CAPÍTULO 1
Introducción

TEMAS

1.1 INTRODUCCIÓN A LA PROGRAMACIÓN

Una computadora es una máquina y, como otras máquinas, como un automóvil o una podadora, debe encenderse y luego conducirse, o controlarse, para hacer la tarea que se pretende realizar. En un automóvil, por ejemplo, el control es proporcionado por el conductor, quien se sienta en su interior y lo dirige. En una computadora, el conductor es un conjunto de instrucciones llamado programa. De manera más formal, un **programa de computadora** es un conjunto independiente de instrucciones usado para operar una computadora con el fin de producir un resultado específico. Otro término para un programa o conjunto de programas es **software**, y se usarán ambos términos de manera indistinta a través del texto.[1]

El proceso de escribir un programa, o software, se llama **programación**, mientras al conjunto que puede usarse para construir un programa se llama **lenguaje de programación**. Los lenguajes de programación disponibles se presentan en una variedad de formas y tipos.

Lenguaje de máquina

En su nivel más fundamental, los únicos programas que pueden usarse en realidad para operar una computadora son los **programas en lenguaje de máquina**. Tales programas, los cuales también se conocen como **programas ejecutables**, o **ejecutables** para abreviar, consisten en una secuencia de instrucciones compuestas por números binarios como:[2]

```
11000000 000000000001 000000000010
11110000 000000000010 000000000011
```

Estas instrucciones en lenguaje de máquina constan de dos partes: una de instrucción y una de dirección. La parte de instrucción, a la cual se conoce como **opcode** (abreviatura de "código de operación"), por lo general es el conjunto de bits en el extremo izquierdo de la instrucción y le indica a la computadora la operación a realizar, como sumar, restar, multiplicar, etc., mientras los bits en el extremo derecho especifican las direcciones de memoria de los datos que se van a usar. Por ejemplo, suponiendo que los ocho bits en el extremo izquierdo de la primera instrucción enlistada antes contienen el código de operación para sumar, y los siguientes dos grupos de doce bits son las direcciones de los dos operandos que se van a sumar, esta instrucción sería un comando para "sumar los datos en la ubicación 1 de la memoria a los datos en la ubicación 2 de la memoria". Del mismo modo, suponiendo que el opcode 11110000 significa multiplicar, la siguiente instrucción es un comando para "multiplicar los datos en la ubicación 2 de la memoria por los datos en la ubicación 3". (La sección 1.6 explica cómo convertir de números binarios a decimales.)

Lenguajes ensambladores

Debido a que cada clase de computadora, como IBM, Apple y Hewlett Packard, tiene su propio lenguaje de máquina particular, es muy tedioso y tardado escribir esos programas en lenguaje de máquina.[3] Uno de los primeros avances en la programación fue la sustitución de

[1] De una manera más incluyente, el término software también se usa para denotar tanto los programas como los datos con los que operarán los programas.

[2] Revise la sección 1.6 al final de este capítulo si no está familiarizado con los números binarios.

[3] En la actualidad, el lenguaje en el nivel de máquina está definido por el procesador alrededor del cual está construida la computadora.

símbolos en forma de palabras, como ADD, SUB, MUL, por los opcodes binarios y los números decimales y etiquetas por las direcciones en memoria. Por ejemplo, usando estos símbolos y valores decimales para las direcciones en memoria, las dos instrucciones en lenguaje de máquina anteriores pueden escribirse como:

```
ADD 1, 2
MUL 2, 3
```

Los lenguajes de programación que usan este tipo de notación simbólica se conocen como **lenguajes ensambladores**. Debido a que las computadoras sólo pueden ejecutar programas en lenguaje de máquina, el conjunto de instrucciones contenido dentro de un programa en lenguaje ensamblador debe traducirse a un programa de lenguaje de máquina antes que pueda ejecutarse en una computadora. Los programas traductores que realizan esta función para los programas en lenguaje ensamblador se conocen como **ensambladores** (véase la figura 1.1).

