si la función regresa un resultado, el enunciado

`return`; *expresión;*

devuelve el valor de *expresión* al llamador.

En nuestro segundo ejemplo se utiliza una función **`maximum`** definida por el programador, para determinar y regresar el mayor de tres enteros (figura 5.4). Los tres enteros son introducidos mediante **`scanf`**. A continuación los enteros son pasados a **`maximum`**, que determina cual es el más grande. Este valor es regresado a **`main`** mediante el enunciado **`return`** existente en **`maximum`**. El valor regresado es asignado a la variable **`largest`** que entonces es impresa con el enunciado **`printf`**.

5.6 Prototipos de funciones

Una de las características más importantes de ANSI C es el *prototipo de función*. Esta característica fue tomada prestada por el comité de ANSI C de los que estaban desarrollando C++. Un prototipo de función le indica al compilador el tipo de dato regresado por la función, el número de parámetros que la función espera recibir, los tipos de dichos parámetros, y el orden en el cual se esperan dichos parámetros. El compilador utiliza los prototipos de funciones para verificar las llamadas de función. Versiones anteriores de C no ejecutaban este tipo de verificación, por lo que era posible llamar de forma incorrecta a las funciones, sin que el compilador se diera cuenta de los errores. Estas llamadas podían resultar en errores fatales en tiempo de ejecución, o en errores no fatales, que causaban sutiles errores lógicos y difíciles de detectar. Los prototipos de funciones de ANSI C corrigen esta deficiencia.

```
/* Finding the maximum of three integers */
#include <stdio.h>

int maximum(int, int, int);   /* function prototype */

main()
{
   int a, b, c;

   printf("Enter three integers: ");
   scanf("%d%d%d", &a, &b, &c);
   printf("Maximum is: %d\n", maximum(a, b, c));

   return 0;
}

/* Function maximum definition */
int maximum(int x, int y, int z)
{
   int max = x;

   if (y > max)
      max = y;

   if (z > max)
      max = z;

   return max;
}
```

```
Enter three integers: 22  85  17
Maximum is: 85
```

```
Enter three integers: 85  22  17
Maximum is: 85
```

```
Enter three integers: 22  17  85
Maximum is: 85
```

Fig. 5.4 Definición-programador de función de **maximum**.

Práctica sana de programación 5.8

Incluya prototipos de funciones para todas las funciones para aprovechar las capacidades de C de verificación de tipo. Utilice las directrices de preprocesador **#include** *para obtener los prototipos de las funciones estándar de biblioteca a partir de los archivos de cabecera de las bibliotecas apropiadas. También utilice* **#include** *para obtener archivos de cabecera que contengan prototipos de funciones utilizadas por usted y los miembros de su grupo.*

El prototipo de función para **maximum** en la figura 5.4 es

```
int maximum(int, int, int);
```

Este prototipo de función indica que **maximum** toma tres argumentos del tipo **int**, y regresa un resultado del tipo **int**. Note que el prototipo de función es la misma que la primera línea de la definición de función de **maximum**, a excepción de los nombres de los parámetros (**x**, **y** y **z**), mismos que no se incluyen.

Práctica sana de programación 5.9

Los nombres de los parámetros a veces se incluyen en los prototipos de función por razones de documentación. El compilador ignora estos nombres.

Error común de programación 5.9

Olvidar el punto y coma al final de un prototipo de función hará que ocurra un error de sintaxis.

Una llamada de función que no coincida con el prototipo de la función causará un error de sintaxis. Un error también se generará si el prototipo de la función y la definición de ésta no están de acuerdo. Por ejemplo, en la figura 5.4, si el prototipo de función hubiera sido escrita

```
void maximum (int, int, int);
```

el compilador habría generado un error, porque el tipo de regreso **void** en el prototipo de función, es diferente del tipo de regreso **int** del encabezado de función.

Otra característica importante de prototipos de función es la *coerción de argumentos*, es decir, obligar a los argumentos al tipo apropiado.

Por ejemplo, la función matemática de biblioteca **sqrt** puede ser llamada con un argumento entero, aun cuando el prototipo de función en **math.h** especifica un argumento **double** y aún así la función operará de forma correcta. El enunciado

```
printf("%.3f\n", sqrt(4));
```

valuará de forma correcta **sqrt(4)** e imprimirá el valor **2.000.** El prototipo de función hace que el compilador convierta el valor entero **4** al valor **double 4.0**, antes de que el valor sea pasado a **sqrt**. En general, los valores de los argumentos que no correspondan precisamente a los tipos de los parámetros de el prototipo de función serán convertidos al tipo apropiado, antes de que la función sea llamada. Estas conversiones pueden llevar a resultados incorrectos si no son seguidas las *reglas de promoción* de C. Las reglas de promoción definen como deben ser convertidos los tipos a otros tipos, sin perder datos. En nuestro ejemplo **sqrt** arriba citado, un **int** es convertido automáticamente a un **double**, sin cambiar su valor. Sin embargo, un **double** convertido a **int** trunca la parte fraccional del valor **double.** Convertir tipos enteros grandes a tipos enteros pequeños (es decir, **long** a **short**), también puede dar como resultado valores cambiados.

Las reglas de promoción se aplican automáticamente a expresiones que contengan valores de dos o más tipos de datos (también conocidas como expresiones de *tipo mixto*). El tipo de cada valor en una expresión de tipo mixto es automáticamente promovida al tipo más alto en la expresión (de hecho se crea una versión temporal de cada valor y se utiliza para la expresión conservándose sin cambio los valores originales). En la figura 5.5 se enlistan los tipos de datos en orden, desde el tipo más alto hasta el tipo más bajo, con cada uno de los tipos de especificaciones de conversión de **printf** y de **scanf**.

Tipos de datos	Especificaciones de conversión printf	Especificaciones de conversión scanf
`long double`	`%Lf`	`%Lf`
`double`	`%f`	`%lf`
`float`	`%f`	`%f`
`unsigned long int`	`%lu`	`%lu`
`long int`	`%ld`	`%ld`
`unsigned int`	`%u`	`%u`
`int`	`%d`	`%d`
`short`	`%hd`	`%hd`
`char`	`%c`	`%c`

Fig. 5.5 Jerarquía de promoción para tipos de datos.

La conversión de valores a tipos inferiores por lo regular resulta en un valor incorrecto. Por lo tanto, un valor puede ser convertido sólo a un valor inferior, asignando de manera explícita el valor a una variable de tipo menor inferior, o mediante el uso de un operador cast. Los valores de los argumentos de las funciones son convertidos a los tipos de parámetro en un prototipo de función, como si hubieran sido asignados directamente a las variables de esos tipos. Si nuestra función **square**, que utiliza un parámetro entero (figura 5.3), es llamada con un argumento de punto flotante, el argumento se convertirá a **int** (un tipo inferior) y **square** normalmente regresará un valor incorrecto. Por ejemplo, **square(4.5)** regresaría **16** y no **20.25**.

Error común de programación 5.10

Convertir de un tipo de datos superior en la jerarquía de promoción a un tipo inferior, puede modificar el valor del dato.

Si el prototipo de función de una función no ha sido incluida en un programa, el compilador forma su propio prototipo de función utilizando la primera ocurrencia de la función ya sea la definición de función o una llamada a dicha función. Por omisión, el compilador asumirá que la función regresa un **int**, y no se supone nada en relación con los argumentos. Por lo tanto, si los argumentos pasados a la función no son correctos, los errores no serán detectados por el compilador.

Error común de programación 5.11

Olvidar un prototipo de función generará un error de sintaxis, si el tipo de regreso de la función no es int y la definición de función aparece después de la llamada a la función dentro del programa. De lo contrario, el olvidar un prototipo de función puede causar un error en tiempo de ejecución o un resultado inesperado.

Observación de ingeniería de software 5.9

Un prototipo de función, colocada fuera de una definición de función, se aplica a todas las llamadas de función que aparezcan dentro del archivo después del prototipo de la función. Una prototipo de función colocado en una función, se aplica sólo a las llamadas efectuadas en esa función.

5.7 Archivos de cabecera

Cada biblioteca estándar tiene su *archivo de cabecera* correspondiente, que contiene los prototipos de función de todas las funciones de dicha biblioteca, y las definiciones de varios tipos de datos y de constantes requeridas por dichas funciones. La figura 5.6 enlista de forma alfabética los archivos de cabecera estándar de biblioteca que pudieran incluirse en programas. El término "macros" que se utiliza varias veces en la figura 5.6, se analiza en detalle en el capítulo 13, "Preprocesador".

Archivo de cabecera de la biblioteca estándar	Explicación
`<asser.h>`	Contiene macros así como información para añadir diagnósticos que ayudan a la depuración de programas.
`<ctype.h>`	Contiene prototipos de funión para funciones que prueban caracteres en relación con ciertas propiedades, y prototipos de funciones que pueden ser utilizadas para la conversión de minúsculas a mayúsculas y viceversa.
`<errno.h>`	Define macros que son útiles para la información de condiciones de error.
`<float.h>`	Contiene los límites de tamaño de punto flotante del sistema.
`<limits.h>`	Incluye los límites de tamaño integral del sistema.
`<locale.h>`	Contiene los prototipos de función y otra información que le permite a un programa ser modificado en relación con la localización actual en la cual se está ejecutando. La noción de localización le permite al sistema de cómputo manejar diferentes reglas convencionales para la expresión de datos como son fechas, horas, monedas y números grandes en diferentes áreas del mundo.
`<math.h>`	Contiene prototipos para funciones matemáticas de biblioteca.
`<setjmp.h>`	Contiene prototipos para funciones que permiten pasar por alto la secuencia usual de llamadas de función y regreso.
`<signal.h>`	Contiene prototipos de función y macros para manejar varias condiciones que pudieran ocurrir durante la ejecución de un programa.
`<stdarg.h>`	Define macros para manejar con una lista de argumentos a una función cuyo número y cuyo tipo son desconocidos.
`<stddef.h>`	Contiene definiciones comunes de tipos utilizados en C para ejecutar ciertos cálculos.
`<stdio.h>`	Contiene prototipos para las funciones de biblioteca de entrada y salida estándar y la información que éstas utilizan.
`<stdlib.h>`	Contiene prototipos de función para las conversiones de números a texto y de texto a números, para la asignación de memoria, para números aleatorios y otras funciones de utilería.
`<string.h>`	Contiene prototipos para las funciones de procesamiento de cadenas.
`<time.h>`	Contiene prototipos de función y tipos para manipular hora y fecha.

Fig. 5.6 Archivo de cabecera de la biblioteca estándar.

El programador puede crear archivos de cabecera personalizados. Los archivos de cabecera personalizados definidos por el usuario, también deben terminar en **.h**. Un archivo de cabecera definido por el programador puede ser incluido utilizando la directriz de preprocesador **#include**. Por ejemplo, el archivo de cabecera **square.h** puede ser incluido en nuestro programa mediante la directriz

```
#include "square.h"
```

en la parte superior del programa. En la sección 13.2 se presenta información adicional sobre la inclusión de archivos de cabecera.

5.8 Cómo llamar funciones: llamada por valor y llamada por referencia

Dos formas de invocar funciones en muchos lenguajes de programación son la *llamada por valor* y la *llamada por referencia*. Cuando los argumentos se pasan en llamada por valor, se efectúa una *copia* del valor del argumento y ésta se pasa a la función llamada. Las modificaciones a la copia no afectan al valor original de la variable del llamador. Cuando un argumento es pasado en llamada por referencia, el llamador de hecho permite que la función llamada modifique el valor original de la variable.

La llamada por valor debería ser utilizada siempre que la función llamada no necesite modificar el valor de la variable original del llamador. Esto evita *efectos colaterales* accidentales, que obstaculizan de forma tan importante el desarrollo de sistemas correctos y confiables de software. La llamada por referencia debe ser utilizada sólo en funciones llamadas confiables, que necesitan modificar la variable original.

En C, todas las llamadas son llamadas por valor. Como veremos en el capítulo 7, es posible *simular* la llamada por referencia mediante la utilización de operadores de dirección y de indirección. En el capítulo 6, veremos que los arreglos son pasados en forma automática, en forma de llamadas por referencia simuladas. Tendremos que esperar hasta que lleguemos al capítulo 7 para una comprensión completa de este tema complejo. Por ahora, nos concentraremos en la llamada por valor.

5.9 Generación de números aleatorios

Ahora nos desviaremos en forma breve y se espera en forma entretenida una popular aplicación de programación, es decir, la simulación y la ejecución de juegos. En esta sección y en la siguiente, desarrollaremos un programa para ejecución de juegos bien estructurado, que incluye muchas funciones. El programa utiliza la mayor parte de las estructuras de control que hemos estudiado.

Existe algo en el aire de los casinos de juego, que excita a todo tipo de personas, desde profesionales de las elegantes mesas de caoba y fieltro para dados, a los bandidos de un solo brazo de las máquinas tragamonedas. Es el *elemento suerte*, la posibilidad que la suerte pueda convertir una simple bolsa de dinero en una montaña de riquezas. El *elemento suerte* puede ser introducido en las aplicaciones de cómputo, mediante el uso de la función **rand** existente en la biblioteca estándar de C

Considere el enunciado siguiente:

```
i = rand();
```

La función **rand** genera un entero entre 0 y **RAND_MAX** (una constante simbólica definida en el archivo de cabecera **<stdlib.h>** El estándar ANSI indica que el valor de **RAND_MAX** debe ser por lo menos 32767, que es el valor máximo de un entero de dos bytes (es decir, 16 bits). Los programas en esta sección fueron probados en un sistema C, con un valor máximo de 32767 para **RAND_MAX**. Si **rand** en verdad produce enteros aleatorios, cualquier número entre 0 y **RAND_MAX** tiene la misma *oportunidad* (*o probabilidad*) de ser elegido, cada vez que **rand** es llamado.

El rango de valores producidos directamente por **rand**, a menudo son distintos de lo que se requiere para una aplicación específica. Por ejemplo, un programa que simule lanzar una moneda al aire, pudiera requerir sólo 0 para "caras" y 1 para "cruces". Un programa de juego de dados, que simule un dado de seis caras, requeriría de enteros aleatorios del rango 1 al 6.

A fin de demostrar **rand**, desarrollemos un programa para simular 20 tiradas de un dado de seis caras, e imprimamos el valor de cada tirada. El prototipo para la función **rand** puede ser encontrada en **<stdlib.h>** Utilizaremos el operador de módulo (%) en conjunción con **rand**, como sigue

```
rand() % 6
```

para producir enteros, en el rango 0 a 5. Esto se llama *dimensionar*. El número 6 se conoce como *factor de dimensionamiento*. Entonces después *desplazamos* el rango de números producidos añadiendo a nuestro resultado anterior la unidad. La figura 5.7 confirma que los resultados quedan dentro del rango de 1 a 6.

