CENGAGE Learning

ESTRUCTURAS DE DATOS CON C++

SEGUNDA EDICIÓN

D.S. Malik

ESTRUCTURAS DE DATOS CON C++

SEGUNDA EDICIÓN

D. S. MALIK

CENGAGE Learning®

Australia • Brasil • Corea • España • Estados Unidos • Japón • México • Reino Unido • Singapur

CENGAGE Learning®

Estructuras de datos con C++
D. S. Malik

Presidente de Cengage Learning Latinoamérica:
Fernando Valenzuela Migoya

Director Editorial, de Producción y de Plataformas Digitales para Latinoamérica:
Ricardo H. Rodríguez

Gerente de Procesos para Latinoamérica:
Claudia Islas Licona

Gerente de Manufactura para Latinoamérica:
Raúl D. Zendejas Espejel

Gerente Editorial de Contenidos en Español:
Pilar Hernández Santamarina

Gerente de Proyectos Especiales:
Luciana Rabuffetti

Coordinador de Manufactura:
Rafael Pérez González

Editores:
Javier Reyes Martínez
Timoteo Eliosa García

Imágenes de portada:
©Fancy Photography/Veer

Composición tipográfica:
Imagen Editorial

Impreso en México
1 2 3 4 5 6 7 15 14 13 12

Traducido del libro *Data Structures using C++*, Second edition.
D. S. Malik
Publicado en inglés por Course Technology, una compañía de Cengage Learning ©2010
ISBN: 978-0-324-78201-1

Datos para catalogación bibliográfica:
D. S. Malik
Estructuras de datos con C++
ISBN: 978-607-481-931-1

Visite nuestro sitio en:
http://latinoamerica.cengage.com

Contenido

CAPÍTULO 1

Principios de ingeniería de software y clases de C++

EN ESTE CAPÍTULO USTED:

- Aprenderá acerca de los principios de ingeniería de software
- Descubrirá lo que es un algoritmo y explorará técnicas de solución de problemas
- Conocerá el diseño estructurado y las metodologías de programación de diseño orientado a objetos
- Aprenderá acerca de las clases
- Conocerá acerca de los miembros **`private`**, **`protected`** y **`public`** de una clase
- Explorará cómo se implementan las clases
- Conocerá acerca de la notación del Lenguaje Unificado de Modelado (UML)
- Examinará constructores y destructores
- Conocerá los tipos de datos abstractos (ADT)
- Explorará cómo las clases se utilizan para implementar ADT

La mayoría de las personas que trabajan con computadoras están familiarizadas con el término *software*. Software son los programas de cómputo diseñados para realizar una tarea específica. Por ejemplo, el software para procesamiento de textos es un programa que permite escribir trabajos escolares finales, crear currículos con excelente presentación e incluso escribir un libro como éste, por ejemplo, el cual fue creado con ayuda de un procesador de textos. Los estudiantes ya no teclean sus documentos en máquinas de escribir ni los redactan a mano. En lugar de ello, utilizan software de procesamiento de textos para presentar sus ensayos. Muchas personas manejan las operaciones de sus chequeras por computadora.

El software, potente y fácil de usar, ha transformado drásticamente la forma en que vivimos y nos comunicamos. Términos que hace apenas una década eran desconocidos, como *Internet*, son muy comunes hoy. Con la ayuda de las computadoras y el software que se ejecuta en ellas, usted puede enviar cartas a sus seres queridos, y también recibirlas, en cuestión de segundos. Ya no necesita enviar su currículo por correo para solicitar un empleo, en muchos casos, simplemente puede enviar su solicitud a través de Internet. Puede ver el desempeño de las acciones en la bolsa en tiempo real, e inmediatamente comprarlas y venderlas.

Sin software, una computadora no tiene ninguna utilidad. Es el software lo que le permite hacer cosas que hace algunos años, quizás eran consideradas ficción. Sin embargo, el software no se crea en una noche. Desde el momento en que un programa de software se concibe hasta su entrega, pasa por varias etapas. Existe una rama de la informática, llamada ingeniería de software, que se especializa en esta área. La mayoría de los colegios y universidades ofrece un curso de ingeniería de software. Este libro no se ocupa de la enseñanza de los principios de la ingeniería de software. No obstante, en este capítulo se describen brevemente algunos de los principios básicos de ingeniería de software que pueden simplificar el diseño de programas.

Ciclo de vida del software

Un programa pasa por muchas etapas desde el momento de su concepción hasta que se le retira, a las cuales se les llama *ciclo de vida* del programa. Las tres etapas fundamentales por las que un programa pasa son *desarrollo*, *uso* y *mantenimiento*. Al principio, un desarrollador de software concibe, por lo general, un programa, porque un cliente tiene algún problema que necesita resolver y el cliente está dispuesto a pagar dinero para que el problema se resuelva. El nuevo programa se crea en la etapa de *desarrollo de software*. En la siguiente sección se describe con detalle esta etapa.

Cuando se considera que el programa está completo, es lanzado (liberado) al mercado para que los usuarios lo utilicen. Una vez que los usuarios comienzan a utilizar el programa, lo más seguro es que descubran problemas o tengan sugerencias para mejorarlo. Los problemas o ideas para hacerle mejoras se hacen llegar al desarrollador de software, y el programa pasa a la etapa de mantenimiento.

En el proceso de *mantenimiento del software*, el programa se modifica para reparar los problemas (identificados) o mejorarlo. Si hay cambios serios o numerosos, por lo común se crea una nueva versión del programa y se lanza a la venta para su uso.

Cuando el mantenimiento de un programa se considera demasiado caro, el desarrollador podría decidir *retirarlo* y ya no realizar una nueva versión del mismo.