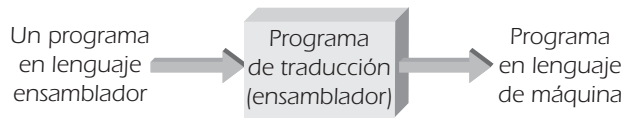


Figura 1.1 *Los programas ensambladores deben traducirse.*

Lenguajes de niveles bajo y alto

Tanto los lenguajes de máquina como los ensambladores se clasifican como **lenguajes de nivel bajo**. Esto se debe a que ambos tipos de lenguaje usan instrucciones que se vinculan en forma directa con un tipo de computadora. Como tal, un programa en lenguaje ensamblador está limitado porque sólo puede usarse con el tipo de computadora específica para el cual se escribió. Sin embargo, estos programas permiten usar las características especiales de un tipo de computadora particular y por lo general se ejecutan en el nivel más rápido posible.

En contraste con los lenguajes de nivel bajo están los lenguajes de alto nivel. Un **lenguaje de alto nivel** usa instrucciones que se parecen a los lenguajes escritos, como el inglés, y pueden ejecutarse en una variedad de tipos de computadora. Visual Basic, C, C++ y Java son ejemplos de lenguajes de alto nivel. Usando C++, una instrucción para sumar dos números y multiplicarlos por un tercer número puede escribirse como:

```
resultado = (primero + segundo) * tercero;
```

Los programas escritos en un lenguaje de computadora (de alto o bajo nivel) se conocen como **programas fuente** y **código fuente**. Una vez que se ha escrito un programa en un lenguaje de alto nivel también debe traducirse, como un programa ensamblador de bajo nivel, al lenguaje de máquina de la computadora en que se va a ejecutar. Esta traducción puede lograrse en dos formas.

Cuando cada declaración en un programa fuente de alto nivel es traducida de manera individual y ejecutada inmediatamente después de la traducción, el lenguaje de programación usado se llama **lenguaje interpretado** y el programa que hace la traducción se llama **intérprete**.

Cuando todas las instrucciones en un programa fuente de alto nivel son traducidas como una unidad completa antes que cualquier declaración sea ejecutada, el lenguaje de programa-

ción usado se llama **lenguaje compilado**. En este caso, el programa que hace la traducción se llama **compilador**. Pueden existir tanto versiones compiladas como interpretadas de un lenguaje, aunque de manera típica predomina una. C++ es predominantemente un lenguaje compilado.

La figura 1.2 ilustra la relación entre un código fuente de C++ y su compilación en un programa ejecutable en lenguaje de máquina. Como se muestra, el programa fuente se introduce usando un programa editor. Éste es en efecto un programa procesador de palabras que es parte del ambiente de desarrollo proporcionado por el compilador. Debe entenderse, sin embargo, que la introducción del código sólo puede comenzar después que una aplicación se ha analizado y comprendido en forma minuciosa y el diseño del programa ha sido planeado con cuidado. La forma en que se logra esto se explica en la siguiente sección.

La traducción del programa fuente C++ en un programa en lenguaje de máquina comienza con el compilador. La salida producida por el compilador se llama **programa objeto**, el cual es una versión en lenguaje de máquina del código fuente. En casi todos los casos, su código fuente usará código preprogramado existente, con código que ha escrito con anterioridad o código proporcionado por el compilador. Éste podría incluir código matemático para encontrar una raíz cuadrada, por ejemplo, o código que se está reutilizando de otra aplicación. Además, un programa C++ grande puede almacenarse en dos o más archivos de programa separados. En todos estos casos, este código adicional debe combinarse con el programa objeto antes que el programa pueda ejecutarse. Es tarea del **ligador** lograr este paso. El resultado del proceso de ligamiento es un programa en lenguaje de máquina completado, que contiene todo el código requerido por el programa, el cual está listo para su ejecución. El último paso en el proceso es cargar este programa en lenguaje de máquina en la memoria principal de su computadora para su ejecución real.