Para mostrar que estos números ocurren aproximadamente con la misma probabilidad, con el programa de la figura 5.8, simularemos 6000 tiradas de un dado. Cada entero del 1 al 6 debería de aparecer aproximadamente 1000 veces.

```c
/* Shifted, scaled integers produced by 1 + rand() % 6 */
#include <stdio.h>
#include <stdlib.h>

main()
{
   int i;

   for (i = 1; i <= 20; i++) {
      printf("%10d", 1 + (rand() % 6));

      if (i % 5 == 0)
         printf("\n");
   }

   return 0;
}
```

```
         5         5         3         5         5
         2         4         2         5         5
         5         3         2         2         1
         5         1         4         6         4
```

Fig. 5.7 Desplazado y dimensionado de enteros producidos por `1 + rand ( ) % 6`.

```
/* Roll a six-sided die 6000 times */
#include <stdio.h>
#include <stdlib.h>

main()
{
   int face, roll, frequency1 = 0, frequency2 = 0,
       frequency3 = 0, frequency4 = 0,
       frequency5 = 0, frequency6 = 0;

   for (roll = 1; roll <= 6000; roll++) {
      face = 1 + rand() % 6;

      switch (face) {
         case 1:
            ++frequency1;
            break;
         case 2:
            ++frequency2;
            break;
         case 3:
            ++frequency3;
            break;
         case 4:
            ++frequency4;
            break;
         case 5:
            ++frequency5;
            break;
         case 6:
            ++frequency6;
            break;
      }
   }

   printf("%s%13s\n", "Face", "Frequency");
   printf("   1%13d\n", frequency1);
   printf("   2%13d\n", frequency2);
   printf("   3%13d\n", frequency3);
   printf("   4%13d\n", frequency4);
   printf("   5%13d\n", frequency5);
   printf("   6%13d\n", frequency6);
   return 0;
}
```

```
Face     Frequency
   1           987
   2           984
   3          1029
   4           974
   5          1004
   6          1022
```

Fig. 5.8 Tirar un dado de seis caras 6000 veces.

Como muestra el resultado del programa, mediante el dimensionamiento y el desplazamiento hemos utilizado la función **rand** para simular en forma realista tirar un dado de seis caras. Advierta que en la estructura **switch** *no* se ha previsto un caso **default**. También note la utilización del especificador de conversión **%s**, para imprimir las cadenas de caracteres **"Face"** y **"Frequency"** como encabezados de columna. Después de que en el capítulo 6 estudiemos los arreglos, mostraremos como remplazar de una manera elegante toda la estructura **switch** con un solo enunciado en una línea.

La ejecución del programa de la figura 5.7 por segunda vez produce

```
5    5    3    5    5
2    4    2    5    5
5    3    2    2    1
5    1    4    6    4
```

Note que apareció impresa exactamente la misma secuencia de valores. ¿Cómo pueden ser aleatorios estos valores? Irónicamente, esta capacidad de repetición es una característica importante de la función **rand**. Al depurar un programa, esta capacidad es esencial para comprobar que las correcciones efectuadas a un programa se ejecuten de manera correcta.

La función **rand** de hecho genera *números seudoaleatorios*. Al llamar repetidamente a **rand** produce una secuencia de números que parecen ser aleatorios. Sin embargo, cada vez que el programa se ejecute la secuencia se repetirá a sí misma. Una vez que el programa se haya depurado con cuidado, puede ser condicionado para producir en cada una de las ejecuciones una secuencia diferente de números aleatorios. A esto se le llama hacerlo *aleatorio* y se lleva a cabo mediante la función estándar de biblioteca **srand**. La función **srand** toma un argumento entero **unsigned** (la semilla), para que en cada ejecución del programa la función **rand** produzca una secuencia diferente de números aleatorios.

El uso de **srand** queda demostrado en la figura 5.9. En el programa, utilizamos el tipo de datos **unsigned**, que es una abreviación de **unsigned.int**. Se almacena un **int** en por lo menos dos bytes de memoria, y puede tener valores positivos y negativos. Una variable del tipo **unsigned** también queda almacenada en por lo menos dos bytes de memoria. Un **unsigned.int** de dos bytes puede tener sólo valores positivos dentro del rango 0 hasta 65535. Un **unsigned.int** de cuatro bytes puede tener sólo valores positivos en el rango desde 0 hasta 4294967295. La función **srand** toma un valor **unsigned** como argumento. Se utiliza el especificador de conversión %u para leer un valor **unsigned** utilizando **scanf**. El prototipo de función **srand** se encuentra en **<stdlib.h.>**

Ejecutemos varias veces el programa y observemos los resultados. Note que cada vez que se ejecuta el programa se obtiene una secuencia *diferente* de números aleatorios, siempre y cuando se le dé una semilla diferente.

Si deseamos hacerlo aleatorio sin necesidad de introducir cada vez una semilla, pudiéramos utilizar un enunciado como

```
srand(time(NULL));
```

Esto hace que la computadora lea su reloj para obtener en forma automática un valor para la semilla. La función **time** devuelve la hora actual del día en segundos. Este valor es convertido a un entero unsigned y es utilizado como semilla para el generador de números aleatorios. La función **time** toma **NULL** como argumento (**time** es capaz de proporcionar al programador una cadena

```
/* Randomizing die-rolling program */
#include <stdlib.h>
#include <stdio.h>

main()
{
   int i;
   unsigned seed;

   printf("Enter seed: ");
   scanf("%u", &seed);
   srand(seed);

   for (i = 1; i <= 10; i++) {
      printf("%10d", 1 + (rand() % 6));

      if (i % 5 == 0)
         printf("\n");
   }

   return 0;
}
```

```
Enter seed: 67
         1         6         5         1         4
         5         6         3         1         2
```

```
Enter seed: 432
         4         2         5         4         3
         2         5         1         4         4
```

```
Enter seed: 67
         1         6         5         1         4
         5         6         3         1         2
```

Fig. 5.9 Programa de tirada del dado para hacerlo aleatorio.

representando la hora del día; **NULL** deshabilita esta capacidad para una llamada específica a **time**). El prototipo de función correspondiente a **time** está en **<time.h>**.

Los valores producidos directamente por **rand** estarán siempre en el rango:

$$0 \leq \texttt{rand()} \leq \texttt{RAND_MAX}$$

Previamente hemos demostrado cómo escribir un solo enunciado en C para simular la tirada de un dado de seis caras:


```
face = 1 + rand() % 6;
```

Este enunciado siempre asigna un valor entero (aleatorio) a la variable **face** en el rango $1 \leq$ **face** ≤ 6. Note que el ancho de este rango (es decir, el número de enteros consecutivos dentro del rango) es 6 y el número inicial dentro del rango es 1. Regresando al enunciado anterior, vemos que el ancho del rango se determina por el número utilizado para dimensionar a rand utilizando el operador de módulo (es decir 6), y el número inicial del rango es igual al número (es decir 1) que se añade a **rand % 6.** Podemos generalizar este resultado como sigue

```
n = a + rand() % b;
```

donde **a** es el *valor de desplazamiento* (que resulta igual al primer número del rango deseado de enteros consecutivos), y **b** es el factor de dimensionamiento (que es igual al ancho del rango deseado de enteros consecutivos). En los ejercicios, veremos que es posible escoger enteros al azar de conjuntos de valores distintos que los rangos de enteros consecutivos.

Error común de programación 5.12

Usar **srand** *en vez de* **rand** *para generar números aleatorios.*

5.10 Ejemplo: un juego de azar

Uno de los juegos de azar más populares en el juego de dados es el juego de dados conocido como "craps" (tiro perdedor), mismo que se juega en casinos y en callejuelas en todo el mundo. Las reglas del juego son sencillas:

> *Un jugador tira dos dados. Cada dado tiene seis caras. Las caras contienen 1, 2, 3, 4, 5, y 6 puntos. Una vez que los dados se hayan detenido, se calcula la suma de los puntos en las dos caras superiores. Si a la primera tirada, la suma es 7, o bien 11, el jugador gana. Si en la primera tirada la suma es 2, 3, o 12 (conocido como "craps"), el jugador pierde (es decir, la casa "gana"). Si en la primera tirada, la suma es 4, 5, 6, 8, 9 ó 10, entonces dicha suma se convierte en el "punto" o en la tirada. Para ganar, el jugador deberá continuar tirando los dados hasta que haga su "tirada". El jugador perderá si antes de hacer su tirada sale una tirada de 7.*

El programa de la figura 5.10 simula el juego de craps. La figura 5.11 muestra varias ejecuciones de muestra.

Note que el jugador debe tirar dos dados en la primera tirada, y debe hacer lo mismo en todas las tiradas subsecuentes. Definimos una función **rollDice** para tirar los dados y calcular e imprimir su suma. La función **rollDice** se define una vez, pero es llamada desde dos lugares en el programa. De forma interesante, **rollDice** no toma argumentos, por lo que en la lista de parámetros hemos indicado **void**. La función **rollDice** sí regresa la suma de los dos dados, por lo que es indicado colocar un regreso de tipo **int** en el encabezado de función.

El juego es razonablemente complicado. El jugador puede perder o ganar en la primera tirada, o puede perder o ganar en cualquier tirada subsecuente. La variable **gameStatus** se utiliza para llevar registro de todo lo anterior.

Cuando se gana el juego, ya sea en la primera tirada o en una tirada subsecuente, **gameStatus** se define en 1. Cuando se pierde el juego, ya sea en la primera tirada o en una tirada subsecuente, **gameStatus** se define como 2. De lo contrario, **gameStatus** es cero y el juego deberá continuar.

```c
/* Craps */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int rollDice(void);

main()
{
   int gameStatus, sum, myPoint;

   srand(time(NULL));
   sum = rollDice();           /* first roll of the dice */

   switch(sum) {
      case 7: case 11:         /* win on first roll */
         gameStatus = 1;
         break;
      case 2: case 3: case 12: /* lose on first roll */
         gameStatus = 2;
         break;
      default:                 /* remember point */
         gameStatus = 0;
         myPoint = sum;
         printf("Point is %d\n", myPoint);
         break;
   }

   while (gameStatus == 0) {   /* keep rolling */
      sum = rollDice();

      if (sum == myPoint)      /* win by making point */
         gameStatus = 1;
      else
         if (sum == 7)         /* lose by rolling 7 */
            gameStatus = 2;
   }

   if (gameStatus == 1)
      printf("Player wins\n");
   else
      printf("Player loses\n");

   return 0;
}

int rollDice(void)
{
   int die1, die2, workSum;

   die1 = 1 + (rand() % 6);
   die2 = 1 + (rand() % 6);
   workSum = die1 + die2;
   printf("Player rolled %d + %d = %d\n", die1, die2, workSum);
   return workSum;
}
```

Fig. 5.10 Programa que simula el juego de craps.


```
Player rolled 6 + 5 = 11
Player wins
```

```
Player rolled 6 + 6 = 12
Player loses
```

```
Player rolled 4 + 6 = 10
Point is 10
Player rolled 2 + 4 = 6
Player rolled 6 + 5 = 11
Player rolled 3 + 3 = 6
Player rolled 6 + 4 = 10
Player wins
```

```
Player rolled 1 + 3 = 4
Point is 4
Player rolled 1 + 4 = 5
Player rolled 5 + 4 = 9
Player rolled 4 + 6 = 10
Player rolled 6 + 3 = 9
Player rolled 1 + 2 = 3
Player rolled 5 + 2 = 7
Player loses
```

Fig. 5.11 Muestra de ejecuciones en el juego de craps.

Después de la primera tirada, si se ha terminado el juego, la estructura `while` es saltada, dado que `gameStatus` no es igual a cero. El programa continúa a la estructura `if/else`, que imprime "`Player wins`", en el caso que `gameStatus` sea `1` y "`Player loses`", si `gameStatus` es igual a `2`.

Después de la primera tirada, si el juego no se ha terminado, entonces `sum` quedará guardada en `myPoint`. La ejecución continúa con la estructura `while` porque `gameStatus` es `0`. Cada vez a través del ciclo `while`, se llama a `rollDice` para producir una nueva `sum`. Si `sum` coincide con `myPoint`, `gameStatus` se define como `1` para indicar que el jugador ganó, entonces la prueba de continuidad `while` falla, y la estructura `if/else` imprimirá "`Player wins`" y la ejecución se terminará. Si `sum` es igual a `7` `gameStatus` se define como `2` para indicar que el jugador perdió, la prueba de terminación `while` falla, el enunciado `if/else` imprime "`Player loses`" y la ejecución termina.

Note la interesante estructura de control de este programa. Hemos utilizado dos funciones `main` y `rollDice` y las estructuras `switch`, `while`, `if/else`, y la `if` anidada. En los ejercicios, investigarémos varias características interesantes del juego de craps.

5.11 Clases de almacenamiento

En los capítulos 2 al 4, utilizamos identificadores para los nombres de variables. Los atributos de variables son: nombre, tipo y valor. En este capítulo, también usamos identificadores como nombre para funciones definidas por usuario. De hecho, cada identificador en un programa tiene otros atributos incluyendo clase de almacenamiento, duración de almacenamiento, alcance y enlace.

C proporciona cuatro clases de almacenamiento, que se indican por los *especificadores de clase de almacenamiento:* **auto**, **register**, **extern** y **static**. La *clase de almacenamiento* de un identificador ayuda a determinar su duración de almacenamiento, su alcance y su enlace. La *duración de almacenamiento* de un identificador es el periodo durante el cual dicho identificador existe en memoria. Algunos identificadores tienen una existencia breve, otros son creados y destruidos en forma repetida, y otros existen durante toda la ejecución de un programa. El *alcance* de un identificador en un programa es dónde puede ser referenciado el mismo. Algunos identificadores pueden ser referenciados a todo lo largo de un programa, y otros a partir de sólo porciones de un programa. El *enlace* de un identificador determina, para un programa de varios archivos fuete (un tema que investigaremos en el capítulo 14), si un identificador es conocido solamente en el archivo fuente actua, o en cualquier archivo fuente, mediante declaraciones apropiadas. Esta sección analiza las cuatro clases de almacenamiento y la duración del almacenamiento. En la sección 5.12 se analiza el alcance de los identificadores. En el capítulo 14, Temas avanzados, se estudia el enlace de identificadores y la programación con varios archivos fuente.

Los cuatro especificadores de clase de almacenamiento (persistencia) pueden ser divididos en dos: *persistencia automática* y *persistencia estática*. Las palabras reservadas **auto** y **register** se utilizan para declarar variables de persistencia automática. Estas variables se crean al introducirse al ámbito del bloque en el cual están declaradas, existen mientras dicho bloque está activo, y se destruyen cuando se sale de ese bloque.

Sólo las variables pueden tener persistencia automática. Las variables locales de una función (aquellas declaradas en la lista de parámetros o en el cuerpo de la función) por lo regular tienen una persistencia automática. La palabra reservada **auto** declara en forma explícita a las variables de persitencia automática. Por ejemplo, la siguiente declaración indica que las variables **float**, **x** e **y** son variables locales automáticas, y existen sólo en el cuerpo de la función en el cual aparece dicha declaración:

```
auto float x, y;
```

Por omisión, las variables locales tienen persistencia automática, por lo que la palabra reservada **auto** es rara vez utilizada. Durante el resto de este texto, nos ocuparemos de las variables con persistencia automática simplemente como variables automáticas.