La etapa de desarrollo del software es la primera y tal vez la más importante del ciclo de vida del mismo. El mantenimiento de un programa bien desarrollado es más fácil y menos costoso. La sección siguiente describe esta etapa.

Etapa de desarrollo del software

Los ingenieros de software dividen el proceso de desarrollo del software en las cuatro fases siguientes:

- Análisis
- Diseño
- Implementación
- Pruebas y depuración

En las secciones siguientes se describen estas cuatro fases con detalle.

Análisis

El análisis del problema es el primer y más importante paso, en el cual se requiere que usted:

- Entienda el problema a fondo.
- Comprenda los requerimientos del problema. Éstos pueden incluir si el programa tendrá interacción con el usuario, si manipulará los datos, si producirá un resultado y la apariencia que tendrá el resultado.

  Supongamos que necesita desarrollar un programa para hacer que un cajero automático (ATM) entre en operación. En la fase de análisis debe determinar la funcionalidad de la máquina. Aquí se establecen las operaciones necesarias que realizará la máquina, como retiro de fondos, depósito de dinero, transferencia del mismo, consulta del estado de cuenta, etc. Durante esta fase, usted también debe consultar con posibles clientes que usarán el cajero. Para lograr que su operación sea sencilla para los usuarios, debe comprender sus necesidades y añadir las operaciones necesarias.

  Si el programa manipulará datos, el programador debe saber de qué datos se trata y cómo los representará. Es decir, usted necesita estudiar una muestra de datos. Si el programa producirá un resultado, usted debe saber cómo se generan los resultados y el formato que tendrán.
- Si el problema es complejo, divídalo en subproblemas, analice cada subproblema y entienda los requerimientos de cada uno.

Diseño

Después de analizar detenidamente el problema, el paso siguiente es diseñar un algoritmo para resolverlo. Si usted divide el problema en subproblemas, necesita diseñar un algoritmo para cada subproblema.

Algoritmo: proceso de solución de problemas, paso a paso, en el cual se llega a una solución en un tiempo finito.

DISEÑO ESTRUCTURADO

La división de un problema en problemas más pequeños o subproblemas se llama **diseño estructurado**. El método del diseño estructurado también se conoce como **diseño descendente, refinamiento por pasos** y **programación modular**. En el diseño estructurado, el problema se divide en problemas más pequeños. Luego se analiza cada subproblema y se obtiene una solución para cada uno. Después se combinan las soluciones de todos los subproblemas para resolver el problema general. Este proceso de implementar un diseño estructurado se conoce como **programación estructurada**.

DISEÑO ORIENTADO A OBJETOS

En el diseño orientado a objetos (DOO), el primer paso en el proceso de solución de problemas es identificar los componentes llamados objetos, que forman la base de la solución, y determinar cómo interaccionarán esos objetos. Por ejemplo, suponga que quiere escribir un programa que automatice el proceso de renta de videos para una tienda local. Los dos objetos principales de este problema son el video y el cliente.

Después de identificar los objetos, el paso siguiente es especificar los datos relevantes para cada objeto y las operaciones posibles que se realizarán con esos datos. Por ejemplo, para un objeto de video, los datos podrían incluir el nombre de la película, los protagonistas, el productor, la empresa productora, el número de copias almacenadas, y así por el estilo. Algunas de las operaciones con el objeto de video podrían ser la verificación del nombre de la película, la reducción en uno del número de copias en reserva cada vez que se alquila una copia, y el incremento en uno del número de copias en bodega después de que un cliente devuelve un video en particular.

Lo anterior muestra que cada objeto se compone de los datos y las operaciones con esos datos. Un objeto combina los datos y operaciones con los datos en una sola unidad. En el DOO, el programa final es una colección de objetos que interaccionan. Un lenguaje de programación que implementa el DOO se llama lenguaje de **programación orientado a objetos** (POO). Usted aprenderá acerca de las muchas ventajas que ofrece el DOO en los capítulos subsecuentes.

El DOO tiene los tres principios básicos siguientes:

- **Encapsulación.** La capacidad para combinar los datos y las operaciones en una sola unidad.
- **Herencia.** La capacidad para crear nuevos tipos de datos a partir de los tipos de datos existentes
- **Polimorfismo.** La capacidad para utilizar la misma expresión para denotar operaciones diferentes.

En C++, la encapsulación se logra mediante el uso de tipos de datos denominados "clases". Más adelante en este capítulo se describe cómo se implementan las clases en C++. En el capítulo 2 se estudian la herencia y el polimorfismo.

En el diseño orientado a objetos, usted decide qué clases necesita y los miembros de datos y funciones relevantes que las compondrán. Luego describirá cómo interaccionarán las clases.

Implementación

En la fase de *implementación*, usted escribe y compila el código de programación para poner en acción las clases y las funciones que se descubrieron en la fase de diseño.

Este libro utiliza la técnica de DOO (junto con la programación estructurada) para resolver un problema en particular. Contiene muchos casos resueltos —llamados "Ejemplos de programación"— para resolver problemas reales.

El programa final consta de varias funciones, cada una de las cuales logra un objetivo específico. Algunas funciones son parte del programa principal, otras se utilizan para implementar varias operaciones con objetos. Desde luego, las funciones interaccionan entre sí, aprovechando las capacidades mutuas. Para utilizar una función, el usuario sólo necesita saber cómo utilizar la función y lo que ésta hace. El usuario no debe preocuparse por los detalles de la función, es decir, cómo se escribe. Ilustremos esto con ayuda del ejemplo siguiente.