Orientaciones a procedimientos y a objetos

Además de clasificar los lenguajes de programación como de alto o bajo nivel, también se clasifican por su orientación a procedimientos u objetos. En un **lenguaje orientado a procedimientos** las instrucciones disponibles se usan para crear unidades independientes, conocidas como **procedimientos**. El propósito de un procedimiento es aceptar datos como entrada y transformarlos de alguna manera para producir un resultado específico como una salida. Hasta la década de los años 90 la mayor parte de los lenguajes de programación de alto nivel eran orientados a procedimientos.

En la actualidad, un segundo enfoque, la orientación a objetos, ha tomado el escenario central. Una de las motivaciones para **lenguajes orientados a objetos** fue el desarrollo de pantallas gráficas y soporte para las interfaces gráficas de usuario (GUI), capaces de desplegar múltiples ventanas que contienen tanto formas gráficas como texto. En tal ambiente, cada ventana en la pantalla puede considerarse un objeto con características asociadas, como color, posición y tamaño. Usando un enfoque orientado a objetos, un programa debe definir primero los objetos que manipulará, incluyendo una descripción de las características generales de los objetos y unidades específicas para manipularlos, como cambiar el tamaño y la posición y transferir datos entre objetos. Es de igual importancia que los lenguajes orientados a objetos tiendan a soportar la reutilización del código existente con más facilidad, lo cual elimina la necesidad de revalidar y reexaminar código nuevo o modificado. C++, el cual se clasifica como un lenguaje orientado a objetos, contiene características que se encuentran en los lenguajes orientados a procedimientos y a objetos. En este texto se diseñarán, desarrollarán y presentarán ambos tipos de código, que es la forma en que se escribe la mayor parte de los programas C++ actuales. Debido a que el código C++ orientado a objetos siempre contiene algún código

de procedimientos, y muchos programas C++ simples se escriben por completo usando sólo código de procedimientos, este tipo de código se presenta primero.

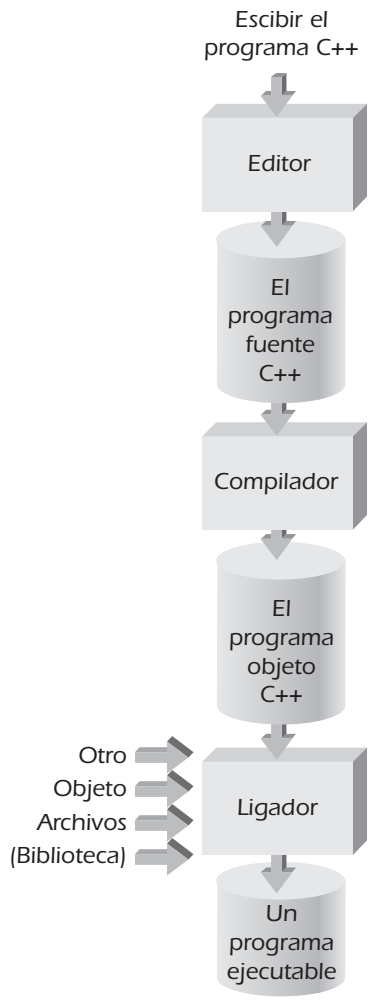


Figura 1.2 Creación de un programa C++ ejecutable.

Software de aplicación y de sistema

El software de aplicación y el software de sistema son dos categorías lógicas de programas de computadora. El **software de aplicación** consiste en aquellos programas escritos para realizar tareas particulares requeridas por los usuarios. Todos los programas en este libro son ejemplos de software de aplicación.

El **software de sistema** es la colección de programas que deben estar disponibles en cualquier sistema de cómputo en el que ha de operar. En los primeros entornos de cómputo de las décadas de los años 50 y 60, el usuario tenía que cargar al inicio el software de sistema en forma manual para preparar la computadora para que hiciera algo. Esto se llevaba a cabo usando hileras de conmutadores en un panel frontal. Se decía que aquellos comandos iniciales

introducidos en forma manual iniciaban (**boot**) la computadora, una expresión derivada de la expresión inglesa *pulling oneself up by the bootstraps* que significa "salir adelante sin ayuda". En la actualidad, el llamado cargador inicial (**bootstrap loader**) es un componente permanente que se ejecuta de manera automática desde el software del sistema de la computadora.