Sugerencia de rendimiento 5.1

La persistencia automática es una forma de conservar memoria, porque las variables automáticas existen sólo cuando son necesitadas. Son creadas al introducirse la función en la cual son declaradas, y son destruidas cuando se sale de dicha función.

Observación de ingeniería de software 5.10

El almacenamiento automático es otra vez otro ejemplo del principio del menor privilegio. ¿Por qué tendrían que estar las variables almacenadas en memoria y accesibles cuando de hecho no son necesarias?

Los datos de un programa en la versión en lenguaje máquina, para cálculos y otros procesos normalmente se cargan en registros.

Sugerencia de rendimiento 5.2

El especificador de clase de almacenamiento ***register*** *puede ser colocado antes de una declaración de variable automática, para sugerir que el compilador conserve la variable en uno de los registros de alta velocidad del hardware de la computadora. Si se puede mantener en registros de hardware las variables muy utilizadas como son contadores y totales, puede ser eliminada la sobrecarga correspondiente a cargar en forma repetida las variables de las memorias a los registros y almacenar los resultados de vuelta en memoria.*

El compilador pudiera ignorar declaraciones **register**. Por ejemplo, quizás no exista un suficiente número de registros disponibles para que los utilice la computadora. La siguiente declaración sugiere que se coloque la variable entera **counter** en uno de los registros de la computadora y se inicialice a 1:

```
register int counter = 1;
```

La palabra reservada **register** puede ser utilizada sólo con variables automáticas.

Sugerencia de rendimiento 5.3

A menudo, son innecesarias las declaraciones **register**. *Los compiladores optimizadores de hoy son capaces de reconocer variables de uso frecuente, y pueden decidir colocarlos en registro, sin necesidad de una declaración* **register** *proveniente del programador.*

Las palabras reservadas **extern** y **static** se utilizan para declarar identificadores de variables y de funciones de persistencia estática. Los identificadores de persistencia estática existen a partir del momento de que el programa se inicia en ejecución. Se asigna y se inicializa almacenamiento para las variables desde el momento que se empieza a operar el programa. Para las funciones, existe el nombre de la función al iniciarse la ejecución del programa. Sin embargo, aunque las variables y los nombres de función existen a partir del inicio de la ejecución del programa, esto no significa que estos identificadores pudieran ser utilizados a todo lo largo del mismo. La persistencia y el alcance (dónde puede ser utilizado un nombre), son temas separados, como veremos en la sección 5.12.

Existen dos tipos de identificadores con persistencia estática: los identificadores externos (como son las variables globales y los nombres de función) y las variables locales declaradas con el especificador de clase de almacenamiento **static**. Las variables globales y los nombres de función son por omisión de la clase de almacenamiento **extern**. Las variables globales se crean colocando declaraciones variables por fuera de cualquier definición de función, y conservan sus valores a todo lo largo de la ejecución del programa. Las variables globales y las funciones pueden ser referenciadas por cualquier función posteriores a sus declaraciones o definiciones dentro del archivo. Esta es una razón para la utilización de prototipos de función. Cuando incluimos **stdio.h** en un programa que hace llamadas a **printf**, el prototipo de función se coloca al principio de nuestro archivo para hacer que el nombre **printf** sea conocido para el resto del archivo.

Observación de ingeniería de software 5.11

La declaración de una variable como global en vez de como local, permite que ocurran efectos colaterales no deseados cuando una función que no requiere de acceso a la variable la modifica accidental o maliciosamente. En general, el uso de las variables globales debería ser evitado, excepto en ciertas situaciones, con requerimientos de rendimiento únicos (como se analizan en el capítulo 14).

UNIVERSIDAD DE LA REPUBLICA FACULTAD DE INGENIERIA DEPARTAMENTO DE

Práctica sana de programación 5.10

Las variables utilizadas sólo en una función particular deberían ser declaradas como variables locales en esa función, en vez de como variables externas.

Las variables locales declaradas con la palabra reservada **static** son aún conocidas sólo en la función para la cual son definidas, pero a diferencia de las variables automáticas, las variables locales **static** conservan su valor cuando sale de la función. La próxima vez que se llama a esa función, la variable local **static** contiene el valor que tenía cuando la función salió por última vez. El siguiente enunciado declara la variable local **count** como **static** y la inicializa a 1.

```
static int count = 1;
```

Todas las variables numéricas **static** se incializan a cero si no son inicializadas de forma explícita por el programador. (Las variables de apuntador, analizadas en el capítulo 7, se inicializan a **NULL**).

Error común de programación 5.13

Usar múltiples especificadores de clase de almacenamiento para un identificador. Sólo puede ser aplicado un especificador de clase de almacenamiento a un identificador.

Las palabras reservadass **extern** y **static** tienen un significado especial, si se aplican explícitamente a identificadores externos. En el capítulo 14, Temas avanzados, analizaremos el uso explícito de **extern** y de **static**, junto con identificadores externos y con programas de archivos de fuentes múltiples.

5.12 Reglas de alcance

El *alcance* de un identificador es la porción del programa en el cual dicho identificador puede ser referenciado. Por ejemplo, cuando en un bloque declaramos una variable local, puede ser referenciada sólo en dicho bloque o en los bloques anidados dentro de dicho bloque. Los cuatro alcances posibles para un identificador son: *alcance de función*, *alcance de archivo, alcance de bloque*, y *alcance del prototipo de función.*

Las etiquetas (un identificador seguido por dos puntos como **start:**) son los únicos identificadores con *alcance de función.* Las etiquetas pueden ser utilizadas en cualquier parte dentro de la función en la cual aparecen, pero no pueden ser referenciadas fuera del cuerpo de la función. Las etiquetas se utilizan en estructuras **switch** (como etiquetas **case**) y en enunciados **goto** (vea el capítulo 14, Temas avanzados). Las etiquetas son detalles de puesta en marcha que las funciones ocultan una de la otra. Este ocultamiento conocido formalmente como *ocultamiento de información* es uno de los principios fundamentales de la buena ingeniería de software.

Un identificador declarado por fuera de cualquier función tiene *alcance de archivo.* Tal indicador es "conocido" en todas las funciones desde el punto donde el identificador se declara hasta el final del archivo. Las variables globales, las definiciones de función y los prototipos de función colocados fuera de una función, todas ellas tienen alcance de archivo.

Los identificadores dentro de un bloque, tienen *alcance de bloque.* El alcance de bloque termina en la llave derecha de terminación (**}**) del bloque. Las variables locales declaradas al principio de una función tienen alcance de bloque, como lo tienen los parámetros de función, que son consideradas por la función como variables locales. Cualquier bloque puede contener declaracio nes variables. Cuando los bloques están anidados, y un identificador de un bloque externo tiene el mismo nombre que un identificador de un bloque interno, el identificador del bloque externo estará "oculto" hasta que el bloque interno termine. Esto significa que, en tanto se ejecute el bloque interno, el bloque interno ve el valor de su propio identificador local y no el valor del identificador de nombre idéntico, del bloque que lo contiene. Las variables locales declaradas **static**, aun tendrán alcance de bloque aunque existan a partir del momento en que empieza a ejecutarse el programa. Por lo tanto, la persitencia no afecta el alcance de un identificador.

Los únicos identificadores con *alcance de prototipo de función* son aquellos que se utilizan en la lista de parámetros del prototipo de una función. Tal y como se mencionó, los prototipos de función no requieren de nombres en la lista de parámetros —solo requieren de tipos. Si en la lista de parámetros de un prototipo de función se utiliza un nombre, el compilador ignorará dicho nombre. Los identificadores utilizados en un prototipo de función, pueden ser reutilizados en cualquier parte del programa, sin ambigüedad.

Error común de programación 5.14

Utilizar accidentalmente el mismo nombre para un identificador en un bloque interno, que se haya usado para un identificador en un bloque externo, cuando de hecho, el programador desea que durante la duración del bloque interno el identificador del bloque externo esté activo.

Práctica sana de programación 5.11

Evite nombres variables que oculten nombres en alcances externos. Esto se puede conseguir dentro de un programa evitando el uso de identificadores duplicados.

El programa de la figura 5.12 demuestra temas de alcance de variables globales, de variables locales automáticas, y de variables locales **static**. Una variable global **x** es declarada e inicializada a 1. Esta variable global se oculta en cualquier bloque (o función) en el cual una variable de nombre **x** está declarada. En **main**, una variable local **x** es declarada e inicializada a 5. Esta variable es impresa a continuación para mostrar que la **x** global está oculta en **main.** A continuación, se define un nuevo bloque en **main**, utilizando otra variable local **x** inicializada a 7. Esta variable se imprime, para mostrar que oculta a **x** en el bloque externo de **main**. La variable **x** con valor 7 de forma automática queda destruida al salir del bloque, y la variable local **x** del bloque externo de **main** se vuelve a imprimir otra vez, para mostrar que ya no está oculta. El programa define tres funciones, dónde cada una de ellas no toma argumento y no regresan nada. La función **a** define una variable automática **x**, y la inicializa a 25. Cuando se llama a **a**, la variable es impresa, incrementada y vuelta a imprimir, antes de salir de la función. Cada vez que esta función es llamada, la variable automática **x** es reinicializada a 25. La función **b** declara una variable **static** de nombre **x**, y la inicializa a 50. Las variables locales declaradas como **static** conservan sus valores, aun si están fuera de alcance. Cuando se llama a **b**, **x** es impreso, incrementado y vuelto a imprimir, antes de salir de la función. En la siguiente llamada a esta función, la variable local **static**, **x** contendrá el valor 51. La función C no declara ninguna variable. Por lo tanto, cuando se refiere a la variable **x**, se utiliza la global **x**. Cuando **c** es llamada, la variable global es impresa, multiplicada por 10, y vuelta a imprimir, antes de salir de la función. La siguiente llamada a la función **c**, la variable global aún tiene su valor modificado, 10. Por último, el programa imprime otra vez la variable local **x** de **main**, para mostrar que ninguna de las llamadas de función modificó el valor de **x**, porque las funciones todas ellas se referían a variables en otros alcances.

5.13 Recursión

Los programas que hemos analizado están estructurados en general como funciones que llaman unas a otras, en forma disciplinada y jerárquica. Para algunos tipos de problemas, es útil tener

```c
/* A scoping example */
#include <stdio.h>

void a(void);   /* function prototype */
void b(void);   /* function prototype */
void c(void);   /* function prototype */

int x = 1;      /* global variable */

main()
{
   int x = 5;   /* local variable to main */

   printf("local x in outer scope of main is %d\n", x);

   {            /* start new scope */
      int x = 7;

      printf("local x in inner scope of main is %d\n", x);
   }            /* end new scope */

   printf("local x in outer scope of main is %d\n", x);

   a();         /* a has automatic local x */
   b();         /* b has static local x */
   c();         /* c uses global x */
   a();         /* a reinitializes automatic local x */
   b();         /* static local x retains its previous value */
   c();         /* global x also retains its value */

   printf("local x in main is %d\n", x);
   return 0;
}

void a(void)
{
   int x = 25;  /* initialized each time a is called */

   printf("\nlocal x in a is %d after entering a\n", x);
   ++x;
   printf("local x in a is %d before exiting a\n", x);
}

void b(void)
{
    static int x = 50;  /* static initialization only */
                        /* first time b is called */
    printf("\nlocal static x is %d on entering b\n", x);
    ++x;
    printf("local static x is %d on exiting b\n", x);
}

void c(void)
{
   printf("\nglobal x is %d on entering c\n", x);
   x *= 10;
   printf("global x is %d on exiting c\n", x);
}
```

Fig. 5.12 Ejemplo de alcance (parte 1 de 2).


```
local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5

local x in a is 25 after entering a
local x in a is 26 before exiting a

local static x is 50 on entering b
local static x is 51 on exiting b

global x is 1 on entering c
global x is 10 on exiting c

local x in a is 25 after entering a
local x in a is 26 before exiting a

local static x is 51 on entering b
local static x is 52 on exiting b

global x is 10 on entering c
global x is 100 on exiting c
local x in main is 5
```

Fig. 5.12 Ejemplo de alcance (parte 2 de 2).

funciones que se llamen a sí mismas. Una *función recursiva* es una función que se llama a sí misma, ya sea directa, o indirecta a través de otra función. La recursión es un tema complejo analizado con profundidad en los cursos de ciencia de cómputo de alto nivel. En esta sección y en la siguiente, se presentan ejemplos simples de recursión. Este libro contiene un extenso tratamiento de la recursión, distribuído a través de los capítulos 5 hasta el 12. En la figura 5.17, localizada al final de la sección 5.15, se resumen los 31 ejemplos de recursión y ejercicios del libro.

Primero consideraremos la recursión en forma conceptual, y a continuación examinarémos varios programas que contienen funciones recursivas. Los enfoques recursivos a la solución de problemas tienen ciertos elementos en común. Una función recursiva es llamada para resolver un problema. La función, de hecho, sabe sólo cómo resolver el caso más simple, es decir el llamado *base case* (caso base o *casos base*). Si la función es llamada con caso base, la función simplemente regresa un resultado. Si la función es llamada con un problema más complejo, la función divide dicho problema en dos partes conceptuales: una parte que la función ya sabe como ejecutar, y una parte que la función no sabe como ejecutar. Para hacer factible la recursión, esta última parte debe parecerse al problema original, pero resulta ligeramente más simple o una versión ligeramente más pequeña del problema original. Dado que este nuevo problema aparenta o se ve similar al problema original, la función emite (llama) a una copia nueva de sí misma, para que empiece a trabajar sobre el problema más pequeño y esto se conoce como una *llamada recursiva* y también se llama el *paso de recursión*. El paso de recursión también incluye la palabra reservada **return**, porque su resultado será combinado con la parte del problema que la función supo como resolver, para formar un resultado que será regresado al llamador original, posiblemente **main**.

El paso de recursión se ejecuta en tanto la llamada original a la función esté abierta, es decir, que no se haya terminado su ejecución. El paso de recursión puede dar como resultado muchas

más llamadas recursivas como éstas, conforme la función continúe dividiendo cada problema sobre el cual es llamada en dos partes conceptuales. A fin de que la recursión de forma eventual se termine, cada vez que la función se llame a sí misma sobre una versión ligeramente más sencilla que el problema original, esta secuencia de problemas cada vez más pequeños, eventualmente deben de convergir en el caso base. Llegado a este punto, la función reconocerá el caso base, regresa un resultado a la copia anterior de la función, y sigue a continuación una secuencia de regresos a todo lo largo de la línea, hasta llegar a la llamada original de la función, devolviendo el resultado final a **main**. Todo esto se escucha muy exótico, en comparación con el tipo de resolución convencional de problemas que hemos estado utilizando hasta este punto. De hecho, se requiere de gran cantidad de experiencia en la escritura de programas recursivos, antes de que el proceso se convierta en natural. Como un ejemplo de estos conceptos en operación, escribamos un programa recursivo para ejecutar un cálculo matemático popular.

El factorial de un entero no negativo *n*. escrito *n!* (y pronunciado " factorial de *n* "), es el producto

$$n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$$

con 1! igual a 1, y 0! definido como 1. Por ejemplo, 5! es el producto 5 * 4 * 3 * 2 * 1, lo cual resulta igual a 120.

