CAPÍTULO 6
Modularidad con el uso de funciones

TEMAS

Los programas profesionales se diseñan, codifican y prueban en forma muy parecida al hardware: como una serie de módulos que están integrados para funcionar en conjunto. Una buena analogía de esto es un automóvil en el cual un módulo fundamental es el motor, otro es la transmisión, un tercero el sistema de frenos, un cuarto la carrocería, etc. Cada uno de estos módulos está vinculado entre sí y colocado al final bajo el control del conductor, el cual puede compararse con un supervisor o un módulo principal de un programa. El conjunto opera ahora como una unidad completa, capaz de hacer trabajo útil, como llevarnos a la tienda. Durante el proceso de montaje, cada módulo se construye, prueba y se encuentra libre de defectos en forma individual antes de ser instalado en el producto final.

Ahora piense en lo que podría hacer si quisiera mejorar el desempeño de su automóvil. Podría alterar el motor existente o eliminarlo por completo y armar un motor nuevo. Del mismo modo, podría cambiar la transmisión o las llantas o los amortiguadores, haciendo cada modificación en forma individual conforme se lo permitan su tiempo y su presupuesto. En cada caso, la mayor parte de los otros módulos pueden permanecer igual, pero el automóvil ahora opera en forma diferente.

En esta analogía, cada uno de los componentes principales de un automóvil puede compararse con una función. Por ejemplo, el conductor llama al motor cuando se oprime el pedal del acelerador. El motor acepta entradas de combustible, aire y electricidad para convertir la solicitud del conductor en un producto útil, potencia, y luego envía esta salida a la transmisión para su procesamiento adicional. La transmisión recibe la salida del motor y la convierte en una forma que pueda ser usada por el eje de transmisión. Una entrada adicional a la transmisión es la selección de velocidades por parte del conductor (primera, reversa, neutral, etcétera).

En cada caso, el motor, la transmisión y otros módulos sólo "conocen" el universo limitado por sus entradas y salidas. El conductor no necesita saber nada de la operación interna del motor, transmisión, aire acondicionado y otros módulos que se están controlando. Todo lo que requiere es comprender qué hace cada unidad y cómo usarla. El conductor tan sólo "llama" a un módulo, como el motor, los frenos, el aire acondicionado y la dirección cuando se requiere la salida de ese módulo. La comunicación entre módulos está restringida a pasar las entradas necesarias a cada módulo conforme se le llame a ejecutar su tarea, y cada módulo opera en forma interna de una manera relativamente independiente. Este mismo enfoque modular es usado por los ingenieros para crear y mantener programas confiables en C++ usando funciones.

Como se ha visto, cada programa en C++ debe contener una función `main()`*. Además de esta función requerida, los programas en C++ también pueden contener cualquier cantidad de funciones adicionales. En este capítulo se aprenderá cómo escribir estas funciones, transmitirles datos, procesar los datos transmitidos y devolver un resultado.*

6.1 DECLARACIONES DE FUNCIONES Y PARÁMETROS

Al crear funciones en C++, debemos poner atención a la función en sí y en la forma en que interactúa con otras funciones, como `main()`. Esto incluye transmitir datos en forma correcta a una función cuando es invocada y devolver valores de una función. En esta sección se describe la primera parte de la interfaz, transmitir datos a una función y hacer que la función reciba, almacene y procese en forma correcta los datos transmitidos.

Como ya se ha visto con las funciones matemáticas, una función se invoca, o utiliza, dando el nombre de la función y transmitiéndole datos, como argumentos, en el paréntesis que sigue al nombre de la función (véase la figura 6.1).

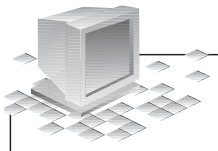
nombre-de-la-función (*datos transmitidos a la función*)

Esto identifica a la función llamada — Esto transmite datos a la función

Figura 6.1 Llamar y transmitir datos a una función.

La función invocada debe ser capaz de aceptar los datos que le son transmitidos por la función que hace la llamada. Sólo después que la función invocada recibe con éxito los datos pueden ser manipulados éstos para producir un resultado útil.

Para aclarar el proceso de envío y recepción de datos, considérese el programa 6.1, el cual invoca a una función nombrada `encontrarMax()`. El programa, como se muestra, no está completo todavía. Una vez que se escribe la función `encontrarMax()` y se incluye en el programa 6.1, el programa completado, consistente en las funciones `main()` y `encontrarMax()`, puede ser compilado y ejecutado.



Programa 6.1

```
#include <iostream>
using namespace std;

void encontrarMax(int, int); // la declaracion de la funcion (prototipo)

int main()
{
  int primernum, segundonum;

  cout << "\nIntroduzca un numero: ";
  cin >> primernum;
  cout << "¡Estupendo! Por favor introduzca un segundo numero: ";
  cin >> segundonum;

  encontrarMax(primernum, segundonum); // aquí se llama la funcion

  return 0;
}
```

Examinemos la declaración y llamada de la función `encontrarMax()` desde `main()`. Luego vamos a escribir `encontrarMax()` para aceptar los datos que se le transmiten y determinar el valor mayor o máximo de los dos valores transmitidos.

La función `encontrarMax()` se conoce como la ***función llamada,*** en vista que es llamada o invocada a la acción por su referencia en `main()`. La función que hace la llamada, en este caso `main()`, se conoce como la ***función que llama.*** Los términos llamada y que llama vienen del uso del teléfono estándar, donde una persona llama a otra. La persona que inicia la llamada se conoce como quien llama, y quien la recibe es la persona llamada. Los mismos términos describen las llamadas a la función. La función llamada, en este caso `encontrarMax()`, es declarada como una función que espera recibir dos números enteros y no devolver ningún valor (void) `a main()`. Esta declaración se conoce de manera formal como un prototipo de función. La función es invocada entonces por la última instrucción en el programa.

Prototipos de función

Antes que una función pueda ser llamada, debe ser declarada la función que hará la llamada. La instrucción de declaración para una función se conoce como un **prototipo de función.** El prototipo de función le indica a la función que llama el tipo de valor que será devuelto formalmente, si es que hay alguno, y el tipo de datos y orden de los valores que la función que llama deberá transmitir a la función llamada. Por ejemplo, el prototipo de función usado antes en el programa 6.1

```
void encontrarMax(int, int);
```

declara que la función `encontrarMax()` espera que se le envíen dos valores enteros, y que esta función particular devuelve de manera formal ningún valor (void). Los prototipos de función pueden colocarse con las instrucciones de declaración de variable de la función que llama, encima del nombre de la función que llama, como en el programa 6.1, o en un archivo de encabezado separado que se incluirá utilizando una instrucción de preprocesamiento `#include.` Por tanto, el prototipo de función para `encontrarMax()` podría haberse colocado antes o después de la instrucción `#include <iostream>,` antes de `main()` o dentro de `main()`. (Las razones para la elección de la colocación se presentan en la sección 6.3.) La forma general de las instrucciones de prototipo de función es:

tipo-de-datos-a-devolver nombre-de-función (lista de tipos de datos para los argumentos);

donde el tipo de datos se refiere al tipo del valor que será devuelto de manera formal por la función. Son ejemplos de prototipos de función

```
int fmax(int, int);
double intercambio(int, char, char, double);
void desplegar(double, double);
```

El prototipo de función para `fmax()` declara que esta función espera recibir dos argumentos en número entero y devolverá de manera formal un valor en número entero. El prototipo de función para `intercambio()` declara que esta función requiere cuatro argumentos consistentes en un número entero, dos caracteres y un argumento de precisión doble, en este orden y formalmente retornará un número de precisión doble. Por último, el prototipo de función para `desplegar()` declara que esta función requiere dos argumentos de precisión doble y no devuelve ningún valor. Dicha función podría ser utilizada para desplegar los resultados de un cálculo en forma directa, sin devolver ningún valor a la función llamada.

El uso de prototipos de función permite la verificación de errores en los tipos de datos por el compilador. Si el prototipo de función no concuerda con los tipos de datos definidos cuando se escribe la función, ocurrirá una advertencia del compilador. El prototipo también sirve para otra tarea: asegura la conversión de todos los argumentos transmitidos a la función al tipo de datos del argumento declarado cuando se llama la función.

Llamada a una función

Llamar a una función es una operación bastante fácil. Los únicos requisitos son usar el nombre de la función y que los datos transmitidos a la función estén encerrados dentro de los paréntesis que siguen al nombre de la función usando el mismo orden y tipo que se declaró en el prototipo de función. Los elementos encerrados dentro de los paréntesis se llaman **argumentos** de la función llamada (véase la figura 6.2).

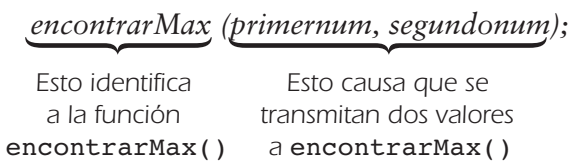


Figura 6.2 *Llamar a* `encontrarMax()` *y transmitirle dos valores.*

Si una variable es uno de los argumentos en una llamada la función, la función llamada recibe una copia del valor almacenado en la variable. Por ejemplo, la instrucción `encontrarMax(primernum, segundonum);` invoca a la función `encontrarMax` y causa que los valores que residen en la actualidad en las variables `primernum` y `segundonum` sean transmitidos a `encontrarMax()`. Los nombres de variable entre paréntesis son argumentos que proporcionan valores a la función llamada. Después que se transmiten los valores, el control es transferido a la función llamada.

Como se ilustra en la figura 6.3, la función `encontrarMax()` *no recibe las variables nombradas* `primernum` *y* `segundonum` *y no tiene conocimiento de estos nombres de variables.*[1] La función tan sólo recibe los valores en estas variables y debe determinar por sí misma dónde almacenarlos antes de hacer algo más. Aunque este procedimiento para transmitir datos a una función puede parecer sorprendente, en realidad es un procedimiento de seguridad para que una función invocada no cambie de manera inadvertida los datos almacenados en una variable. La función obtiene una copia de los datos que va a usar. Puede cambiar su copia y, por supuesto, cambiar cualesquiera variables declaradas dentro de sí misma. Sin embargo, a menos que se den pasos específicos para hacerlo, no se le permite a una función cambiar el contenido de las variables declaradas en otras funciones.

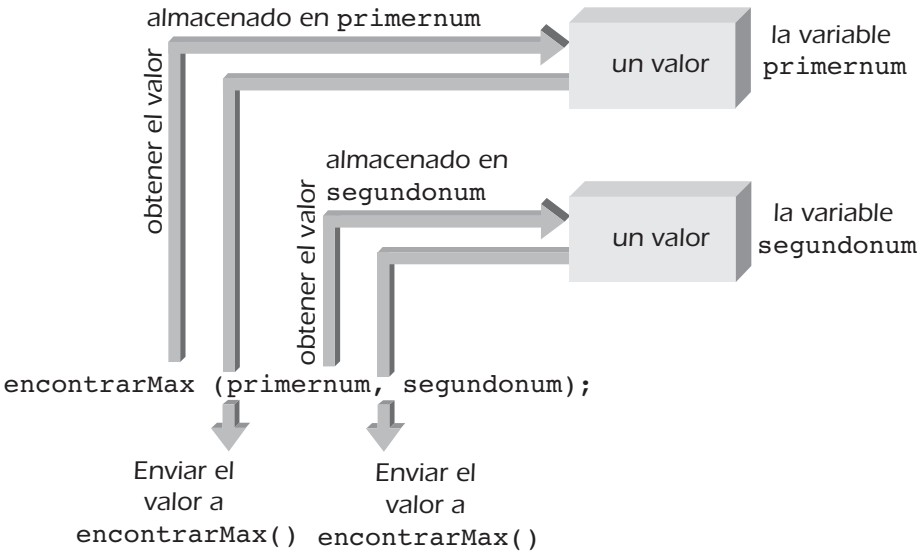


Figura 6.3 `encontrarMax()` *recibe valores actuales.*

Ahora comenzaremos a escribir la función `encontrarMax()` para procesar los valores que se le transmitieron.

[1] En la sección 6.3 se verá cómo, al usar variables de referencia, C++ permite también el acceso directo a las variables de la función que invoca.

Definición de una función

Una función se define cuando se escribe. Cada función es definida una vez (es decir, escrita una vez) en un programa y puede ser usada entonces por cualquier otra función en el programa que la declare en forma adecuada.

Como la función `main()`, toda función en C++ consta de dos partes, un **encabezado de función** y un **cuerpo de función**, como se ilustra en la figura 6.4. El propósito del encabezado de función es identificar el tipo de datos del valor devuelto por la función, proporcionarle un nombre a la función y especificar el número, orden y tipo de los argumentos esperados por ella. El propósito del cuerpo de función es operar sobre los datos transmitidos y devolver en forma directa, cuando mucho, un valor a la función que llama. (En la sección 6.3, se verá, cómo puede hacerse que una función devuelva múltiples valores.)

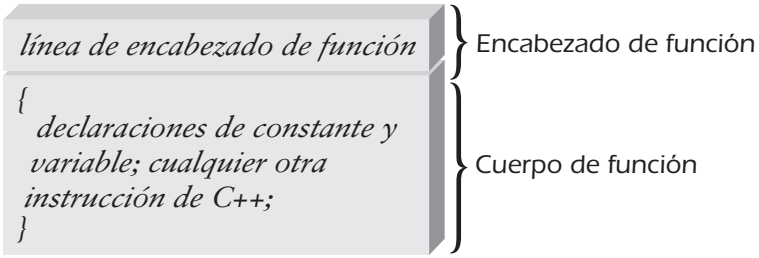


Figura 6.4 Formato general de una función.

El encabezado de función siempre es la primera línea de una función y contiene el tipo de valor devuelto de la función, su nombre y los nombres y tipos de datos de sus argumentos. En vista que `encontrarMax()` no devolverá de manera formal ningún valor y recibirá dos argumentos en número entero, puede usarse la siguiente línea de encabezado:

```
void encontrarMax(int x, int y)
```
← sin punto y coma

Los nombres de argumento en el encabezado se conocen como **parámetros formales** de la función.[2] Por tanto, el parámetro x se usará para almacenar el primer valor transmitido a `encontrarMax()` y el parámetro y se utilizará para almacenar el segundo valor transmitido en el momento de la llamada a la función. La función no sabe de dónde provienen los valores cuando se hace la llamada desde `main()`. La primera parte del procedimiento de llamada ejecutado por la computadora implica ir a las variables `primernum` y `segundonum` y recuperar los valores almacenados. Estos valores son transmitidos luego a `encontrarMax()` y almacenados al final en los parámetros `x` y `y` (véase la figura 6.5).

[2]La porción del encabezado de función que contiene el nombre de la función y los parámetros se conoce de manera formal como un *declarador de función*.

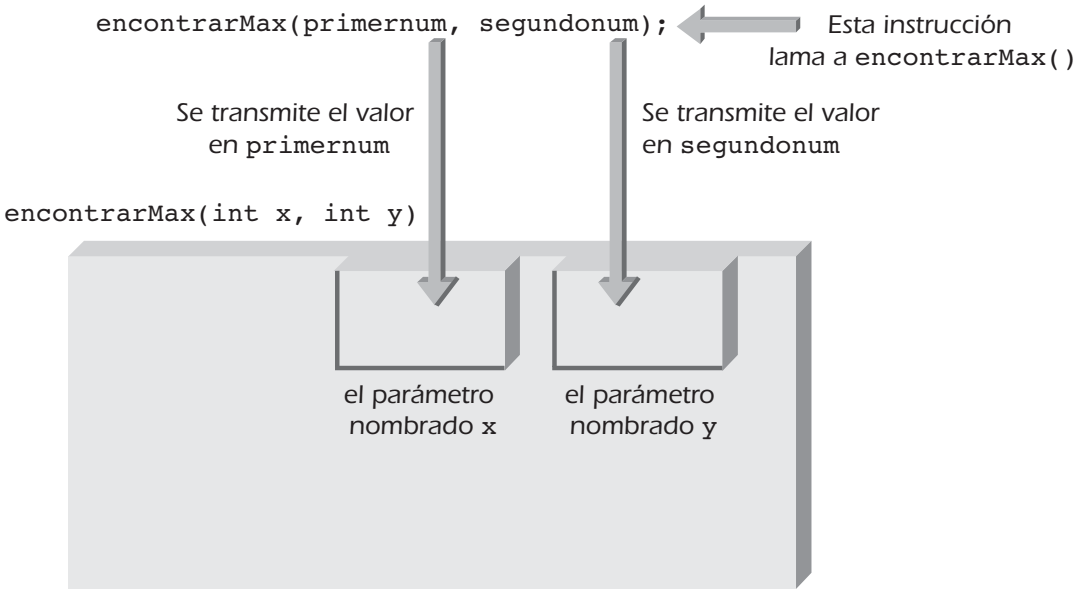


Figura 6.5 *Almacenar valores en parámetros.*

El nombre de la función y todos los nombres de parámetros en el encabezado, en este caso `encontrarMax, x` y `y`, son elegidos por el programador. Pueden usarse los nombres seleccionados de acuerdo con las reglas usadas para elegir nombres de variables. Todos los parámetros enumerados en la línea de encabezado de la función deben separarse con comas y deben tener los tipos de datos individuales declarados por separado.

Ahora que hemos escrito el encabezado de función para la función `encontrarMax(),` se puede construir su cuerpo. Supongamos que la función `encontrarMax()` selecciona y despliega el más grande de los dos números que se le transmiten.

Como se ilustra en la figura 6.6, un cuerpo de función comienza con una llave de apertura, `{,` contiene las declaraciones necesarias y otras instrucciones de C++, y termina con una llave de cierre, `}.` Esto debería serle familiar porque es la misma estructura usada en todas las funciones `main()` que se han escrito. Esto no debería ser sorpresa porque `main()` en sí es una función y debe apegarse a las reglas requeridas para construir todas las funciones legítimas.

```
{
  declaraciones de constantes simbólicas,
  declaraciones de variables y otras
  instrucciones de C++
}
```

Figura 6.6 Estructura de un cuerpo de función.

Punto de Información

Definiciones de función y prototipos de función

Cuando se escribe una función, se crea de manera formal una definición de función. Cada definición comienza con una línea de encabezado que incluye una lista de parámetros, si es que hay alguna, encerrada entre paréntesis y termina con la llave de cierre que finaliza el cuerpo de función. Los paréntesis se requieren use o no algún parámetro la función. Una sintaxis utilizada por lo común para una definición de función es:

```
tipo-de-datos-que-devuelve nombre-de-funcion (lista de parámetros)
{
  declaraciones de constantes
  declaraciones de variables

  otras instrucciones de C++

  return valor
}
```

Un **prototipo de función** declara una función. La sintaxis de un prototipo de función, el cual proporciona el tipo de datos que devuelve la función, el nombre de la función y la lista de argumentos de la función es:

```
tipo-de-datos-que-devuelve nombre-de-funcion (lista de tipos de
datos de los parámetros);
```

Como tal, el prototipo junto con comentarios previos y posteriores a la condición (véase el siguiente recuadro Punto de información) deberán proporcionar al usuario toda la información de programación necesaria para invocar a la función con éxito.

En general, todos los prototipos de funciones se colocan al principio del programa, y todas las definiciones se colocan después de la función `main()`. Sin embargo, esta colocación puede cambiarse. El único requisito en C++ es que una función no puede ser llamada antes que haya sido declarada o definida.

En el cuerpo de la función `encontrarMax()`, se declarará una variable para almacenar el máximo de los dos números transmitidos a ella. Entonces se usará una instrucción `if-else` para encontrar el máximo de los dos números. Por último, se usará una instrucción `cout` para desplegar el máximo. La definición de función completa para la función `encontrarMax()` es:

```
void encontrarMax(int x, int y)
{                   // inicio del cuerpo de funcion
  int numMax;       // declaración de variable

  if (x >= y)       // find the maximum number
    numMax = x;
  else
    numMax = y;

  cout << "\nEl máximo de los dos números es "
       << numMax << endl;

}  // fin del cuerpo de funcion y fin de la funcion
```

Punto de información

Punto de información

Las condiciones previas son cualquier conjunto de condiciones que una función requiere que sean verdaderas para operar en forma correcta. Por ejemplo, si una función usa la constante simbólica `MAXCHARS`, la cual debe tener un valor positivo, una condición previa es que `MAXCHARS` sea declarado con un valor positivo antes que la función sea invocada.

Del mismo modo, una condición posterior es una condición que será verdadera después que se ejecute la función, suponiendo que se cumplen las condiciones previas.

Las condiciones previas y posteriores se documentan de manera típica como comentarios al usuario. Por ejemplo, considere la siguiente declaración y los comentarios:

```
bool bisiesto(int)
// Condiciones previas: los numeros enteros deben representar un año en un formato
//              : de cuatro digitos, como 2006
// Condiciones posteriores: debe devolver un valor de verdadero si el año
                            es bisiesto;
//              : de lo contrario se devolvera falso
```

Los comentarios de las condiciones previas y posteriores deberán incluirse tanto en los prototipos de función como en las definiciones de función siempre que se necesiten aclaraciones.

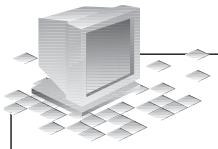
Nótese que las declaraciones de parámetros se hacen dentro de la línea de encabezado y la declaración de variables se hace inmediatamente después de la llave de apertura del cuerpo de función. Ello está de acuerdo con el concepto que los valores de parámetro son transmitidos a una función desde fuera de la función, y que las variables son declaradas y se les asignan valores desde dentro del cuerpo de función.

El programa 6.2 incluye la función `encontrarMax()` dentro del código del programa enlistado antes en el programa 6.1.

El programa 6.2 puede usarse para seleccionar e imprimir el máximo de dos números enteros cualesquiera introducidos por el usuario. A continuación se ve una muestra de la ejecución del programa 6.2:

```
Introduzca un número: 25
¡Estupendo! Por favor introduzca un segundo número: 5

El máximo de los dos números es 25
```

La colocación de la función `encontrarMax()` después de la función `main()` en el programa 6.2 es cuestión de elección. Por lo general enlistaremos primero `main()` porque es la función conductora que le dará a cualquiera que lea el programa una idea de lo que se refiere el programa completo antes de encontrar los detalles de cada función. Sin embargo, en ningún caso puede colocarse la definición de `encontrarMax()` dentro de `main()`. Esto es verdad para todas las funciones de C++, las cuales deben definirse por sí mismas fuera de cualquier otra función. Cada función en C++ es una entidad separada e independiente con sus propios parámetros y variables; *nunca se permite el anidamiento de funciones*.