De manera colectiva, el conjunto de programas de sistema usados para operar y controlar una computadora se llama **sistema operativo**. Los sistemas operativos modernos incluyen las siguientes tareas: administración de memoria; asignación de tiempo de CPU; control de unidades de entrada y salida como teclado, pantalla e impresoras, y la administración de todos los dispositivos de almacenamiento secundarios. Muchos sistemas operativos manejan programas grandes y múltiples usuarios, en forma concurrente, dividiendo los programas en segmentos que son movidos entre el disco y la memoria conforme se necesita. Tales sistemas operativos permiten que más de un usuario ejecute un programa en la computadora, lo cual le da a cada usuario la impresión que la computadora y los periféricos son sólo suyos. Esto se conoce como un sistema **multiusuario**. Además, muchos sistemas operativos, incluyendo la mayor parte de los ambientes con ventanas, permiten a cada usuario ejecutar múltiples programas. Dichos sistemas operativos se conocen como sistemas **multiprogramados** y **multitareas**.

El desarrollo de C++

En un nivel básico, el propósito de casi todos los programas de aplicación es procesar datos para producir uno o más resultados específicos. En un lenguaje de procedimientos, un programa se construye a partir de conjuntos de instrucciones, con cada conjunto nombrado como un procedimiento, como se señaló con anterioridad. En efecto, cada procedimiento mueve los datos un paso más cerca de la salida final deseada a lo largo de la ruta mostrada en la figura 1.3.

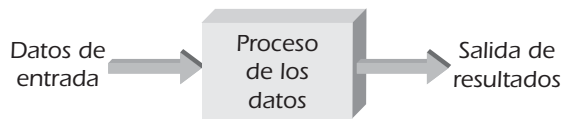


Figura 1.3 Operaciones de procedimiento básicas.

El proceso de programación ilustrado en la figura 1.3 refleja en forma directa las unidades de hardware de entrada, procesamiento y salida usadas para construir una computadora (véase la sección 1.6). Esto no fue accidental porque los primeros lenguajes de programación fueron diseñados de manera específica para corresponder y controlar en forma directa, lo más óptimamente posible, a las unidades de hardware apropiadas.

El primer lenguaje de procedimientos, llamado FORTRAN, cuyo nombre se deriva de *FOR*mula *TRAN*slation, fue introducido en 1957 y siguió siendo popular durante la década de los años 60 y principios de la década de los años 70. (Otro lenguaje de programación de nivel alto desarrollado en forma casi concurrente con FORTRAN, pero que nunca logró la aceptación abrumadora de FORTRAN, fue nombrado ALGOL.) FORTRAN tiene instrucciones tipo álgebra que se concentran en la fase de procesamiento mostrada en la figura 1.3 y fue desarrollado para aplicaciones científicas y de ingeniería que requerían salidas numéricas de gran precisión, incluyendo muchos lugares decimales. Por ejemplo, calcular la trayectoria de un cohete o el nivel de concentración bacteriana en un estanque contaminado, como se ilustra en la figura 1.4, requiere evaluar una ecuación matemática a un alto grado de precisión numérica y es típico de las aplicaciones basadas en FORTRAN.

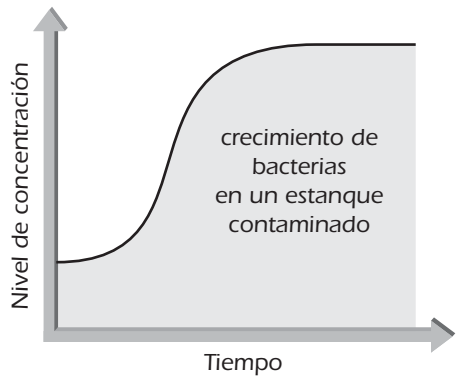


Figure 1.4 *FORTRAN fue desarrollado para aplicaciones científicas y de ingeniería.*

En orden de aparición, el siguiente lenguaje de aplicación de nivel alto significativo fue COBOL, el cual fue introducido en la década de los años 60 y permaneció como un lenguaje de procedimientos importante hasta la década de los años 80. La palabra COBOL se formó por las siglas de COmmon Business-Oriented Language. Este lenguaje tenía características enfocadas hacia aplicaciones de negocios que requerían cálculos matemáticos más simples que los necesarios para aplicaciones de ingeniería. Uno de los beneficios más notables de COBOL fue que proporcionaba formatos de salida extensos que facilitaban la creación de informes que contenían muchas columnas de números y totales en dólares y centavos formateados con esmero, como se ilustra en la figura 1.5. Esto obligó a los programadores a construir procedimientos estructurados bien definidos que seguían un patrón más consistente que el requerido por FORTRAN.