Suponga que quiere escribir una función que convierte una medición dada en pulgadas en su equivalente en centímetros. La fórmula de conversión es 1 pulgada = 2.54 centímetros. La función siguiente realiza la tarea:

```
double inchesToCentimeters(double inches)
{
    if (inches < 0)
    {
        cerr << "La medida dada no puede ser negativa". << endl;
        return -1.0;
    }
    else
        return 2.54 * inches;
}
```

El objeto `cerr` corresponde al flujo de errores estándar sin memoria intermedia. A diferencia del objeto `cout` (cuya salida primero pasa a la memoria intermedia), la salida de `cerr` se envía de inmediato al flujo de errores estándar, que por lo general es la pantalla.

Si analiza el cuerpo de la función, puede reconocer que si el valor de las pulgadas es menor que 0, es decir, negativo, la función devuelve –1.0; de lo contrario, la función devuelve la longitud equivalente en centímetros. El usuario de esta función no necesita conocer los detalles específicos de cómo se implementa el algoritmo que calcula la longitud equivalente en centímetros, pero sí debe saber que para obtener la respuesta válida, la entrada debe ser un número no negativo. Si la entrada a esta función es un número negativo, el programa devuelve –1.0. Esta información puede proporcionarse como parte de la documentación de esta función utilizando sentencias específicas, llamadas precondiciones y poscondiciones.

Precondición: una sentencia que especifica la(s) condición(es) que deben ser verdaderas antes de asignarle un nombre a la función.

Poscondición: una sentencia que especifica lo que es verdadero después de que la asignación del nombre de la función se completa.

La precondición y la poscondición para la función `inchesToCentimeters` pueden especificarse como sigue:

```
  //Precondición: El valor de inches debe ser no negativo.
  //Poscondición: Si el valor de inches es < 0, la función
  //   devuelve -1.0; de lo contrario, la función devuelve la
  //   longitud equivalente en centímetros.
double inchesToCentimeters(double inches)
{
    if (inches < 0)
    {
        cerr << "La medida dada tiene que ser no negativa". << endl;
        return -1.0;
    }
    else
        return 2.54 * inches;
}
```

En ciertas situaciones, usted puede utilizar la sentencia `assert` de C++ para validar la entrada. Por ejemplo, la función anterior puede escribirse como sigue:

```
  //Precondición: El valor de inches debe ser no negativo.
  //Poscondición: Si el valor de inches es < 0, la función
  //   termina; de lo contrario, la función devuelve la
  //   longitud equivalente en centímetros.
double inchesToCentimeters(double inches)
{
      assert(inches >= 0);
      return 2.54 * inches;
}
```

Sin embargo, si la expresión `assert` falla, todo el programa terminará, lo cual puede ser apropiado si el resultado del programa depende de la ejecución de la función. Por otra parte, el usuario puede comprobar el valor devuelto por la función, determinar si el valor devuelto es apropiado y proceder en consecuencia. Para utilizar la función `assert`, usted necesita incluir el archivo con el encabezado `cassert` en su programa.

Para desactivar las expresiones `assert` en un programa, utilice la directiva de preprocesador `#define NDEBUG`. Esta directiva debe colocarse antes de la sentencia `#include <cassert>`.

Como es posible observar, la misma función puede ser implementada de manera diferente por distintos programadores. Debido a que el usuario de una función no necesita preocuparse por los detalles de la función, las precondiciones y poscondiciones se especifican con la función `prototype`. Es decir, el usuario recibe la información siguiente:

```
double inchesToCentimeters(double inches);
  //Precondición: El valor de inches debe ser no negativo.
  //Poscondición: Si el valor de inches es < 0, la función
  //    devuelve -1.0; de lo contrario, la función devuelve la
  //    longitud equivalente en centímetros.
```

Como otro ejemplo, para utilizar una función que busque un elemento específico en una lista, ésta debe existir antes de que la función sea solicitada. Una vez que la búsqueda está completa, la función devuelve `true` o `false`, dependiendo de si la búsqueda fue exitosa o no.

```
bool search(int list[], int listLength, int searchItem);
  //Precondición: La lista debe existir.
  //Poscondición: La función devuelve true si searchItem está en
  //    la lista; de lo contrario, la función devuelve false.
```

Pruebas y depuración

El término *prueba* se refiere a probar la exactitud del programa; es decir, asegurarse de que el programa hace lo que se supone debe hacer. El término *depuración* se refiere a encontrar y corregir los errores, si es que éstos existen.

Una vez que una función o un algoritmo se escriben, el paso siguiente es comprobar que funciona correctamente. No obstante, en un programa grande y complejo, es casi seguro que existan errores. Por tanto, para aumentar la confiabilidad del programa, los errores deben descubrirse y repararse antes de que el programa se distribuya (libere) a los usuarios.

Desde luego, esto se puede demostrar mediante el uso de algunos análisis (quizás matemáticos) de la exactitud de un programa. Sin embargo, para los programas grandes y complejos, esta técnica por sí sola puede no ser suficiente, debido a que es posible cometer errores durante la prueba. Por consiguiente, también nos basamos en ensayos para determinar la calidad del programa, el cual se somete a una serie de pruebas específicas, llamada "casos de prueba", en un intento por detectar problemas.

Un caso de prueba consiste en una serie de entradas de información, acciones por parte del usuario y otras condiciones iniciales, y el resultado esperado. Dado que un caso de prueba puede repetirse varias veces, debe documentarse de manera apropiada. Por lo general, un programa manipula un conjunto grande de datos. De ahí que resulte poco práctico crear casos de prueba para todas las entradas posibles. Por ejemplo, imagine que un programa manipula los enteros. Está claro que no es posible crear un caso de prueba para cada entero. Usted puede clasificar los casos de prueba en categorías separadas llamadas "categorías de equivalencia". Una categoría de equivalencia es un conjunto de valores de entrada que es probable que produzca la misma salida. Por ejemplo, suponga que tiene una función que toma un entero como entrada y devuelve `true` si el entero es no negativo, y `false` en caso contrario. En este caso, usted puede formar dos categorías de equivalencia —una compuesta por números negativos, y la otra por números no negativos.