El factorial de un entero, **number**, mayor que o igual a 0, puede ser calculado en forma *iterativa* (y no recursiva) utilizando a **for** como sigue:

```
factorial = 1;
for (counter = number; counter >= 1; counter--)
   factorial *= counter;
```

Una definición recursiva de la función factorial se consigue al observar la siguiente relación:

$$n! = n \cdot (n - 1)!$$

Por ejemplo, 5! claramente es lo mismo que 5 * 4!, como se muestra mediante el siguiente:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$
$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$
$$5! = 5 \cdot (4!)$$

La evaluación de 5! se haría como se muestra en la figura 5.13. La figura 5.13a muestra, como la sucesión de llamadas recursivas continúa hasta que 1! se evalúa al valor 1, lo que termina la recursión. En la figura 5.13b se muestran los valores regresados por cada llamada recursiva a su llamador, hasta que el valor final es calculado y regresado.

El programa de la figura 5.14 utiliza la recursión para calcular e imprimir los factoriales de los enteros 0 a 10 (la selección del tipo de datos **long** se explicará a continuación). La función recursiva **factorial** primero prueba para ver si una condición de terminación es verdadera, es decir, es **number** menor que o igual a 1. Si **number** es en verdad menor que o igual a 1, **factorial** regresa 1, ya no es necesaria mayor recursión, y el programa termina. Si **number** es mayor que 1, el enunciado

```
return number * factorial (number - 1);
```

expresa el problema como el producto de **number** y una llamada recursiva a **factorial** evaluando el **factorial** de **number - 1**. Note que el **factorial (number - 1)** es un problema ligeramente más simple que el cálculo original **factorial (number)**.

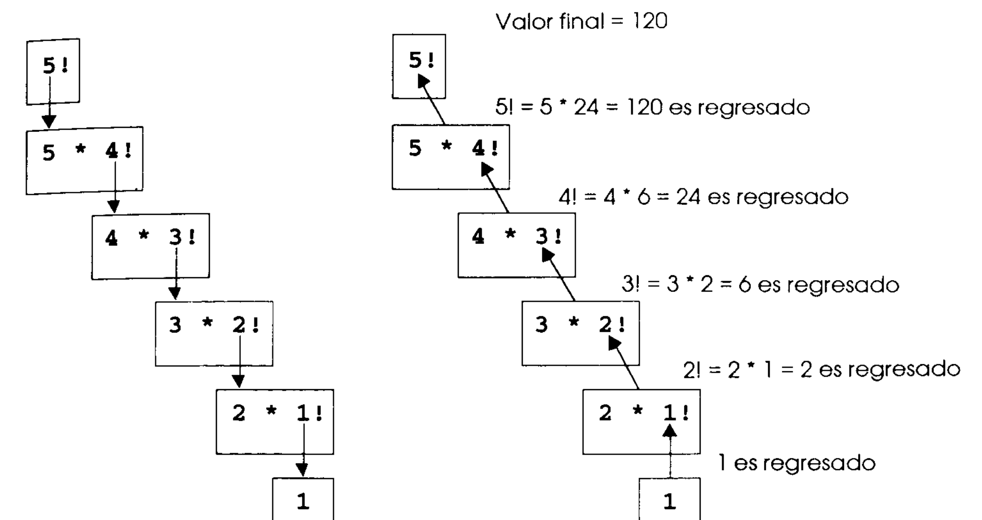


Fig. 5.13 Evaluación recursiva de 5!

La función **factorial** ha sido declarada para recibir un parámetro del tipo **long** y regresar un resultado del tipo **long**. Esto es una notación abreviada correspondiente a **long int**. El estándar ANSI define que una variable del tipo **long int** se almacena en por lo menos 4 bytes, y, por lo tanto, puede contener un valor tan grande como +2147483647. Como se puede ver en la figura 5.14, los valores factoriales con rápidez se hacen grandes. Hemos escogido el tipo de datos **long**, a fin de que el programa pueda calcular factoriales mayores que 7! en computadoras con enteros pequeños (como de 2 bytes). El especificador de conversión **%ld** se utiliza para imprimir los valores long. Desafortunadamente la función factorial produce tan rápido valores grandes que inclusive **long int** no nos ayuda a imprimir muchos valores factoriales, antes de que el tamaño de la variable **long int** quede excedida.

Conforme exploremos los ejercicios, **float** y **double** pudieran al final ser necesarios para el usuario que desee calcular factoriales de número grandes. Esto apunta a una debilidad en C (y en la mayor parte de otros lenguajes de programación), es decir, que el lenguaje no se extiende con fácilidad para manejar los requisitos únicos de varias aplicaciones. Como veremos más adelante, C++ es un lenguaje extensible, que nos permite, si así lo deseamos, crear enteros grandes en forma arbitraria.

Error común de programación 5.15

Olvidar el regresar un valor de una función recursiva cuando se requiere de uno.

Error común de programación 5.16

Omitir ya sea el caso base, o escribir el paso de recursión en forma incorrecta, de tal forma que no converja al caso base, causando recursión infinita, y agotando de forma eventual la memoria. Esto es análogo al problema de un ciclo infinito en una solución iterativa (no recursiva). La recursión infinita también puede ser causada al proporcionarle una entrada no esperada.

```
/* Recursive factorial function */
#include <stdio.h>

long factorial(long);

main()
{
   int i;

   for (i = 1; i <= 10; i++)
      printf("%2d! = %ld\n", i, factorial(i));

   return 0;
}

/* Recursive definition of function factorial */
long factorial(long number)
{
   if (number <= 1)
      return 1;
   else
      return (number * factorial(number - 1));
}
```

```
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

Fig. 5.14 Cálculos fáctoriales con una función recursiva.

5.14 Ejemplo utilizando recursión: la serie Fibonacci

La serie Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

empieza con 0 y 1, y tiene la propiedad que cada número Fibonacci subsecuente es la suma de los dos números Fibonacci previos.

La serie ocurre en la naturaleza y en particular describe una forma de espiral. La relación de números sucesivos Fibonacci converge a un valor constante de 1.618.... Este número, también, ocurre en forma repetida en la naturaleza y ha sido llamada la *regla áurea* o la *media áurea*.

Los seres humanos tienen tendencia a encontrar la media áurea estéticamente agradable. Los arquitectos a menudo diseñan las ventanas, habitaciones y edificios cuya longitud de ancho están en la relación de la media áurea. Las tarjetas postales a menudo se diseñan en una relación de longitud/ancho de la media áurea.

La serie Fibonacci puede ser definida de forma recursiva como sigue:

$fibonacci(0) = 0$
$fibonacci(1) = 1$
$fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)$

El programa de la figura 5.15 calcula el número Fibonacci i^a en forma recursiva, utilizando la función **fibonacci**. Advierta que los números Fibonacci tienden con rápidez hacerse grandes. Por lo tanto, en la función **fibonacci** hemos escogido el tipo de datos **long** para el tipo de parámetros y para el de regreso. En la figura 5.15, cada par de líneas de salida muestra una ejecución separada del programa.

La llamada de **fibonacci** a partir de **main** no es una llamada recursiva, pero todas las llamadas subsecuentes a **fibonacci**, sí son recursivas. Cada vez que se invoca a **fibonacci**, de inmediato prueba por el caso base **n** es igual a 0 ó a 1. Si esto es verdadero, se regresa **n**. De forma interesante, si **n** es mayor que 1, el paso de recursión genera *dos* llamadas recursivas, cada una de las cuales es para un problema ligeramente más sencillo que la llamada original a **fibonacci**. La figura 5.16 muestra cómo la función **fibonacci** evaluaría **fibonacci(3)**, sólo abreviamos **fibonacci** como una **f**, a fin de hacer la figura más legible.

```
/* Recursive fibonacci function */
#include <stdio.h>

long fibonacci(long);

main()
{
   long result, number;

   printf("Enter an integer: ");
   scanf("%ld", &number);
   result = fibonacci(number);
   printf("Fibonacci(%ld) = %ld\n", number, result);
   return 0;
}

/* Recursive definition of function fibonacci */
long fibonacci(long n)
{
   if (n == 0 || n == 1)
      return n;
   else
      return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Fig. 5.15 Los números Fibonacci generan en forma recursiva (parte 1 de 2).

```
Enter an integer: 0
Fibonacci(0) = 0

Enter an integer: 1
Fibonacci(1) = 1

Enter an integer: 2
Fibonacci(2) = 1

Enter an integer: 3
Fibonacci(3) = 2

Enter an integer: 4
Fibonacci(4) = 3

Enter an integer: 5
Fibonacci(5) = 5

Enter an integer: 6
Fibonacci(6) = 8

Enter an integer: 10
Fibonacci(10) = 55

Enter an integer: 20
Fibonacci(20) = 6765

Enter an integer: 30
Fibonacci(30) = 832040

Enter an integer: 35
Fibonacci(35) = 9227465
```

Fig. 5.15 Los números FIbonacci generan en forma recursiva (Parte 2 de 2)

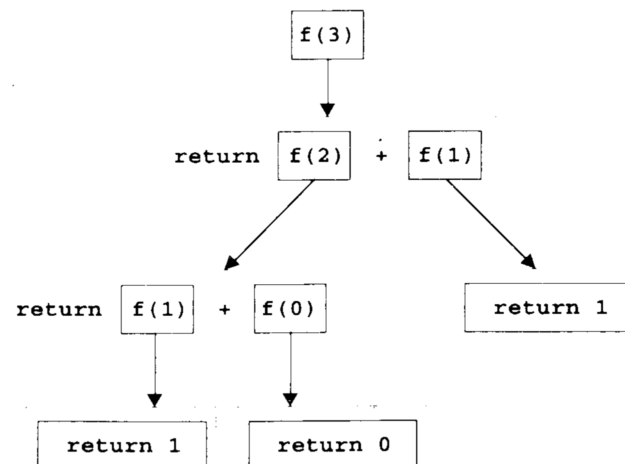


Fig. 5.16 Conjunto de llamadas recursivas a la función fibonacci.

Esta figura plantea algunos temas interesantes en relación con el orden en el cual los compiladores de C evaluarán los operandos de operadores. Se trata de un tema distinto al orden en el cual los operadores son aplicados a sus operandos, es decir al orden dictado por las reglas de precedencia de operadores. A partir de la figura 5.16 aparece que durante la evaluación de `f(3)`, se harán dos llamadas recursivas, es decir `f(2)` y `f(1)`. ¿Pero en qué orden se harán estas llamadas? La mayor parte de los programadores supone que los operandos serán evaluados de izquierda a derecha. De forma extraña, el estándar ANSI no especifica el orden en que deban ser evaluados los operandos de la mayor parte de los operadores (incluyendo `+`). Por lo tanto, el programador no debería hacer ninguna suposición en relación con el orden en el cual se ejecutarán estas llamadas. Las llamadas de hecho podrían ejecutar `f(2)` primero y a continuación `f(1)`, o las llamadas podrían ejecutarse en el orden inverso, `f(1)` y a continuación `f(2)`. En este programa y en la mayor parte de otros programas, de hecho el resultado final sería el mismo. Pero en algunos programas la evaluación de un operando pudiera tener efectos colaterales que pudieran afectar el resultado final de la expresión. De los muchos operadores de C, el estándar ANSI especifica el orden de evaluación de los operandos de sólo cuatro operadores —es decir, `&&`, `||`, el operador coma (`,`) e `?:`. Los primeros tres son operadores binarios, cuyos dos operandos están garantizados en su evaluación de izquierda a derecha. El último operador es el único operador ternario de C. Su operando más a la izquierda es el que primero se evalúa siempre; si el operando más a la izquierda tiene el valor de no cero, se evalúa a continuación el operando de enmedio y el último se ignora; si el operando más a la izquierda se evalúa a cero, el tercer operando se evalúa a continuación y el operando central es ignorado.

Error común de programación 5.17

Escribir programas que dependan del orden de evaluación de los operandos de operadores distintos que `&&`, `||`, `?:`, y el operador coma (`,`) puede llevar a errores, porque los compiladores no necesariamente evalúan los operandos en el orden en que los programas esperan.

Sugerencia de portabilidad 5.2

Los programas que dependen del orden de evaluación de los operandos o de operadores, diferentes a &&, ||, ?:, y el operador coma (,) pueden funcionar de forma distinta sobre sistemas con compiladores diferentes.

Una palabra de precaución merece ser dicha en relación con programas recursivos, como el que estamos utilizando aquí para generar números Fibonacci. Cada nivel de recursión en la función **fibonacci** tiene un efecto duplicador sobre el número de llamadas, es decir, el número de llamadas recursivas que se ejecutarán para calcular el número Fibonacci del orden *n* será del orden de 2^n. Esto se sale rápido de control. Simplemente calcular el 20º número Fibonacci requeriría del orden de 2^{20} o aproximadamente de un millón de llamadas, y calcular el número Fibonacci del orden treintavo requeriría del orden de 2^{30} o aproximadamente de mil millones de llamadas, y así en lo sucesivo. Los científicos de la computación se refieren a lo anterior como *complejidad exponencial*. ¡Problemas de esta naturaleza inclusive humillan a las computadoras más poderosas del mundo!. Los temas de complejidad en general, y la complejidad exponencial en particular, se analizan en detalle en un curso del curriculum de ciencia de las computadoras de alto nivel conocido como "Algoritmos".

Sugerencia de rendimiento 5.4

Evite programas recursivos de tipo fibonacci que resultan en una "explosión" exponencial de llamadas.

5.15 Recursión en comparación con iteración

En las secciones anteriores, hemos estudiado dos funciones que pueden con fácilidad instaurarse, ya sea en forma recursiva o en forma iterativa. En esta sección compararemos los dos enfoques y analizaremos por qué el programador debe escoger un enfoque sobre el otro, en una situación en particular.

Tanto la iteración como la recursión se basan en una estructura de control: la iteración utiliza una estructura de repetición; la recursión utiliza una estructura de selección. Tanto la iteración como la recursión implican repetición: la iteración utiliza la estructura de repetición en forma explícita; la recursión consigue la repetición mediante llamadas de función repetidas. La iteración y la recursión ambas involucran una prueba de terminación: la iteración termina cuando falla la condición de continuación de ciclo; la recursión termina cuando se reconoce un caso base. La iteración con una repetición controlada por contador y la recursión ambas se acercan de forma gradual a la terminación: la iteración continúa modificando un contador hasta que éste asume un valor que hace que la condición de continuación del ciclo falle; la recursión continúa produciendo versiones más sencillas del problema original hasta que se alcanza el caso base. Tanto la iteración como la recursión pueden ocurrir en forma indefinida: ocurre un ciclo infinito en la iteración si la prueba de continuación de ciclo nunca se hace falsa; la recursión infinita ocurre si el paso de recursión no reduce el problema en cada ocasión, de tal forma que converja al caso base.