Programa 6.2

```
#include <iostream>
using namespace std;

void encontrarMax(int, int); // el prototipo de la función

int main()
{
  int primernum, segundonum;

  cout << "\nIntroduzca un número: ";
  cin >> primernum;
  ccout << "¡Estupendo! Por favor introduzca un segundo numero: ";
  cin >> segundonum;

  encontrarMax(primernum, segundonum); // aquí se llama a la función

  return 0;
}

// en seguida está la función encontrarMax()

void encontrarMax(int x, int y)
{                   // inicio del cuerpo de función
  int numMax;       // declaración de variable

  if (x >= y)       // encontrar el número máximo
    numMax = x;
  else
    numMax = y;

  cout << "\nEl máximo de los dos números es "
       << numMax << endl;

  return;
}  // fin del cuerpo de función y fin de la función
```

Colocación de instrucciones

C++ no impone una estructura de ordenamiento de instrucciones rígida al programador. La regla general para colocar instrucciones en un programa en C++ es tan sólo que todas las directivas del preprocesador, constantes nombradas, variables y funciones deben declararse o definirse antes que puedan usarse. Como se ha señalado antes, aunque esta regla permite que tanto las directivas del preprocesador como las instrucciones de declaración se coloquen a través del programa, hacerlo así produce una estructura de programa muy pobre.

Como una cuestión de buena forma de programación, el siguiente ordenamiento de instrucciones deberá formar la estructura básica alrededor de la cual se construyan todos sus programas en C++.

```
directivas del preprocesador

prototipos de función

int main()
{
  constantes simbólicas
  declaraciones de variables

  otras instrucciones ejecutables

  return valor
}

definiciones de función
```

Como siempre, pueden entremezclarse comentarios a las instrucciones en cualquier parte dentro de esta estructura básica.

Cabos de función

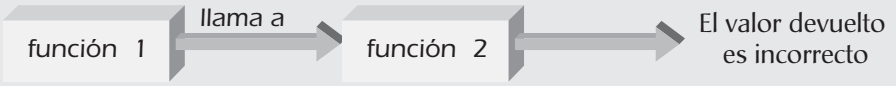
Una alternativa para completar cada función requerida en un programa completo es escribir primero la función `main()`, y agregar las funciones más adelante, conforme se desarrollan. El problema que surge con este enfoque, sin embargo, es el mismo que ocurrió con el programa 6.1; es decir, el programa no puede ejecutarse hasta que todas las funciones se han incluido. Por conveniencia a continuación se ha reproducido el código para el programa 6.1.

```
#include <iostream>
using namespace std;

void encontrarMax(int, int); // la declaración de la función
(prototipo)

int main()
{
  int primernum, segundonum;

  cout << "\nIntroduzca un número: ";
  cin >> primernum;
  cout << "¡Estupendo! Por favor introduzca un segundo número: ";
  cin >> segundonum;

  encontrarMax(primernum, segundonum); // aquí se llama la fúnción

  return 0;
}
```

Punto de Información

Prueba de aislamiento

Uno de los métodos más exitosos que se conoce para probar software es incorporar siempre el código que se está probando dentro de un ambiente de código funcional. Por ejemplo, suponga que tiene dos funciones no probadas que son llamadas en el orden que se muestra a continuación, y el resultado devuelto por la segunda función es incorrecto.



A partir de la información mostrada en esta figura, una de las funciones o quizás ambas podrían estar operando en forma incorrecta. El primer paso para solucionar el problema es aislar una función específica.

Uno de los métodos más potentes para llevar a cabo este aislamiento de código es desacoplar las funciones. Esto se hace probando cada función en forma individual o probando una función primero y, sólo cuando se sabe que está operando en forma correcta, reconectarla a la segunda función. Luego, si ocurre un error, lo habrá aislado a la transferencia de datos entre funciones o a la operación interna de la segunda función.

Este procedimiento específico es un ejemplo de la *regla básica de la prueba,* la cual establece que cada función sólo debería probarse en un programa en el que se sabe que todas las demás funciones son correctas. Esto significa que una función debe probarse primero por sí misma, usando cabos si es necesario para las funciones llamadas, y una segunda función debería probarse ya sea por sí misma o con una función probada con anterioridad, y así en forma sucesiva. Esto asegura que cada función nueva es aislada dentro de una capa de prueba de funciones correctas, con el programa final de código de funciones probadas construido de manera efectiva.

Este programa estaría completo si existiera una definición de función para `encontrarMax`. Pero en realidad no se necesita una función `encontrarMax` *correcta* para probar y ejecutar lo que se ha escrito, sólo necesitamos una función que *actúe* como si fuera una `encontrarMax` "falsa" que acepte el número y tipos apropiados de parámetros y devuelva valores de la manera apropiada para invocar la función es todo lo que se necesita para permitir la prueba inicial. Esta función falsa se llama cabo. Un **cabo** es el comienzo de una función final que se puede usar como un sustituto de la unidad final hasta que la unidad esté completa. Un cabo para `encontrarMax` es como sigue:

```
void encontrarMax(int x, int y)
{
  cout << "En encontrarMax()\n";
  cout << "El valor de x es " << x << endl;
  cout << "El valor de x es " << y << endl;
}
```

Esta función cabo puede ahora compilarse y vincularse con el código completado con anterioridad para obtener un programa ejecutable. Entonces el código para la función puede desarrollarse más con el código "real" cuando es completado, reemplazando la porción cabo.

El requerimiento mínimo de una función cabo es que se compila y se vincula con el módulo que llama. En la práctica, es una buena idea hacer que un cabo despliegue un mensaje que se ha introducido con éxito y el valor o los valores de sus parámetros recibidos, como en el cabo para `encontrarMax()`.

Conforme se depura la función, puede permitir que haya cada vez más, quizá devolver resultados intermedios o incompletos. Este refinamiento creciente, o por etapas, es un concepto importante en el desarrollo eficiente de programas que le proporcionan el medio para ejecutar un programa que todavía no satisface todos sus requerimientos finales.

Funciones con listas de parámetros vacías

Aunque las funciones útiles que tienen una lista de parámetros vacía son limitadas en extremo (se proporciona una de dichas funciones en el ejercicio 11), pueden ocurrir. El prototipo de función para dicha función requiere escribir la palabra clave void o no poner nada en absoluto entre los paréntesis que siguen al nombre de la función. Por ejemplo, ambos prototipos

```
int despliegue();
```

e

```
int despliegue(void);
```

indican que la función `despliegue()` no tiene parámetros y devuelve un número entero. Una función con una lista de parámetros vacía es llamada por su nombre sin nada escrito dentro del paréntesis requerido después del nombre de la función. Por ejemplo, la instrucción `despliegue();` llama en forma correcta a la función `despliegue()` cuyo prototipo se proporcionó antes.

Argumentos por omisión[3]

Una característica conveniente de C++ es su flexibilidad para proporcionar argumentos por omisión mediante una llamada a la función. El uso primario de los argumentos por omisión es extender la lista de parámetros de funciones existentes sin requerir algún cambio en las listas de argumentos invocados que ya están en su lugar dentro de un programa.

Los valores del argumento por omisión se enlistan en el prototipo de función y son transmitidos en forma automática a la función llamada cuando los argumentos correspondientes son omitidos de la llamada a la función. Por ejemplo, el prototipo de función

```
void ejemplo (int, int = 5, double = 6.78);
```

proporciona valores por omisión para los últimos dos argumentos. Si alguno de estos argumentos se omite cuando la función es llamada en realidad, el compilador de C++ suministrará estos valores por omisión. Por tanto, todas las siguientes llamadas a la función son válidas:

```
ejemplo(7, 2, 9.3)  // no se usan valores por omisión
ejemplo(7, 2)       // igual que ejemplo(7, 2, 6.78)
ejemplo(7)          // igual que ejemplo(7, 5, 6.78)
```

Cuando se usan parámetros por omisión deben seguirse cuatro reglas. La primera es que los valores por omisión deberían asignarse en el prototipo de función.[4] La segunda es que si a cualquier parámetro se le da un valor por omisión en el prototipo de función, a todos los

[3] Este tema puede omitirse en la primera lectura sin perder la continuidad de la materia.

[4] Algunos compiladores aceptan asignaciones por omisión en la definición de la función.

parámetros que siguen también deben asignárseles valores por omisión. La tercera es que si un argumento se omite en la llamada a la función real, entonces todos los argumentos a su derecha también deben omitirse. Estas dos reglas le dejan claro de C++ cuáles argumentos se están omitiendo y le permite suministrar valores por omisión correctos para los argumentos faltantes, empezando por el argumento de la extrema derecha y trabajando hacia la izquierda. La última regla especifica que el valor por omisión utilizado en el prototipo de función puede ser una expresión consistente en constantes y variables declaradas con anterioridad. Si se usa una expresión así, debe pasar la verificación del compilador para variables declaradas con validez, aun cuando el valor real de la expresión sea evaluado y asignado en tiempo de ejecución.

Los argumentos por omisión son útiles en extremo cuando se extiende una función existente para incluir más características que requieren argumentos adicionales. Agregar los argumentos nuevos a la derecha de los argumentos existentes y proporcionarle a cada argumento nuevo un valor por omisión permite que todas las llamadas a funciones existentes permanezcan como están. Por tanto, el efecto de los cambios nuevos es aislado convenientemente del código existente en el programa.

Reutilización de nombres de función (sobrecarga)[5]

C++ proporciona la capacidad de usar el mismo nombre de función para más de una función, lo cual se conoce como **sobrecarga de función.** El único requisito para crear más de una función con el mismo nombre es que el compilador debe ser capaz de determinar cuál función usar con base en los tipos de datos de los parámetros (no los tipos de datos del valor devuelto, si es que hay alguno). Por ejemplo, considere las tres funciones siguientes, todas nombradas `cdabs()`.

```
void cdabs(int x) // calcula y despliega el valor absoluto de un número entero
{
  if ( x < 0 )
    x = -x;
  cout << "El valor absoluto del número entero es " << x << endl;
}

void cdabs(float x) // calcula y despliega el valor absoluto de un número de
punto flotante
{
  if ( x < 0 )
    x = -x;
  cout << "El valor absoluto del número de punto flotante es " << x << endl;
}

void cdabs(double x) // calcula y despliega el valor absoluto de un número
en doble precision
{
  if ( x < 0 )
   x = -x;
  cout << "El valor absoluto del número de doble precisión es " << x << endl;
}
```

[5]Este tema puede omitirse en la primera lectura sin que pierda la continuidad de la materia.

Cuál de las tres funciones nombradas `cdabs()` se llama en realidad depende de los tipos de argumento suministrados en el momento de la llamada. Por tanto, la llamada a la función `cdabs(10);` causaría que el compilador utilizara la función nombrada `cdabs()` que espera un argumento de número entero, y la llamada a la función `cdabs(6.28f);` causaría que el compilador usara la función nombrada `cdabs()` que espera un argumento de precisión simple.[6]

Hay que observar que sobrecargar un nombre de función tan sólo significa usar el mismo nombre para más de una función. Cada función que utiliza el nombre debe escribirse aún y existe como una entidad separada. El uso del mismo nombre de función no requiere que el código dentro de las funciones sea similar, aunque la buena práctica de programación dicta que las funciones con el mismo nombre deberían ejecutar en esencia las mismas operaciones. Todo lo que se requiere de manera formal para usar el mismo nombre de función es que el compilador pueda distinguir cuál función seleccionar con base en los tipos de datos de los argumentos cuando es llamada la función. Sin embargo, es evidente que si todo lo que es diferente acerca de las funciones sobrecargadas son los tipos de argumentos, una mejor solución de programación es tan sólo crear una plantilla de función. El uso de funciones sobrecargadas, sin embargo, es útil en extremo con funciones constructoras, un tema que se presenta en la sección 8.3.

Plantillas de función[7]

En la mayor parte de los lenguajes de nivel alto, incluyendo C, el predecesor inmediato de C++, cada función requiere su propio nombre único. En teoría esto tiene sentido, pero en la práctica puede conducir a una profusión de nombres de función, incluso para funciones que realizan en esencia las mismas operaciones. Por ejemplo, considere determinar y desplegar el valor absoluto de un número. Si el número transmitido a la función puede ser un número entero, uno de precisión simple o un valor de precisión doble, deben escribirse tres funciones distintas para manejar en forma correcta cada caso. Por supuesto, podría dársele a cada una de estas funciones un nombre único, como `abs()`, `fabs()` y `dabs()`, respectivamente, teniendo los prototipos de función:

```
void abs(int);
void fabs(float);
void dabs(double);
```

Es evidente que cada una de estas funciones realiza en esencia la misma operación, pero con diferentes tipos de datos en los parámetros. Una solución mucho más limpia y elegante es escribir una función general que maneje todos los casos, pero cuyos parámetros, variables e incluso tipo devuelto puedan ser establecidos por el compilador basado en la llamada a la función real. Esto es posible en C++ usando plantillas de función.

[6] Esto se logra por un proceso conocido como *mutilación del nombre*. Usando este proceso el nombre de función generado en realidad por el compilador de C++ difiere del nombre de función usado en el código fuente. El compilador añade información al nombre de función del código fuente dependiendo del tipo de datos que se transmiten, y se dice que el nombre resultante es una versión mutilada del nombre del código fuente.

[7] Este tema puede omitirse en la primera lectura sin perder la continuidad de la materia.

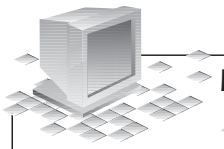
Una **plantilla de función** es una función única completa que sirve como modelo para una familia de funciones. Cuál función de la familia se creará en realidad depende de las llamadas a la función subsiguientes. Para hacer esto más concreto, considere una plantilla de función que calcule y despliegue el valor absoluto de un argumento transmitido. Una plantilla de función apropiada es:

```
template <class T>
void mostrarabs(T numero)
{
  if (numero < 0)
    numero = -numero;
  cout << "El valor absoluto del numero "
       << " es " << numero << endl;

  return
}
```

Por el momento, ignore la primera línea `template <class T>` y observe la segunda línea, la cual consiste del encabezado de `función mostrarabs(T numero).` Hay que observar que esta línea de encabezado tiene la misma sintaxis que se ha usado para todas las definiciones de función, excepto por la `T` en el lugar donde se coloca por lo general un tipo de datos. Por ejemplo, si la línea de encabezado fuera `void mostrarabs(int numero),` lo reconocería como una función llamada `mostrarabs` que espera que se le transmita un argumento en número entero y que no devuelve ningún valor. Del mismo modo, si la línea de encabezado fuera `void mostrarabs(float numero),` la reconocería como una función que espera que se transmita un argumento de punto flotante cuando se llame a la función.

La ventaja al usar la `T` dentro de la línea de encabezado de la plantilla de función es que representa un tipo de datos general que es reemplazado por un tipo de datos real, como `int`, `float`, `double`, etc., cuando el compilador encuentra una llamada a la función real. Por ejemplo, si se encuentra una llamada a la función con un argumento en número entero, el compilador usará la plantilla de función para construir el código para una función que espera un parámetro en número entero. Del mismo modo, si se hace una llamada con un argumento de punto flotante, el compilador construirá una función que espera un parámetro en punto flotante. Como un ejemplo específico de esto, considérese el programa 6.3.

Primero observe las tres llamadas a función que se hacen en la función `main()` mostrada en el programa 6.3, la cual llama a la función `mostrarabs()` con un valor entero, flotante y doble, respectivamente. Ahora revise la plantilla de función para `mostrarabs()` y considere la primera línea `template <class T>.` Esta línea, la cual se llama **prefijo de plantilla,** se usa para informar al compilador que la función que sigue inmediatamente es una plantilla que usa un tipo de datos nombrado `T`. Dentro de la plantilla de función se usa la `T` de la misma manera que cualquier otro tipo de datos, como `int`, `float`, `double`, etc. Luego, cuando el compilador encuentra una llamada a la actual función para `mostrarabs()`, o el tipo de datos del argumento transmitido en la llamada es sustituido por `T` a lo largo de la función. En efecto, el compilador crea una función específica, usando la plantilla, que espera el tipo de argumento en la llamada. En vista que el programa 6.3 hace tres llamadas a `mostrarabs`, cada una con un tipo de datos diferente para el argumento, el compilador creará tres funciones `mostrarabs()` separadas. El compilador sabe cuál



Programa 6.3

```
#include <iostream>
using namespace std;

template <clases T>
void mostrarabs(T número)
{

  if (numero < 0)
    numero = -numero;
  cout << "El valor absoluto del número es "
       << numero << endl;

  return;
}

int main()
{
  int num1 = -4;
  float num2 = -4.23f;
  double num3 = -4.23456;

  mostrarabs(num1);
  mostrarabs(num2);
  mostrarabs(num3);

  return 0;
}
```

función usar con base en los argumentos transmitidos en el momento de la llamada. La salida desplegada cuando el programa 6.3 se ejecuta es:

```
El valor absoluto del numero es 4
El valor absoluto del numero es 4.23
El valor absoluto del numero es 4.23456
```

La letra `T` usada en el prefijo de plantilla `template <class T>` tan sólo es un sustituto para un tipo de datos que se define cuando la función es invocada en realidad. Puede

utilizarse en cambio cualquier letra o identificador que no sea una palabra clave. Por tanto, la plantilla de la función `mostrarabs()` podría haberse definido también como:

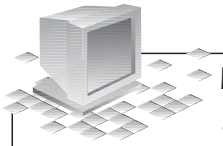
```
template <class TIPOD>
void abs(TIPOD numero)
{
  if (numero < 0)
    numero = -numero;
  cout << "El valor absoluto del número es "
       << numero << endl;

  return;
}
```

Así, en ocasiones es más simple y claro leer la palabra *clase* en el prefijo de plantilla que las palabras *tipo de datos*. Por tanto, el prefijo de plantilla `template <clases T>` puede leerse como "estamos definiendo una plantilla de función que tiene un tipo de datos nombrado `T`". Luego, dentro de la línea de encabezado y el cuerpo de la función definida se usa el tipo de datos `T` (o cualquier otra letra o identificador definido en el prefijo) de la misma manera que cualquier tipo de datos incorporado, como `int`, `float`, `double`, etcétera.

Ahora suponga que se quiere crear una plantilla de función para incluir el tipo de retorno y una variable declarada en forma interna. Por ejemplo, considere la siguiente plantilla de función:

```
template <clase T> // prefijo de plantilla
T abs(T valor) // linea de encabezado
{
  T numabs; // declaracion de variable

  if (valor < 0)
    numabs = -valor;
  else
    numabs = valor;

  return numabs;
}
```

En esta definición de plantilla, se ha usado el tipo de datos `T` para declarar tres elementos: el tipo devuelto de la función, el tipo de datos de un parámetro de función único llamado `valor`, y una variable declarada dentro de la función. El programa 6.4 ilustra cómo esta plantilla de función podría utilizarse dentro del contexto de un programa completo.



Programa 6.4

```
#include <iostream>
using namespace std;

template <class T> // prefijo de plantilla
T abs(T valor) // linea de encabezado
{
  T numabs; // declaracion de variable

  if (valor < 0)
    numabs = -valor;
  else
    numabs = valor;

  return numabs;
}
int main()
{
  int num1 = -4;
  float num2 = -4.23f;
  double num3 = -4.23456;

  cout << "El valor absoluto de " << num1
       << " es " << abs(num1) << endl;
  cout << "El valor absoluto de " << num2
       << " es " << abs(num2) << endl;
  cout << "El valor absoluto de " << num3
       << " es " << abs(num3) << endl;

  return 0;
}
```

En la primera llamada a `abs()` hecha dentro de `main()`, se transmite un valor entero como un argumento. En este caso, el compilador sustituye un tipo de datos `int` para el tipo de datos `T` en la plantilla de función y crea la siguiente función:

```
int abs(int valor) // linea de encabezado
{
  int numabs; // declaracion de la variable

  if (valor < 0)
    numabs = -valor;
  else
    numabs = valor;
```

```
    return numabs;
}
```

Del mismo modo, en la segunda y tercera llamadas a la función, el compilador crea dos funciones más, una en la que el tipo de datos `T` es reemplazado por la palabra clave `float`, y otra en la que el tipo de datos `T` es reemplazado por la palabra clave `double`. La salida producida por el programa 6.4 es:

```
El valor absoluto de -4 es 4
El valor absoluto de -4.23 es 4.23
El valor absoluto de -4.23456 es 4.23456
```

El valor de usar la plantilla de función es que se ha usado una definición de función para crear tres funciones diferentes, cada una de las cuales usa la misma lógica y operaciones pero opera sobre diferentes tipos de datos.

Por último, aunque los programas 6.3 y 6.4 definen una plantilla de función que usa un solo tipo de datos sustituto, pueden definirse plantillas de función con más de un tipo de datos. Por ejemplo, el prefijo de plantilla

```
template <class TIPOD1, class TIPOD2, class TIPOD3>
```

puede utilizarse para crear una plantilla de función que requiere tres tipos de datos diferentes. Como antes, dentro del encabezado y el cuerpo de la plantilla de función los tipos de datos `TIPOD1`, `TIPOD2` y `TIPOD3` se usarían de la misma manera que cualquier tipo de datos incorporado, como un `int`, `float`, `double`, etc. Además, como se señaló antes, los nombres `TIPOD1`, `TIPOD2` y `TIPOD3` pueden ser cualquier identificador que no sea una palabra clave. De manera convencional, se usaría la letra `T` seguida por cero o más dígitos, como `T`, `T1`, `T2`, `T3`, etcétera.

Ejercicios 6.1

1. Para los siguientes encabezados de función, determine el número, tipo y orden (secuencia) de los valores que deben transmitirse a la función:
   a. `void factorial(int n)`
   b. `void voltios(int res, double induc, double cap)`
   c. `void potencia(int tipo, double induc, double cap)`
   d. `void marcador(char tipo, double corriente, double tiempo)`
   e. `void total(double cantidad, double tasa)`
   f. `void roi(int a, int b, char c, char d, double e, double f)`
   g. `void obtener_valor(int elemento, int iter, char decmarcador, char delim)`

2. a. Escriba una función nombrada `revisar()` que tenga tres parámetros. El primer parámetro deberá aceptar un número entero, el segundo parámetro un número de precisión doble y el tercer parámetro un número de precisión doble. El cuerpo de la función deberá desplegar sólo los valores de los datos transmitidos a la función cuando es llamada. (*NOTA*: Cuando se rastrean errores en las funciones, es muy útil hacer que la función despliegue los valores que se le han transmitido. Con bastante frecuencia, el error no está en lo que el cuerpo de la función hace con los datos, sino en los datos recibidos y almacenados.)

b. Incluya la función escrita en el ejercicio 2a en un programa que funcione. Asegúrese que su función es llamada desde `main().` Pruebe la función transmitiéndole varios datos.

3. a. Escriba una función llamada `encontrarAbs()` que acepte un número trasmitido de precisión doble, calcule su valor absoluto y despliegue el valor absoluto. El valor absoluto de un número es el mismo número si éste es positivo y el negativo del número si éste es negativo.
b. Incluya la función escrita en el ejercicio 3a en un programa que funcione. Asegúrese que su función es llamada desde `main()`. Pruebe la función transmitiéndole varios datos.