Existen dos tipos de pruebas: de *caja blanca* y de *caja negra*. En las pruebas de caja negra usted no conoce el trabajo interno del algoritmo o la función, sólo sabe lo que hace la función. Las pruebas de caja negra se basan en entradas y salidas. Los casos de prueba para las pruebas de caja negra, por lo general se seleccionan al crear categorías de equivalencia. Si una función trabaja

bien para una entrada de la categoría de equivalencia, se espera que trabaje también para otras entradas de la misma categoría.

Suponga que la función `isWithInRange` devuelve un valor `true` si un entero es mayor o igual que 0 y menor o igual que 100. En las pruebas de caja negra, la función se prueba con valores que rodean y entran en los límites, llamados **valores límite**, así como valores generales de las categorías de equivalencia. Para la función `isWithInRange`, en las pruebas de caja negra, los valores límite podrían ser: –1, 0, 1, 99, 100 y 101, por tanto, los valores de prueba pueden ser –500, –1, 0, 1, 50, 99, 100, 101 y 500.

Las pruebas de caja blanca se basan en la estructura interna y la implementación de una función o algoritmo. El objetivo es asegurarse de que cada parte de la función o algoritmo se ejecuta cuando menos una vez. Suponga que quiere asegurarse de que una sentencia trabaja de manera apropiada. Los casos de prueba deben constar de una entrada, por lo menos, para la cual la sentencia `if` se evalúa como `true` y por lo menos un caso para el cual se evalúa como `false`. Los bucles y otras estructuras pueden probarse de modo parecido.

Análisis de algoritmos: la notación O grande

Así como un problema se analiza antes de escribir el algoritmo y el programa de computadora, después de que un algoritmo se diseña también debe analizarse. Existen varias maneras de diseñar un algoritmo en particular. La ejecución de ciertos algoritmos requiere muy poco tiempo de computadora, mientras que la ejecución de otros toma mucho tiempo.

Considere el problema siguiente. La temporada navideña se acerca y una tienda de regalos espera que la cantidad normal de ventas se duplique e incluso se triplique. Se ha contratado más personal de entrega para asegurarse de que los paquetes sean entregados a tiempo. La empresa calcula la distancia más corta desde la tienda a un destino en particular y pasa la ruta al repartidor. Suponga que se deben entregar 50 paquetes en 50 casas diferentes. La tienda, mientras prepara la ruta, se da cuenta de que las 50 casas están a una milla de distancia y se encuentran en la misma zona. (Vea la figura 1-1, donde cada punto representa una casa y la distancia entre las casas es de 1 milla).

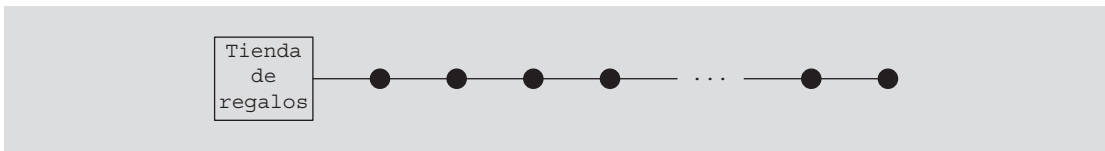


FIGURA 1-1 La tienda de regalos y cada punto que representa una casa

Para entregar 50 paquetes a sus destinos, uno de los repartidores recoge los 50 paquetes, maneja una milla a la primera casa y entrega el primer paquete. Luego maneja otra milla y entrega el segundo paquete, después maneja otra milla y entrega el tercer paquete, etcétera. La figura 1-2 ilustra este esquema de entrega.

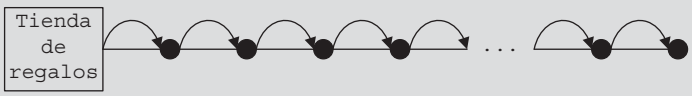


FIGURA 1-2 Esquema de entrega de paquetes

Por tanto, al utilizar este esquema, la distancia que condujo el repartidor para entregar los paquetes es:

$1 + 1 + 1 + \ldots + 1 = 50$ millas

Por consiguiente, la distancia total recorrida por el repartidor para entregar los paquetes y luego regresar a la tienda es:

$50 + 50 = 100$ millas

Otro repartidor tiene una ruta semejante para entregar otro grupo de 50 paquetes. El repartidor estudia la ruta y entrega los paquetes como sigue: recoge el primer paquete, maneja una milla a la primera casa y entrega el paquete, luego regresa a la tienda. Después recoge el segundo paquete, maneja 2 millas y lo entrega, y regresa a la tienda. Ahí, el repartidor recoge el tercer paquete, maneja 3 millas, entrega el paquete y regresa a la tienda. La figura 1-3 muestra este esquema de entrega.

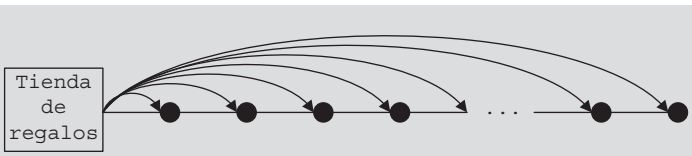


FIGURA 1-3 Otro esquema de entrega de paquetes

El repartidor entrega sólo un paquete a la vez. Después de entregar un paquete, regresa a la tienda para recoger y entregar el segundo paquete. Bajo este esquema, la distancia total recorrida por el repartidor para entregar los paquetes y luego regresar a la tienda es:

$2 \cdot (1 + 2 + 3 + \ldots + 50) = 2550$ millas

Ahora suponga que hay n paquetes para entregar a n casas y que cada casa está a una milla de distancia de la otra, como se aprecia en la figura 1-1. Si los paquetes se entregan utilizando el primer esquema, la ecuación siguiente proporciona la distancia total recorrida:

$$1 + 1 + \ldots + 1 + n = 2n \tag{1-1}$$

Si los paquetes se entregan utilizando el segundo método, la distancia recorrida es:

$$2 \cdot (1 + 2 + 3 + \ldots + n) = 2 \cdot (n(n + 1)/2) = n^2 + n \tag{1-2}$$

En la ecuación (1-1), se dice que la distancia recorrida es una función de n. Considere la ecuación (1-2). En esta ecuación, para los valores grandes de n, encontraremos que el término conformado por n^2 se convertirá en el término dominante y el término que contiene a n será insignificante. En este caso, se dice que la distancia recorrida es una función de n^2. La tabla 1-1 evalúa las ecuaciones (1-1) y (1-2) para ciertos valores de n. (La tabla también muestra el valor de n^2.)

TABLA 1-1 Varios valores de n, $2n$, n^2 y $n^2 + n$

n	$2n$	n^2	$n^2 + n$
1	2	1	2
10	20	100	110
100	200	10,000	10,100
1000	2000	1,000,000	1,001,000
10,000	20,000	100,000,000	100,010,000

Cuando se analiza un algoritmo en particular, por lo general se cuenta el número de operaciones realizadas por el algoritmo. Nos concentramos en el número de operaciones, no en el tiempo de computadora real para ejecutar el algoritmo. Esto se debe a que un algoritmo particular puede implementarse en diversas computadoras y la rapidez de la computadora puede afectar el tiempo de ejecución. Sin embargo, el número de operaciones realizadas por el algoritmo sería el mismo en cada computadora. Piense en los ejemplos siguientes.

EJEMPLO 1-1

Considere el siguiente algoritmo. (Suponga que todas las variables se declararon correctamente.)

```
cout << "Especificar dos números";                    //Línea 1

cin >> num1 >> num2;                                  //Línea 2

if (num1 >= num2)                                     //Línea 3
    max = num1;                                       //Línea 4
else                                                  //Línea 5
    max = num2;                                       //Línea 6

cout << "El número máximo es: " << max << endl;       //Línea 7
```

La línea 1 tiene una operación, <<; la línea 2 tiene dos operaciones; la línea 3 tiene una operación, >=; la línea 4 tiene una operación, =; la línea 6 tiene una operación; y la línea 7 tiene tres operaciones. Se ejecuta ya sea la línea 4 o la línea 6. Por tanto, el número total de operaciones ejecutadas en el código anterior es 1 + 2 + 1 + 1 + 3 = 8. En este algoritmo, el número de operaciones ejecutadas es fijo.

CAPÍTULO 3

APUNTADORES Y LISTAS BASADAS EN ARREGLOS (ARRAYS)

EN ESTE CAPÍTULO USTED:

- Aprenderá acerca del tipo de datos apuntador y las variables apuntador
- Explorará cómo se declara y manipula un apuntador
- Aprenderá acerca de la dirección del operador y la desreferenciación
- Descubrirá las variables dinámicas
- Examinará cómo utilizar los operadores **`new`** y **`delete`** para manipular variables dinámicas
- Aprenderá acerca de la aritmética de apuntadores
- Descubrirá los arreglos dinámicos
- Se enterará de las copias de datos profunda y superficial
- Descubrirá las peculiaridades de las clases con miembros de datos de apuntador
- Explorará cómo se utilizan las arreglos dinámicos para procesar listas
- Aprenderá acerca de las funciones virtuales
- Se enterará de las clases abstractas

Los tipos de datos en C++ se clasifican en tres categorías: simples, estructurados y apuntadores. Hasta el momento, usted ha trabajado sólo con los dos primeros. Este capítulo estudia el tercer tipo de datos. Primero se explicará cómo se declaran las variables apuntador (o apuntadores) y se manipulan los datos a los cuales apuntan. Estos conceptos se utilizarán más adelante cuando se estudien los arreglos dinámicos y las listas vinculadas. Las listas vinculadas se estudian en el capítulo 5.

El tipo de datos apuntador y las variables apuntador

Los valores que pertenecen a los tipos de datos apuntador son direcciones de memoria de la computadora. Sin embargo, no existe un nombre asociado con el tipo de datos apuntador en C++. Debido a que el dominio (es decir, los valores de un tipo de datos apuntador), está compuesto por direcciones (ubicaciones o espacios en la memoria), una variable apuntador es una variable cuyo contenido es una dirección, es decir, una ubicación en la memoria.

Variable apuntador: una variable cuyo contenido es una dirección (es decir, una dirección de memoria).

Declaración de variables apuntador

El valor de una variable apuntador es una dirección. Esto significa que el valor hace referencia a otra ubicación en la memoria. Por lo general, los datos se almacenan en este espacio de memoria. Por consiguiente, cuando se declara una variable apuntador, también se especifica el tipo de datos del valor que se almacenará en la ubicación de memoria a la cual apunta la variable apuntador.

En C++, una variable apuntador se declara al utilizar el símbolo asterisco (`*`) entre el tipo de datos y el nombre de la variable. La sintaxis general para declarar una variable apuntador es la siguiente:

```
dataType *identifier;
```

Como ejemplo, considere las sentencias siguientes:

```
int *p;
char *ch;
```

En estas sentencias, tanto `p` como `ch` son variables apuntador. El contenido de `p` (cuando se asigna de manera apropiada) apunta a una ubicación en la memoria del tipo `int`, y el contenido de `ch` apunta a una ubicación en la memoria del tipo `char`. Por lo general, `p` se conoce como una variable apuntador del tipo `int`, y `ch` se denomina como una variable apuntador del tipo `char`.