La recursión tiene muchas negativas. Invoca en forma repetida al mecanismo y, por lo tanto, a la sobrecarga de las llamadas de función. Esto puede resultar costoso tanto en tiempo de procesador como en espacio en memoria. Cada llamada recursiva genera otra copia de la función (de hecho sólo de las variables de función); esto puede consumir gran cantidad de memoria. La iteración por lo regular ocurre dentro de una función, por lo que no ocurre la sobrecarga de llamadas repetidas de función y asignación extra de memoria. Por lo tanto, ¿por qué elegir la recursión?

Observación de ingeniería de software 5.12

Cualquier problema que puede ser resuelto en forma recursiva, también puede ser resuelto en forma iterativa (no recursiva). Normalmente se escoge un enfoque recursivo en preferencia a uno iterativo cuando el enfoque recursivo es más natural al problema y resulta en un programa que sea más fácil de comprender y de depurar. Otra razón para seleccionar una solución recursiva, es que la solución iterativa pudiera no resultar aparente.

Sugerencia de rendimiento 5.5

Evite el uso de la recursión cuando se requiera de rendimiento. Las llamadas recursivas toman tiempo y consumen memoria adicional.

Error común de programación 5.18

Hacer que accidentalmente una función no recursiva se llame a sí misma, ya sea directa o indirecta a través de otra función.

Muchos libros de texto de programación presentan la recursión mucho más adelante de lo que nosotros hemos hecho aquí. Pensamos que la recursión es un tema lo suficiente rico y complejo que es mejor presentarlo pronto y repartir los ejemplos sobre el resto del libro. La figura 5.17 resume por capítulos los 31 ejemplos de recursión y los ejercicios del libro.

Cerremos este capítulo con algunas observaciones que hacemos en forma repetida a todo lo largo del libro. Es importante la buena ingeniería de software. Es importante un alto rendimiento. Desafortunadamente, a menudo estas metas se contraponen. Es clave una buena ingeniería de

software para hacer manejable la tarea de desarrollar sistemas mayores y más complejos de software que estamos necesitando. Un alto rendimiento es clave también para llevar a cabo los sistemas del futuro que colocarán crecientes demandas sobre el hardware. ¿Dónde entran en este panorama las funciones?

Observación de ingeniería de software 5.13

La funcionalización de los programas de una forma nítida y jerárquica promueve buena ingeniería de software. Pero tiene un costo.

Sugerencia de rendimiento 5.6

Un programa muy funcionalizado en comparación con uno monolítico (es decir, de una pieza) sin funciones potencialmente hace grandes cantidades de llamadas de función y estas consumen tiempo de ejecución en el procesador de una computadora. Pero los programas monolíticos son difíciles de programar, de probar, de depurar, de mantener y de modificar.

Por lo tanto, funcionalice sus programas con juicios, manteniendo en mente siempre un delcado equilibrio entre rendimiento y buena ingeniería de software.

Resumen

- La mejor forma de desarrollar y mantener un programa grande es dividirlo en varios módulos de programa más pequeños, siendo cada uno de ellos más manejables que el programa original. En C los módulos se escriben como funciones.
- Una función se invoca mediante una llamada de función. La llamada de función menciona a la función por su nombre y proporciona información (en forma de argumentos) que la función llamada requiere para ejecutar su tarea.
- El objeto del ocultamiento de información es para que las funciones tengan acceso sólo a la información que requieren para completar sus tareas. Esto significa instaurar el principio del mínimo privilegio, uno de los principios más importantes de la buena ingeniería de software.
- Las funciones por lo regular se invocan en un programa escribiendo el nombre de la función, seguido por un paréntesis izquierdo, a su vez por el *argumento* (o una lista separada por comas de los argumentos) de la función, seguida por un paréntesis derecho.
- El tipo de datos `double` es un tipo de punto flotante similar a `float`. Una variable de tipo `double` puede almacenar un valor de una magnitud mayor y con una precisión mayor de lo que puede almacenar un `float`.
- Cada argumento de una función puede ser una constante, una variable o una expresión.
- Una variable local es conocida sólo en una definición de función. A otras funciones no se les permite conocer los nombres de las variables locales de una función, ni se le permite a ninguna función conocer los detalles de instauración de cualquier otra función.
- El formato general para una definición de función es

  tipo de valor de regreso nombre de función (lista de parámetros)
  {
  declaraciones

  enunciados
  }

Capítulo	Ejemplos y ejercicios de recursión
Capítulo 5	Función factorial
	Funciones Fibonacci
	Máximo común divisor
	Sumar dos enteros
	Multiplicar dos enteros
	Elevar un entero a una potencia entera
	Torres de Hanoi
	main recursivo
	Imprimir entradas del teclado a la inversa
	Visualización de la recursión
Capítulo 6	Sumar los elementos de un arreglo
	Imprimir un arreglo
	Imprimir un arreglo a la inversa
	Imprimir una cadena a la inversa
	Verificar si una cadena es un palíndromo
	Valor mínimo en un arreglo
	Clasificación de selección
	Clasificación rápida
	Búsqueda lineal
	Búsqueda binaria
Capítulo 7	Ocho reinas
	Atravesar Maze
Capítulo 8	Imprimir una entrada de cadena desde el teclado a la inversa
Capítulo 12	Inserción de lista enlazada
	Eliminación de lista enlazada
	Búsqueda en una lista enlazada
	Impresión a la inversa de una lista enlazada
	Inserción de un árbol binario
	Recorrido en preorden de un árbol binario
	Recorrido en orden de un árbol binario
	Recorrido en postorden de un árbol binario

Fig. 5.17 Resumen de ejemplos y ejercicios de recursión del libro.

- El *tipo de valor de regreso*, indica el tipo de valor regresado a la función llamadora. Si la función no regresa un valor, el *tipo de valor de regreso* se declara como **void**. El *nombre de función* es cualquier identificador válido. La *lista de parámetros* es una lista separada por comas, que contiene las declaraciones de las variables que serán pasadas a la función. Si una función no recibe ningún valor, la *lista de parámetros* se declara como **void**. El *cuerpo de función* es el conjunto de declaraciones y de enunciados que la constituyen.
- Los argumentos pasados a una función deberán coincidir en número, tipo y orden con los parámetros en la definición de función.

- Cuando un programa encuentra una función, el control se transfiere desde el punto de invocación a la función llamada, los enunciados de la función llamada se ejecutan, y el control se regresa al llamador.
- Una función llamada puede devolver el control al llamador de una de tres formas diferentes. Si la función no regresa un valor, el control se regresa con la llave derecha de terminación de función, o con la ejecución del enunciado

```
return;
```

- Si la función regresa un valor, el enunciado

```
return expression;
```

devolverá el valor de **expression**.

- Un prototipo de función declara el tipo devuelto por de la función y declara el número, tipos y órdenes de los parámetros que la función espera recibir.
- Los prototipos de función permiten al compilador verificar que las funciones están llamadas correctamente.
- El compilador ignorará los nombres de variable mencionadas en el prototipo de función.
- Cada biblioteca estándar tiene su correspondiente archivo de cabecera, conteniendo los prototipos de función para todas las funciones en dicha biblioteca, así como las definiciones de varias constates simbólicas necesitadas por dichas funciones.
- Los programadores pueden crear e incluir sus propios archivos de cabecera.
- Cuando un argumento se pasa en llamada por valor, se efectúa una *copia* del valor de la variable y la copia se pasa a la función llamada. Las modificaciones a la copia de la función llamada no afectan al valor original de la variable.
- Todas las llamadas en C son llamadas por valor.
- La función **rand** genera un entero entre 0 y **RAND_MAX** definido por el estándar ANSI C a ser por lo menos 32767.
- Los prototipos de función de **rand** y **srand** están contenidas en **<stdlib.h>**.
- Los valores producidos por **rand** pueden ser dimensionados y desplazados para producir valores dentro de un rango específico.
- Para aleatorizar un programa, utilice la función estándar **srand** de biblioteca de C.
- El enunciado **srand** se inserta normalmente en un programa, sólo después de que el programa haya sido en totalidad depurado. Durante la depuración, es mejor omitir **srand**. Esto asegura la capacidad de repetición, que es esencial para comprobar que las correcciones a un programa de generación de números aleatorios funcione correctamente.
- A fin de poder tener números aleatorios sin necesidad de introducir cada vez una semilla, podemos utilizar srand(time(NULL)). La función time regresa el número de segundos a partir del principio del día. El prototipo de función **time** está localizada en la cabecera **<time.h>**
- La ecuación general para dimensionar y desplazar un número aleatorio es

```
n = a + rand() % b;
```

donde **a** es el valor a desplazar (que es igual al primer número del rango deseado de enteros consecutivos), y **b** es el factor de dimensionamiento (que es igual al ancho del rango deseado de enteros consecutivos).

UNIVERSIDAD DE LA REPÚBLICA FACULTAD DE INGENIERÍA

- Cada identificador en un programa tiene los atributos de clase de almacenamiento, persistencia, alcance y enlace.
- C proporciona cuatro clases de almacenamiento indicados por los especificadores de clase de almacenamiento (persistencia): **auto**, **register**, **extern** y **static**.
- La persistencia de un identificador es el tiempo que el identificador existe en memoria.
- El alcance de un identificador es dónde puede ser referenciado dicho identificador dentro de un programa.
- El enlace de un identificador determina para un programa de archivos de fuentes múltiples si un identificador es conocido sólo en el archivo fuente actual o en cualquiera de los archivos fuentes mediante declaraciones apropiadas.
- Las variables con persistencia automático son creadas cuando se introduce el bloque en el cual están declaradas, existen mientras el bloque está activo, y se destruyen cuando el bloque se termina. Las variables locales de una función por lo regular tienen persistencia automática.
- El especificador de clase de almacenamiento **register** puede ser colocado delante de una declaración de variable automática, para sugerir que el compilador mantenga la variable en alguno de los registros de hardware de alta velocidad de la computadora. El compilador pudiera ignorar las declaraciones **register**. La palabra reservada **register** puede ser utilizada sólo con variables de persistencia automática.
- Las palabras reservadas **extern** y **static** se utilizan para declarar identificadores de variables y de función de persistencia estática.
- Las variables con persistencia estática, son asignadas e inicializadas una vez cuando el programa inicie su ejecución.
- Existen dos tipos de identificadores para la persistencia estática: los identificadores externos (como las variables y nombres de funciones globales) y las variables locales declaradas utilizando el especificador de clase de almacenamiento **static**.
- Las variables globales se crean colocando declaraciones variables fuera de cualquier definición de función, y conservan sus valores a todo lo largo de la ejecución del programa.
- Las variables locales declaradas como **static** conservan su valor cuando la función en las cuales han sido declaradas se termina.
- Todas las variables numéricas de persistencia estática se inicializan a cero, si no son de forma explícita incializadas por el programador.
- Los cuatro alcances para un identificador son: alcance de función, alcance de archivo, alcance de bloque y alcance de prototipo de función.
- Las etiquetas son los únicos identificadores con alcance de función. Las etiquetas pueden ser utilizadas en cualquier parte en la función en la cual aparecen, pero no pueden ser referenciadas fuera del cuerpo de la función.
- Un identificador declarado fuera de cualquier función tiene un alcance de archivo. Tal identificador es "conocido" en todas las funciones, desde el momento en el cual es declarado el identificador, hasta el final del archivo.
- Los identificadores declarados dentro de un bloque tienen un alcance de bloque. El alcance de bloque termina al terminar la llave derecha (**}**) de dicho bloque.

- Las variables locales declaradas al principio de una función tienen alcance de bloque, igual que los parámetros de función, que son considerados como variables locales por la función.
- Cualquier bloque puede contener declaraciones variables. Cuando los bloques están anidados, y un identificador de un bloque exterior tiene el mismo nombre que un identificador de un bloque interior, el identificador del bloque exterior está "oculto" hasta que termine el bloque interior.
- Los únicos identificadores con alcance de prototipo de función son aquellos utilizados en la lista de parámetros de un prototipo de función. Los identificadores utilizados en un prototipo de función pueden ser vueltos a utilizar sin ambigüedad en alguna otra parte del programa.
- Una función recursiva es una función que se llama a sí misma, ya sea directa o indirecta.
- Si una función recursiva es llamada con un caso base, la función simplemente regresa un resultado. Si la función es llamada con un problema más complejo, la función divide el problema en dos partes conceptuales: una parte que la función sabe cómo ejecutar y una versión ligeramente más pequeña del problema original. Dado que este nuevo problema asemeja el problema original, la función emite una llamada recursiva, para trabajar en el problema menor.
- Para que termine una recursión, cada vez que la función recursiva se llama a sí misma, con una versión ligeramente más simple del problema original, la secuencia de problemas más y más pequeños debe de converger al caso base. Cuando la función reconoce el caso base, el resultado es regresado a la llamada de función previa, y a continuación sigue una secuencia de regresos hacia atrás, hasta que la llamada original de la función eventualmente devuelve el resultado final.
- El estándar ANSI no especifica el orden en el cual deben ser evaluados los operandos de la mayor parte de los operadores (incluyendo el signo de **+**). De los muchos operadores de C, el estándar especifica el orden de evaluación de operandos de los operadores **&&**, **||**, el operador coma (**,**) y **?:**. Los primeros tres anteriores son operadores binarios cuyos operandos se evalúan de izquierda a derecha. El último operador es el único operador ternario de C. Su operando más a la izquierda se evalúa primero; si el operando más a la izquierda toma el valor diferente de cero, el operando intermedio se evalúa a continuación ignorándose el operando a la derecha; si el operando más a la izquierda tiene el valor de cero, el tercer operando se evalúa a continuación y el operando de en medio es ignorado.
- Tanto la iteración como la recursión se basan en una estructura de control: la iteración utiliza una estructura de repetición; la recursión una estructura de selección.
- Tanto la iteración como la recursión incluyen repetición: la iteración utiliza una estructura de repetición en forma explícita; la recursión consigue la repetición mediante llamadas de función repetidas.
- La iteración y la recursión ambas incluyen una prueba de terminación: la iteración termina cuando falla la condición de continuación de ciclo; la recursión termina cuando se reconoce el caso base.
- Tanto la iteración como la recursión pueden ocurrir en forma infinita: ocurrirá un ciclo infinito en el caso de la iteración si la prueba de continuación de ciclo nunca se convierte en falsa; ocurrirá la recursión infinita si el paso de recursión no reduce el problema de tal forma que converja en el caso base.
- La recursión invoca el mecanismo en forma repetida y, por lo tanto, sobrecarga las llamadas de función. Esto puede resultar costoso tanto en tiempo de procesador como en espacio de memoria.