No. de parte	Descripción	Cantidad	Precio
12225	#4 Clavos, normales	25 cajas	1.09
12226	#6 Clavos, normales	30 cajas	1.09
12227	#8 Clavos, normales	65 cajas	1.29
12228	#10 Clavos, normales	57 cajas	1.35
12229	#12 Clavos, normales	42 cajas	
12230	#16 Clavos, normales		

Figure 1.5 *COBOL fue desarrollado para aplicaciones de negocios.*

Otro lenguaje, BASIC (o Beginners All-purpose Symbolic Instruction Code), fue desarrollado en Dartmouth College más o menos al mismo tiempo que COBOL. BASIC era en esencia una versión ligeramente reducida de FORTRAN y pretendía ser un lenguaje introductorio para estudiantes universitarios. Era un lenguaje relativamente sencillo, fácil de entender, que no requería un conocimiento detallado de una aplicación específica. Su principal desventaja era que no requería ni imponía un enfoque consistente o estructurado para crear programas. Con frecuencia, el programador no podía comprender con facilidad qué hacía su programa BASIC después de un tiempo breve.

Para remediar esto y adecuar la programación a una base más científica y racional que hiciera más fácil entender y reutilizar el código, se desarrolló el lenguaje Pascal. (Pascal no es una sigla, sino que se le puso este nombre en honor al matemático del siglo XVII Blaise Pascal.) Introducido en 1971, proporcionó a los estudiantes un fundamento más firme en el diseño de programación estructurada que lo aportado por versiones anteriores de BASIC.

Los programas estructurados se crean usando un conjunto de estructuras bien definidas organizadas en secciones de programación individuales, cada una de las cuales ejecuta una tarea específica que puede probarse y modificarse sin perturbar otras secciones del programa. Sin embargo, el lenguaje Pascal estaba estructurado en forma tan rígida que no existían escapes de las secciones estructuradas cuando hubieran sido útiles. Esto era una limitante para muchos proyectos del mundo real y es una de las razones por las que Pascal no fue aceptado en forma amplia en los campos científico y de ingeniería. En cambio, el lenguaje C, el cual es un lenguaje de procedimientos estructurado desarrollado en la década de los años 70 en AT&T Bell Laboratories por Ken Thompson, Dennis Ritchie y Brian Kernighan, se convirtió en el lenguaje para aplicaciones de ingeniería dominante de la década de los años 80. Este lenguaje tiene un amplio conjunto de capacidades que permite que se escriba como un lenguaje de nivel alto mientras conserva la capacidad de acceso directo a las características del nivel de máquina de una computadora.

C++ fue desarrollado a principios de la década de los años 80, cuando Bjarne Stroustrup (también en AT&T) usó sus conocimientos en lenguaje de simulación para crear un lenguaje de programación orientado a objetos. Una característica central de los lenguajes de simulación es que modelan situaciones de la vida real como objetos. Esta orientación a objetos, la cual era ideal para objetos gráficos presentados en pantalla como rectángulos y círculos, se combinó con características de C, existentes para formar el lenguaje C++. Por tanto, C++ conservó el conjunto extenso de capacidades estructuradas y de procedimientos proporcionadas por C, pero agregó su propia orientación a objetos para convertirse en un verdadero lenguaje de programación de uso general. Como tal, C++ puede usarse desde programas interactivos simples, hasta programas de ingeniería y científicos sofisticados y complejos, dentro del contexto de una estructura en verdad orientada a objetos.