4. a. Escriba una función llamada `mult()` que acepte dos números en punto flotante como parámetros, multiplique estos dos números y despliegue el resultado.
b. Incluya la función escrita en el ejercicio 4a en un programa que funcione. Asegúrese que su función es invocada desde `main()`. Pruebe la función transmitiéndole varios datos.

5. a. Escriba una función llamada al_`cuadrado()` que calcule el cuadrado del valor que se le transmite y despliegue el resultado. La función deberá ser capaz de elevar al cuadrado números con puntos decimales.
b. Incluya la función escrita en el ejercicio 5a en un programa que funcione. Asegúrese que su función es llamada desde `main()`. Pruebe la función transmitiéndole varios datos.

6. a. Escriba una función nombrada `funpot()` que eleve un número entero que se le transmita a una potencia en número entero positivo y despliegue el resultado. El número entero positivo deberá ser el segundo valor transmitido a la función. Declare la variable usada para almacenar el resultado como un tipo de datos entero largo para asegurar suficiente almacenamiento para el resultado.
b. Incluya la función escrita en el ejercicio 6a en un programa que funcione. Asegúrese que su función es llamada desde `main()`. Pruebe la función transmitiéndole varios datos.

7. a. Escriba un programa en C++ que devuelva la parte fraccionaria de cualquier número introducido por el usuario. Por ejemplo, si se introduce el número 256.879, debería desplegarse el número 0.879. (*Sugerencia:* Use un `int cast`.)
b. Introduzca, compile y ejecute el programa escrito para el ejercicio 7a.

8. a. Escriba un programa en C++ que acepte un argumento en número entero y determine si el entero transmitido es par o non. (*Sugerencia:* Utilice el operador %.)
b. Introduzca, compile y ejecute el programa escrito para el ejercicio 8a.

9. a. Escriba una función que produzca una tabla de los números del 1 al 10, sus cuadrados y sus cubos. La función deberá producir el mismo despliegue que el producido por el programa 5.11.
b. Incluya la función escrita en el ejercicio 9a en un programa que funcione. Asegúrese que su función es llamada desde `main()`. Pruebe la función transmitiéndole varios datos.

10. a. Modifique la función escrita para el ejercicio 9 para aceptar el valor inicial de la tabla, el número de valores que se van a desplegar y el incremento entre valores. Si el incremento no se envía de manera explícita, la función deberá usar un valor por omisión de 1. Nombre su función `selTab()`. Una llamada a `selTab(6, 5, 2);` deberá producir una tabla de cinco líneas, la primera línea iniciando con el número 6 y cada número subsiguiente incrementándose en 2.

b. Incluya la función escrita en el ejercicio 10a en un programa que funcione. Asegúrese que su función es llamada desde `main()`. Pruebe la función transmitiéndole varios datos.

11. Puede construirse una función útil que no utilice parámetros para que devuelva un valor para π que sea preciso al número máximo de lugares decimales permitido por su computadora. Este valor se obtiene tomando el arco seno de 1.0, el cual es π/2, y multiplicando el resultado por 2. En C++, la expresión requerida es 2.0* asin(1.0), donde la función `asin()` es proporcionada en la biblioteca matemática estándar de C++ (recuerde incluir `cmath`). Usando esta expresión, escriba una función en C++ llamada `pi()` que calcule y despliegue el valor de π.

12. a. Escriba una plantilla de función llamada `despliegue()` que despliegue el valor del argumento único que se le transmite cuando es invocada la función.

b. Incluya la plantilla de función creada en el ejercicio 12a dentro de un programa en C++ completo que llame a la función tres veces: una con un argumento de carácter, una con un argumento en número entero y una con un argumento de número de precisión doble.

13. a. Escriba una plantilla de función llamada `entero()` que devuelva el valor entero de cualquier argumento que se le transmita cuando la función sea llamada.

b. Incluya la plantilla de función creada en el ejercicio 13a dentro de un programa en C++ completo que llame a la función tres veces: una con un argumento de carácter, una con un argumento en número entero y una con un argumento de número de precisión doble.

14. a. Escriba una plantilla de función llamada `maximo()` que devuelva el valor máximo de tres argumentos que se transmitan a la función cuando sea llamada. Suponga que los tres argumentos serán del mismo tipo de datos.

b. Incluya la plantilla de función creada para el ejercicio 14a dentro de un programa en C++ completo que llame a la función con tres números enteros y luego con tres números de precisión doble.

15. a. Escriba una plantilla de función llamada al_`cuadrado()` que calcule y devuelva el cuadrado del argumento único transmitido a la función cuando es llamada.

b. Incluya la plantilla de función creada para el ejercicio 15a dentro de un programa en C++ completo.

6.2 DEVOLVER UN SOLO VALOR

Al utilizar el método de transmitir datos a una función presentado en la sección anterior, la función llamada sólo recibe copias de los valores contenidos en los argumentos en el momento de la llamada (repase la figura 6.3 si no le ha quedado claro esto). Cuando se transmite un valor a una función invocada de esta manera, el argumento transmitido se conoce como ***transmitido por valor*** y es una ventaja distintiva de C++.[8] En vista que la función llamada no tiene acceso directo a las variables usadas como argumentos por la función que llama, no puede alterar de manera inadvertida el valor almacenado en una de estas variables.

La función que recibe los argumentos transmitidos por valor puede procesar los valores que se le envían en cualquier forma deseada y devolver en forma directa cuando mucho uno, y sólo un, valor "legítimo" a la función que llama (véase la figura 6.7). En esta sección se verá cómo se devuelve un valor así a la función que llama. Como podría esperarse, dada la flexibilidad de C++, hay una forma de devolver más de un solo valor, pero éste es el tema de la siguiente sección.

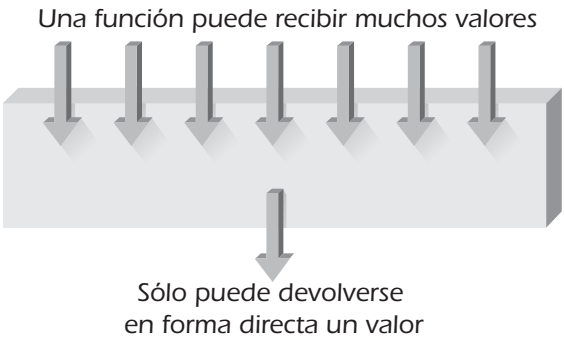


Figura 6.7 *Una función devuelve en forma directa cuando mucho un valor.*

Como ocurre con la llamada a una función, devolver en forma directa un valor requiere que la interfaz entre la función llamada y la que llama se maneje en forma correcta. Desde el lado de la transacción devuelta, la función llamada debe proporcionar los siguientes elementos:

- el tipo de datos del valor devuelto
- el valor real que se devuelve

Una función que devuelve un valor debe especificar, en su línea de encabezado, el tipo de datos del valor que se devolverá. Recuérdese que la línea de encabezado de la función es la primera línea de la función, la cual incluye tanto el nombre de la función como una lista de nombres de parámetros. Como ejemplo, considérese la función `encontrarMax()` escrita en la sección anterior. Determina el valor máximo de dos números transmitidos a la función. Por conveniencia, se enlista de nuevo el código `encontrarMax()`:

[8]Esto también se conoce como ***llamada por valor.*** Sin embargo, el término no se refiere a la llamada de la función en conjunto, sino a la forma en que se transmite un argumento individual cuando se hace la llamada a una función.

```
void encontrarMax(int x, int y)
{                     // inicio del cuerpo de la funcion
  int numMax;         // declaracion de la variable

  if (x >= y)         // encontrar el numero maximo
    numMax = x;
  else
    numMax = y;

  cout << "\nEl máximo de los dos números es "
       << numMax << endl;

}  // fin del cuerpo de la funcion y fin de la funcion
```

Tal como está escrito, la línea de encabezado de la función es

```
void encontrarMax(int x, int y)
```

donde `x` y `y` son los nombres elegidos para los parámetros de la función.

Si ahora `encontrarMax()` va a devolver un valor, la línea de encabezado de la función debe ser modificado para incluir el tipo de datos del valor que se devuelve. Por ejemplo, si se va a devolver un valor entero, la línea de encabezado de la función apropiada es

```
int encontrarMax(int x, int y)
```

Del mismo modo, si la función va a recibir dos parámetros de precisión simple y devolverá un valor de precisión simple, la línea de encabezado de la función correcta es

```
float encontrarMax(float x, float y)
```

y si la función va a recibir dos parámetros de precisión doble y a devolver un valor de precisión doble, la línea de encabezado sería[9]

```
double encontrarMax(double x, double y)
```

Ahora se modificará la función `encontrarMax()` para devolver el valor máximo de los dos números que se le transmiten. Para hacer esto, primero se debe determinar el tipo de datos del valor que se va a devolver e incluir este tipo de datos en la línea de encabezado de la función.

En vista que el valor máximo determinado por `encontrarMax()` se almacena en la variable de número entero `numMax`, es el valor de esta variable el que deberá devolver la función. Devolver un valor entero de `encontrarMax()` requiere que la declaración de la función sea

```
int encontrarMax(int x, int y)
```

Obsérvese que ésta es igual a la línea de encabezado de la función original para `encontrarMax()` con la sustitución de la palabra clave `int` en lugar de la palabra clave `void`.

[9]El tipo de datos devuelto sólo se relaciona con los tipos de datos de los parámetros en la medida en que el valor devuelto se calcula a partir de valores de parámetros. En este caso, en vista que la función se usa para devolver el valor máximo de sus parámetros, tendría poco sentido devolver un tipo de datos que no correspondiera con los tipos de parámetros de la función.

Habiendo declarado el tipo de datos que devolverá `encontrarMax()`, todo lo que resta es incluir una instrucción dentro de la función que cause la devolución del valor correcto. Para devolver un valor, una función debe usar una **instrucción de `devolución`**, la cual tiene la forma:[10]

return expresión;

Cuando se encuentra la instrucción de devolución, la expresión se evalúa primero. El valor de la expresión es convertido entonces de manera automática en el tipo de datos declarado en el encabezado de la función antes de ser enviado de regreso a la función que llama. Después que es devuelto el valor, el control del programa se revierte a la función que llama. Por tanto, para devolver el valor almacenado en `numMax`, todo lo que se necesita hacer es agregar la instrucción `return numMax;` antes de la llave de cierre de la función `encontrarMax()`. El código de la función completo es:

Éstos deberán ser del mismo tipo de datos

```
int encontrarMax(int x, int y) // linea de encabezado
                                  de la funcion
{                              // inicio del cuerpo de
                                  la funcion
  int numMax;                  // declaración de la variable

   if (x >= y)
     numMax = x;
   else
     numMax = y;

   return numMax;              // instruccion de devolucion
}
```

En este nuevo código para la función `encontrarMax()` hay que observar que el tipo de datos de la expresión contenida en la instrucción de devolución corresponde en forma correcta al tipo de datos en la línea de encabezado de la función. Le corresponde al programador asegurar que esto sea así en todas las funciones que devuelven un valor. Si hay una falla en hacer que corresponda con exactitud el valor devuelto con el tipo de datos declarado en la función, puede ser que no se produzca un error cuando su programa sea compilado, pero esto puede conducir a resultados indeseables debido a que el valor devuelto siempre es convertido al tipo de datos declarado en la declaración de la función. Por lo general esto es un problema sólo cuando la parte fraccionaria de un número de punto flotante o de precisión doble devuelta es truncada debido a que se declaró que la función devolvería un valor entero.

Habiendo cuidado el lado emisor de la transacción de devolución, ahora se debe preparar a la función que llama para recibir el valor enviado por la función llamada. En el lado que llama (receptor), la función que llama debe

- ser alertada del tipo de valor a esperar
- usar de manera apropiada el valor devuelto

[10]Muchos programadores colocan la expresión entre paréntesis, produciendo la instrucción `return (expresión);`. Aunque puede usarse cualquier forma, por consistencia sólo se adoptará una.

Para alertar a la función que llama acerca del tipo de valor devuelto que debe esperar se utiliza el prototipo de función. Por ejemplo, incluir el prototipo de función

```
int encontrarMax(int, int);
```

antes de la función `main()` es suficiente para alertar a `main()` que `encontrarMax()` es una función que devolverá un valor entero.

Para usar en verdad un valor devuelto hay que proporcionar una variable para almacenar el valor o usar el valor en forma directa en una expresión. Se logra almacenar el valor devuelto en una variable usando una instrucción de asignación estándar. Por ejemplo, puede usarse la instrucción de asignación

```
max = encontrarMax (primernum, segundonum);
```

para almacenar el valor devuelto por `encontrarMax()` en la variable llamada `max`. Esta instrucción de asignación hace dos cosas. Primera, el lado derecho de la instrucción de asignación llama a `encontrarMax()`, luego el resultado devuelto por `encontrarMax()` se almacena en la variable `max`. En vista que el valor devuelto por `encontrarMax()` es un número entero, la variable `max` deberá declararse también como una variable en número entero dentro de las declaraciones de variables de la función que llama.

El valor devuelto por una función no necesita almacenarse en forma directa en una variable, pero puede usarse siempre que una expresión sea válida. Por ejemplo la expresión `2 * encontrarMax(primernum, segundonum)` multiplica el valor devuelto por `encontrarMax()` por dos, y la instrucción

```
cout << encontrarMax(primernum, segundonum);
```

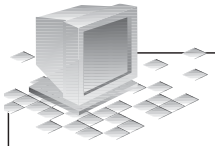
despliega el valor devuelto.

El programa 6.5 ilustra la inclusión de instrucciones de prototipo y de asignación para que `main()` declare, llame y almacene en forma correcta un valor devuelto por `encontrarMax()`. Como antes, y para conservar la convención de colocar primero la función `main()`, se ha colocado la función `encontrarMax()` después de `main()`.

Al revisar el programa 6.5 es importante señalar los cuatro elementos que se han introducido en esta sección. El primer elemento es el prototipo para `encontrarMax()`. Esta instrucción, la cual termina con punto y coma, como todas las instrucciones de declaración, alerta a `main()` y a las funciones subsiguientes que usen `encontrarMax()` con el tipo de datos que devolverá `encontrarMax()`. El segundo elemento que se puede observar en `main()` es el uso de una instrucción de asignación para almacenar el valor devuelto de la llamada a `encontrarMax()` en la variable `numMax.` También se ha asegurado de declarar en forma correcta `numMax` como un número entero dentro de las declaraciones de variables de `main()` de modo que concuerde con el tipo de datos del valor devuelto.

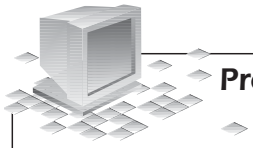
Los últimos dos elementos que hay que observar conciernen a la codificación de la función `encontrarMax()`. La primera línea de `encontrarMax()` declara que la función devolverá un valor entero, y la expresión en la instrucción de devolución evalúa un tipo de datos que corresponda. Por tanto `encontrarMax()` es consistente internamente en enviar un valor entero de vuelta a `main()`, y `main()` ha sido alertado en forma correcta para recibir y usar el número entero devuelto.

Al escribir sus propias funciones debe tener en cuenta siempre estos cuatro elementos. Como ejemplo, vea si puede identificar estos cuatro elementos en el programa 6.6.



Programa 6.5

```
#include <iostream>
using namespace std;

int encontrarMax(int, int); // el prototipo de la función

int main()
{
  int primernum, segundonum, max;


  cout << "\nIntroduzca un numero: ";
  cin >> primernum;
  cout << "¡Estupendo! Por favor introduzca un segundo número: ";
  cin >> segundonum;

  max = encontrarMax(primernum, segundonum); // aquí se llama a la función

  cout << "\nEl máximo de los dos números es " << max << endl;

  return 0;
}

int encontrarMax(int x, int y)
{                     // inicio del cuerpo de la función
  int numMax;         // declaración de variable

  if (x >= y)         // encontrar el número máximo
    numMax = x;
  else
    numMax = y;

  return numMax;      // instrucción de devolución
}
```



Programa 6.6

```
#include <iostream>
using namespace std;

double convertir_temp(double); // prototipo de la función

int main()
{
  const CONVERSIONES = 4; // numero de conversiones que se harán
  int cuenta;
  double fahren;

  for(cuenta = 1; cuenta <= CONVERSIONES; cuenta++)
  {
    cout << "\nIntroduzca una temperatura en grados Fahrenheit: ";
    cin  >> fahren;
    cout << "El equivalente en grados Celsius es "
         << convertir_temp(fahren) << endl;
  }

  return 0;
}

// convertir Fahrenheit a Celsius
double convertir_temp(double in_temp)
{
  return (5.0/9.0) * (in_temp - 32.0);
}
```

Al revisar el programa 6.6, se analizará primero la función convertir_temp(). La definición completa de la función comienza con la línea de encabezado de la función y termina con la llave de cierre después de la instrucción de devolución. La función es declarada como double; esto significa que la expresión en la instrucción de devolución de la función debe evaluar a un número de precisión doble, lo cual hace. En vista que la línea de encabezado de la función no es una instrucción sino el inicio del código que define la función, la línea de encabezado de la función no termina con punto y coma.

En el lado receptor, main() tiene un prototipo para la función convertir_temp() que concuerda con la definición de la función de convertir_temp(). No se declara una variable en main() para almacenar el valor devuelto de convertir_temp() porque el valor devuelto es transmitido de inmediato a cout para su despliegue.

Vale la pena mencionar un punto más aquí. Uno de los propósitos de las declaraciones, como se aprendió en el capítulo 2, es alertar a la computadora de la cantidad de almacenamiento interno reservado para los datos. El prototipo para `convertir_temp()` ejecuta esta tarea y alerta al compilador del tipo de almacenamiento necesario para el valor devuelto. En vista que se ha elegido siempre enlistar `main()` como la primera función en un archivo, se deben incluir prototipos de función para todas las funciones llamadas por `main()` y cualesquiera funciones subsiguientes.

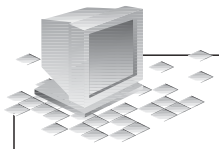
Funciones `inline`[11]

Llamar a una función pone una cierta cantidad de sobrecarga en la computadora, ya que consiste en colocar valores de argumentos en una región de memoria reservada a la que la función tiene acceso (esta región de memoria se conoce como **pila**), transmitir el control a la función, proporcionar una ubicación de memoria reservada para cualquier valor devuelto (de nuevo, la región de pila de la memoria se usa para este propósito) y por último regresar al punto apropiado en el programa que llama. Esta sobrecarga se justifica cuando una función es invocada muchas veces, porque puede reducir de manera significativa el tamaño de un programa. En lugar de repetir el mismo código cada vez que es necesario, el código se escribe una sola vez, como una función, y se llama cada vez que es necesario.

Sin embargo, para funciones pequeñas que no son llamadas muchas veces no se justifica la sobrecarga de transmitir y devolver valores. No obstante, aún sería conveniente agrupar las líneas repetitivas de código bajo un nombre de función común y hacer que el compilador coloque este código en forma directa en el programa siempre que se llame a la función. Esta capacidad es proporcionada por funciones `inline`.

Indicarle al compilador de C++ que una función es `inline` causa que una copia del código de la función sea colocada en el programa en el punto en que es invocada. la función. Por ejemplo, considérese la función `convertir_temp()` definida en el programa 6.6. En vista que ésta es una función relativamente corta, es ideal para ser una función `inline`. Para hacer a ésta, o a cualquier otra función, una función `inline`, tan sólo se requiere colocar la palabra reservada `inline` antes del nombre de la función y definir la función antes que se haga cualquier llamada. Esto se lleva a cabo para la función `convertir_temp()` en el programa 6.7.

[11] Esta sección es opcional y puede omitirse en la primera lectura sin perder la continuidad de la materia.



Programa 6.7

```
#include <iostream>
using namespace std;

inline double convertir_temp(double in_temp) // una función inline
{
  return (5.0/9.0) * (in_temp - 32.0);
}

int main()
{
  const CONVERSIONES = 4; // numero de conversiones que se harán
  int cuenta;
  double fahren;

  for(cuenta = 1; cuenta <= CONVERSIONES; cuenta++)
  {
    cout << "\nIntroduzca una temperatura en grados Fahrenheit: ";
    cin  >> fahren;
    cout << "El equivalente en grados Celsius es "
         << convertir_temp(fahren) << endl;
  }

  return 0;
}
```

En el programa 6.7 hay que observar que la función `inline` se coloca antes de cualesquier llamada a ella. Esto es un requisito de todas las funciones `inline` y evita la necesidad de un prototipo de función antes de cualquier función subsiguiente que llama. En vista que la función ahora es una `inline`, su código será expandido en forma directa siempre que sea llamada en el programa.

La ventaja de usar una función `inline` es que aumenta la velocidad de ejecución. En vista que la función `inline` es expandida en forma directa e incluida en cada expresión o instrucción que la llame, no se pierde tiempo de ejecución debido a la sobrecarga de la llamada y devolución requeridas por una función que no es `inline`. La desventaja es el aumento en el tamaño del programa cuando una función `inline` es llamada en forma repetida. Cada vez que se hace referencia a una función `inline`, se reproduce el código completo de la función y se almacena como una parte integral del programa. Una función que no es `inline`, sin embargo, se almacena en la memoria una sola vez. Sin importar cuántas veces sea llamada la función, se usa el mismo código. Por consiguiente, las funciones `inline` sólo deberán usarse para funciones pequeñas que no son llamadas en forma eshaustiva en un programa.

Ejercicios 6.2

1. Vuelva a escribir el programa 6.5 de modo que la función `encontrarMax()` acepte dos argumentos de precisión doble y devuelva un valor de precisión doble a `main()`. Asegúrese de modificar `main()` a fin de transmitir dos valores de punto flotante a `encontrarMax()` y aceptar y almacenar el valor de precisión doble devuelto por `encontrarMax()`.

2. Para los siguientes encabezados de función, determine el número, tipo y orden (secuencia) de los valores que deberán ser transmitidos a la función cuando sea llamada y el tipo de datos del valor devuelto por la función.

   a. `int factorial(int n)`
   b. `double voltios(int res, double induc, double cap)`
   c. `double potencia(int tipo, double induc, double cap)`
   d. `char marcador(char tipo, float corriente, float tiempo)`
   e. `int total(float cantidad, float tasa)`
   f. `float roi(int a, int b, char c, char d, float e, float f)`
   g. `void obtener_valor(int elemento, int iter, char decmarcador, char delim)`

3. Escriba encabezados de función para lo siguiente:

   a. una función llamada `verificar` que tenga tres parámetros. El primer parámetro deberá aceptar un número entero, el segundo un número de precisión doble y el tercer parámetro un número de precisión doble. La función no devuelve ningún valor.
   b. una función nombrada `encontrarAbs()` que acepte un número de precisión doble que se le transmite y devuelva su valor absoluto.
   c. una función nombrada `mult` que acepte dos números de punto flotante como parámetros, multiplique estos dos números y devuelva el resultado.
   d. una función nombrada `al_cuadrado()` que calcule y devuelva el cuadrado del valor entero que se le transmite.
   e. una función llamada `funpot()` que eleve un número entero que se le transmita a una potencia entera positiva (como un argumento) y devuelva el resultado como un número entero.
   f. una función que produzca una tabla de los números del 1 al 10, sus cuadrados y sus cubos. Que no se transmitan argumentos a la función y la función no devuelve ningún valor.