Antes de estudiar cómo funcionan los apuntadores, reflexione sobre estas observaciones. Las sentencias siguientes que declaran que `p` es una variable apuntador del tipo `int` son equivalentes:

```
int *p;
int* p;
int * p;
```

Por tanto, el carácter * puede aparecer en cualquier parte entre el nombre del tipo de datos y el nombre de la variable.

Ahora considere la sentencia siguiente:

```
int* p, q;
```


En esta sentencia, sólo p es una variable apuntador, q no lo es. Aquí, q es una variable **int**. Para evitar confusiones, es preferible adjuntar el carácter * al nombre de la variable. Así que la sentencia anterior se escribe así:

```
int *p, q;
```

Desde luego, la sentencia

```
int *p, *q;
```

declara que tanto p como q son variables apuntador del tipo **int**.

Ahora que usted ya sabe cómo declarar apuntadores, se explicará cómo hacer que un apuntador apunte a una ubicación en la memoria y cómo manipular los datos almacenados en esos espacios de memoria.

Como el valor del apuntador es una dirección de memoria, un apuntador puede almacenar la dirección de un espacio de memoria del tipo designado. Por ejemplo, si p es un apuntador del tipo int, p puede almacenar la dirección de cualquier ubicación en la memoria del tipo **int**. C++ proporciona dos operadores, la dirección del operador (&) y el operador de desreferenciación (*), que funcionan con apuntadores. Las dos secciones siguientes estudian estos operadores.

Dirección del operador (&)

En C++, el símbolo & (ampersand), llamado **dirección del operador**, es un operador unitario que devuelve la dirección de su operando. Por ejemplo, dadas las sentencias

```
int x;
int *p;
```

la sentencia

```
p = &x;
```

asigna la dirección de x a p. Es decir, x y el valor de p hacen referencia a la misma ubicación en la memoria.

Operador de desreferenciación (*)

En los capítulos anteriores se utilizó el carácter asterisco, *, como el operador de multiplicación binario. C++ también utiliza el símbolo * como operador unitario. Cuando el asterisco, *, que comúnmente se conoce como **operador de desreferenciación** u **operador de indirección**,

se utiliza como cualquier operador unitario, hace referencia al objeto al cual apunta el operando de * (es decir, al apuntador). Por ejemplo, dadas las sentencias

```
int x = 25;
int *p;
p = &x;   //almacenar la dirección de x en p
```

la sentencia

```
cout << *p << endl;
```

imprime el valor almacenado en el espacio de memoria al cual apunta p, que es el valor de x. Asimismo, la sentencia

```
*p = 55;
```

almacena 55 en la ubicación de memoria a la cual apunta p, es decir, 55 se almacena en x.

El ejemplo 3-1 muestra cómo funciona una variable apuntador.

EJEMPLO 3-1

Considere las sentencias siguientes:

```
int *p;
int num;
```

En estas sentencias, p es una variable apuntador del tipo **int** y num es una variable del tipo **int**. Suponga que la ubicación 1200 de la memoria se asignó a p y la ubicación 1800 de la memoria se asignó a num. (Vea la figura 3-1.)

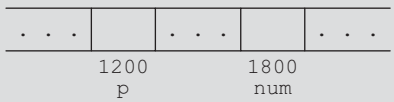
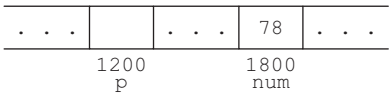
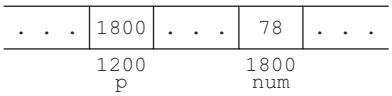
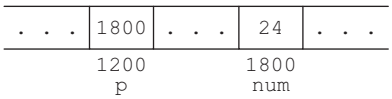


FIGURA 3-1 Las variables p y num

Considere las sentencias siguientes:

1. `num = 78;`
2. `p = &num;`
3. `*p = 24;`

A continuación se muestran los valores de las variables después de la ejecución de cada sentencia.

Después de la sentencia	Valores de las variables	Explicación
1		La sentencia `num = 78;` almacena `78` en `num`.
2		La sentencia `p = &num;` almacena la dirección de `num`, que es `1800`, en `p`.
3		La sentencia `*p = 24;` almacena `24` en la ubicación de memoria a la cual apunta `p`. Como el valor de `p` es `1800`, la sentencia 3 almacena `24` en la ubicación de memoria `1800`. Observe que el valor de `num` también cambió.

Hagamos un resumen del análisis anterior.

1. Una declaración como **`int`** `*p;` asigna memoria sólo a `p`, no a `*p`. Más adelante se explica cómo asignar memoria a `*p`.
2. El contenido de `p` apunta sólo a una ubicación de memoria del tipo **`int`**.
3. `&p`, `p` y `*p` tienen significados diferentes.
4. `&p` significa la dirección de `p`, es decir, `1200` (como se muestra en la figura 3-1).
5. `p` significa el contenido de `p`, que es `1800`, después de que se ejecuta la sentencia `p = &num;`.
6. `*p` significa el contenido de la ubicación de memoria a la cual apunta `p`. Observe que el valor de `*p` es `78` después de que se ejecuta la sentencia `p = &num;` el valor de `*p` es `24` después de que se ejecuta la sentencia `*p = 24`.

El programa del ejemplo 3-2 ilustra de una manera más amplia cómo funciona una variable apuntador.