Terminología

abstracción
argumento en una llamada de función
almacenamiento automático
persistencia automática
variable automática
especificador de clase de almacenamiento **auto**
caso base en recursión
bloque
alcance de bloque
biblioteca estándar de C
llamada a función
llamada por referencia
llamada por valor
función llamada
llamador
función llamadora
clock
coerción de argumentos
copia de un valor
dividir y vencer
elemento de azar
especificador de clase de almacenamiento **extern**
función factorial
simulación
función
llamada de función
declaración de función
definición de función
prototipo de función
alcance de prototipo de función
alcance de función
variable global
archivo de cabecera
ocultamiento de información
invocar una función
iteración
enlace
variable local
funciones matemáticas de biblioteca
expresión de tipo mixto
programa modular
compilador optimizador
parámetro en una definición de función
principio del mínimo privilegio
función definida por el programador
jerarquía de promoción
números seudoaleatorios
rand
RAND_MAX
hacer aleatorio
generación de números aleatorios
recursión
llamada recursiva
función recursiva
especificador de clase de almacenamiento **register**
return
tipo de valor de regreso
dimensionamiento
especificador de conversión **%s**
alcance
desplazamiento
efectos colaterales

ingeniería de software
reutilización de software
srand
archivos de cabecera de la biblioteca estándar
especificador de clase de almacenamiento **static**
persistencia estática
variable **static**
clases de almacenamiento
especificador de clase de almacenamiento
persistencia
time
unsigned
void

Errores comunes de programación

5.1 Olvidar incluir el archivo de cabecera matemático, al usar las funciones matemáticas de biblioteca, puede causar resultados extraños

5.2 Omitir el tipo de valor de regreso en una definición de función causa un error de sintaxis, si el prototipo de función especifica un regreso de tipo distinto a `int`.

5.3 Olvidar regresar un valor de una función, que se supone debe regresar un valor, puede llevar a errores inesperados. El estándar ANSI indica que el resultado de esta omisión queda indefinido.

5.4 Regresar un valor de una función, cuyo tipo de regreso se ha declarado como `void`, causará un error de sintaxis.

5.5 Declarar parámetros de función del mismo tipo como `float x, y` en vez de `float x, float y`. La declaración de parámetros `float x, y` convertiría de hecho a *y* en un parámetro del tipo `int`, porque `int` es el valor por omisión.

5.6 Es un error de sintaxis colocar un punto y coma después del paréntesis derecho que encierra una lista de parámetros de una definición de función.

5.7 Volver a definir dentro de la función un parámetro de función como variable local es un error de sintaxis.

5.8 Definir una función en el interior de otra función es un error de sintaxis.

5.9 Olvidar el punto y coma al final de prototipo de función hará que ocurra un error de sintaxis.

5.10 Convertir de un tipo de datos superior en la jerarquía de promoción a un tipo inferior, puede modificar el valor del dato.

5.11 Olvidar un prototipo de función generará un error de sintaxis, si el tipo de regreso de la función no es `int` y la definición de función aparece después de la llamada a la función dentro del programa. De lo contrario, el olvidar un prototipo de función puede causar un error en tiempo de ejecución o un resultado inesperado.

5.12 Usar `srand` en vez de `rand` para generar números aleatorios.

5.13 Usar múltiples especificadores de clase de almacenamiento para un identificador. Sólo puede ser aplicado un especificador de clase de almacenamiento a un identificador.

5.14 Utilizar accidentalmente el mismo nombre para un identificador en un bloque interno, que se haya usado para un identificador en un bloque externo, cuando de hecho, el programador desea que durante la duración del bloque interno el identificador del bloque externo esté activo.

5.15 Olvidar el regresar un valor de una función recursiva cuando se requiere de uno.

5.16 Omitir ya sea el caso base, o escribir el paso de recursión en forma incorrecta, de tal forma que no converja al caso base, causando recursión infinita, y agotando de forma eventual la memoria. Esto es análogo al problema de un ciclo infinito en una solución iterativa (no recursiva). La recursión infinita también puede ser causada al proporcionarle una entrada no esperada.

5.17 Escribir programas que dependan del orden de evaluación de los operandos de operadores distintos que `&&`, `||`, `?:`, y el operador coma (`,`) puede llevar a errores, porque los compiladores no necesariamente evalúan los operandos en el orden en que los programas esperan.

5.18 Hacer que accidentalmente una función no recursiva se llame a sí misma, ya sea directa o indirecta a través de otra función.

Prácticas sanas de programación

5.1 Familiarícese con la amplia colección de funciones de la biblioteca estándar ANSI C.

5.2 Incluya el archivo de cabecera de matemáticas utilizando la directiva de preprocesador `#include <math.h>` cuando esté utilizando funciones de la biblioteca de matemáticas.

5.3 Coloque una línea en blanco entre definiciones de función, para separarlas y para mejorar la legibilidad del programa.

5.4 Aun cuando un tipo de regreso omitido resulte en `int` por omisión, declare siempre en forma explícita el tipo de regreso. Sin embargo, normalmente, se omite el tipo de regreso correspondiente a `main`.

5.5 Incluya en la lista de parámetros el tipo de cada parámetro, inclusive si algún parámetro es del tipo por omisión `int`.

5.6 Aunque hacerlo no es incorrecto, no utilice los mismos nombres para argumentos pasados a una función y parámetros correspondientes de la definición de función. Con ello ayuda a evitar ambigüedad.

5.7 Seleccionar nombres significativos para funciones y nombres significativos para parámetros, hace que sean más legibles los programas y ayuda a evitar un uso de comentarios excesivo.

5.8 Incluya prototipos de función para todas las funciones para aprovechar las capacidades de C de verificación de tipo. Utilice las directivas de preprocesador `#include` para obtener prototipos para las funciones estándar de biblioteca a partir de los archivos de cabecera de las bibliotecas apropiadas. También utilice `#include` para obtener archivos de cabecera que contengan prototipos de función utilizados por usted y los miembros de su grupo.

5.9 Los nombres de los parámetros a veces se incluyen en los prototipos de función por razones de documentación. El compilador ignora estos nombres.

5.10 Las variables utilizadas sólo en una función particular deberían ser declaradas como variables locales en esa función, en vez de variables externas.

5.11 Evite nombres variables que oculten nombres en alcances externos. Esto se puede conseguir dentro de un programa evitando el uso de identificadores duplicados.

Sugerencias de portabilidad

5.1 Usar funciones de la biblioteca estándar ANSI C auxilia a que los programas sean más portables.

5.2 Los programas que dependen del orden de evaluación de los operandos o de operadores, diferentes a `&&`, `||`, `?:`, y el operador coma (`,`) pueden funcionar de forma distinta sobre sistemas con compiladores diferentes.

Sugerencias de rendimiento

5.1 El almacenamiento automático es una forma de conservar memoria, porque las variables automáticas existen sólo cuando son necesitadas. Son creadas al introducirse la función en la cual son declaradas, y son destruidas cuando se sale de dicha función.

5.2 El especificador de clase de almacenamiento `register` puede ser colocado antes de una declaración de variable automática, para sugerir que el compilador conserve la variable en uno de los registros de alta velocidad del hardware de la computadora. Si se puede mantener en registros de hardware las variables intensamente utilizadas como son contadores y totales, puede ser eliminada la sobrecarga correspondiente a cargar en forma repetida las variables de las memorias a los registros y almacenar los resultados de vuelta en memoria.

5.3 A menudo, son innecesarias las declaraciones `register`. Los compiladores optimizadores de hoy día son capaces de reconocer variables de uso frecuente, y pueden decidir colocarlos en registro, sin necesidad de una declaración `register` proveniente del programador.

5.4 Evite programas recursivos de tipo fibonacci que resultan en una "explosión" exponencial de llamadas.

5.5 Evite el uso de la recursión cuando se requiera de rendimiento. Las llamadas recursivas toman tiempo y consumen memoria adicional.

5.6 Un programa muy funcionalizado en comparación con uno monolítico (es decir de una pieza) sin funciones potencialmente hace grandes cantidades de llamadas de función y estas consumen tiempo de ejecución en el procesador de una computadora. Pero los programas monolíticos son difíciles de programar, probar, depurar, mantener y de modificar.

Observaciones de ingeniería de software

5.1 Evite reinventar la rueda. Siempre que sea posible, utilice funciones estándar de biblioteca ANSI C, en vez de escribir nuevas funciones. Esto reduce el tiempo de desarrollo del programa.

5.2 En programas que contengan muchas funciones, `main` deberá de ser organizada como un grupo de llamadas a funciones que ejecuten la mayor parte del trabajo del programa.

5.3 Cada función debería limitarse a ejecutar una tarea sencilla y bien definida, y el nombre de la función debería expresar dicha tarea con claridad. Ello facilitaría la abstracción y promovería la reutilización del software.

5.4 Si no puede elegir un nombre conciso, que exprese lo que la función ejecuta, es probable que su función está intentando ejecutar demasiadas tareas diversas. A menudo es mejor dividir dicha función en varias funciones más pequeñas.

5.5 Una función no debería tener una longitud mayor que una página. Aún mejor, una función no debería ser más larga que media página. Las funciones pequeñas promueven la reutilización del software.

5.6 Los programas deberían ser escritos como recopilaciones de pequeñas funciones. Esto haría que los programas fuesen más fáciles de escribir, depurar, mantener y de modificar.

5.7 Una función que requiera un gran número de parámetros quizás esté ejecutando demasiadas tareas. Piense en dividir esta función en funciones más pequeñas, que ejecuten las tareas por separado. El encabezado de función, si es posible, debería poder caber en una línea.

5.8 El prototipo de función, el encabezado de función y las llamadas de función deberán todas estar de acuerdo en lo que se refiere al número, tipo y orden de argumentos y de parámetros, así como en el tipo de valor de regreso.

5.9 Un prototipo de función, colocado fuera de una definición de función, se aplica a todas las llamadas de función que aparezcan dentro del archivo después del prototipo de función. Un prototipo de función colocado en una función, se aplica sólo a las llamadas efectuadas en esa función.

5.10 El almacenamiento automático es otra vez un ejemplo del principio del menor privilegio. ¿Por qué tendrían que estar las variables almacenadas en memoria y accesibles cuando de hecho no son necesarias?

5.11 La declaración de una variable como global en vez de como local, permite que ocurran efectos colaterales no deseados cuando una función que no requiere de acceso a la variable la modifica accidental o maliciosamente. En general, el uso de las variables globales debería ser evitado, excepto en ciertas situaciones, con requerimientos de rendimiento únicos (como se analizan en el capítulo 14).

5.12 Cualquier problema que puede ser resuelto en forma recursiva, también puede ser resuelto en forma iterativa (no recursiva). Por lo regular se escoge un enfoque recursivo en preferencia a uno iterativo cuando el enfoque recursivo es más natural al problema y resulta en un programa que sea más fácil de comprender y de depurar. Otra razón para seleccionar una solución recursiva, es que la solución iterativa pudiera no resultar aparente.

5.13 La funcionalización de los programas de una forma nítida y jerárquica promueve buena ingeniería de software. Pero tiene un costo.

Ejercicios de autoevaluación

5.1 Llene cada uno de los siguientes espacios en blanco:

a) En C un módulo de programas se conoce como un _______.
b) Una función se invoca mediante un _______.
c) Una variable que es conocida solo dentro de la función en la cual está definida se conoce como una _______.
d) Un enunciado _______ en una función llamada se utiliza para pasar el valor de una expresión de regreso a la función llamada.
e) La palabra reservada _______ se utiliza en un encabezado de función para indicar que una función no regresará un valor o para indicar que la función no contiene parámetros.
f) El _______ de un identificador es la parte del programa en el cual se puede utilizar dicho identificador.
g) Las tres distintas maneras para regresar control a partir de una función llamada hacia su llamador son _______, _______, y _______.
h) Una _______ le permite al compilador verificar el número, tipos y orden de los argumentos pasados a una función.
i) La función _______ se utiliza para producir números aleatorios.

j) La función _______ se utiliza para establecer la semilla de números aleatorios, a fin de hacer aleatorio un programa.
k) Los especificadores de clase de almacenamiento son _______, _______, _______, y _______.
l) Las variables declaradas en un bloque o en la lista de parámetros de una función se suponen son de la clase de almacenamiento _______, a menos de que se especifique lo contrario.
m) El especificador de clase de almacenamiento _______ es una recomendación para el compilador para que almacene una variable en uno de los registros de la computadora.
n) Una variable declarada por fuera de cualquier bloque o de función es una variable _______.
o) Para que la variable local en una función conserve su valor entre llamadas a la función, debe de ser declarada como de especificador de clase de almacenamiento _______.
p) Los cuatro posibles alcances de un identificador son _______, _______, _______, y _______.
q) Una función que se llama a sí misma, ya sea directa o indirecta es una función _______.
r) Una función recursiva típica consta de dos componentes: uno que proporciona una forma para que la recursión se termine mediante la prueba buscando un caso _______, y otra que expresa el problema como una llamada recursiva para un problema ligeramente más simple que la llamada original.

5.2 Para el programa siguiente, indique el alcance (ya sea alcance de función, archivo, bloque o de prototipo de función) de cada uno de los elementos siguientes:

a) La variable `x` en `main`.
b) La variable `y` en `cube`.
c) La función `cube`.
d) La función `main`.
e) El prototipo de función correspondiente a `cube`.
f) El identificador `y` en el prototipo de función correspondiente a `cube`.

```
#include <stdio.h>
int cube(int y);

main()
{
   int x;

   for (x = 1; x <= 10; x++)
      printf("%d\n", cube(x));
}

int cube(int y)
{
   return y * y * y;
}
```

5.3 Escriba un programa que compruebe si los ejemplos de las llamadas de funciones matemáticas de biblioteca mostradas en la figura 5.2 de verdad producen los resultados indicados.

5.4 Proporcione el encabezamiento de función para cada una de las función siguientes.

a) Función `hypotenuse` que toma dos argumentos de punto flotante de doble precisión, `side1` y `side2`, y regresa un resultado de punto flotante de doble precisión.
b) Función `smallest` que toma tres enteros, `x`, `y`, `z` y regresa un entero.
c) Función `instructions` que no recibe ningún argumento y no regresa ningún valor. (Nota: se utilizan estas funciones por lo regular para mostrar instrucciones a un usuario.)
d) Función `intToFloat` que toma un argumento entero, `number` y regresa un resultado en punto flotante.

5.5 Proporcione el prototipo de función de cada uno de los siguientes:

a) La función descrita en el ejercicio 5.4a

b) La función descrita en el ejercicio 5.4b
c) La función descrita en el ejercicio 5.4c
d) La función descrita en el ejercicio 5.4d

5.6 Escriba una declaración para cada uno de los siguientes:

a) `count` entero que debe de ser conservado en un registro. Inicialice `count` a `0`.
b) Variable de punto flotante `lastVal` que debe de conservar su valor entre llamadas a la función en la cual ha sido definida.
c) Entero externo `number` cuyo alcance debe de quedar restringido al resto del archivo en el cual ha sido definido.