4. a. Escriba una función llamada `trianguloRectangulo()` que acepte el largo de dos lados de un triángulo rectángulo como los argumentos *a* y *b*, respectivamente. La subrutina deberá determinar y devolver la hipotenusa, *c*, del triángulo. (*Sugerencia*: Use el teorema de Pitágoras, $c^2 = a^2 + b^2$).
   b. Incluya la función escrita para el ejercicio 4a en un programa que funcione. La unidad de función `main()` deberá llamar en forma correcta a `triangulo-Rectangulo()` y desplegar el valor devuelto por la función.

5. a. Escriba una función en C++ llamada `encontrarAbs()` que acepte un número de precisión doble que se le transmite, calcule su valor absoluto y devuelva el valor absoluto a la función que llama. El valor absoluto de un número es el mismo número si éste es positivo y el negativo del número si éste es negativo.

b. Incluya la función escrita en el ejercicio 5a en un programa que funcione. Asegúrese que su función es llamada desde `main()` y que devuelve en forma correcta un valor a `main()`. Haga que `main()` use una instrucción `cout` para desplegar el valor devuelto. Pruebe la función transmitiéndole varios datos.

6. a. El volumen, v, de un cilindro está dado por la fórmula

$$v = \pi r^2 l$$

donde r es el radio del cilindro y l es su largo. Usando esta fórmula, escriba una función C++ nombrada `vol_cil()` que acepte el radio y el largo de un cilindro y devuelva su volumen.

b. Incluya la función escrita en el ejercicio 6a en un programa que funcione. Asegúrese que su función es llamada desde `main()` y devuelve en forma correcta un valor a `main()`. Haga que `main()` use una instrucción `cout` para desplegar el valor devuelto. Pruebe la función transmitiéndole varios datos.

7. a. El área de la superficie, s, de un cilindro está dada por la fórmula

$$s = 2\pi rl$$

donde r es el radio del cilindro y l su largo. Usando esta fórmula escriba una función en C++ nombrada `areasup()` que acepte el radio y largo de un cilindro y devuelva su superficie.

b. Incluya la función escrita en el ejercicio 7a en un programa que funcione. Asegúrese que su función es llamada desde `main()` y devuelva en forma correcta un valor a `main()`. Haga que `main()` use una instrucción `cout` para desplegar el valor devuelto. Pruebe la función transmitiéndole varios datos.

8. Un polinomio de segundo grado en x está dado por la expresión $ax^2 + bx + c$, donde a, b, y c son números conocidos y a no es igual a cero. Escriba una función en C++ nombrada *poli_dos(a, b, c, x)* que calcule y devuelva el valor de un polinomio de segundo grado para valores transmitidos de *a, b, c* y *x*.

9. a. La deflexión máxima permisible de una viga depende de su función. Para un piso, la deflexión máxima permisible típica, en pulgadas, es $D_{máx} = L / 240$, mientras para una viga del techo $D_{máx} = L / 180$, donde L es el largo de la viga en pulgadas. Usando estas fórmulas, escriba y pruebe una función nombrada `deflexMax()` que acepte el largo de una viga, en pies, y el tipo de viga (piso o techo) como un código de carácter y devuelva la deflexión máxima permisible.

b. Incluya la función escrita en el ejercicio 9a en un programa que funcione. Asegúrese que su función es llamada desde `main()` y devuelve en forma correcta un valor a `main()`. Haga que `main()` use una instrucción `cout` para desplegar el valor devuelto. Pruebe la función transmitiéndole varios datos.

10. a. La carga, P_{cr}, en unidades de kilolibras (klb), aplicada a una columna que causará que la columna se combe se conoce como la carga de combamiento crítica. Esta carga puede determinarse usando la ecuación

$$P_{cr} = \pi^2 E A / (L / r)^2$$

donde E es el módulo de elasticidad del material usado en la columna, A es el área de corte transversal, L es el largo de la columna y r es su radio de giro. Usando esta fórmula, escriba una función en C++ llamada `cargaC()` que acepte valores de E, A, L y r, y devuelva la carga crítica.

b. Incluya la función escrita en el ejercicio 10a en un programa que funcione. Asegúrese que su función es llamada desde `main()` y devuelve en forma correcta un valor a `main()`. Haga que `main()` use una instrucción `cout` para desplegar el valor devuelto. Pruebe la función transmitiéndole varios datos.

11. a. Un algoritmo de programación útil en extremo para redondear un número real a n lugares decimales es

Paso 1: Multiplicar el número por 10^n

Paso 2: Sumar 0.5

Paso 3: Eliminar la parte fraccionaria del resultado

Paso 4: Dividir entre 10^n

Por ejemplo, usar este algoritmo para redondear el número 78.374625 a tres lugares decimales produce:

Paso 1: $78.374625 \times 10^3 = 78374.625$

Paso 2: $78374.625 + 0.5 = 78375.125$

Paso 3: Conservar la parte entera = 78375

Paso 4: 78375 dividido entre $10^3 = 78.375$

Usando este algoritmo, escriba una función en C++ que acepte un valor introducido por un usuario y devuelva el resultado redondeado a dos lugares decimales.

b. Introduzca, compile y ejecute el programa escrito para el ejercicio 11a.

12. a. Escriba una función en C++ llamada `entero()` que devuelva la parte entera de cualquier número que se transmita a la función. (*Sugerencia:* Asigne el argumento transmitido a una variable entera.)

b. Incluya la función escrita en el ejercicio 12a en un programa que funcione. Asegúrese que su función es llamada desde `main()` y devuelve en forma correcta un valor a `main()`. Haga que `main()` use una instrucción `cout` para desplegar el valor devuelto. Pruebe la función transmitiéndole varios datos.

13. a. Escriba una función en C++ nombrada `partefrac()` que devuelva la parte fraccionaria de cualquier número transmitido a la función. Por ejemplo, si se transmite el número 256.879 a `partefrac()`, debería devolverse el número 0.879. Haga que la función `partefrac()` llame a la función `entero()` que escribió en el ejercicio 12. El número devuelto puede determinarse entonces como el número transmitido a `partefrac()` menos el valor devuelto cuando el

mismo argumento es transmitido a `entero()`. El programa completo deberá consistir de `main()` seguido por `partefrac()` seguido por `entero()`.

b. Incluya la función escrita en el ejercicio 13a en un programa que funcione. Asegúrese que su función es llamada desde `main()` y devuelve en forma correcta un valor a `main()`. Haga que `main()` use una instrucción `cout` para desplegar el valor devuelto. Pruebe la función transmitiéndole varios datos.

14. Todos los años que son divisibles entre 400 o son divisibles entre cuatro y no son divisibles entre 100 son años bisiestos. Por ejemplo, en vista que 1600 es divisible entre 400, el año 1600 fue un año bisiesto. Del mismo modo, en vista que 1988 es divisible entre cuatro pero no entre 100, el año 1988 también fue un año bisiesto. Usando esta información, escriba una función en C++ que acepte el año como entrada de un usuario y devuelva un uno si el año transmitido es un año bisiesto o un cero si no lo es.

6.3 DEVOLVER VALORES MÚLTIPLES

En una invocación de una función típica, la función llamada recibe valores de la función que la llama, almacena y manipula los valores transmitidos y devuelve en forma directa cuando mucho un valor único. Cuando los datos son transmitidos en esta manera se conoce como ***transmisión por valor***.

Llamar a una función y transmitir argumentos por valor es una ventaja distintiva de C++. Permite que se escriban funciones como entidades independientes que pueden usar cualquier nombre de variable o parámetro sin preocuparse por que otras funciones puedan usar también el mismo nombre. También alivia cualquier preocupación de que alterar un parámetro o variable en una función pueda alterar de manera inadvertida el valor de una variable en otra función. Bajo este enfoque, los parámetros pueden considerarse como variables inicializadas o variables a las que se les asignarán valores cuando se ejecute la función. Sin embargo, en ningún momento la función llamada tiene acceso directo a cualquier variable definida en la función que llama, aun si la variable se usa como un argumento en la llamada a la función.

Hay ocasiones, sin embargo, en que es necesario alterar este enfoque dándole a una función llamada acceso directo a las variables de su función que llama. Esto le permite a la función, la cual es la función llamada, usen y cambien el valor de variables que se han definido en la función que llama. Para hacer esto se requiere que la dirección de la variable se transmita a la función llamada. Una vez que la función llamada tiene la dirección de la variable, "sabe dónde vive la variable", por así decirlo, y puede tener acceso y cambiar el valor almacenado ahí en forma directa.

La transmisión de direcciones se conoce como función de ***transmisión por referencia,***[12] en vista que la función llamada puede hacer referencia, o tener acceso, a la variable cuya dirección se ha transmitido. C++ proporciona dos tipos de parámetros de dirección, referencias y apuntadores. En esta sección se describe el método que usa parámetros de referencia.

[12]También se conoce como ***llamada por referencia,*** donde, una vez más, el término se aplica sólo a los argumentos cuya dirección se ha transmitido.

Transmisión y uso de parámetros de referencia

Como siempre, al intercambiar datos entre dos funciones se deben atender los lados emisor y receptor del intercambio de datos. Desde el lado emisor, sin embargo, llamar a una función y transmitir una dirección como un argumento que se aceptará como parámetro de referencia es exactamente lo mismo que llamar a una función y transmitir un valor; la función llamada es convocada a la acción proporcionando su nombre y una lista de argumentos. Por ejemplo, la instrucción `valnuevo(primernum, segundonum);` llama a la función nombrada `valnuevo` y le transmite dos argumentos. Que se transmita en realidad un valor o una dirección depende de los tipos de parámetros declarados para `valnuevo()`. Ahora se escribirá la función y el prototipo `valnuevo` de modo que reciban las direcciones de las variables `primernum` y `segundonum`, las cuales se supondrá que son variables de precisión doble, en lugar de sus valores.

Uno de los primeros requisitos al escribir `valnuevo()` es declarar dos parámetros de referencia para aceptar direcciones transmitidas. En C++ se declara un parámetro de referencia usando la sintaxis

tipo-de-datos& nombre-de-referencia

Por ejemplo, la declaración de referencia

```
double& num1;
```

declara que num1 es un parámetro de referencia que se utilizará para almacenar la dirección de un `double`. Del mismo modo, `int& segundonum` declara que `segundonum` es una referencia a un número entero y `char& clave` declara que `clave` es una referencia a un carácter.

Se recordará de la sección 2.4 que el símbolo ampersand, &, en C++ significa "la dirección de". Además, cuando se usa un símbolo & dentro de una declaración se refiere a "la dirección del" tipo de datos precedente. Usando esta información, declaraciones como `double& num1` e `int& segundonum` a veces se entienden con más claridad si se leen al revés. Leer la declaración `double& num1` de esta manera produce la información que "num1 es la dirección de un valor de precisión doble".

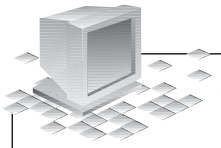
En vista de que es necesario aceptar dos direcciones en la lista de parámetros para `valnuevo()`, pueden usarse las declaraciones `double& num1`, `double& num2`. Al incluir estas declaraciones dentro de la lista de parámetros para `valnuevo()`, y suponer que la función no devuelve ningún valor `(void)`, el encabezado de función para `valnuevo()` se vuelve:

```
void valnuevo(double& num1, double& num2)
```

Para la línea de encabezado de esta función, un prototipo de función apropiado es

```
void valnuevo(double&, double&);
```

Este prototipo y línea de encabezado se han incluido en el programa 6.8, el cual incorpora un cuerpo de función `valnuevo()` completo que despliega y altera en forma directa los valores almacenados en estas variables de referencia desde dentro de la función que se ha llamado.



Programa 6.8

```
#include <iostream>
using namespace std;

void valnuevo(double&, double&); // prototipo con dos parámetros de referencia

int main()
{
  double primernum, segundonum;

  cout << "Introduzca dos números: ";
  cin >> primernum >> segundonum;
  cout << "\nEl valor en primernum es: " << primernum << endl;
  cout << "El valor en segundonum es: " << segundonum << "\n\n";

  valnuevo(primernum, segundonum); // llamada a la función

  cout << "Ahora el valor en primernum es: " << primernum << endl;
  cout << "Ahora el valor en segundonum es: " << segundonum << endl;

  return 0;
}

void valnuevo(double& xnum, double& ynum)
{
  ccout << "El valor en xnum es: " << xnum << endl;
  cout << "El valor en ynum es: " << ynum << "\n\n";
  xnum = 89.5;
  ynum = 99.5;

  return;
}
```

Al llamar a la función `valnuevo()` dentro del programa 6.8, es importante entender la conexión entre los argumentos, `primernum` y `segundonum`, usados en la llamada a la función y los parámetros, `xnum` y `ynum`, usados en el encabezado de la función. *Ambos se refieren a los mismos elementos de datos.* La importancia de esto es que los valores en los argumentos (`primernum` y `segundonum`) ahora pueden alterarse desde dentro de `valnuevo()` usando los nombres de parámetros (`xnum y ynum`). Por tanto, el parámetro `xnum` y `ynum` no almacena copias de los valores en `primernum` y `segundonum`, sino que tiene acceso directo a las ubicaciones en la memoria apartados para estos dos argumentos. La equivalencia de los nombres de argumentos en el programa 6.8, la cual es la esencia de una transmisión por referencia, se ilustra en la figura 6.8. Como se muestra en esta figura, los nombres de argumentos y sus nombres de parámetros correspondientes tan sólo son nombres diferentes que se refieren a las mismas áreas de almacenamiento en la memoria. En `main()` se hace referencia a estas ubicaciones de memoria por los nombres

primernum y segundonum, respectivamente, mientras en valnuevo() se hace referencia a las mismas ubicaciones con los nombres de parámetro xnum y ynum, respectivamente.

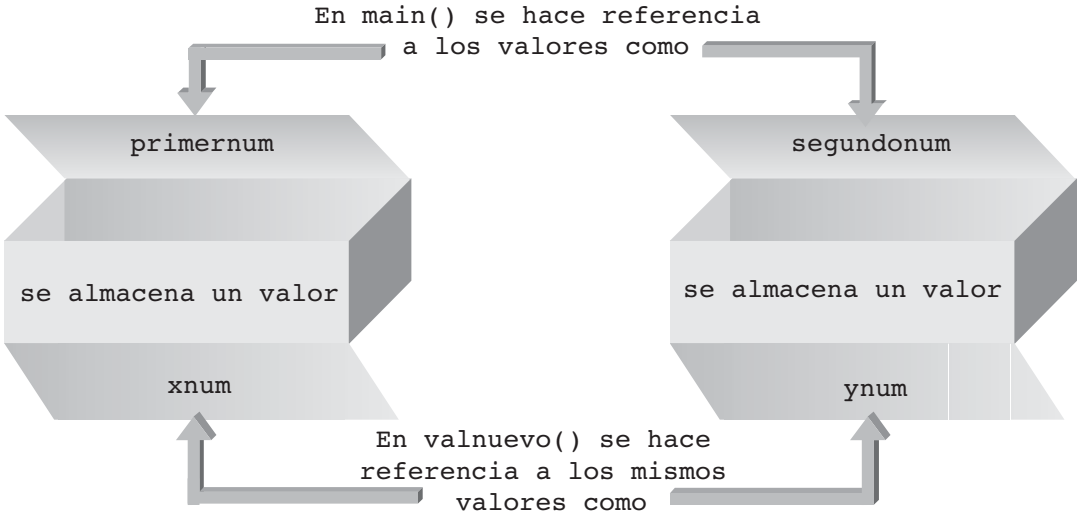


Figura 6.8 *La equivalencia de argumentos y parámetros en el programa 6.8.*

El siguiente ejemplo muestra la ejecución que se obtuvo usando el programa 6.8:

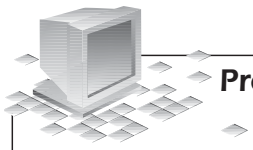
```
Introduzca dos números: 22.5 33.0

El valor en primernum es: 22.5
El valor en segundonum es:   33

El valor en xnum es: 22.5
El valor en ynum es: 33

Ahora el valor en primernum es: 89.5
Ahora el valor en segundonum es: 99.5
```

Al revisar esta salida, hay que observar que los valores desplegados al principio para los parámetros xnum y ynum son los mismos que los desplegados para los argumentos primernum y segundonum. Sin embargo, en vista que xnum y ynum son parámetros de referencia, valnuevo() ahora tienen acceso directo a los argumentos primernum y segundonum. Por tanto, cualquier cambio a xnum dentro de valnuevo() altera en forma directa el valor de primernum en main() y cualquier cambio a ynum cambia en forma directa el valor de segundonum. Como lo ilustran los valores finales desplegados, la asignación de valores a xnum y ynum dentro de valnuevo() se refleja en main() como la alteración de los valores de primernum y segundonum.

La equivalencia entre los argumentos reales que llaman y los parámetros de función ilustrados en el programa 6.8 proporcionan la base para devolver valores múltiples desde adentro de una función. Por ejemplo, suponga que se requiere una función para aceptar tres valores, calcular la suma y el producto de estos valores y devolver estos resultados calculados a la rutina que llama. Al nombrar a la función calcular() y proporcionar cinco parámetros (tres para los datos de entrada y dos referencias para los valores devueltos), puede usarse la siguiente función.

```
void calcular(double num1, double num2, double num3, double& total, double& producto)
{
  total = num1 + num2 + num3;
  producto = num1 * num2 * num3;
  return;
}
```

Esta función tiene cinco parámetros, nombrados `num1`, `num2`, `num3`, `total` y `producto`, de los cuales sólo los últimos dos son declarados como referencias. Por tanto, los primeros tres argumentos son transmitidos por valor y los últimos dos argumentos son transmitidos por referencia. Dentro de la función sólo se alteran los dos últimos parámetros. El valor del cuarto parámetro, `total`, se calcula como la suma de los primeros tres parámetros y el último parámetro, `producto`, se calcula como el producto de los parámetros `num1`, `num2` y `num3`. El programa 6.9 incluye esta función en un programa completo.



Programa 6.9

```
#include <iostream>
using namespace std;

void calcular(double, double, double, double&, double&); // prototipo

int main()
{
  double primernum, segundonum, tercernum, suma, producto;

  cout << "Introduzca tres números: ";
  cin >> primernum >> segundonum >> tercernum;

  calcular(primernum, segundonum, tercernum, suma, producto); // llamada a la función

  cout << "\nLa suma de los números es: " << suma << endl;
  cout << "El producto de los números es: " << producto << endl;

  return 0;
}

void calcular(double num1, double num2, double num3, double& total, double& producto)
{
  total = num1 + num2 + num3;
  producto = num1 * num2 * num3;
  return;
}
```

Dentro de `main()`, la función `calcular()` es llamada usando los cinco argumentos `primernum`, `segundonum`, `tercernum`, suma y producto. Como se requiere, estos argumentos concuerdan en número y tipo de datos con los parámetros declarados por `calcular()`. De los cinco argumentos transmitidos, sólo a `primernum`, `segundonum` y `tercernum` se les han asignado valores cuando se hace la llamada a `calcular()`. Los dos argumentos restantes no se han inicializado y se usarán para recibir valores devueltos por `calcular()`. Dependiendo del compilador utilizado para compilar el programa, estos argumentos contendrán al principio ceros o valores "basura". La figura 6.9 ilustra la relación entre los nombres reales y de parámetro y los valores que contienen después de ser devueltos de `calcular()`.

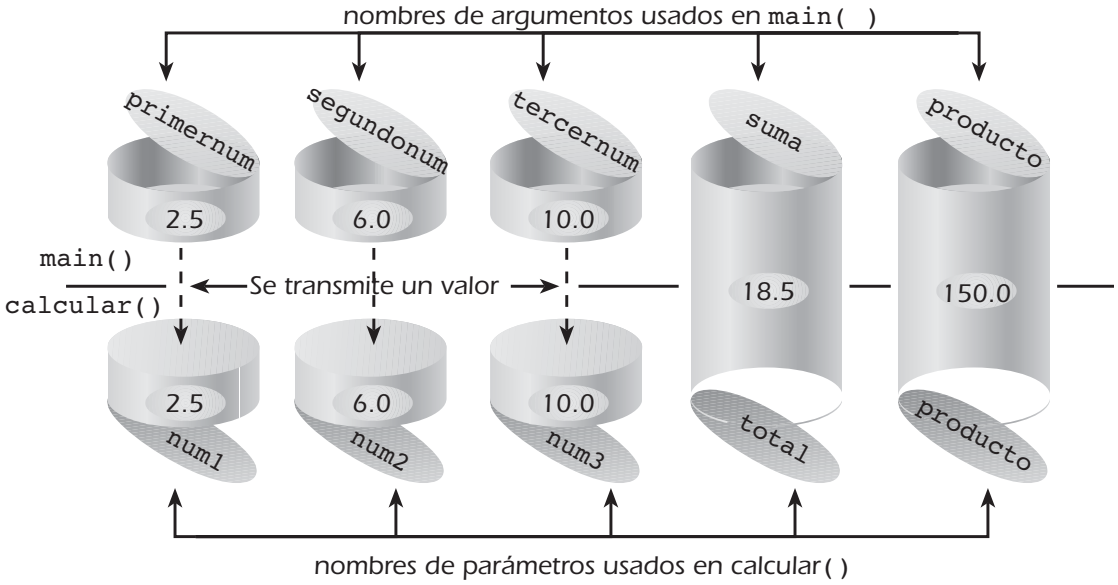


Figura 6.9 *Relación entre los nombres de argumento y de parámetro.*

Una vez que es llamada `calcular()`, usa sus primeros tres parámetros para calcular valores para total y producto y luego devuelve el control a `main()`. Debido al orden de los argumentos reales de la función que llama, `main()` conoce los valores calculados por `calcular()` como suma y producto, los cuales son desplegados luego. A continuación se presenta una muestra de ejecución usando el programa 6.9.

```
Introduzca tres números: 2.5 6.0 10.0
La suma de los números introducidos es: 18.5
El producto de los números introducidos es: 150
```

Como un ejemplo final que ilustra la utilidad de transmitir referencias a una función llamada, se construirá una función llamada `intercambio()` que intercambia los valores de dos de las variables de precisión doble de `main()`. Una función así es útil cuando se clasifica una lista de números.