EJEMPLO 3-2

```
//*********************************************************
// Autor: D.S. Malik
//
// Este programa ilustra cómo funciona una variable apuntador.
//*********************************************************

#include <iostream>                                    //Línea 1

using namespace std;                                   //Línea 2
```

```
int main()                                                    //Línea 3
{                                                             //Línea 4
    int *p;                                                   //Línea 5
    int num1 = 5;                                             //Línea 6
    int num2 = 8;                                             //Línea 7

    p = &num1; //almacenar la dirección de num1 en p; Línea 8

    cout << "Línea 9: &num1 = " << &num1
         << ", p = " << p << endl;                            //Línea 9
    cout << "Línea 10: num1 = " << num1
         << ", *p = " << *p << endl;                          //Línea 10

    *p = 10;                                                  //Línea 11
    cout << "Línea 12: num1 = " << num1
         << ", *p = " << *p << endl << endl;                  //Línea 12

    p = &num2; //almacenar la dirección de num2 en p; Línea 13

    cout << "Línea 14: &num2 = " << &num2
         << ", p = " << p << endl;                            //Línea 14
    cout << "Línea 15: num2 = " << num2
         << ", *p = " << *p << endl;                          //Línea 15

    *p = 2 * (*p);                                            //Línea 16
    cout << "Línea 17: num2 = " << num2
         << ", *p = " << *p << endl;                          //Línea 17

    return 0;                                                 //Línea 18
}                                                             //Línea 19
```

Corrida de ejemplo:

```
Línea 9: &num1 = 0012FF54, p = 0012FF54
Línea 10: num1 = 5, *p = 5
Línea 12: num1 = 10, *p = 10

Línea 14: &num2 = 0012FF48, p = 0012FF48
Línea 15: num2 = 8, *p = 8
Línea 17: num2 = 16, *p = 16
```

En su mayor parte, el resultado anterior es sencillo. Echemos un vistazo a algunas de estas sentencias. La sentencia de la línea 8 almacena la dirección de `num1` en `p`. La sentencia de la línea 9 produce la salida del valor de `&num1`, la dirección de `num1` y el valor de `p`. (Observe que los valores de salida de la línea 9 dependen de la máquina. Cuando ejecute este programa en su computadora, es probable que obtenga diferentes valores de `&num1` y `p`.) La sentencia de la línea 10 produce la salida del valor de `num1` y `*p`. Debido a que `p` apunta a la ubicación de memoria de `num1`, `*p` proporciona el valor de esta ubicación de memoria, es decir, `num1`. La sentencia de la línea 11 cambia el valor de `*p` a `10`. Como `p` apunta a la ubicación de memoria `num1`, el valor de `num1` también cambia. La sentencia de la línea 12 proporciona el valor de `num1` y `*p`.

La sentencia de la línea 13 almacena la dirección de `num2` en `p`. Por lo que después de la ejecución de esta sentencia, `p` apunta a `num2`. Así que cualquier cambio que ocurra en `*p` modifica

inmediatamente el valor de num2. La sentencia de la línea 14 produce la salida de la dirección de num2 y el valor de p. La sentencia de la línea 16 multiplica el valor de *p, que es el valor de num2, por 2, y almacena el valor nuevo en *p. Esta sentencia también cambia el valor de num2. La sentencia de la línea 17 produce la salida del valor de num2 y *p.

Apuntadores y clases

Considere las sentencias siguientes:

```
string *str;
str = new string;
*str = "Sunny Day";
```

La primera sentencia declara que str es una variable apuntador del tipo string. La segunda sentencia asigna memoria del tipo string y almacena la dirección de la memoria asignada en str. La tercera sentencia almacena la cadena "Sunny Day" en la memoria a la cual apunta str. Ahora suponga que se quiere utilizar la función de cadena length para encontrar la longitud de la cadena "Sunny Day". La sentencia (*str).length() devuelve la longitud de la cadena. Observe los paréntesis que encierran a *str. La expresión (*str).length() es una mezcla de una desreferenciación de apuntador y la selección del componente de clase. En C++, el operador punto, ., tiene una mayor precedencia que el operador de desreferenciación, *. Expliquemos esto con mayor detalle. En la expresión (*str).length(), el operador * se evalúa primero, por lo que la expresión *str se evalúa primero. Como str es una variable apuntador del tipo string, *str hace referencia a un espacio de memoria del tipo string. Por consiguiente, en la expresión (*str).length() se ejecuta la función length de la **clase** string. Ahora considere la expresión *str.length(). Veamos cómo se evalúa esta expresión. Debido a que . tiene una precedencia mayor que *, la expresión str.length() se evalúa primero. La expresión str.length() daría como resultado un error de sintaxis debido a que str *no* es un objeto string, así que no puede utilizar la función length de la clase string.

Como se aprecia, en la expresión (*str).length() son importantes los paréntesis que encierran a *str. No obstante, es inevitable cometer errores. Por esta razón, para simplificar el acceso de los componentes class o struct mediante un apuntador, C++ proporciona otro operador, llamado **operador flecha de acceso a miembros,** ->. El operador -> está compuesto por dos símbolos consecutivos: un guión y el símbolo "mayor que".

La sintaxis para tener acceso a un miembro de class (struct) utilizando el operador -> es la siguiente:

```
pointerVariableName->classMemberName
```

Por tanto, la expresión

```
(*str).length()
```

es equivalente a la expresión

```
str->length()
```

Al acceder a los componentes de `class` (`struct`) por medio de apuntadores utilizando el operador `->` se eliminan tanto el uso de paréntesis como del operador de desreferenciación. Debido a que los errores son inevitables y la falta de los paréntesis puede producir que el programa se termine de manera anómala o dé resultados erróneos, cuando se accede a los componentes de `class` (`struct`) por medio de apuntadores, este libro utiliza la notación de flecha.

Inicialización de variables apuntador

Como C++ no inicializa las variables en forma automática, las variables apuntador deben inicializarse cuando no se quiere que apunten a algo. Las variables apuntador se inicializan utilizando el valor constante `0`, llamado **apuntador nulo**. Por tanto, la sentencia `p = 0;` almacena el apuntador nulo en `p`, es decir, `p` apunta a nada. Algunos programadores utilizan la constante llamada `NULL` para inicializar las variables apuntador. Las dos sentencias siguientes son equivalentes:

```
p = NULL;
p = 0;
```

El número `0` es el único número que puede asignarse directamente a una variable apuntador.