5.7 Encuentre el error en cada uno de los segmentos siguientes de programa y explique cómo puede corregirse dicho error (vea también el ejercicio 5.50):

a)
```
int g(void) {
   printf("Inside function g\n");

   int h(void) {
      printf("Inside function h\n");
   }
}
```
b)
```
int sum(int x, int y) {
   int result;

   result = x + y;
}
```
c)
```
int sum(int n) {
   if (n == 0)
      return 0;
   else
      n + sum(n - 1);
}
```
d)
```
void f(float a); {
   float a;

   printf("%f", a);
}
```
e)
```
void product(void) {
   int a, b, c, result;

   printf("Enter three integers: ")
   scanf("%d%d%d", &a, &b, &c);
   result = a * b * c;
   printf("Result is %d", result);
   return result;
}
```

Respuestas a los ejercicios de autoevaluación

5.1 a) Función. b) Llamada de función. c) Variable local. d) `return`. e) `void`. f) Alcance. g) `return`; o bien `return expression`; o bien al encontrar la llave de cierre izquierda de una función. h) Prototipo de función. i) `rand`. j) `srand`. k) `auto`, `register`, `extern`, `static`. l) Automático. m) `register`. n) External, global. o) `static`. p) Alcance de función, alcance de archivo, alcance de bloque, alcance de prototipo de función. q) Recursivo. r) Base.

5.2 a) Alcance de bloque. b) Alcance de bloque. c) Alcance de archivo. d) Alcance de archivo. e) Alcance de archivo. f) Alcance de prototipo de función.

5.3

```
/* Testing the math library functions */
#include <stdio.h>
#include <math.h>

main()
{
   printf("sqrt(%.1f) = %.1f\n", 900.0, sqrt(900.0));
   printf("sqrt(%.1f) = %.1f\n", 9.0, sqrt(9.0));
   printf("exp(%.1f) = %f\n", 1.0, exp(1.0));
   printf("exp(%.1f) = %f\n", 2.0, exp(2.0));
   printf("log(%f) = %.1f\n", 2.718282, log(2.718282));
   printf("log(%f) = %.1f\n", 7.389056, log(7.389056));
   printf("log10(%.1f) = %.1f\n", 1.0, log10(1.0));
   printf("log10(%.1f) = %.1f\n", 10.0, log10(10.0));
   printf("log10(%.1f) = %.1f\n", 100.0, log10(100.0));
   printf("fabs(%.1f) = %.1f\n", 13.5, fabs(13.5));
   printf("fabs(%.1f) = %.1f\n", 0.0, fabs(0.0));
   printf("fabs(%.1f) = %.1f\n", -13.5, fabs(-13.5));
   printf("ceil(%.1f) = %.1f\n", 9.2, ceil(9.2));
   printf("ceil(%.1f) = %.1f\n", -9.8, ceil(-9.8));
   printf("floor(%.1f) = %.1f\n", 9.2, floor(9.2));
   printf("floor(%.1f) = %.1f\n", -9.8, floor(-9.8));
   printf("pow(%.1f, %.1f) = %.1f\n", 2.0, 7.0, pow(2.0, 7.0));
   printf("pow(%.1f, %.1f) = %.1f\n", 9.0, 0.5, pow(9.0, 0.5));
   printf("fmod(%.3f/%.3f) = %.3f\n",
      13.675, 2.333, fmod(13.675, 2.333));
   printf("sin(%.1f) = %.1f\n", 0.0, sin(0.0));
   printf("cos(%.1f) = %.1f\n", 0.0, cos(0.0));
   printf("tan(%.1f) = %.1f\n", 0.0, tan(0.0));
}
```

```
sqrt(900.0) = 30.0
sqrt(9.0) = 3.0
exp(1.0) = 2.718282
exp(2.0) = 7.389056
log(2.718282) = 1.0
log(7.389056) = 2.0
log10(1.0) = 0.0
log10(10.0) = 1.0
log10(100.0) = 2.0
fabs(13.5) = 13.5
fabs(0.0) = 0.0
fabs(-13.5) = 13.5
ceil(9.2) = 10.0
ceil(-9.8) = -9.0
floor(9.2) = 9.0
floor (-9.8) = -10.0
pow(2.0, 7.0) = 128.0
```

continúa

continuación

```
pow(9.0, 0.5) = 3.0
fmod(13.675/2.333) = 2.010
sin(0.0) = 0.0
cos(0.0) = 1.0
tan(0.0) = 0.0
```

5.4 a) `double hypotenuse(double side1, double side2)`
b) `int smallest(int x, int y, int z)`
c) `void instructions(void)`
d) `float intToFloat(int number)`

5.5 a) `double hypotenuse(double, double);`
b) `int smallest(int, int, int);`
c) `void instructions(void)`
d) `float intToFloat(int)`

5.6 a) `register int count = 0;`
b) `static float lastVal;`
c) `static int number;`
Nota: Esto aparecería por fuera de cualquier definición de función.

5.7 a) Error: la función **h** está definida en la función **g**.
Corrección: mueva la definición de **h** fuera de la definición de **g**
b) Error: la función debe supuestamente regresar un entero, pero no lo hace.
Corrección: borrar la variable **result** y colocar el siguiente enunciado en la función:

```
return x + y;
```

c) Error: el resultado de **n + sum(n-1)** no es regresado; **sum** regresa un resultado inadecuado.
Corrección: volver a escribir el enunciado en la cláusula **else** como

```
return n + sum(n - 1);
```

d) Error: punto y coma, antes del paréntesis derecho que encierra la lista de parámetros, y redefinir el parámetro **a** en la definición de función.
Corrección: Borre el punto y coma después del paréntesis derecho de la lista de parámetros, y borre la declaración **float a;**
e) Error: la función regresa un valor cuando no se supone que lo haga.
Corrección: eliminar el enunciado **return**.

Ejercicios

5.8 Muestre el valor de **x** después de que se hayan ejecutado los siguientes enunciados:
a) `x = fabs(7.5)`
b) `x = floor(7.5)`
c) `x = fabs(0.0)`
d) `x = ceil(0.0)`
e) `x = fabs(-6.4)`
f) `x = ceil(-6.4)`
g) `x = ceil(-fabs(-8+floor(-5.5)))`

5.9 Un estacionamiento público carga $2.00 de estacionamiento mínimo por las primeras tres horas. El estacionamiento carga $0.50 adicionales por cada hora o *parte de la misma* en exceso de tres horas. El cargo máximo para cualquier periodo de 24 horas es $10.00. Suponga que no existe ningún vehículo que se quede más de 24 horas a la vez. Escriba un programa que calcule e imprima los cargos por estacionamiento

para cada uno de tres clientes que ayer estacionaron sus automóviles en este garaje. Deberá de introducir las horas de estacionamiento para cada uno de los clientes. Su programa deberá imprimir los resultados en un formato tabular nítido, y deberá calcular e imprimir el total de los ingresos de ayer. El programa deberá utilizar la función **`calculate-Charges`** para determinar los cargos de cada cliente. Sus salidas deberán de aparecer en el formato siguiente:

```
Car        Hours        Charge
1            1.5          2.00
2            4.0          2.50
3           24.0         10.00
TOTAL       29.5         14.50
```

5.10 Una aplicación de la función `floor` es redondear un valor al entero más cercano. El enunciado

```
y = floor(x + .5);
```

redondeará el número `x` al entero más cercano, y asignará el resultado a `y`. Escriba un programa que lea varios números y que utilice el enunciado anterior para redondear cada uno de estos números al entero más cercano. Para cada número procesado, imprima tanto el número original como el número redondeado.

5.11 La función `floor` puede ser utilizada para redondear un número a una cantidad específica de lugares decimales. El enunciado

```
y = floor(x * 10 + .5) / 10;
```

redondea `x` a la posición de décimos (la primera posición a la derecha del punto decimal). El enunciado

```
y = floor(x * 100 + .5) / 100;
```

redondea `x` a la posición de las centésimos (es decir, a la segunda posición a la derecha del punto decimal). Escriba un programa que defina cuatro funciones para redondear un número `x` de varias formas:

a) **`roundToInteger(number)`**
b) **`roundToTenths(number)`**
c) **`roundToHundreths(number)`**
d) **`roundToThousandths(number)`**

Para cada uno de los valores leídos, su programa debe imprimir el valor original, el número redondeado para el entero más cercano, el número redondeado a la décima más cercana, el número redondeado a la centésima más cercana, y el número redondeado a la milésima más cercana.

5.12 Conteste cada una de las preguntas siguientes:

a) ¿Qué significa seleccionar números "al azar"?
b) ¿Por qué la función **rand** es útil para la simulación de juegos de azar?.
c) ¿Por qué tendría que hacer aleatorio un programa utilizando a **`srand`**? ¿y bajo qué circunstancias no sería deseable hacerlo?.
d) ¿Por qué resulta necesario ha menudo el dimensionar y/o desplazar los valores producidos por **`rand`**?
e) ¿Por qué es una técnica útil la simulación computarizada de situaciones del mundo real?

5.13 Escriba enunciados que asignen números enteros aleatorios a la variable n en los rangos siguientes:

a) $1 \leq n \leq 2$
b) $1 \leq n \leq 100$
c) $0 \leq n \leq 9$
d) $1000 \leq n \leq 1112$
e) $-1 \leq n \leq 1$
f) $-3 \leq n \leq 11$

5.14 Para cada uno de los conjuntos siguientes de enteros, escriba un solo enunciado que imprima un número al azar del conjunto.

a) 2, 4, 6, 8, 10.
b) 3, 5, 7, 9, 11.
c) 6, 10, 14, 18, 22.

5.15 Defina una función **`hypotenuse`** que calcule la longitud de la hipotenusa de un triángulo rectángulo, cuando son conocidos los otros dos lados. Utilice esta función en un programa para determinar la longitud de la hipotenusa de los triángulos siguientes. La función debe de tomar dos argumentos del tipo `double` y regresar la hipotenusa también como **`double`**.

Triángulo	Lado 1	Lado 2
1	3.0	4.0
2	5.0	12.0
3	8.0	15.0

5.16 Escriba una función `integerPower(base, exponent)` que devuelva el valor de

base ^{exponente}

Por ejemplo, `integerPower(3,4) = 3 * 3 * 3 * 3`. Suponga que **`exponent`** es un entero positivo, no cero, y **`base`** es un entero. La función `integerPower` deberá utilizar `for` para controlar el cálculo. No utilice ninguna función matemáticas de biblioteca.

5.17 Escriba una función **`multiple`** que determine para un par de enteros, si el segundo de ellos es múltiplo del primero. La función debe tomar dos argumentos enteros y regresar `1` (verdadero) si el segundo es un múltiplo del primero, y `0` (falso) de no ser así. Utilice esta función en un programa que introduzca una serie de pares de enteros.

5.18 Escriba un programa que introduzca una serie de enteros y que los pase uno a la vez a la función **`even`** que utiliza el operador de módulo, para determinar si el entero es par. La función deberá tomar un argumento entero y regresar `1` si el entero es par, y `0` si no lo es.

5.19 Escriba una función que despliegue en el margen izquierdo de la pantalla un cuadrado sólido de asteriscos, cuyo costado o lado está especificado en el parámetro entero **`side`**. Por ejemplo, si **`side`** es **`4`**, la función mostrará

```
****
****
****
****
```

5.20 Modifique la función creada en el ejercicio 5.19 para formar el cuadrado en base a cualquier carácter que esté contenido en el parámetro de carácter **`fillCharacter`**. Por lo tanto si **`side`** es **5** y **`fillCharacter`** es "**#**" entonces esta función debería imprimir.

```
#####
#####
#####
#####
#####
```

5.21 Utilice técnicas similares a las desarrolladas en los ejercicios 5.19 y 5.20 para producir un programa que grafique una amplia gama de formas

5.22 Escriba segmentos de programa que lleven a cabo cada uno de ellos lo siguiente:

a) Calcule la parte entera del cociente cuando el entero `a` se divide por el entero `b`.

b) Calcule el residuo entero cuando el entero `a` es dividido por el entero `b`.

c) Utilice las porciones de programa desarrolladas en `a)` y en `b)` para describir una función que introduzca un entero entre `1` y `32767` y lo imprima como una serie de dígitos, estando separado cada par de dígitos por dos espacios. Por ejemplo el entero `4562` deberá ser impreso como

```
4  5  6  2
```

5.23 Escriba una función que obtenga el tiempo como tres argumentos enteros (para horas, minutos y segundos) y regrese el número de segundos desde la última vez que el reloj "llegó a las 12". Utilice esta función para calcular la cantidad de tiempo en segundos entre dos horas, cuando ambas estén dentro de un ciclo de 12 horas del reloj.

5.24 Ponga en marcha las siguientes funciones enteras:

a) La función `celsius` que regresa el equivalente Celsius de una temperatura en Fahrenheit.

b) La función `fahrenheit` que regresa el equivalente en Fahrenheit de una temperatura en Celsius.

c) Utilice ambas funciones para escribir un programa que imprima gráficas mostrando los equivalentes Fahrenheit de todas las temperaturas Celsius desde 0 hasta 100 grados, y los equivalentes Celsius de todas las temperaturas Fahrenheit entre 32 y 212 grados. Imprima las salidas en un formato tabular nítido, que minimice el número de líneas de salida manteniéndose legible.

5.25 Escriba una función que regrese el más pequeño de tres números de punto flotante.

5.26 Un número entero se dice que se trata de un *número perfecto* si sus factores, incluyendo a 1 (pero excluyendo en el número mismo), suman igual que el número. Por ejemplo, 6 es un número perfecto porque 6 = 1+2+3. Escriba una función `perfect` que determine si el parámetro `number` es un número perfecto. Utilice esta función en un programa que determine e imprima todos los números perfectos entre 1 y 1000. Imprima los factores de cada número perfecto para confirmar que el número de verdad es perfecto. Ponga en acción la potencia de su computadora para probar números más grandes que 1000.

5.27 Se dice que un entero es *primo* si es divisible sólo entre 1 y sí mismo. Por ejemplo, 2, 3, 5, y 7 son primos, pero 4, 6, 8 y 9 no lo son.

a) Escriba una función que determine si un número es primo.

b) Utilice esta función en un programa que determine e imprima todos los números primos entre 1 y 10,000. ¿Cuántos de estos 10,000 números tendrá que probar verdaderamente antes de estar seguro de que se han encontrado todos los números primos?

c) Inicialmente pudiera pensar que $n/2$ es el límite superior para el cual debe usted probar para ver si un número es primo, pero sólo necesita llegar hasta la raíz cuadrada de n. ¿Por qué? Vuelva a escribir el programa, y ejecútelo de ambas formas. Estime la mejoría en rendimiento.

5.28 Escriba una función que tome un valor entero y regrese el número con sus dígitos invertidos. Por ejemplo, dado el número 7631, la función debería regresar 1367.

5.29 El *máximo común divisor* de dos enteros es el entero más grande que divide de forma uniforme cada uno de los dos números. Escriba una función `gcd` que regrese el máximo común divisor de dos enteros.

5.30 Escriba una función `qualityPoints` que introduzca el promedio de un alumno y regrese 4 si el promedio es entre 90 - 100, 3 si el promedio es entre 80 - 89, 2 si el promedio es entre 70 - 79, 1 si el promedio está entre 60 - 69 0 si el promedio es menor de 60.

5.31 Escriba un programa que simule lanzar una moneda. Para cada lanzamiento de la moneda el programa deberá imprimir `Heads` o `Tails`. Permita que el programa lance la moneda 100 veces, y cuente el número de veces que aparece alguno de los dos lados de la moneda. Imprima los resultados. El programa deberá llamar una función separada o distinta `flip`, que no toma argumentos y que regresa `0` para las caras, y `1` para las cruces. *Nota*: si el programa simula en forma realista el lanzamiento de la moneda, entonces cada cara de la misma deberá aparecer aproximadamente la mitad del tiempo para un total de aproximadamente 50 caras y 50 cruces.