En vista que es afectado el valor de más de una variable, `intercambio()` no puede escribirse como una función de transmisión por valor que devuelve un valor único. El intercambio deseado de las variables de `main()` por `intercambio()` sólo puede obtenerse dándole a `intercambio()` acceso a las variables de `main()`. Una forma de hacer esto es usar parámetros de referencia.

Ya se ha visto cómo transmitir referencias a dos variables en el programa 6.8. Ahora se construirá una función para intercambiar los valores en los argumentos de referencia transmitidos. Intercambiar valores en dos variables se logra usando el algoritmo de intercambio en tres pasos:

1. Guardar el valor del primer parámetro en una ubicación temporal (véase la figura 6.10a).
2. Almacenar el valor del segundo parámetro en la primera variable (véase la figura 6.10b).
3. Almacenar el valor temporal en el segundo parámetro (véase la figura 6.10c).

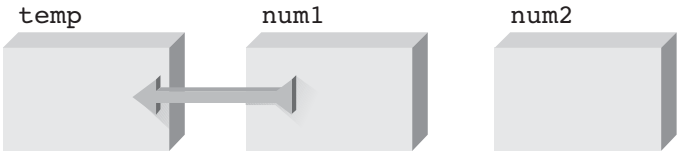


Figura 6.10a *Guardar el primer valor.*

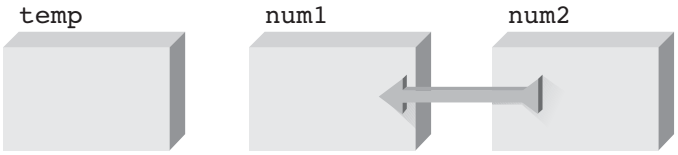


Figura 6.10b *Reemplazar el primer valor con el segundo valor.*

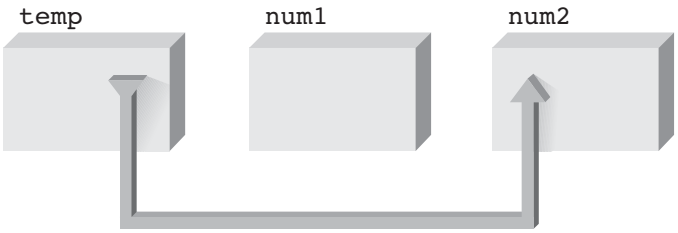


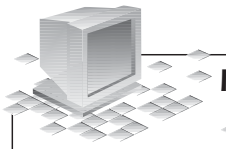
Figura 6.10c *Cambiar el segundo valor.*

A continuación está la función `intercambio()` escrita de acuerdo con estas especificaciones:

```
void intercambio(double& num1, double& num2)
{
  double temp;

  temp = num1;      // guarda el valor de num1
  num1 = num2;      // almacena el valor de num2 en num1
  num2 = temp;      // cambia el valor de num2

  return;
}
```

Hay que observar que el uso de referencias en la línea de encabezado de `intercambio()` le da acceso a `intercambio()` a los argumentos equivalentes en la función que llama. Por

tanto, cualquier cambio a los dos parámetros de referencia en `intercambio()` de manera automática cambia los valores en los argumentos de la función que llama. El programa 6.10 contiene `intercambio()` en un programa completo.



Programa 6.10

```
#include <iostream>
using namespace std;

void intercambio(double&, double&); // la función recibe dos referencias

int main()
{
  double primernum = 20.5, segundonum = 6.25;

  cout << "El valor almacenado en primernum es: " << primernum << endl;
  cout << "El valor almacenado en segundonum es : " << segundonum << "\n\n";

  intercambio(primernum, segundonum); // llama a la función con referencias

  cout << "Ahora el valor almacenado en primernum es: "
       << primernum << endl;
  cout << "Ahora el valor almacenado en segundonum es: "
       << segundonum << endl;

  return 0;
}

void intercambio(double& num1, double& num2)
{
  double temp;

  temp = num1;      // guarda el valor de num1
  num1 = num2;      // almacena el valor de num2 en num1
  num2 = temp;      // cambia el valor de num2

  return;
}
```

La siguiente muestra de ejecución se obtuvo usando el programa 6.10:

```
El valor almacenado en primernum es: 20.5
El valor almacenado en segundonum es: 6.25


Ahora el valor almacenado en primernum es: 6.25
Ahora el valor almacenado en segundonum es:  20.5
```

Como lo ilustra su salida, los valores almacenados en las variables de `main()` se han modificado desde dentro de `intercambio()`, lo cual fue posible por el uso de parámetros de referencia. Si se hubiera utilizado una transmisión por valor, el intercambio dentro de `intercambio()` sólo habría afectado los parámetros de `intercambio()` y no lograría nada con respecto a las variables de `main()`. Por tanto, una función como `intercambio()` sólo puede escribirse usando referencias o algún otro medio que proporcione acceso a las variables de `main()` (este otro medio se llava a cabo por apuntadores, tema del capítulo 12).

Al usar argumentos de referencia, es necesario mencionar dos precauciones. La primera es que los argumentos de referencia deben ser variables (es decir, no pueden usarse para cambiar constantes). Por ejemplo, llamar a `intercambio()` con dos constantes, como en la llamada `intercambio(20.5, 6.5)` transmite dos constantes a la función. Aunque `intercambio()` puede ejecutarse, no cambiará los valores de estas constantes.[13]

La segunda precaución es que una llamada a función en sí no da indicio de que la función llamada usará parámetros de referencia. La opción por omisión en C++ es hacer transmisiones por valor en lugar de transmisiones por referencia, precisamente para limitar la capacidad de una función llamada para alterar variables en la función que llama. Este procedimiento de llamada deberá acatarse siempre que sea posible, lo cual significa que los parámetros de referencia sólo deberían usarse en situaciones muy restringidas que en realidad requieran múltiples valores devueltos, como en la función `intercambio()` ilustrada en el programa 6.10. La función `calcular()`, incluida en el programa 6.9, aunque útil con propósitos ilustrativos, también podría ser escrita como dos funciones separadas, cada una devolviendo un valor único.

Ejercicios 6.3

1. Escriba declaraciones de parámetros para lo siguiente:
   a. un parámetro nombrado cantidad que será una referencia a un valor de precisión doble.
   b. un parámetro nombrado precio que será una referencia a un número de precisión doble.
   c. un parámetro nombrado minutos que será una referencia a un número entero.
   d. un parámetro nombrado `clave` que será una referencia a un carácter.
   e. un parámetro nombrado `rendimiento` que será una referencia a un número de precisión doble.
2. Se van a usar tres argumentos en número entero en una llamada a una función denominada `tiempo()`. Escriba un encabezado de función adecuado para `tiempo()`, suponiendo que `tiempo()` acepte estas variables como los parámetros de referencia `seg`, `min` y `horas`, y no devuelva ningún valor a la función que llama.
3. Vuelva a escribir la función `encontrarMax()` del programa 6.5 de modo que la variable `max`, declarada en `main()`, se use para almacenar el valor máximo de los dos números transmitidos. El valor de `max` deberá establecerse en forma directa desde dentro de `encontrarMax()`. (*Sugerencia:* `encontrarMax()` tendrá que aceptar una referencia a `max`.)

[13] Muchos compiladores tomarán esto como un error.

4. Escriba una función nombrada `cambio()` que tenga un parámetro en número entero y seis parámetros de referencia en número entero nombrados cien, `cincuenta`, `veinte`, `diez`, `cinco y uno`, respectivamente. La función tiene que considerar el valor entero transmitido como una cantidad en dólares y convertir el valor en el número menor de billetes equivalentes. Usando las referencias, la función deberá alterar de manera directa los argumentos respectivos en la función que llama.

5. Escriba una función nombrada `tiempo()` que tenga un parámetro en número entero llamado segundos y tres parámetros de referencia enteros nombrados `horas`, `min` y `seg`. La función es convertir el número de segundos transmitido en un número equivalente de horas, minutos y segundos. Usando las referencias la función deberá alterar de manera directa los argumentos respectivos en la función que llama.

6. Escriba una función nombrada `calc_años()` que tenga un parámetro entero que represente el número total de días desde la fecha 1/1/1900 y parámetros de referencia nombrados `año`, `mes` y `día.` La función es calcular el año, mes y día actual para el número dado de días que se le transmitan. Usando las referencias, la función deberá alterar en forma directa los argumentos respectivos en la función que llama. Para este problema suponga que cada año tiene 365 días y cada mes tiene 30 días.

7. El siguiente programa usa los mismos nombres de argumento y parámetro tanto en la función que llama como en la función llamada. Determine si esto causa algún problema para la computadora.

```
#include <iostream>
using namespace std;

void tiempo(int&, int&); // prototipo de la función

int main()
{
  int min, hora;

  cout << "Introduzca dos números :";
  cin >> min >> hora;
  tiempo(min, hora);

  return 0;
}

void tiempo(int& min, int& hora) // acepta dos referencias
{
  int seg;

  seg = (hora * 60 + min) * 60;
  ccout << "El número total de segundos es " << seg <<
  endl;

  return;
}
```

6.4 APLICACIONES

Preparar un programa de computadora bien diseñado es muy parecido a preparar un reporte semestral bien diseñado; ambos deben comenzar con un esquema, el cual puede ser escrito o, para programas muy pequeños, puede tan sólo mantenerse en la mente conforme se desarrolla el programa. Sin embargo, como sucede con un esquema para un reporte semestral, el cual enumera los temas principales del reporte, el esquema inicial de un programa de computadora debe proporcionar un listado de las tareas primarias que debe cumplir el programa.

En forma escrita, el esquema inicial de un programa de computadora por lo general es una descripción en seudocódigo (véase la sección 1.3) o un diagrama de estructura de primer nivel (véase la sección 1.2). Este esquema inicial comienza el proceso de definir un problema más complicado en un conjunto de tareas más pequeñas y más manejables. Cada una de estas tareas puede subdividirse o depurarse más, en tareas aún más pequeñas, si se requiere. Una vez que las tareas están bien definidas, el trabajo real de codificación puede comenzar, iniciando con cualquier tarea, en cualquier orden. Si hay más tareas de las que puedan ser manejadas por un programador, pueden distribuirse entre tantos programadores como se requiera. Esto es equivalente a tener a muchas personas trabajando en un proyecto de investigación grande, con cada persona responsable de un tema individual. Un esquema general aplicable a muchas tareas de ingeniería y científicas es el siguiente algoritmo:

Obtener las entradas del problema
Calcular el resultado deseado
Reportar los resultados del cálculo

Estas tres tareas son las responsabilidades primarias de todo programa, y se hará referencia a este algoritmo como el **algoritmo para resolver problemas.** Un diagrama de estructura de primer nivel de este algoritmo se muestra en la figura 6.11.

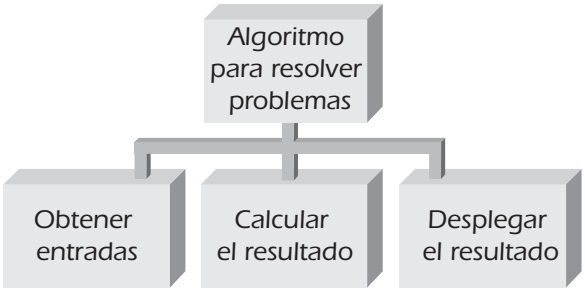


Figura 6.11 *Diagrama de estructura de primer nivel del algoritmo para resolver problemas.*

Cada tarea en el algoritmo para resolver problemas puede trabajarse en forma independiente como una función, una especie de "mini" programa en C++ que de manera típica es más fácil de completar que un programa entero. Cada una de estas tareas de función puede refinarse y codificarse en cualquier orden deseado, aunque completar primero la sección de entrada por lo general hace más fácil la prueba y el desarrollo. Ahora aplicaremos este procedimiento de desarrollo a un problema de programación real.

Aplicación 1: Conversión de coordenadas rectangulares a polares

Suponga que debemos escribir un programa en C++ para convertir las coordenadas rectangulares (x,y) de un punto en forma polar. Es decir, dada una posición x y y en un sistema de coordenadas cartesiano, como se ilustra en la figura 6.12, se debe calcular la distancia desde el origen, r, y el ángulo desde el eje x, θ, especificado por el punto. Los valores de r y θ se conocen como las *coordenadas polares* del punto.

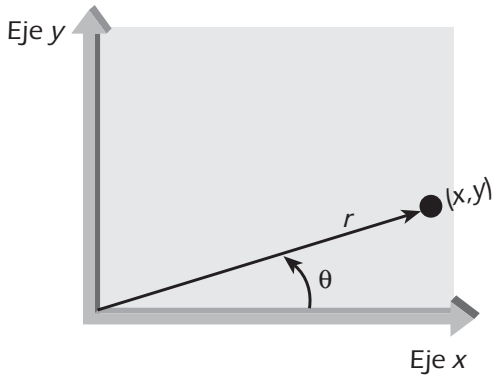


Figura 6.12 Correspondencia entre coordenadas polares (distancia y ángulo) y cartesianas (x,y).

Cuando se conocen las coordenadas x y y de un punto, las coordenadas r y θ equivalentes pueden calcularse usando las fórmulas:

$$r = \sqrt{x^2 + y^2}$$

$$\theta = \tan^{-1}(y / x) \quad x \neq 0$$

Se inicia el desarrollo del programa con un esquema de lo que éste ha de lograr. Puede construirse una descripción primaria en seudocódigo del programa deseado usando el algoritmo para resolver problemas en lo que se refiere a los detalles de esta aplicación. Las entradas requeridas son una coordenada x y una coordenada y, el cálculo es convertir los valores introducidos en su forma de coordenadas polares, y el despliegue son las coordenadas polares calculadas. Por tanto, la descripción de seudocódigo inicial es

Obtener los valores de las coordenadas x y y
Calcular los valores de las coordenadas polares (r y θ)
Desplegar los valores de las coordenadas polares

El diagrama de estructura de primer nivel o nivel superior equivalente para este algoritmo se ilustra en la figura 6.13.

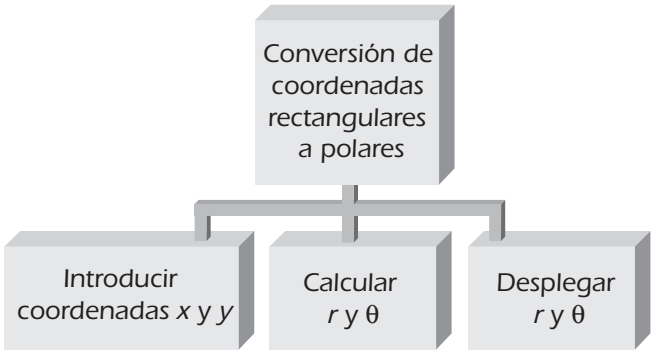


Figura 6.13 *Diagrama de estructura de nivel superior.*

Como éste es un programa relativamente simple y cada tarea descrita por el algoritmo está bien definida, se puede iniciar la codificación de cada tarea. Para ilustrar que cada tarea puede ser codificada de manera independiente de cualquier otra tarea, se comenzará de manera arbitraria con la codificación de la función que realiza el cálculo de las coordenadas polares. Como una característica adicional, esta función devolverá el ángulo θ en grados en lugar de la medida radianes devuelta por la función intrínseca `atan()`. En vista que esta función debe recibir dos entradas, las coordenadas *x* y *y*, y devolver dos salidas, las coordenadas *r* y θ, se le proporcionan a la función cuatro parámetros, dos para sus entradas y dos para sus salidas. Seleccionando de manera arbitraria los nombres de parámetros de `x`, `y`, `r`, y `theta`, y nombrando a la función `polar()`, el siguiente código lleva a cabo el cálculo requerido de coordenadas polares.

```
void polar(double x, double y, double& r, double& theta)
{
  const double A_GRADOS = 180.0/3.141593;

  r = sqrt(x * x + y * y);
  theta = atan(y/x) * A_GRADOS;

  return;
}
```

La función `polar()` es bastante sencilla. La línea de encabezado de la función declara que la función no devolverá en forma directa ningún valor y cada uno de sus parámetros es declarado como un tipo de datos en precisión doble. Los primeros dos parámetros se usarán para aceptar un valor *x* y uno *y*, mientras los últimos dos parámetros, los cuales son parámetros de referencia, se usarán para transmitir los valores de distancia y ángulo convertidos de vuelta a la función que llama. Dentro del cuerpo de la función se define una constante llamada `A_GRADOS` como el factor 180.0/3.142593. Las siguientes dos instrucciones de asignación usan los dos parámetros, `x` y `y`, para asignar valores a los parámetros *r* y `theta`. La constante nombrada `A_GRADOS` se usa para convertir el valor en radianes devuelto de la función `atan()` a grados. Como está escrita, la función `polar()` puede ser complicada para verificar cualquier error en tiempo de compilación.

Para entender cómo se transmiten los valores devueltos es útil pensar en los parámetros de referencia `r` y `theta` como recipientes (o variables) a través de los cuales pueden transmitirse valores en cualquier dirección. Esta situación se muestra en la figura 6.14, la

cual es útil para entender las características fundamentales de los parámetros de referencia: *tan sólo proporcionan la capacidad para que una función llamada y una que llama tengan acceso a la misma área de almacenamiento usando nombres diferentes.*

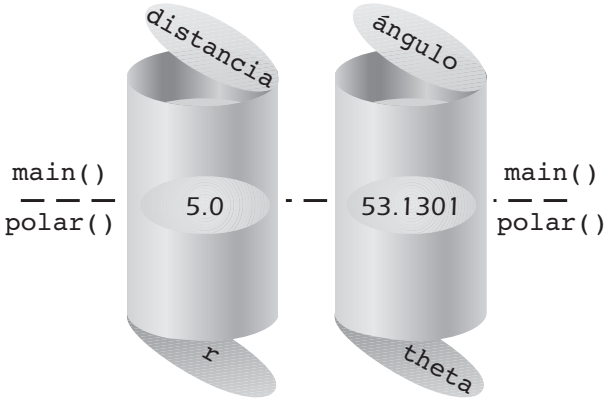
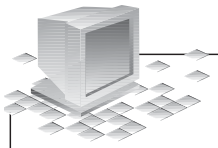


Figura 6.14 *Valores de los parámetros cuando se llama a* `polar()`.

Como se muestra en la figura 6.14, la función que llama puede tener acceso a los valores asignados a `r` y `theta` dentro de `polar()` usando los nombres de argumento `distancia` y `angulo`, o cualesquier otros nombres de argumentos seleccionados por el programador.

Probar la función

Una vez que se ha escrito `polar()`, puede probarse en forma independiente de cualquier otra función. Esto se hace escribiendo una función controladora dedicada que sólo llame a `polar()`, como se hace en el programa 6.11.



Programa 6.11

```
#include <iostream>
#include <cmath>
using namespace std;

void polar(double, double, double&, double&); // prototipo de la función

int main()
{
  double distancia, angulo;
  polar(3.0, 4.0, distancia, angulo);
  cout << "r = " << distancia << endl;
  cout << "ángulo = " << angulo << endl;

  return 0;
}
```

(Continúa)

(Continuación)

```
void polar(double x, double y, double& r, double& theta)
{
  const double A_GRADOS = 180.0/3.141593;

  r = sqrt(x * x + y * y);
  theta = atan(y/x) * A_GRADOS;

  return;
}
```

Hay que observar que en `main()` se transmitieron las constantes 3.0 y 4.0 a `polar()`. La función acepta estas entradas como los parámetros `x` y `y`, y usa estos parámetros para calcular valores para los parámetros r y theta. Dentro de `main()`, estos dos últimos parámetros se conocen como distancia y angulo, cuyos valores se despliegan inmediatamente después que se hace la llamada a `polar()`. La salida producida cuando se ejecuta el programa 6.11 es

```
r = 5
angulo = 53.1301
```

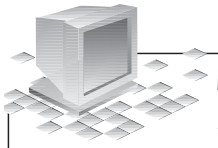
Éstos son los mismos resultados que se obtendrían por medio de un cálculo manual. Cómo la función lleva a cabo sólo dos cálculos, y el resultado desplegado por el programa de prueba concuerda por medio del obtenido con el cálculo manual, la función ha sido probada por completo por sí misma. Aún resta probarla en grupo con las dos funciones restantes requeridas por el programa completo para asegurar que los valores de argumentos correctos se intercambian entre cada función.

Completar el Programa

El diagrama de estructura para el programa completo (figura 6.13) requiere que también se escriban funciones para aceptar dos coordenadas rectangulares y desplegar las coordenadas polares calculadas, respectivamente. La siguiente función, `obtrec()`, puede utilizarse para aceptar los datos de entrada.

```
void obtrec(double& x, double& y)
{
  cout << "Programa de conversión de coordenadas"
       << " rectangulares a polares\n" << endl;
  cout << "Introduzca la coordenada x: ";
  cin  >> x;
  cout << "Introduzca la coordenada y: ";
  cin  >> y;

  return;
}
```

En esta función se usan los parámetros de referencia `x` y `y` para devolver los valores que se introducen en respuesta a los dos indicadores `cin`. Como con la función `polar()`, esta función puede probarse por sí misma usando un pequeño programa controlador dedicado. La función con su programa controlador se ilustra en el programa 6.12.



Programa 6.12

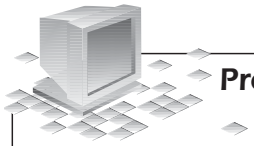
```
#include <iostream>
using namespace std;

void obtrec(double&, double&); // prototipo de la función

int main()
{
  double xcoord, ycoord;

  obtrec(xcoord, ycoord);
  cout << "El valor introducido para x es " << xcoord << endl;
  cout << "El valor introducido para y es " << ycoord << endl;

  return 0;
}


void obtrec(double& x, double& y)
{
  cout << "Programa de conversión de coordenadas"
       << " rectangulares a polares\n" << endl;
  cout << "Introduzca la coordenada x: ";
  cin  >> x;
  cout << "Introduzca la coordenada y: ";
  cin  >> y;

  return;
}
```

Observe que el programa controlador dedicado, el cual también se conoce como controlador de "aplicación para el usuario" o "de interfaz", se ha usado tanto para llamar a `obtrec()` como para desplegar los valores devueltos por esta función. La siguiente salida producida por el programa 6.12 verifica la operación correcta de la función `obtrec()`:

```
Programa de conversión de coordenadas rectangulares a
polares

Introduzca la coordenada x: 3
Introduzca la coordenada y: 4
```

```
El valor introducido para x es 3
El valor introducido para y es 4
```

De una manera similar, se construye la función para desplegar las coordenadas polares. El programa 6.13 contiene tanto la función, la cual se ha nombrado `mostrarlo()`, como un controlador de interfaz usado para probar la función. Hay que observar que los nombres de parámetros usados en la línea de encabezado para `mostrarlo()` no necesitan ser iguales que los usados en cualquier otra función. `mostrarlo()` se construye tan sólo para desplegar los valores en sus dos parámetros, los cuales en este caso se han nombrado `radio` y `angulo`.