Variables dinámicas

En las secciones previas, usted aprendió a declarar variables apuntador, a almacenar la dirección de una variable en una variable apuntador del mismo tipo que la variable, y a manipular los datos utilizando apuntadores. Sin embargo, aprendió cómo utilizar los apuntadores para manipular los datos sólo en espacios de memoria que se crearon utilizando otras variables. En otras palabras, los apuntadores manipularon los datos en espacios de memoria existentes. Así que, ¿cuál es el beneficio de utilizar apuntadores? Usted puede acceder a estos espacios de memoria al trabajar con las variables que se utilizaron para crearlos. En esta sección aprendió acerca del poder que conllevan los apuntadores. En particular a asignar y desasignar memoria durante la ejecución de programas utilizando apuntadores.

Las variables que se crean durante la ejecución de un programa se llaman **variables dinámicas**. Con la ayuda de apuntadores, C++ crea variables dinámicas. C++ proporciona dos operadores, `new` y **`delete`**, para crear y destruir variables dinámicas, respectivamente. Cuando un programa requiere una variable nueva, se utiliza el operador `new`. Cuando un programa ya no necesita una variable dinámica, se utiliza el operador `delete`.

En C++, **`new`** y **`delete`** son palabras reservadas.

Operador `new`

El operador **`new`** tiene dos formas: una para asignar una sola variable y otra para asignar un arreglo de variables. La sintaxis para utilizar el operador **`new`** es la siguiente:

```
new dataType;              //para asignar una sola variable
new dataType[intExp];      //para asignar un arreglo de variables
```

donde `intExp` es una expresión que produce un entero positivo al evaluarse.

El operador new asigna memoria (una variable) del tipo designado y devuelve un apuntador al mismo, es decir, la dirección de esta memoria asignada. Además, la memoria asignada no se inicializa.

Considere la declaración siguiente:

```
int *p;
char *q;
int x;
```

La sentencia

```
p = &x;
```

almacena la dirección de x en p. Sin embargo, no se asigna nueva memoria. Por otro lado, considere la sentencia siguiente:

```
p = new int;
```

Esta sentencia crea una variable durante la ejecución de un programa en alguna otra parte de la memoria, y almacena la dirección de la memoria asignada en p. El acceso a la memoria asignada es por medio de la desreferenciación de apuntadores, en concreto, de *p. Del mismo modo, la sentencia

```
q = new char[16];
```

crea un arreglo de 16 componentes del tipo char y almacena la dirección base del arreglo en q.

Debido a que una variable dinámica no tiene nombre, no se puede acceder directamente a ella. El acceso a la misma es indirecto, por medio del apuntador devuelto por **new**. Las sentencias siguientes ilustran este concepto:

```
int *p;    //p es un apuntador de tipo int

p = new int;   //asigna memoria de tipo int y almacena la dirección
               //de la memoria asignada en p
*p = 28;       //almacena 28 en la memoria asignada
```

NOTA

El operador **new** asigna una ubicación de memoria de un tipo específico y devuelve la dirección (inicial) del espacio de memoria asignado. Sin embargo, si el operador new es incapaz de asignar el espacio de memoria requerido (por ejemplo, no hay suficiente espacio de memoria), el programa podría terminar con un mensaje de error.

Operador `delete`

Suponga que tiene la declaración siguiente:

```
int *p;
```

Esta sentencia declara que p es una variable apuntador del tipo **int**. Ahora considere las sentencias siguientes:

```
p = new int;     //Línea 1
*p = 54;         //Línea 2
p = new int;     //Línea 3
*p = 73;         //Línea 4
```

ESTRUCTURAS DE DATOS CON C++ Segunda edición

D.S. Malik

Este libro representa la culminación de un proyecto desarrollado por su autor a lo largo de más de 25 años de exitosa enseñanza de programación y de estructuras de datos para estudiantes de ciencias de la computación.

CARACTERÍSTICAS DEL LIBRO

Las características del libro favorecen el aprendizaje autónomo. Los conceptos se presentan de principio a fin a un ritmo adecuado, lo cual permite al lector aprender con comodidad y confianza. El estilo de redacción de la obra es accesible y sencillo, similar al estilo de enseñanza en un aula:

+ Los *objetivos de aprendizaje* ofrecen un esquema de los conceptos de programación de C++ que se estudiarán a detalle dentro del capítulo.
+ Las *notas* resaltan los hechos importantes respecto a los conceptos presentados en el capítulo.
+ Los diagramas, amplios y exhaustivos, ilustran los conceptos complejos. El libro contiene más de 295 figuras.
+ Los *Ejemplos* numerados dentro de cada capítulo ilustran los conceptos clave con el código correspondiente.
+ Los *Ejemplos de programación* son programas que aparecen al final de cada capítulo y contienen las etapas precisas y concretas de Entrada, Salida, el Análisis de problemas, y el Algoritmo de diseño, así como un Listado de programas. Además, los problemas en estos ejemplos de programación se resuelven y programan mediante el uso de DOO.
+ El *Repaso rápido* ofrece un resumen de los conceptos estudiados en el capítulo.
+ Los *Ejercicios* refuerzan aún más el aprendizaje y aseguran que el lector ha aprendido el material.
+ Los *Ejercicios de programación* desafían al lector a escribir programas en C++ con un resultado específico.

CENGAGE Learning®

Visite nuestro sitio en http://latinoamerica.cengage.com

ISBN-13: 978-607481931-1
ISBN-10: 607481931-9

9 786074 819311