5.32 Las computadoras están jugando un papel creciente en la educación. Escriba un programa que ayudaría a un alumno de escuela primaria a aprender a multiplicar. Utilice `rand` para producir dos enteros positivos de un dígito. A continuación debería escribir una pregunta como la siguiente:

```
How much is 6 times 7?
```

A continuación el alumno escribe la respuesta. Su programa verifica la respuesta del alumno. Si es correcta, imprime `"Very good!"` y a continuación solicita otra multiplicación. Si la respuesta es incorrecta, imprimirá `"No. Please try again."` y a continuación permitirá que el alumno vuelva a intentar la misma pregunta en forma repetida, hasta que al final la conteste correctamente.

5.33 La utilización de las computadoras en la educación se conoce como *instrucción asistida por computadora* (CAI). Un problema que se desarrolla en los entornos CAI es la fatiga del alumno. Esto puede ser eliminado variando el diálogo de la computadora para retener la atención del alumno. Modifique el programa del ejercicio 5.32 de tal forma que los comentarios que se impriman para respuesta correcta y cada respuesta incorrecta sean como sigue:

Respuestas a las contestaciones correctas

```
Very good!
Excellent!
Nice work!
Keep up the good work!
```

Respuestas a las contestaciones incorrectas

```
No. Please try again.
Wrong. Try once more.
Don't give up!
No. Keep trying.
```

Utilice el generador de números aleatorios para escoger el número de 1 a 4 y seleccionar una respuesta apropiada para cada una de las contestaciones. Utilice una estructura `switch` con enunciados `printf` para emitir las respuestas.

5.34 Sistemas más avanzados de instrucción asistida por computadora vigila el rendimiento del alumno a lo largo de un periodo de tiempo. La decisión para empezar un nuevo tema, a menudo se basa en el léxico del alumno en relación con temas anteriores. Modifique el programa del ejercicio 5.33 para contar el número de respuestas correctas e incorrectas escritas por el estudiante. Una vez que el alumno escriba 10 respuestas, su programa deberá calcular el porcentaje de respuestas correctas. Si el porcentaje es menor de 75%, su programa deberá de imprimir `"Please ask you instructor for extra help"` y a continuación terminar.

5.35 Escriba un programa en C que juegue el juego de "adivine el número" como sigue: su programa escoge el número que se debe de adivinar seleccionando un entero al azar en el rango del 1 al 1000. El programa a continuación escribe:

```
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
```

El jugador entonces escribe su primera estimación. El programa responde con una de las siguientes:

```
1. Excellent! You guessed the number!
   Would you like to play again (y or n)?
2. Too low. Try again.
3. Too high. Try again.
```

Si la adivinanza del jugador es incorrecta, su programa deberá de ciclar hasta que el jugador obtiene al final el número correcto. Su programa debe de insistir en indicarle al jugador `Too high` o bien `Too low` para ayudarlo a centrarse a la contestación correcta. Nota: la técnica de búsqueda empleada en este problema es conocida como *búsqueda binaria*. Diremos más en relación con lo anterior en el siguiente problema.

5.36 Modifique el programa del ejercicio 5.35 para contar el número de veces que intenta adivinar el jugador. Si el número es 10 o menor, imprima `Either you know the secret or you got lucky!` Si el jugador adivina el número en 10 intentos, entonces imprima `Ahah! You Know the secret!` Si el jugador hace más de 10 intentos, entonces imprima `You should be able to do better!` ¿Por qué debería de tomar no más de 10 intentos? "Bueno", con cada una de las estimaciones buenas, el jugador tendría que estar en posición de eliminar la mitad de los números. Ahora muestre porque cualquier número del 1 al 1000 puede ser encontrado en 10 o menos intentos.

5.37 Escriba una función recursiva `power (base, exponente)` que al ser invocada regrese

$$base^{exponente}$$

Por ejemplo, `power(3, 4) = 3 * 3 * 3 * 3`. Suponga que `exponente` es un entero mayor o igual a 1. *Sugerencia:* el paso de recursión deberá de utilizar la relación

$$base^{exponente} = base \cdot base^{exponente-1}$$

y la condición de terminación ocurrirá cuando `exponente` es igual a `1` porque

$$base^{1} = base$$

5.38 La serie Fibonacci

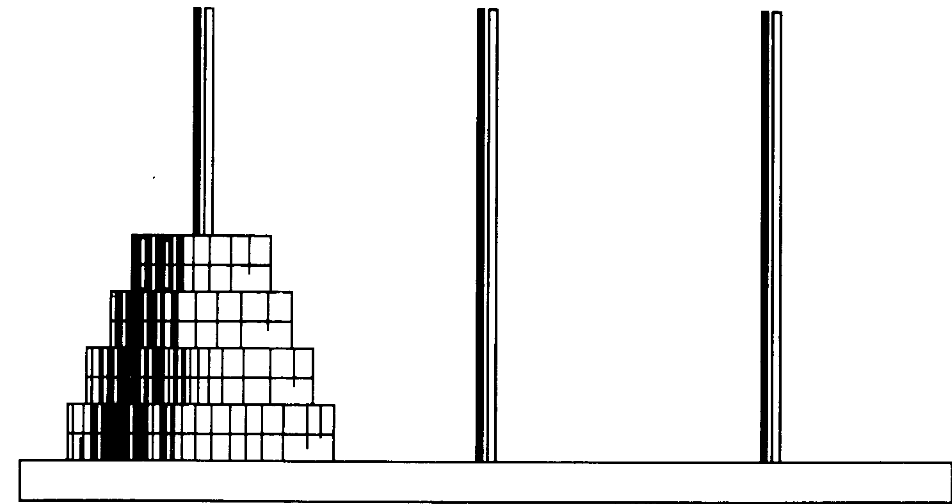
0, 1, 1, 2, 3, 5, 8, 13, 21,...

empieza con los términos 0 y 1 y tiene la propiedad que cada término siguiente es la suma de los dos términos precedentes. a) Escriba una función *no recursiva* `fibonacci(n)` que calcule el número Fibonacci de orden `n`. b) Determine el número Fibonacci más grande que pueda ser impreso en su sistema. Modifique el programa de la parte a) para utilizar `double` en vez de `int`, a fin de calcular y regresar números Fibonacci. Deje que el programa cicle hasta que falle debido a valores en exceso altos.

5.39 (*Torres de Hanoi*) Todos los científicos de cómputo incipientes deben de enfrentarse con ciertos problemas clásicos, y las Torres de Hanoi (vea la figura 5.18) es uno de los más famosos. Dice la leyenda que en un templo del Lejano Este, los monjes están intentando mover una pila de discos de una estaca hacia otra. La pila inicial tenía 64 discos ensartados en una estaca y acomodados de la parte inferior a la superior en tamaño decreciente. Los monjes están intentando mover la pila de esta estaca a la segunda con las limitaciones que exactamente un disco debe de ser movido a la vez, y en ningún momento se puede colocar un disco mayor por encima de un disco menor. Existe una tercera estaca disponible para almacenamiento temporal de discos. Se supone que cuando los monjes terminen su tarea llegará el fin del mundo, por lo cual para nosotros existe poca motivación en ayudarles en sus esfuerzos.

Supongamos que los monjes están intentando mover los discos de la estaca 1 a la estaca 3. Deseamos desarrollar un algoritmo que imprima la secuencia precisa de las transferencias disco a disco entre estacas.

Si fuéramos a enfocar este problema con métodos convencionales, nos encontraríamos rápidamente enmarañados y sin esperanza de poder manejar los discos. En vez de ello, si atacamos el problema teniendo en mente la recursión, de inmediato se vuelve manejable. El mover n discos puede ser visualizado en términos de sólo mover n-1 discos (y de ahí la recursión), como sigue:



UNIVERSIDAD DE LA REPUBLICA
FACULTAD DE INGENIERIA
DEPARTAMENTO DE
DOCUMENTACION Y BIBLIOTECA

Fig. 5.18 Las Torres de Hanoi para el caso de cuatro discos.

1. Mover n-1 discos de la estaca 1 a la estaca 2, utilizando a la estaca 3 como un área de almacenamiento temporal.
2. Mover el último disco (el más grande) de la estaca 1 a la estaca 3.
3. Mover los n-1 discos de la estaca 2 a la estaca 3, utilizando la estaca 1 como área de almacenamiento temporal.

El proceso termina cuando la última tarea consiste en mover el disco $n = 1$, es decir, el caso base. Esto se lleva a cabo en forma trivial moviendo el disco, sin necesidad de utilizar el área temporal de almacenamiento.

Escriba un programa para resolver el problema de las Torres de Hanoi. Utilice una función recursiva con cuatro parámetros:

1. El número de discos a moverse
2. La estaca en la cual se acumularán estos discos al inicio
3. La estaca a la cual esta pila de discos se moverá
4. La estaca a utilizarse como área de almacenamiento temporal

Su programa deberá imprimir las instrucciones precisas que deberán seguirse para mover los discos de la estaca de arranque a la estaca destino. Por ejemplo, para mover una pila de tres discos de la estaca 1 a la estaca 3, su programa deberá imprimir la serie siguiente de movimientos:

1 → 3 (Esto significa mover un disco de la estaca 1 a la estaca 3)
1 → 2
3 → 2
1 →3
2 → 1
2 → 3
1 → 3

5.40 Cualquier programa que puede ser organizado en forma recursiva, puede ser organizado también en forma iterativa, aunque algunas veces con mayor dificultad y con menor claridad. Intente escribir una versión iterativa de las Torres de Hanoi. Si tiene éxito, compare su versión iterativa con la versión recursiva que desarrolló en el ejercicio 5.39. Investigue los temas correspondientes a rendimiento, claridad y a su capacidad para demostrar la corrección de los programas.

5.41 (Visualización de la recursión). Es interesante poder observar la recursión en "acción". Modifique la función factorial de la figura 5.14 para imprimir su variable local y su parámetro de llamada recursiva. Para cada llamada recursiva, despliegue las salidas en una línea por separado, y añada un nivel de sangría. Haga todo lo posible para que las salidas resulten claras, interesantes y significativas. Su meta aquí es diseñar e implantar un formato de salida que le ayude a una persona a mejor comprender la recursión. Quizás desee añadir capacidades de despliegue, tales como éstas, a los muchos otros ejemplos de recursión así como a los ejercicios a todo lo largo del texto.

5.42 El máximo común divisor de los enteros `x` e `y` es el entero más grande que divide en forma completa tanto a `x` como a `y`. Escriba una función recursiva `gcd` que regrese el máximo común divisor de `x` y de `y`. El `gcd` de `x` y de `y` se define en forma recursiva como sigue: Si `y` es igual a `0`, entonces `gcd(de x, y)` es `x`; de lo contrario `gcd(de x, y)` es igual a `gcd(y, x%y)` donde `%` es el operador de módulo.

5.43 ¿Es posible llamar recursivamente a `main`? Escriba un programa que contenga una función `main`. Incluya la variable local `count` de tipo `static` incializada a 1. Postincremente e imprima el valor de `count` cada vez que `main` es llamada. Ejecute su programa. ¿Qué ocurre?.

5.44 Los ejercicios 5.32 al 5.34 desarrollaron un programa de instrucciones asistido por computadora, para enseñar la multiplicación a un alumno de escuela primaria. Este ejercicio sugiere mejoras a dicho programa.

a) Modifique el programa para permitir al usuario la introducción de una capacidad de nivel de grado. Un nivel de grado 1 significa que utilice en los problemas sólo números de un dígito, un nivel de grado 2 significa la utilización de números de hasta dos dígitos de grande, etcétera.

b) Modifique el programa para permitirle al usuario escoger el tipo de problemas aritméticos que él o ella desee estudiar. Una opción 1 significa sólo problemas de suma, 2 significa problemas de resta, 3 significa problemas de multiplicación, 4 significa sólo problemas de división y 5 significa problemas entremezclados al azar, de todos los tipos anteriores.

5.45 Escriba la función `distance` que calcule la distancia entre dos puntos (x1, y1) y (x2, y2). Todos los valores de los números y valores de regreso deberán de ser del tipo `float`.

5.46 ¿Qué es lo que hace el siguiente programa?

```
main()
{
   int c;

   if ((c = getchar()) != EOF) {
      main();
      printf("%c", c);
   }

   return 0;
}
```

5.47 ¿Qué es lo que hace el siguiente programa?

```
int mystery(int, int);

main()
{
   int x, y;

   printf("Enter two integers: ");
   scanf("%d%d", &x, &y);
   printf("The result is %d\n", mystery(x, y));
   return 0;
}
```



```
/* Parameter b must be a positive
   integer to prevent infinite recursion */
int mystery(int a, int b)
{
   if (b == 1)
      return a;
   else
      return a + mystery(a, b - 1);
}
```

5.48 Una vez que haya determinado lo que hace el programa del ejercicio 5,47, modifíquelo para que funcione correctamente, después de eliminar la restricción del segundo argumento que no sea negativo.

5.49 Escriba un programa que pruebe todas las funciones matemáticas posibles de biblioteca en la figura 5.2 como pueda. Ejecute cada una de estas funciones haciendo que su programa imprima tablas de valores de regreso para una diversidad de valores de argumentos.

5.50 Encuentre el error en cada uno de los segmentos de programa, y explique cómo corregirlo:

a)
```
float cube(float);    /* function prototype */
...
cube(float number)    /* function definition */
{
   return number * number * number;
}
```

b)
```
register auto int x = 7;
```

c)
```
int randomNumber = srand();
```

d)
```
float y = 123.45678;
int x;

x = y;
printf("%f\n", (float) x);
```

e)
```
double square(double number)
{
   double number;

   return number * number;
}
```

f)
```
int sum(int n)
{
   if (n == 0)
      return 0;
   else
      return n + sum(n);
}
```

5.51 Modifique el programa de craps de la figura 5.10 para permitir apuestas. Empaquete como una función aquella parte del programa que ejecuta un juego de craps. Inicialice la variable `bankBalance` a 1000 dólares. Solicite al jugador que introduzca una apuesta. Utilice un ciclo `while` para verificar que `wager` es menor o igual a `bankBalance` y de lo contrario indíquele al usuario de que vuelva a entrar `wager` hasta que se introduzca un `wager` válido. Una vez introducido un `wager` válido, ejecute un juego de craps. Si el jugador gana, aumente `bankBalance` por la cantidad `wager` e imprima el nuevo `bankBalance`. Si el jugador pierde, reduzca `bankBalance` por la cantidad `wager`, imprima el nuevo `bankBalance`, verifique si `bankBalance` se ha convertido en cero, y si es así, imprima el mensaje `"Sorry. You busted!"` Conforme progrese el juego, imprima varios mensajes para crear algo de "plática" como es `"Oh, you're going for broke, huh?"`, o bien `"Aw cmon, take a chance!"` o bien `"You're up big. Now's the time to cash in your chips!"`