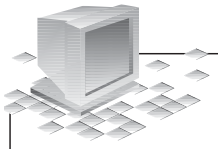
Programa 6.13

```
#include <iostream>
using namespace std;

void mostrarlo(double, double); // prototipo de la función

int main()
{
  mostrarlo(5.0, 53.1301);

  return 0;
}

void mostrarlo(double radio, double angulo)
{
  cout << "\nLas coordenadas polares son: " << endl;
  cout << " Distancia desde el origen: " << radio << endl;
  cout << " Angulo (en grados) respecto al eje x: " << angulo << endl;

  return;
}
```

La salida del programa 6.13, la cual se muestra a continuación, verifica que `mostrarlo()` despliega en forma correcta los valores que se le transmiten.

```
Las coordenadas polares son:
  Distancia desde el origen: 5
  Angulo (en grados) respecto al eje x: 53.1301
```

Ahora falta crear un programa `main()` que llame a cada una de las funciones desarrolladas en el orden correcto. Esto se hace en el programa 6.14, el cual también incluye las funciones `obtrec()`, `polar()` y `mostrarlo()`.



Programa 6.14

```
// Este programa convierte coordenadas rectangulares en coordenadas polares
// Funciones usadas: obtrec() - obtener las coordenadas rectangulares
//                 : polar() - calcular las coordenadas polares
//                 : mostrarlo() - desplegar las coordenadas polares
//
#include <iostream>
#include <cmath>
using namespace std;

void obtrec(double&, double&);                  // prototipo de la función
void polar(double, double, double&, double&);  // prototipo de la función
void mostrarlo(double, double);                 // prototipo de la función

int main()
{
  double x, y, distancia, angulo;

  obtrec(x, y);
  polar(x, y, distancia, angulo);
  mostrarlo(distancia, angulo);

  return 0;
}

void obtrec(double& x, double& y)
{
  cout << "Programa de conversión de coordenadas"
       << " rectangulares a polares\n" << end;
  cout << "Introduzca la coordenada x: ";
  cin  >> x;
  cout << "Introduzca la coordenada y: ";
  cin  >> y;

  return;
}
```

(Continúa)

(Continuación)

```
void polar(double x, double y, double& r, double& theta)
{
  const double A_GRADOS = 180.0/3.141593;

  r = sqrt(x * x + y * y);
  theta = atan(y/x) * A_GRADOS;

  return;
}

void mostrarlo(double radio, double angulo)
{
  cout << "\nLas coordenadas polares son: " << endl;
  cout << " Distancia desde el origen: " << radio << endl;
  cout << " Ángulo (en grados) respecto al eje x: " << angulo << endl;

  return;
}
```

La siguiente salida se produjo con una ejecución utilizando el programa 6.14:

```
Programa de conversión de coordenadas rectangulares
a polares

Introduzca la coordenada x: 3
Introduzca la coordenada y: 4

Las coordenadas polares son:
  Distancia desde el origen: 5
  Ángulo (en grados) respecto al eje x: 53.1301
```

Antes de dejar el programa 6.14, hay que señalar que una alternativa al escribir programas controladores para cada subrutina conforme se desarrollan las subrutinas es escribir primero un programa `main()` y agregar después las subrutinas conforme se desarrollan. Esto se logra utilizando cabos para cada función (véase la sección 6.1) y luego reemplazando cada cabo, uno a la vez, con la función completada.

Aplicación 2: Simulación

Hay muchos problemas de simulación científicos y de ingeniería en los que hay que considerar la probabilidad o deben utilizarse técnicas de muestreo estadístico. Por ejemplo, al simular el flujo de tráfico de automóviles o los patrones de uso de teléfono, se requieren modelos estadísticos. Además, aplicaciones como juegos de computadora simples y escenarios más vinculados con la ingeniería sólo pueden describirse en forma estadística. Todos estos modelos estadísticos requieren la **generación de números** aleatorios; es decir, una serie de números cuyo orden no pueda predecirse.

En la práctica, no hay números que en verdad sean aleatorios. Los dados nunca son perfectos; las cartas nunca se barajan en forma completamente aleatoria; los movimientos de las moléculas que se suponen aleatorios son influidos por el ambiente; y las computado-

ras digitales sólo pueden manejar números dentro de un rango finito y con precisión limitada. Lo mejor que puede hacerse es generar números **seudoaleatorios,** los cuales son suficientemente aleatorios para la tarea a realizar.

Algunos lenguajes de computadora contienen una función de biblioteca que produce números aleatorios; otros no. Las funciones proporcionadas por C++ se llama `rand()`, la cual genera números aleatorios, y `srand()`, la cual establece valores "semilla" aleatorios iniciales. Se presentarán estas dos funciones y luego se usarán en una aplicación que simula el lanzamiento de una moneda para determinar el número de caras y cruces resultante.

Generación de números seudoaleatorios

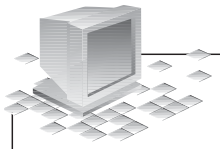
Los compiladores de C++ proporcionan dos funciones para crear números aleatorios: `rand()` y `srand()`. La función `rand()` produce una serie de números aleatorios en el rango $0 \leq$ `rand()` $\leq$ `RAND_MAX`, donde la constante `RAND_MAX` se define en el archivo de encabezado `cmath`. La función `srand()` proporciona un valor "semilla" inicial para `rand()`. Si no se usa `srand()` o alguna otra técnica de "sembrado" equivalente, `rand()` producirá siempre la misma serie de números aleatorios.

El procedimiento general para crear una serie de *n* números aleatorios usando funciones de biblioteca de C++ se ilustra con el siguiente código:

```
srand(time(NULL)); // esto genera el primer valor "semilla"

for (int i = 1; i <= N; i++) // esto genera N números aleatorios
{
  rvalor = rand();
  cout << rvalor << endl;
}
```

Aquí, el argumento para la función `srand()` es una llamada a la función `time()` con un argumento `NULL`. Con este argumento la función `time()` lee el tiempo del reloj interno de la computadora, en segundos. Entonces la función `srand()` usa este tiempo, convertido a un `int` sin signo para inicializar la función generadora de números aleatorios `rand()`.[14] El programa 6.15 usa este código para generar una serie de 10 números aleatorios.

[14]De manera alternativa, muchos compiladores de C++ tienen una rutina `randomize()` que se define usando la función `srand()`. Si esta rutina está disponible, la llamada `randomize()` puede usarse en lugar de la llamada `srand(time(NULL))`. En cualquier caso, la rutina "semilla" de inicialización es llamada una sola vez, después de lo cual se usa la función `rand()` para generar una serie de números.



Programa 6.15

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <ctime>
using namespace std;

// este programa genera diez números seudoaleatorios
// usando la funcion rand() de C++

int main()
{
  const int NUMEROS = 10;

  double valor_azar;
  int i;

  srand(time(NULL));
  for (i = 1; i <= NUMEROS; i++)
  {
    valor_azar = rand();
    cout << valor_azar << endl;
  }

  return 0;
}
```

La siguiente es la salida producida por una ejecución del programa 6.15:

```
20203
21400
15265
26935
 8369
10907
31299
15400
 5074
20663
```

Debido a la llamada a la función `srand()` en el programa 6.15, la serie de diez números aleatorios diferirá cada vez que se ejecute el programa. Sin el efecto de "sembrado" aleatorizador de esta función, siempre se produciría la misma serie de números aleatorios.

Nótese también la inclusión de los archivos de encabezado `cmath` y `ctime`. El archivo cmath contiene los prototipos de función para las funciones `srand()` y `rand()`, mientras el archivo de encabezado `ctime` contiene el prototipo de función para la función `time()`.

Escalamiento

En la práctica por lo general debe hacerse una modificación al número aleatorio producido por la función `rand()`. En la mayor parte de las aplicaciones, se requiere que los números aleatorios sean valores de punto flotante dentro del rango 0.0 a 1.0 o que sean números enteros dentro de un rango especificado, como 1 a 100. El método para ajustar los números aleatorios producidos por un generador de números aleatorios para que residan dentro de dichos rangos se llama **escalamiento.**

El escalamiento de números aleatorios para que residan dentro del rango 0.0 a 1.0 se logra con facilidad dividiendo el valor devuelto de `rand()` entre `RAND_MAX`. Por tanto, la expresión `double (rand())/RAND_MAX` produce un número aleatorio en precisión doble entre 0.0 y 1.0.

El escalamiento de un número aleatorio como un valor entero entre 0 y *N*-1 se logra usando cualquiera de las expresiones `rand() % N` o `int(double(rand())/RAND_MAX * N)`. Por ejemplo, la expresión `int(double(rand())/RAND_MAX * 100)` produce un número entero aleatorio entre 0 y 99.

Para producir un número entero aleatorio entre 1 y *N* puede usarse la expresión `1 + rand() % N`. Por ejemplo, al simular el lanzamiento de un dado, la expresión `1 + rand() % 6` produce un número entero aleatorio entre 1 y 6. Puede usarse la expresión de escalamiento más general `a + rand() % (b + 1 – a)` para producir un número entero aleatorio entre los números *a* y *b*.

Un uso común de los números aleatorios es simular eventos usando un programa, en lugar de dedicar tiempo y costo a construir un experimento real. Por ejemplo, la teoría estadística indica que la probabilidad de obtener cara en un solo lanzamiento de una moneda es $\frac{1}{2}$. Del mismo modo, hay una probabilidad de 50% de obtener cruz en un solo lanzamiento de una moneda.

Usando estas probabilidades se esperaría que una sola moneda que es lanzada 1000 veces produzca 500 caras y 500 cruces. En la práctica, sin embargo, esto no se realiza nunca con exactitud en un solo experimento consistente de 1000 lanzamientos. En lugar de lanzar en realidad una moneda 1000 veces, podemos usar un generador de números aleatorios para simular esos lanzamientos.

Paso 1 *Analizar el problema*

Para este problema se requieren dos salidas: el porcentaje de caras y el porcentaje de cruces que resultan cuando se simula que una moneda es lanzada 1000 veces. No se requerirá ningún elemento de entrada para la función del generador de números aleatorios.

Paso 2 Desarrollar una solución

El porcentaje de caras y cruces se determina como

$$\text{percentaje de caras} = \frac{\text{número de caras}}{1000} \quad 100\%$$

$$\text{percentaje of cruces} = \frac{\text{número de cruces}}{1000} \quad 100\%$$

Para determinar el número de caras y cruces, se tienen que simular 1000 números aleatorios de tal manera que se pueda definir un resultado de "caras" o "cruces" de cada número generado. Hay diversas formas de hacer esto.

Una forma es usar la función `rand()` para generar números enteros entre 0 y `RAND_MAX`. Sabiendo que cualquier lanzamiento individual tiene una probabilidad de 50% de ser cara o cruz, podría designarse una "cara" como un número aleatorio par y una "cruz" como un número aleatorio non. Otro método sería escalar el valor devuelto de `rand()` para que resida entre 0.0 y 1.0 como se describió antes. Entonces se podría definir una "cara" como cualquier número mayor que 0.5 y cualquier otro resultado como una "cruz". Éste es el algoritmo que se adoptará.

Habiendo definido cómo se creará un solo lanzamiento que tiene una probabilidad de 50% de producir caras o cruces, la generación de 1000 lanzamientos es bastante simple: se usa un ciclo de cuenta fija que genera 1000 números aleatorios. Para cada generación se identifica el resultado como cara o cruz y se acumulan los resultados en un contador de caras y cruces. Por tanto, el algoritmo de simulación completo está dado por el seudocódigo

```
Inicializar un contador de caras en cero
Inicializar un contador de cruces en cero
For 1000 veces
  generar un número aleatorio entre 0 y 1
  If el número aleatorio es mayor que 0.5
    considérelo como cara y
    sume uno a la cuenta de caras
  Else
    considérelo cruz y
    sume uno a la cuenta de cruces
  Endif
Endfor
Calcular el porcentaje de caras como el número de caras dividido entre 1000 × 100%
Calcular el porcentaje de cruces como el número de cruces dividido entre 1000 × 100%
Imprimir el porcentaje de caras y cruces obtenido
```

Paso 3 Codificar la solución

El programa 6.16 codifica este algoritmo en C++.

Programa 6.16

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <ctime>
using namespace std;

// un programa para simular el lanzamiento de una moneda NUMLANZ veces
int main()
{
  const int NUMLANZ = 1000;

  int caras = 0; // inicializa el contador de caras
  int cruces = 0; // inicializa el contador de cruces
  int i;
  double volado, porcaras, porcruces;

    // simular los lanzamientos de NUMLANZ de una moneda
  srand(time(NULL));
  for (i = 1; i <= NUMLANZ; i++)
  {
    volado = double(rand())/RAND_MAX; // escala el número entre 0 y 1
    if (volado > 0.5)
      cabezas = cabezas + 1;
    else
      cruces = cruces + 1;
  }
  porcaras = (caras / double (NUMLANZ)) * 100.0; // calcula el porcentaje de caras
  porcruces = (cruces / double (NUMLANZ)) * 100.0; // calcula el porcentaje de cruces
  cout << "\nCaras salio " << porcaras << " por ciento de las veces";
  cout << "\nCruces salio " << porcruces << " por ciento de las veces";

  return 0;
}
```

A continuación hay dos muestras de ejecuciones usando el programa 6.16.

```
Caras salio 50.9 por ciento de las veces
Cruces salio 49.1 por ciento de las veces
```

y

```
Caras salio 49.7 por ciento de las veces
Cruces salio 50.3 por ciento de las veces
```

Por supuesto es más fácil escribir y ejecutar el programa 6.16 que lanzar una moneda 1000 veces en forma manual. Debe señalarse que la validez de los resultados producidos por el programa depende de cuán aleatorios son en realidad los números producidos por `rand()`.

Paso 4 Probar y depurar el programa

En realidad hay dos pruebas que el programa 6.16 debe pasar. La prueba más importante se refiere a la aleatoriedad de cada número generado. Ésta, por supuesto, es en realidad una prueba de la función de números aleatorios. Para nuestros propósitos, se ha usado una función escrita con anterioridad suministrada por el compilador. Así que en este punto aceptamos la "aleatoriedad" del generador. (Véase el ejercicio 13 para un método de verificación de la aleatoriedad de la función.)

Una vez que se ha resuelto la cuestión del generador de números aleatorios, la segunda prueba requiere que se generen en forma correcta 1000 números y acumular una cuenta de caras y cruces. Que esto se logra en forma correcta se verifica de manera adecuada con una simple verificación de escritorio del ciclo `for` dentro del programa 6.16. Además, sabemos que el resultado de la simulación debe estar cerca del 50% de caras y 50% de cruces. Los resultados de la simulación verifican que esto fue así.

Ejercicios 6.4

1. El volumen, v, y el área superficial, s, de un cilindro están dados por las fórmulas $v = \pi r^2 l$ y $s = 2\pi rl$, donde r es el radio del cilindro y l es su largo. Usando estas fórmulas, escriba y pruebe una función nombrada `cilindro()` que acepte el radio y el largo de un cilindro y devuelva su volumen y su área superficial.

2. Escriba y pruebe una función en C++ que calcule el radio, r, y el área, a, de un círculo cuando se da su circunferencia, c. Las fórmulas relevantes son $r = c/(2\pi)$ y $a = \pi r^2$. Pruebe su función usando un programa que tenga una función controladora dedicada.

3. Un líquido puede fluir por un tubo con un flujo en un patrón suave conocido como flujo *laminar* o en un patrón violento conocido como flujo *turbulento*. Las velocidades que produce cada tipo de flujo dentro del tubo pueden determinarse usando las fórmulas:

   $v_{lam} = 2100\mu / \rho\, d$ and $v_{tur} = 4000\mu / \rho\, d$,

   donde

   v_{lan} es la velocidad del líquido, en pies/s, que produce un flujo laminar determinado

   v_{tur} es la velocidad del líquido, en pies/s, que produce un flujo turbulento determinado

   μ es la viscosidad del líquido, en libras-pie por s/pie^2

   ρ es la densidad del líquido, en slug/pies^3

   d es el diámetro interior del tubo, en pies

   Usando estas fórmulas, escriba y pruebe una función en C++ llamada `flujo()` que devuelva tanto la velocidad de flujo laminar, v_{lan}, y la velocidad de flujo turbulento, v_{tur}, usando parámetros de referencia. La función deberá calcular estas velocidades para el agua, la cual tiene una viscosidad, μ, de 1.9×10^5 libras-pie por s/pie^2 y una densidad, ρ, de 1.94 slug/pies^3. El diámetro del tubo deberá ser transmitido por valor en la función `flujo()`.

4. La viscosidad y densidades de tres líquidos comunes se enumeran a continuación:

	Viscosidad (libras-pies s/pies^2)	**Densidad (slug/pies^3)**
Alcohol etílico	2.29×10^5	1.527
Alcohol metílico	1.17×10^5	1.531
Alcohol propílico	4.01×10^5	1.556

Usando estos datos, escriba y pruebe una función en C++ llamada `viscDen()` que devuelva la viscosidad y la densidad del líquido seleccionado usando parámetros de referencia. El tipo de líquido deberá introducirse a la función como un carácter que es transmitido por valor.

5. Escriba un programa en C++ que acepte las coordenadas rectangulares de dos puntos (x_1, y_1) y (x_2, y_2), calcule la distancia de cada punto a partir del origen y la distancia entre los dos puntos. La distancia entre dos puntos, d, está dada por la fórmula

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

6. Modifique el programa 6.16 de modo que solicite el número de lanzamientos del usuario. (*Sugerencia:* Asegúrese de hacer que el programa determine en forma correcta los porcentajes de caras y cruces obtenidas.)

7. Se han desarrollado muchos algoritmos para generar números seudoaleatorios. Algunos de estos algoritmos utilizan un esquema de conteo, como contar bits comenzando en alguna ubicación arbitraria en una memoria cambiante. Otro esquema, el cual crea números seudoaleatorios realizando un cálculo, es el método del residuo de potencias.

El **método del residuo de potencias** comienza con un número entero non de n dígitos, el cual se conoce como en número "semilla". La semilla se multiplica por el valor $(10^{n/2} - 3)$. Usar los n dígitos inferiores del resultado (el "residuo") produce una nueva semilla. Continuar este procedimiento produce una serie de números aleatorios, con cada nuevo número usado como la semilla para el siguiente número. Si la semilla original tiene cuatro o más dígitos (n igual a o mayor que 4) y no es divisible entre dos ni cinco, este procedimiento produce $5 \times 10^{(n-2)}$ números aleatorios antes que una secuencia de números se repita. Por ejemplo, si se comienza con una semilla de seis dígitos ($n = 6$), como 654321, puede generarse una serie de $5 \times 10^4 = 50\,000$ números aleatorios.

Como un algoritmo, los pasos específicos en la generación de números seudoaleatorios usando un procedimiento de residuo de potencias son los siguientes:

Paso 1: Hacer que un usuario introduzca una semilla en número entero de seis dígitos que no sea divisible entre 2 ni 5; esto significa que el número deberá ser un número non que no termine en 5.
Paso 2: Multiplicar el número semilla por 997, lo cual es $10^3 - 3$.
Paso 3: Extraer los seis dígitos inferiores del resultado producido por el paso 2. Use este número aleatorio como la siguiente semilla.
Paso 4: Repita los pasos 2 y 3 para tantos números aleatorios como sean necesarios.

Por tanto, si el número semilla introducido por el usuario es 654321 (paso 1), el primer número aleatorio generado se calcula como sigue:

Paso 2: 654321 * 997 = 652358037
Paso 3: Extraer los seis dígitos inferiores del número obtenido en el paso 2. Esto se logra usando un "truco" de programación estándar.

El truco implica:

Paso 3a: Dividir el número entre 10^6 = 1000000.
Por ejemplo, 652358037 / 1000000 = 652.358037
Paso 3b: Tomar la parte entera del resultado del paso 3a.
Por ejemplo, la parte entera de 652.358037 = 652
Paso 3c: Multiplicar el resultado anterior por 10^6
Por ejemplo, $652 \times 10^6 = 652000000$
Paso 3d: Restar este resultado del número original.
Por ejemplo, 652358037 – 652000000 = 358037

La parte entera de un número en punto flotante puede tomarse asignando el número en punto flotante a una variable en número entero, o mediante un molde de C++ (véase la sección 3.3). En nuestro procedimiento se utilizará el mecanismo de molde. Por tanto, el algoritmo para producir un número aleatorio puede lograrse usando el siguiente código:

```
i = int(997.0 * x / 1.e6);    // tomar la parte entera
x = 997.0 * x - i * 1.e6;
```

Usando esta información,

a. Cree una función llamada `numalea()` que acepte una "semilla" en punto flotante como un parámetro y devuelva un número aleatorio en punto flotante entre 0 y 1.e6.
b. Incorpore la función `numalea()` creada en el ejercicio 7a en un programa en C++ que produzca diez números aleatorios entre 0 y 1.e6.
c. Pruebe la aleatoriedad de la función `numalea()` creada en el ejercicio 7a usando el método descrito en el ejercicio 13. Inténtelo con algunos valores semilla pares y algunos valores semilla nones que terminen en 5 para determinar si esto afecta la aleatoriedad de los números.

8. Escriba una función en C++ que determine en cuál cuadrante reside una línea trazada desde el origen. La determinación del cuadrante se hace usando el ángulo que forma la línea con el eje *x* positivo como sigue:

Ángulo desde el eje *x* positivo	Cuadrante
Entre 0 y 90 grados	1
Entre 90 y 180 grados	2
Entre 180 y 270 grados	3
Entre 270 y 360 grados	4

NOTA: Si el ángulo tiene exactamente 0, 90, 180 o 270 grados la línea correspondiente no reside en ningún cuadrante sino que se encuentra en un eje. Para este caso su función deberá devolver un cero.

9. Escriba un programa para simular el lanzamiento de dos dados. Si el total de los dos dados es 7 u 11 usted gana; de lo contrario pierde. Adorne este programa tanto como quiera, con apuestas, posibilidades diferentes, combinaciones diferentes para ganar o perder, dejar de jugar cuando no le quede dinero o alcance el límite de la casa, desplegar los dados, etc. (*Sugerencia:* Calcule los puntos que se muestran en cada dado con la expresión `puntos = (int)(6.0 * numero aleatorio + 1)`, donde el número aleatorio esté entre 0 y 1.)

10. Un valor que a veces es útil es el máximo común divisor de dos números enteros *n*1 y *n*2. Hace más de dos mil años un matemático famoso, Euclides, descubrió un método eficiente para hacer esto. Sin embargo, por ahora nos conformaremos con un cabo. Escriba la función en número entero `stub gcd(n1, n2)`. Tan sólo haga que devuelva un valor que sugiera que recibió sus parámetros en forma correcta. (*Sugerencia: n*1 + *n*2 es una buena elección de valores de devolución. ¿Por qué *n*1/*n*2 no es una buena elección?)