Ejercicios 1.1

1. Defina los siguientes términos:
   - **a.** programa de computadora
   - **b.** programación
   - **c.** lenguaje de programación
   - **d.** lenguaje de alto nivel
   - **e.** lenguaje de bajo nivel
   - **f.** lenguaje de máquina
   - **g.** lenguaje ensamblador
   - **h.** lenguaje orientado a procedimientos
   - **i.** lenguaje orientado a objetos
   - **j.** programa fuente
   - **k.** compilador
   - **l.** intérprete
2. Describa el propósito y usos principales del software de aplicación y de sistema.

3. **a.** Describa la diferencia entre lenguajes de alto y bajo nivel.
b. Describa la diferencia entre lenguajes orientados a procedimientos y a objetos.

4. Describa las semejanzas y diferencias entre ensambladores, intérpretes y compiladores.

5. **a.** Dados los siguientes códigos de operación,

```
11000000 significa sumar el 1er. operando al 2o. operando
10100000 significa restar el 1er. operando del 2o. operando
11110000 significa multiplicar el 2o. operando por el 1er. operando
11010000 significa dividir el 2o. operando entre el 1er. operando
```

traduzca las siguientes instrucciones al español:

opcode	Dirección del 1er. operando	Dirección del 2o. operando
11000000	000000000001	0000000000010
11110000	000000000010	0000000000011
10100000	000000000100	0000000000011
11010000	000000000101	0000000000011

b. Suponiendo que las siguientes ubicaciones contienen los datos proporcionados, determine el resultado producido por las instrucciones listadas en el ejercicio 5a. Para este ejercicio, suponga que cada instrucción es ejecutada de manera independiente de cualquier instrucción.

Dirección	Valor inicial (en decimales) almacenado en esta dirección
00000000001	5
00000000010	3
00000000011	6
00000000100	14
00000000101	4

6. Reescriba las instrucciones en el nivel de máquina enlistadas en el ejercicio 5a usando notación de lenguaje ensamblador. Use los nombres simbólicos ADD, SUB, MUL y DIV para operaciones de adición, sustracción, multiplicación y división, respectivamente. Al escribir las instrucciones use valores decimales para las direcciones.

1.2 Solución de problemas y desarrollo de software

Sin importar cuál campo de trabajo elija o cuál pueda ser su estilo de vida, tendrá que resolver problemas. Muchos de éstos, como sumar el cambio en su bolsillo, pueden resolverse rápido y fácil. Otros, como montar en bicicleta, requieren algo de práctica pero pronto se vuelven automáticos. Otros más requieren de una planeación y premeditación considerables para que la solución sea apropiada y eficiente. Por ejemplo, construir una red telefónica celular o crear un sistema de administración de inventarios para un gran almacén son problemas para los cuales las soluciones por ensayo y error podrían resultar costosas y desastrosas.

Crear un programa no es diferente porque un programa es una solución desarrollada para resolver un problema particular. Por ello, escribir un programa casi es el último paso en un proceso de determinar primero cuál es el problema y el método que se usará para resolverlo. Cada campo de estudio tiene su propio nombre para el método sistemático usado para resolver problemas mediante el diseño de soluciones adecuadas. En las ciencias y la ingeniería el enfoque se conoce como el **método científico**, mientras en el análisis cuantitativo el enfoque se denomina **enfoque de sistemas**.

El método usado por los profesionales que desarrollan software para entender el problema que se va a solucionar y para crear una solución de software efectiva y apropiada se llama **procedimiento de desarrollo de software**. Este procedimiento, como se ilustra en la figura 1.6, consiste en tres fases que se superponen:

- Diseño y desarrollo
- Documentación
- Mantenimiento

Como disciplina, la **ingeniería de software** se encarga de crear programas y sistemas legibles, eficientes, confiables y mantenibles, utilizando el procedimiento de desarrollo de software para lograr esta meta.