11. El método de Euclides para encontrar el máximo común divisor (GCD, por sus siglas en inglés) de dos números enteros positivos consiste en los siguientes pasos:

a. Divida el número mayor entre el menor y conserve el residuo.
b. Divida el número menor entre el residuo, conservando de nuevo el residuo.
c. Continúe dividiendo el residuo anterior entre el residuo actual hasta que el residuo sea cero, punto en el cual el último residuo diferente de cero es el máximo común divisor.

Por ejemplo, suponga que los dos números enteros positivos son 84 y 49, tenemos:

Paso a: 84/49 produce un residuo de 35
Paso b: 49/35 produce un residuo de 14
Paso c: 35/14 produce un residuo de 7
Paso d: 14/7 produce un residuo de 0

Por tanto, el último residuo diferente de cero, el cual es 7, es el máximo común divisor de 84 y 49.

Utilizando el algoritmo de Euclides, reemplace la función cabo escrita para el ejercicio 10 con una función real que determine y devuelva el GCD de sus dos parámetros en número entero.

12. El siguiente programa usa los mismos nombres de variables tanto en la función que llama como en la función llamada. Determine si esto causa algún problema para el compilador.

```
#include <iostream.h>

int tiempo(int, int); // prototipo de la función

int main()
```

```
{
  int min, hora, seg;

  cout << "Introduzca dos números: ";
  cin >> min, hora;
  seg = tiempo(min, hora);
  cout << "El número total de segundos es " << seg <<
endl;

  return 0;
}

int tiempo(int min, int hora)
{
  int seg;

  seg = (hora * 60 + min) * 60;
  return seg;
}
```

13. Escriba un programa que pruebe la efectividad de la función de biblioteca `rand()`. Empiece inicializando en cero 10 contadores, como `cuentacero`, `cuentauno`, `cuentados`,..., `cuentanueve`. Luego genere un número grande de números enteros seudoaleatorios entre 0 y 9. Cada vez que ocurra un 0 incremente `cuentacero`, cuando ocurra un 1 incremente `cuentauno`, etc. Por último, imprima el número de 0, 1, 2, etc., que ocurrieron y el porcentaje de veces que ocurrieron.

14. El determinante de la matriz de 2 por 2

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}$$

es $a_{11}a_{22} - a_{21}a_{12}$.

Del mismo modo, el determinante de una matriz de 3 por 3

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} =$$

$$a_{11}\begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{21}\begin{vmatrix} a_{12} & a_{13} \\ a_{32} & a_{33} \end{vmatrix} + a_{31}\begin{vmatrix} a_{12} & a_{13} \\ a_{22} & a_{23} \end{vmatrix}$$

Utilizando esta información, escriba y pruebe dos funciones, llamadas `det2()` y `det3()`. La función `det2()` deberá aceptar los cuatro coeficientes de una matriz de 2 por 2 y devolver su determinante. La función `det3()` deberá aceptar los nueve coeficientes de una matriz de 3 por 3 y devolver su determinante llamando a `det2()` para calcular los determinantes 2 por 2 requeridos.

6.5 ALCANCE DE UNA VARIABLE

Ahora que hemos comenzado a escribir programas que contienen más de una función, se pueden observar con más detenimiento las variables declaradas dentro de cada función y su relación con las variables en otras funciones.

Por su misma naturaleza, las funciones en C++ están construidas como módulos independientes. Como se ha visto, los valores se transmiten a la función usando la lista de parámetros de la función y se devuelve un valor usando una instrucción de devolución. Visto desde esta perspectiva, una función puede considerarse como una caja cerrada, con ranuras en la parte superior para recibir valores y una sola ranura en la parte inferior de la caja para devolver un valor (véase la figura 6.15).

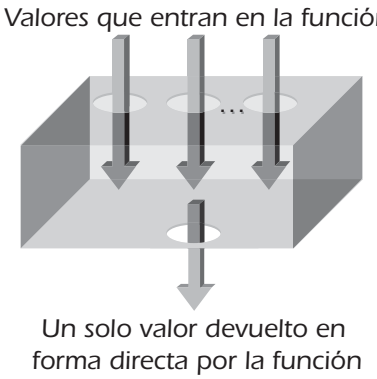


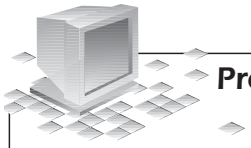
Figura 6.15 *Una función puede considerarse una caja cerrada.*

La metáfora de una caja cerrada es útil porque enfatiza el hecho que todo lo que va dentro de la función, incluyendo todas las declaraciones de variables dentro del cuerpo de la función, está oculto de la vista de todas las demás funciones. Dado que las variables creadas dentro de una función están disponibles de manera convencional sólo para la función en sí, se dice que son locales para la función o ***variables locales***. Este término se refiere al alcance de un identificador, donde ***alcance*** se define como la sección del programa donde el identificador, como una variable, es válido o "conocido". Se dice que esta sección del programa es donde la variable es visible. Una variable puede tener ***alcance*** local o global. Una variable con un ***alcance local*** es aquella a la que una instrucción de declaración hecha dentro del cuerpo de una función le ha designado ubicaciones de almacenamiento. Las variables locales sólo son significativas cuando se usan en expresiones o instrucciones dentro de la función que las declaró. Esto significa que el mismo nombre de variable puede declararse y usarse en más de una función. Para cada función que declara la variable, se crea una variable separada y distinta.

Todas las variables que se han utilizado hasta ahora han sido variables locales. Éste es un resultado directo de colocar las instrucciones de declaración dentro de las funciones y usarlas como instrucciones de definición que causan que la computadora reserve almacenamiento para la variable declarada. Como se verá, las instrucciones de declaración pueden colocarse fuera de las funciones y no necesitan actuar como definiciones para que se reserven nuevas áreas de almacenamiento para la variable declarada.

Una variable con ***alcance global,*** por lo general denominada ***variable global,*** el almacenamiento se crea mediante una instrucción de declaración localizada fuera de cualquier

función. Estas variables pueden ser utilizadas por todas las funciones que se colocan físicamente después de la declaración de la variable global. Esto se muestra en el programa 6.17, donde se utiliza a propósito el mismo nombre de variable dentro de ambas funciones contenidas en el programa.



Programa 6.17

```
#include <iostream>
using namespace std;

int primernum; // crea una variable global llamada primernum

void valfun(); // prototipo de la función (declaración)

int main()
{
  int segundonum;          // crea una variable local llamada segundonum

  primernum = 10; // almacena un valor en la variable global
  segundonum = 20; // almacena un valor en la variable local

  cout << "De main(): primernum = " << primernum << endl;
  cout << "De main(): segundonum = " << segundonum << endl;

  valfun(); // llama a la función valfun

  cout << "\nDe main() de nuevo: primernum = " << primernum << endl;
  cout << "De main() de nuevo: segundonum = " << segundonum << endl;

  return 0;
}

void valfun() // no se transmiten valores a esta función
{
  int segundonum; // crea una segunda variable local llamada segundonum

  segundonum = 30; // esto sólo afecta al valor de esta variable local

  cout << "\nDe valfun(): primernum = " << primernum << endl;
  cout << "De valfun(): segundonum = " << segundonum << endl;

  primernum = 40; // esto cambia primernum para ambas funciones

  return;
}
```

La variable `primernum` en el programa 6.17 es una variable global debido a que su almacenamiento fue creado por una instrucción de definición localizada fuera de una función. En vista que ambas funciones, `main()` y `valfun()`, siguen a la definición de `primernum`, ambas funciones pueden usar esta variable global sin que necesiten ninguna otra declaración.

El programa 6.17 también contiene dos variables locales separadas, ambas nombradas `segundonum`. El almacenamiento para la variable `segundonum` nombrada en `main()` es creado por la instrucción de definición localizada en `main()`. Un área de almacenamiento diferente para la variable `segundonum` en `valfun()` es creada por la instrucción de definición localizada en la función `valfun()`. La figura 6.16 ilustra las tres áreas de almacenamiento distintas reservadas por las tres instrucciones de definición que se encuentran en el programa 6.17.

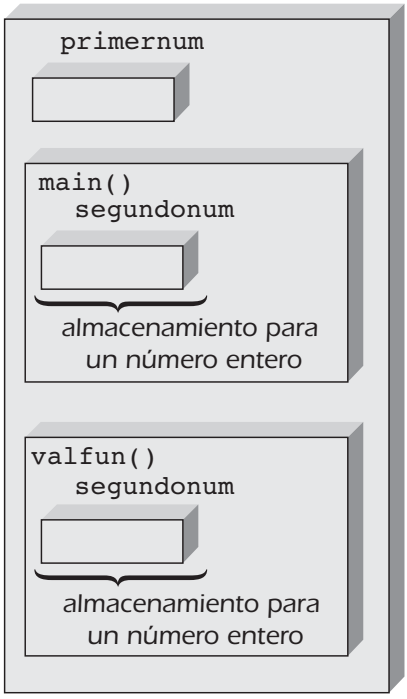


Figura 6.16 *Las tres áreas de almacenamiento creadas por el programa 6.17.*

Cada una de las variables nombradas `segundonum` es local para la función en la que se crea su almacenamiento, y cada una de estas variables sólo puede usarse desde dentro de la función apropiada. Por tanto, cuando se usa `segundonum` en `main()`, se tiene acceso al área de almacenamiento reservada por `main()` para su variable segundonum, y cuando se usa `segundonum` en `valfun()`, se tiene acceso al área de almacenamiento reservada por `valfun()` para su variable `segundonum`. La siguiente salida se produce cuando se ejecuta el programa 6.17:

```
De main(): primernum = 10
De main(): segundonum = 20

De valfun(): primernum = 10
De valfun(): segundonum = 30
```

```
De main() de nuevo: primernum = 40
De main() de nuevo: segundonum = 20
```

Vamos a analizar esta salida. En vista que `primernum` es una variable global, las funciones `main()` y `valfun()` pueden usar y cambiar su valor. Al principio, ambas funciones imprimen el valor de 10 que almacenó `main()` en `primernum`. Antes de devolver, `valfun()` cambia el valor de primernum a 40, que es el valor que corresponde cuando a continuación la variable `primernum` se despliega desde dentro de `main()`.

Debido a que cada función sólo "conoce" sus propias variables locales, `main()` sólo puede enviar el valor de su `segundonum` al objeto `cout`. Por tanto, siempre que se obtiene `segundonum` desde `main()` se despliega el valor 20, y siempre que se obtiene `segundonum` desde `valfun()` se despliega el valor 30.

C++ no confunde las dos variables `segundonum` porque sólo puede ejecutarse una función en un momento determinado. Mientras se ejecuta una función, sólo se puede tener acceso a aquellas variables y parámetros que están "al alcance" de esa función (global y local).

El alcance de una variable de ninguna manera influye o restringe el tipo de datos de la variable. Del mismo modo en que una variable local puede ser un carácter, número entero, booleano, de precisión doble o cualquiera de los otros tipos de datos (largo/corto) que se han introducido, así también pueden ser las variables globales de estos tipos de datos, como se ilustra en la figura 6.17. El alcance de una variable es determinado por la colocación de la instrucción de definición que reserva almacenamiento para ella y de manera opcional por una instrucción de declaración que la hace visible, mientras el tipo de datos de la variable se determina usando la palabra clave apropiada (`char`, `int`, `bool`, `double`, etc.) antes del nombre de la variable en una instrucción de declaración.

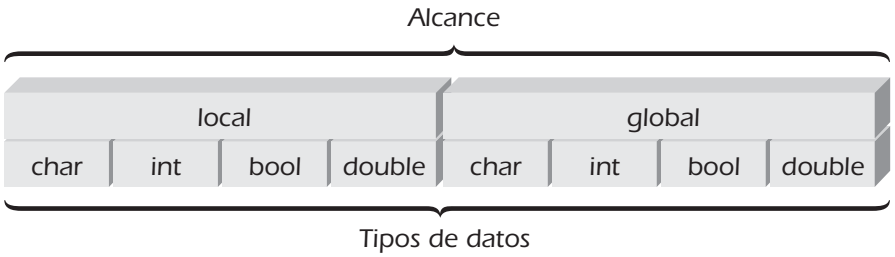
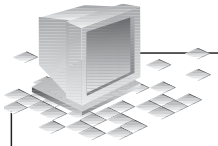


Figura 6.17 *Relación del alcance y el tipo de una variable.*

Operador de resolución de alcance

Cuando una variable local tiene el mismo nombre que una variable global, todas las referencias al nombre de la variable hechas dentro del alcance de la variable local se refieren a esta variable local. Esta situación se ilustra en el programa 6.18, donde se define el nombre de la variable `numero` como una variable global y como una local.



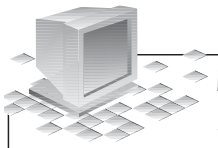
Programa 6.18

```
#include <iostream>
using namespace std;

double numero = 42.8;     // una variable global llamada número

int main()
{
  double numero = 26.4;     // una variable local llamada número

  cout << "El valor de número es " << número << endl;

  return 0;
}
```

Cuando se ejecuta el programa 6.18, se despliega la siguiente salida.

```
El valor de número es 26.4
```

Como lo muestra esta salida, el nombre de la variable local toma precedencia sobre la variable local. En tales casos, aún se puede tener acceso a la variable global utilizando el operador de resolución de alcance de C++. Este operador, el cual tiene el símbolo **::**, debe colocarse inmediatamente antes del nombre de la variable, como en **::** número. Cuando se usa de esta manera, **::** le indica al compilador que use la variable global. Como un ejemplo, se usa el operador de resolución de alcance en el programa 6.18a.



Program 6.18a

```
#include <iostream>
using namespace std;

double numero = 42.5;       // una variable global nombrada número

int main()
{
  double numero = 26.4;       // una variable local nombrada número

  cout << "El valor de número es " << ::número << endl;

  return 0;
}
```

La salida producida por el programa 6.18a es:

```
El valor de número es 42.5
```

Como lo indica esta salida, el operador de resolución de alcance causa que se tenga acceso a la variable global en lugar de a la local.

Mal uso de las variables globales

Las variables globales permiten al programador "brincarse" las salvaguardas normales proporcionadas por las funciones. En lugar de transmitir variables a una función, es posible hacer que todas las variables sean globales. **No haga esto.** Al hacer todas las variables globales de manera indiscriminada, al instante destruye las salvaguardas que C++ proporciona para hacer que las funciones sean independientes y aisladas entre sí, incluyendo la necesidad de designar con cuidado el tipo de argumentos que necesita una función, las variables usadas en la función y el valor devuelto.

Usar sólo variables globales puede ser desastroso, en especial en programas más grandes que tienen muchas funciones creadas por el usuario. En vista que todas las variables en una función deben ser declardas, crear funciones que utilicen variables globales requiere recordar escribir las declaraciones globales apropiadas al principio de cada programa que use la función; ya no se encuentran junto con la función. Sin embargo, más devastador que esto es tratar de rastrear un error en un programa grande que utiliza variables globales. Dado que cualquier función que siga a la declaración global puede tener acceso a una variable global y cambiarla, localizar el origen de un valor erróneo es una tarea frustrante y que consume tiempo.

Sin embargo, las definiciones globales, en ocasiones son útiles para crear variables y constantes que deben ser compartidas entre muchas funciones. En lugar de transmitir la misma variable a cada función, es más fácil definir la variable una vez como global. Hacer esto alerta a cualquiera que lea el programa respecto a que muchas funciones usan esta variable. La mayor parte de los programas grandes casi siempre usan unas cuantas variables o constantes globales. Sin embargo, los programas más pequeños que contienen sólo algunas funciones casi nunca deben contener variables globales.

El mal uso de las variables globales no se aplica a los prototipos de funciones, los cuales por lo general son globales. Debe señalarse que todos los prototipos de función que hasta ahora se han usado han sido de alcance global, declaran el prototipo para todas las funciones subsiguientes. Colocar un prototipo de función dentro de una función hace del prototipo una declaración local disponible sólo para la función que se declara adentro.

Ejercicios 6.5

1. a. Para la siguiente sección de código, determine el tipo de datos y el alcance de todas las funciones declaradas. Para hacer esto, utilice una hoja de papel separada y anote los tres encabezados de columna que siguen (se han llenado las entradas para la primera variable):

Nombre del a variable	Tipo de datos	Alcance
`voltios`	`int`	`global para main, roi y paso`

```
#include <iostream>
using namespace std;

int voltios;
long int resistencia;
double corriente;

int main()
{
  int potencia;
  double factor, tiempo;
     .
     .
  return 0;
}

double roi(int mat1, int mat2)
{
  int cuenta;
  double peso;
       .
       .
  return peso;
}

int paso(double primero, double ultimo)
{
  int horas;
  double partefrac;
       .
       .
  return 10*horas;
}
```

b. Dibuje cuadros alrededor de la sección apropiada del código anterior para encerrar el alcance de cada variable.

c. Determine el tipo de datos de los parámetros que esperan las funciones `roi` y `paso`, y el tipo de datos del valor devuelto por estas funciones.

2. a. Para la siguiente sección de código, determine el tipo de datos y el alcance de todas las variables declaradas. Para hacer esto, use una hoja de papel separada y anote los tres encabezados de columna que siguen (se han llenado las entradas de la primera variable):

Nombre de la variable	Tipo de datos	Alcance
`clave`	`char`	`global para main, func1 y func2`

```
#include <iostream>
using namespace std;

char clave;
long int numero;

int main()
{
  int a,b,c;
  double x,y;
     .
     .
  return 0;
}

double segundonum;

int func1(int num1, int num2)
{
  int o,p;
  float q;
     .
     .
  return p;
}

double func2(double primero, double ultimo)
{
  int a,b,c,o,p;
  double r;
  double s,t,x;
     .
     .
  return s * t;
}
```

b. Dibuje un cuadro alrededor de la sección apropiada del código anterior para encerrar el alcance de las variables `clave`, `segundonum`, `y` y `r`.

c. Determine el tipo de datos de los argumentos que esperan las funciones `func1` y `func2`, y el tipo de datos del valor devuelto por estas funciones.

3. Además de hablar sobre el alcance de una variable, también se puede aplicar el término a los parámetros de una función. ¿Cuál piensa que es el alcance de todos los parámetros de función?

4. Considere la siguiente estructura de programa:

```
#include <iostream>
using namespace std;

int a, b;
double Uno(float);
void Dos(void);
int main()
{
  int c, d;
  double e, f;
    .
    .
  return 0;
}

double Uno(double p2)
{
  char m, n;
    .
    .
}

void Dos(void)
{
  int p, d;
  double q, r;
    .
    .
}
```

Defina el alcance del parámetro p2 y las variables `a`, `b`, `c`, `d`, `m`, `n`, `p`, `d`, `q` y `r`.

5. Determine los valores desplegados por cada instrucción `cout` en el siguiente programa:

```
#include <iostream>
using namespace std;

int primernum = 10; // declara e inicializa una variable
global
void desplegar(); // prototipo de la función
int main()
{
  int primernum = 20; // declara e inicializa una variable
  local
  cout << "\nEl valor de primernum es " << primernum << endl;
  desplegar();

  return 0;
}
```

```
void desplegar(void)
{
  cout << "Ahora el valor de primernum es " << primernum
  << endl;
  return;
}
```

6.6 CLASES DE ALMACENAMIENTO DE VARIABLES

El alcance de una variable define la ubicación dentro de un programa donde ésta puede usarse. Dado un programa, se podría tomar un lápiz y trazar un cuadro alrededor de la sección del programa donde es válida cada variable. El espacio dentro del cuadro representaría el alcance de una variable. Desde este punto de vista, el alcance de una variable puede considerarse como el espacio dentro del programa donde es válida la variable.

Además de la dimensión espacial representada por su alcance, las variables también tienen una dimensión temporal. La dimensión temporal se refiere al tiempo que las ubicaciones de almacenamiento son reservadas para una variable. La dimensión temporal se conoce como la "vida" de la variable. Por ejemplo, todas las ubicaciones de almacenamiento de la variable se devuelven a la computadora cuando un programa termina su ejecución. Sin embargo, mientras un programa aún está en ejecución, las áreas de almacenamiento de la variable provisional se reservan y en lo subsiguiente se devuelven a la computadora. Puede determinarse dónde y cuánto se conservan las áreas de almacenamiento de una variable antes que sean liberadas por la **clase de almacenamiento** de la variable.

Además de tener un tipo de datos y un alcance, cada variable tiene también una clase de almacenamiento. Las cuatro clases de almacenamiento disponibles se llaman `auto`, `static`, `extern` y `register`. Si se usa uno de estos nombres de clase, debe colocarse antes del tipo de datos de la variable en una instrucción de declaración. Son ejemplos de instrucciones de declaración que incluyen una designación de clase de almacenamiento:

```
auto int num;           // clase de almacenamiento auto y tipo de datos int
static int millas;      // clase de almacenamiento static y tipo de datos int
register int dist;      // clase de almacenamiento register y tipo de datos int
extern int voltios;     // clase de almacenamiento extern y tipo de datos int
auto float cupon;       // clase de almacenamiento auto y tipo de datos float
static double años;     // clase de almacenamiento static y tipo de datos double
extern float resultado; // clase de almacenamiento extern y tipo de datos float
auto char tecla_en;     // clase de almacenamiento auto y variable char
```

Para entender lo que significa la clase de almacenamiento de una variable, se considerarán primero las variables locales (aquellas variables creadas dentro de una función) y luego las variables globales (aquellas variables creadas fuera de una función).

Clases de almacenamiento de variables locales

Las variables locales sólo pueden ser miembros de las `clases de almacenamiento auto`, `static o register.` Si no se incluye una descripción de clase en la instrucción de declaración, la variable es asignada de manera automática a la categoría `auto`. Por tanto, `auto`

es la clase por omisión usada por C++. Todas las variables locales que se han usado, en vista que se omitió la designación de la clase de almacenamiento, han sido variables `auto`.

El término `auto` es una abreviatura de ***automático.*** El almacenamiento para variables automáticas locales es reservado o creado en forma automática cada vez que se llama una función que declara variables automáticas. En tanto la función no haya regresado el control a la función que la llama, todas las variables automáticas locales para la función están "vivas"; es decir, el almacenamiento de las variables está disponible. Cuando la función devuelve el control a la función que la llama, sus variables automáticas locales "mueren"; es decir, el almacenamiento para las variables es devuelto a la computadora. Este proceso se repite cada vez que es llamada una función. Por ejemplo, considere el programa 6.19, donde se llama tres veces a la función `probarauto()` desde `main()`.

Programa 6.19

```
#include <iostream>
using namespace std;

void probarauto();    // prototipo de la función

int main()
{
  int cuenta;              // cuenta es una variable automática local

  for(cuenta = 1; cuenta <= 3; cuenta++)
    probarauto();

  return 0;
}

void probarauto()
{
  int num = 0;     // num es una variable automática local
                   // inicializada en cero
  cout << "El valor de la variable automática num es "
       << num << endl;
  num++;

  return;
}
```

La salida producida por el programa 6.19 es:

```
El valor de la variable automática num es 0
El valor de la variable automática num es 0
El valor de la variable automática num es 0
```

Cada vez que es llamada `probarauto()`, se crea e inicializa en cero la variable automática `num`. Cuando la función devuelve el control a `main()`, la variable `num` se destruye junto con cualquier valor almacenado en `num`. Por tanto, se pierde el efecto de incrementar num en `probarauto()`, antes de la instrucción de devolución de la función, cuando se regresa el control a `main()`.

Para la mayor parte de las aplicaciones, el uso de variables automáticas funciona bien. Sin embargo, hay casos en los que nos gustaría que una función recordara valores entre las llamadas a la función. Éste es el propósito de la clase de almacenamiento `static`. Una vez creadas, las variables `static` locales permanecen en existencia durante la vida del programa. Esto significa que el último valor almacenado en la variable cuando termina de ejecutarse la función está disponible para la función la próxima vez que es llamada.

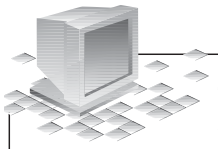
```
static int rate;
static double resistance;
static char inKey;
```

Una variable local `static` no se crea y se destruye cada vez que se llama a la función que declara dicha variable. Una vez creadas, las variables locales `static` siguen existiendo durante la vida del programa. Esto significa que el valor menor almacenado en la variable cuando termina la función, queda disponible cuando la función vuelve a ser llamada.

Debido a que las variables `static` locales conservan sus valores, no se inicializan dentro de una instrucción de declaración en la misma forma que las variables automáticas. Para entender por qué, considérese la declaración automática `int num = 0;` la cual causa que se cree la variable automática `num` y se coloque en cero cada vez que se encuentra la declaración. Esto se llama **inicialización en tiempo de ejecución** porque la inicialización ocurre cada vez que se encuentra la instrucción de declaración. Este tipo de inicialización sería desastroso para una variable `static`, porque reestablecer el valor de la variable en cero cada vez que se llama la función destruiría el valor mismo que se está tratando de guardar.

La inicialización de las variables `static` (tanto locales como globales) sólo se hace una vez, cuando el programa se compila por primera vez. En tiempo de compilación se crea la variable y se coloca cualquier valor de inicialización en ella.[15] A partir de entonces, el valor en la variable es conservado sin mayor inicialización. Para ver cómo funciona esto, considere el programa 6.20.

[15] Algunos compiladores inicializan las variables static locales la primera vez que se ejecuta la instrucción de definición en vez de hacerlo cuando se compila el programa.



Programa 6.20

```
#include <iostream>
using namespace std;

void probarstat();    // prototipo de la función

int main()
{
  int cuenta;              // cuenta es una variable auto local

  for(cuenta = 1; cuenta <= 3; cuenta++)
    probarstat();

  return 0;
}

void probarstat()
{
  static int num = 0;     // num es una variable static local
  cout << "El valor de la variable static num ahora es "
       << num << endl;
  num++;

  return;
}
```

La salida producida por el programa 6.20 es

```
El valor de la variable static num ahora es 0
El valor de la variable static num ahora es 1
El valor de la variable static num ahora es 2
```

Como lo ilustra esta salida, la variable `static num` se pone en cero sólo una vez. Luego la función `probarstat()` incrementa esta variable justo antes de regresar el control a `main()`. El valor que tenía `num` cuando deja la función `probarstat()` es conservado y desplegado cuando se vuelve a llamar a la función.

A diferencia de las variables automáticas que pueden ser inicializadas por constantes o por expresiones que usan tanto constantes como variables inicializadas con anterioridad, las variables `static` sólo pueden inicializarse usando constantes o expresiones constantes, como `3.2 + 8.0`. Además, a diferencia de las variables automáticas, todas las variables `static` se colocan en cero cuando no se da una inicialización explícita. Por tanto, en el programa 6.19 no se requiere la inicialización específica de `num` en cero.

La clase de almacenamiento restante disponible para variables locales, la clase `register`, no se usa en forma tan extensa como las `variables auto` o `static`. Son ejemplos de declaraciones de variable `register`

```
register int tiempo;
register double diferencia;
register float cupon;
```

Las variables `register` tienen la misma duración que las variables `auto`; es decir, una variable `register` local se crea cuando se introduce la función que la declara y se destruye cuando la función completa su ejecución. La única diferencia entre las variables `register` y `auto` es dónde se localiza el almacenamiento para la variable.

El almacenamiento para todas las variables (locales y globales), con excepción de las variables `register`, se reserva en el área de memoria de la computadora. La mayor parte de las computadoras tienen unas cuantas áreas de almacenamiento de alta velocidad adicionales localizadas en forma directa en la unidad de procesamiento de la computadora que también pueden usarse para almacenamiento de variables. Estas áreas de almacenamiento de alta velocidad especiales se llaman **registros**. Dado que los registros se localizan físicamente en la unidad de procesamiento de la computadora, también puede tenerse acceso a ellos con más rapidez que a las áreas de almacenamiento de memoria normales localizadas en la unidad de memoria. Además, las instrucciones de computadora que hacen referencia a registros por lo general requieren menos espacio que las instrucciones que hacen referencia a ubicaciones de memoria debido a que hay menos registros a los que se pueda tener acceso que ubicaciones de memoria. Cuando el compilador sustituye la ubicación de un registro por una variable durante la compilación del programa, se necesita menos espacio en la instrucción del que se requiere para dirigirse a una memoria que tiene millones de ubicaciones.

Además de disminuir el tamaño de un programa en C++ compilado, usar variables `register` puede aumentar la velocidad de ejecución de un programa en C++ si su computadora admite este tipo de datos. Los programas de aplicaciones que están pensados para ser ejecutados en varias computadoras no deberán usar registros. Los intentos de hacerlo por lo general serán frustrados por el compilador cambiando de manera automática las variables declaradas con la clase de almacenamiento `register` a una clase de almacenamiento `auto`.

La única restricción al usar la clase de almacenamiento `register` es que no se puede utilizar la dirección de una variable `register` con el operador de direccionamiento `&`. Esto se comprende con facilidad cuando se recuerda que los registros no tienen direcciones de memoria estándares.

Clases de almacenamiento de variables globales

Las variables globales son creadas por instrucciones de definición externas a una función. Por su naturaleza, estas variables definidas en forma externa no llegan y se van con la llamada de cualquier función. Una vez que se crea una variable global, existe hasta que el programa en el que se declaró termina de ejecutarse. Por tanto, las variables globales no pueden declararse como variables `auto` o `register` que se crean y destruyen conforme se ejecuta el programa. Además, las variables globales pueden declararse como miembros de las clases de almacenamiento `static` o `extern` (pero no de ambas). Son ejemplos de instrucciones de declaración que incluyen estas dos descripciones de clases

```
extern int suma;
extern double voltios;
static double corriente;
```

Las clases `static` y `extern` sólo afectan el alcance, no la duración, de variables globales. Como con las variables `static` locales, todas las variables globales se inicializan en cero en tiempo de compilación.

El propósito de la clase de almacenamiento `extern` es extender el alcance de una variable global más allá de sus límites normales. Para entender esto, primero hay que observar que todos los programas que se han escrito hasta ahora siempre se habían contenido en un solo archivo. Por tanto, cuando guardaba o recuperaba programas sólo necesitaba darle a la computadora un nombre para su programa. Esto no se requiere en C++.

Los programas más grandes por lo general consisten en muchas funciones que son almacenadas en múltiples archivos. Un ejemplo de esto se muestra en la figura 6.18, donde las tres funciones `main()`, `func1()` y `func2()` se almacenan en un archivo y las dos funciones `func3()` y `func4()` se almacenan en un segundo archivo.

archivo1

```
int voltios;
double corriente;
static double potencia;
     .
     .
     .
int main( )
{
     func1( );
     func2( );
     func3( );
     func4( );
}
int func1( )
{
     .
     .
     .
}
int func2( )
{
     .
     .
     .
}
```

archivo2

```
double factor;
int func3( )
{
     .
     .
     .
}
int func4( )
     .
     .
     .
}
```

Figura 6.18 *Un programa puede estar almacenado en más de un archivo.*

Para los archivos ilustrados en la figura 6.18, las variables globales `voltios`, `corriente` y `potencia` declaradas en archivo1 sólo pueden ser usadas por las funciones `main()`, `func1()` y `func2()` en este archivo. La variable global única, factor, declarada en archivo2 sólo puede ser usada por las funciones `func3()` y `func4()` en el archivo2.

Aunque la variable `voltios` se ha creado en el archivo1, se puede utilizar en el archivo2. Colocar la instrucción de declaración `extern int voltios;` en el archivo2, como se muestra en la figura 6.19, nos permite hacerlo. Poner esta instrucción en la parte superior de archivo2 extiende el alcance de la variable `voltios` al archivo2 de modo que pueda usarse tanto en `func3()` como en `func4()`. Por tanto, la designación `extern` tan sólo declara una variable global que es definida en otro archivo. Así, colocar la instrucción `extern double corriente;` en `func4()` extiende el alcance de esta variable global, creada en archivo1, a la `func4()`, y el alcance de la variable global `factor`, creada en archivo2, se extiende a `func1()` y `func2()` por la instrucción de declaración `extern double factor;` colocada antes de `func1()`. Hay que observar que `factor` no está disponible para `main()`.

archivo1

```
int voltios;
double curriente;
static double potencia;
     .
     .
     .
int main( )
{
   func1( );
   func2( );
   func3( );
   func4( );
}
extern double factor;
int func1( )
{
     .
     .
     .
}
int func2( )
{
     .
     .
     .
}
```

archivo2

```
double factor;
extern int voltios;
int func3( )
{
     .
     .
     .
}
int func4( )
{
   extern double corriente;
     .
     .
     .
}
```

Figura 6.19 *Extensión del alcance de variables globales.*

Una instrucción de declaración que contiene de manera específica la palabra `extern` es diferente de todas las otras instrucciones de declaración porque no origina una variable nueva reservando nuevo almacenamiento para la variable. Una instrucción de declaración `extern` tan sólo informa a la computadora que ya existe una variable global y que ahora puede usarse. El almacenamiento real para la variable debe crearse en alguna otra parte del programa usando una, y sólo una, instrucción de declaración global en la que no se ha usado

Punto de información

Clases de almacenamiento

Las variables del tipo `auto` y `register` siempre son variables locales. Sólo variables globales no estáticas pueden declararse usando la palabra clave `extern`. Hacerlo así extiende el alcance de la variable a otro archivo o función.

Hacer `static` una variable global reduce una variable privada del archivo en el que se declara. Por tanto, las variables `static` *no pueden* utilizar la palabra clave `extern`. Excepto para variables `static`, todas las variables son inicializadas cada vez que ingresan a un alcance. Las variables `static` sólo se inicializan una vez cuando se definen.

la palabra `extern`. La inicialización de la variable global puede hacerse, por supuesto, con la declaración original de la variable global. La inicialización dentro de una instrucción de declaración `extern` no se permite y causará un error de compilación.

La existencia de la clase de almacenamiento `extern` es la razón por la cual se ha tenido tanto cuidado en distinguir entre la creación y declaración de una variable. Las instrucciones de declaración que contienen la palabra `extern` no crean nuevas áreas de almacenamiento; sólo extienden el alcance de variables globales existentes.

La última clase global, las variables globales `static`, se utilizan para prevenir la extensión de una variable global a un segundo archivo. Las variables globales `static` se declaran en la misma forma que las variables locales `static`, excepto que la instrucción de declaración se coloca afuera de cualquier función.

El alcance de una variable global `static` no puede extenderse más allá del archivo en el que es declarada. Esto proporciona un grado de privacidad para las variables globales `static`. Dado que sólo son "conocidas" y sólo pueden usarse en el archivo en el que son declaradas, otros archivos no pueden tener acceso a ellas ni cambiar sus valores. Por tanto, las variables globales `static` no pueden extenderse después a un segundo archivo usando una instrucción de declaración `extern`. Intentar hacerlo producirá un error de compilación.

Ejercicios 6.6

1. **a.** Enumere las clases de almacenamiento disponibles para variables locales.
   b. Enumere las clases de almacenamiento disponibles para variables globales.
2. Describa la diferencia entre una variable local `auto` y una variable local `static`.
3. ¿Cuál es la diferencia entre las siguientes funciones?

```
void init1()
{
  static int años = 1;
  cout << "El valor de anios es " << años << endl;
  años = años + 2;
}
```

```
void init2()
{
  static int años;
  anios = 1;
  cout << "El valor de años es " << años << endl;
  años = años + 2;
}
```

4. **a.** Describa la diferencia entre una variable global `static` y una variable global `extern`.
   b. Si una variable es declarada con una clase de almacenamiento `extern`, ¿qué otra instrucción de declaración debe estar presente en alguna otra parte del programa?
5. La instrucción de declaración `static double resistencia;` puede usarse para crear ya sea una variable local o una global. ¿Qué determina el alcance de la variable `resistencia`?
6. Para las declaraciones de función y variable ilustradas en la figura 6.20, coloque una declaración `extern` para lograr de manera individual lo siguiente:
   a. Extender el alcance de la variable global `eleccion` a todo el archivo2.
   b. Extender el alcance de la variable global `marcador` sólo a la función `promedio()`.
   c. Extender el alcance de la variable global fecha a `promedio()` y `variancia()`.
   d. Extender el alcance de la variable global `fecha` sólo a `roi()`.
   e. Extender el alcance de la variable global `factor` sólo a `roi()`.
   f. Extender el alcance de la variable global `tipo_b` a todo el archivo1.
   g. Extender el alcance de la variable global `resistencia` a `vatios()` y `empuje()`.

[16] En la práctica deberá usarse un buen programa depurador.

archivo1

```
char eleccion;
int marcador;
long fecha, tiempo;
int main( )
{
     .
     .
     .
}
double factor;
ddouble vatios ( )
{
     .
     .
     .
}
double empuje( )
{
     .
     .
     .
}
```

archivo2

```
char tipo_b;
double resistencia;
double roi( )
{
     .
     .
     .
}
double promedio( )
{
     .
     .
     .
}
double variancia( )
{
     .
     .
     .
}
```

Figura 6.20 *Archivos para el ejercicio 6.*

6.7 ERRORES COMUNES DE PROGRAMACIÓN

Un error de programación demasiado común relacionado con las funciones es transmitir tipos de datos incorrectos. Los valores transmitidos a una función deben corresponder a los tipos de datos de los parámetros declarados para la función. Una forma de verificar que se han recibido valores correctos es desplegar todos los valores transmitidos dentro del cuerpo de una función antes que se hagan cálculos. Una vez que ha tenido lugar esta verificación, puede prescindirse del despliegue.[16]

Otro error común puede ocurrir cuando se declara la misma variable en forma local dentro de la función que llama y la función que es invocada. Aun cuando el nombre de la variable es el mismo, un cambio a una variable local no altera el valor en la otra variable local.

Un error relacionado con esto puede ocurrir cuando una variable local tiene el mismo nombre que una variable global. Dentro de la función que la declara, el uso del nombre de la variable sólo afecta al contenido de la variable local a menos que se utilice el operador de resolución de alcance, **::**.

Otro error común es omitir el prototipo de la función llamada ya sea antes o dentro de la función que llama. La función invocada debe ser alertada del tipo de valor que se devolverá, y esta información es proporcionada por el prototipo de la función. El prototipo puede omitirse si la función llamada está colocada físicamente en un programa antes de la función que la llama. Aunque también es permisible omitir el prototipo y el tipo de devolución para funciones que devuelven un número entero, es una mala práctica de documentación hacerlo de esta manera. El valor real devuelto por una función puede verificarse desplegándolo antes y después de que es devuelto.

Los últimos dos errores comunes son terminar una línea de encabezado de una función con un punto y coma, y olvidar incluir el tipo de datos de los parámetros de una función dentro de la línea de encabezado.

6.8 RESUMEN DEL CAPÍTULO

1. Una función es invocada dando su nombre y transmitiéndole cualesquier datos que haya en el paréntesis que sigue al nombre. Si una variable es uno de los argumentos en una llamada a la función, la función llamada recibe una copia del valor de la variable.
2. La forma usada de una función escrita por el usuario por lo general es

   tipo-de-datos-a-devolver nombre-de-función(lista de parámetros)

   {

   declaraciones y otras instrucciones de C++;

   return expresion;

   }

   La primera línea de la función se llama **encabezado de la función**. Las llaves de apertura y cierre de la función y todas las instrucciones entre estas llaves constituyen el **cuerpo** de la función. El tipo de datos devuelto, por omisión, es un número entero cuando no se especifica ningún tipo de datos devuelto. La lista de parámetros es una lista de declaraciones de parámetros separadas por comas.
3. El tipo de devolución de una función es el tipo de datos del valor devuelto por la función. Si no se declara ningún tipo se asume que la función devuelve un valor entero. Si la función no devuelve un valor deberá declararse como un tipo `void`.
4. Las funciones pueden devolver en forma directa cuando mucho un valor de un solo tipo de datos a las funciones que las llaman. Este valor es el valor de la expresión en la instrucción de devolución.

5. Usando argumentos de referencia, se puede transmitir a una función la dirección de una variable. Si a una función llamada se le transmite una dirección tiene la capacidad de tener acceso directo a la variable de la función que llama respectiva. Usar direcciones transmitidas permite que una función llamada devuelva de manera efectiva múltiples valores.

6. Las funciones pueden declararse a todas las funciones que llaman por medio de un **prototipo de función.** El prototipo proporciona una declaración para una función que especifica el tipo de datos devuelto por la función, su nombre y los tipos de datos de los argumentos esperados por la función. Como con todas las declaraciones, un prototipo de función se termina con un punto y coma y puede incluirse dentro de declaraciones de variable local o como una declaración global. La forma más común de un prototipo de función es:

   tipo-de-datos nombre-de-función(lista de tipos de datos de los parámetros);

   Si la función llamada es colocada físicamente arriba de la función que llama, no se requieren más declaraciones porque la definición de la función sirve como una declaración global para todas las funciones que siguen.

7. Todas las variables usadas en un programa tienen un **alcance,** el cual determina en qué parte del programa puede utilizarse la variable. El alcance de una variable es local o global y está determinado por el lugar donde se coloca la instrucción de definición de la variable. Una variable local se define dentro de una función y sólo puede usarse dentro de la función o bloque que la define. Una variable global se define fuera de una función y puede usarse en cualquier función que siga a la definición de la variable. Todas las variables globales que no son inicializadas de manera específica por el usuario son inicializadas en cero por el compilador y pueden ser compartidas entre archivos que usen la palabra clave `extern`.

8. Todas las variables tienen una **clase.** La clase de una variable determina cuánto se conservará el valor en la variable, lo cual también se conoce como la duración de la variable: las variables `auto` son variables locales que sólo existen mientras se está ejecutando la función que las define; las variables `register` son similares a las variables `auto` pero son almacenadas en los registros internos de una computadora y no en la memoria; las variables `static` pueden ser globales o locales y conservan sus valores mientras dura la ejecución de un programa. Todas las variables `static` se colocan en cero cuando se definen si no son inicializadas de manera explícita por el usuario.

Consideración de opciones de carrera

Ingeniería química

La ingeniería química es la aplicación del conocimiento o técnicas de la ciencia, en particular de la química, a la industria. Los ingenieros químicos son responsables del diseño y operación de plantas manufactureras, a gran escala, para materiales que experimentan cambios químicos en su producción. Estos materiales incluyen todos los productos nuevos y mejorados que han afectado en forma tan profunda a la sociedad, como petroquímicos, hules y polímeros, nuevas aleaciones de metales, sustancias químicas industriales y refinados, alimentos, pinturas, detergentes, cementos, pesticidas, gases industriales y medicinas. Los ingenieros químicos también desempeñan un papel importante en el abatimiento de la contaminación y la administración de los recursos energéticos existentes. Debido a que el campo de la ingeniería química ha crecido tanto, es difícil clasificar las actividades de los ingenieros químicos. A grandes rasgos pueden dividirse en sistemas de producción a gran escala, o procesamiento químico, y en sistemas a escala más pequeña, o moleculares.

Procesamiento químico

El procesamiento químico tiene que ver con todos los aspectos del diseño y operación de plantas de procesamiento químico grandes. Incluye las siguientes áreas:

1. Petroquímicos. La destilación y refinación de combustibles como gasolina, gas natural sintético, licuefacción y gasificación del carbón, y la producción de una variedad infinita de productos hechos de petróleo, desde cosméticos hasta productos farmacéuticos.
2. Materiales sintéticos. El proceso de polimerización, la unión de moléculas más simples en moléculas complejas grandes, es responsable de muchos materiales modernos como el nailon, hules sintéticos, poliestireno, y una gran variedad de plásticos y fibras sintéticas.
3. Ingeniería de alimentos y bioquímica. La manufactura de alimentos empacados, aditivos alimenticios mejorados, esterilización y la utilización de bacterias, hongos y levaduras industriales en procesos como la fermentación.
4. Operaciones unitarias. El análisis del transporte de calor o líquidos, como el bombeo de sustancias químicas a través de una tubería o la transferencia de calor entre sustancias. Esta área incluye también el efecto de la transferencia de calor en reacciones químicas como la oxidación, la clorinación, etcétera.
5. Ingeniería criogénica. El diseño de plantas que operan a temperaturas cercanas al cero absoluto.
6. Ingeniería electroquímica. El uso de electricidad para alterar reacciones químicas, como el electrochapado, o el diseño de baterías o celdas de energía.
7. Control de la contaminación. Un campo que ha tenido un crecimiento rápido que busca vigilar y reducir los efectos perjudiciales del procesamiento químico en el ambiente. Sus áreas de interés son el control del desperdicio de agua, el abatimiento de la contaminación del aire y los aspectos económicos del control de la contaminación.

(continúa)

Consideración de opciones de carrera

Sistemas moleculares

Este campo implica la aplicación de técnicas de laboratorio en procesos en gran escala. Incluye las siguientes áreas:

1. Ingeniería bioquímica. Aplicación de enzimas, bacterias u otros microorganismos para mejorar los procesos químicos a gran escala.
2. Síntesis de polímeros. La base molecular de las propiedades de los polímeros y la síntesis química de nuevos polímeros adaptados a la producción a gran escala.
3. Investigación y desarrollo en todas las áreas del procesamiento químico.

La preparación para una carrera en la ingeniería química requiere una educación sólida en física, química y matemáticas y un conocimiento de termodinámica y química física, analítica y orgánica. Aunque con una capacitación extensa en química, los ingenieros químicos difieren de los químicos, ya que su principal interés es la adaptación de las técnicas de laboratorio a plantas manufactureras a gran escala.