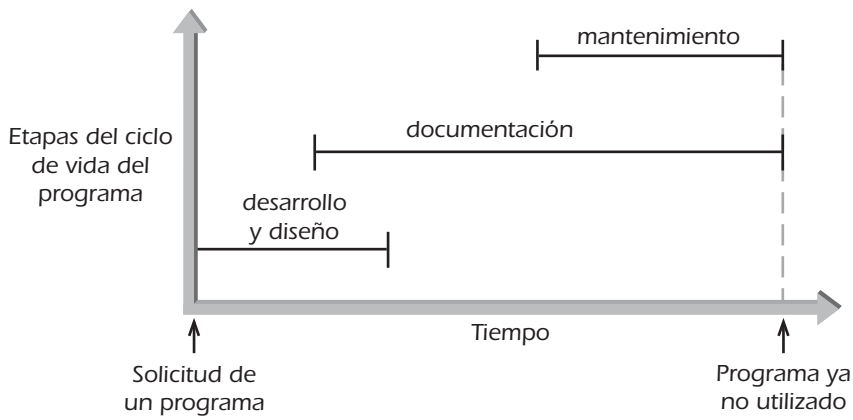


Figura 1.6 *Las tres fases del desarrollo de programas.*

Fase I. Desarrollo y diseño

La fase I comienza con el planteamiento de un problema o con una solicitud específica de un programa, lo cual se conoce como **requerimiento de programa**. Una vez que se ha planteado un problema o se ha hecho una solicitud específica para un programa, comienza la fase de diseño y desarrollo. Esta fase consta de cuatro pasos bien definidos, como se ilustra en la figura 1.7 y se resume a continuación.

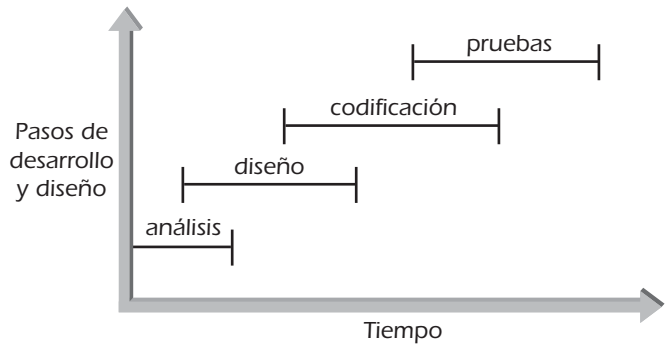


Figura 1.7 Los pasos de diseño y desarrollo.

Paso 1 Analizar el problema

Este paso es necesario para asegurar que el problema está definido y se entiende con claridad. La determinación de que el problema está definido en forma clara se hace sólo después que quien realiza el análisis entiende qué salidas se requieren y qué entradas se necesitarán. Para lograr esto el analista debe tener una comprensión de la forma en que se pueden usar las entradas para producir la salida deseada. Por ejemplo, suponga que recibe la siguiente tarea:

> ***Escriba un programa que nos proporcione la información que necesitamos sobre los círculos. Termínelo para mañana.***
>
> ***— La gerencia***

Un análisis simple de esta solicitud de programa revela que no es un problema bien definido en absoluto, porque no sabemos con exactitud qué información de salida se requiere. Por ello, sería un error enorme comenzar de inmediato a escribir un programa para solucionarlo. Para aclarar y definir el planteamiento del problema, su primer paso deberá ser ponerse en contacto con "La gerencia" para definir con exactitud qué va a producir el programa (sus salidas). Suponga que hizo esto y se enteró que lo que en realidad se deseaba es un programa para calcular y mostrar la circunferencia de un círculo cuando se da el radio. Debido a que existe una fórmula para convertir la entrada en la salida, puede proceder al siguiente paso. Si no se está seguro de cómo obtener la salida requerida o exactamente cuáles entradas se necesitan, se requiere un análisis más profundo. Esto de manera típica significa obtener más información antecedente acerca del problema o aplicación. Con frecuencia también implica hacer uno o más cálculos manuales para asegurar que se entiende qué entradas son necesarias y cómo deben combinarse para lograr la salida deseada.

Innumerables horas se han dedicado a escribir programas de computadora que nunca se han usado o han causado una animosidad considerable entre el programador y el usuario debido a que el programador no produjo lo que el usuario necesitaba o esperaba. Los programadores exitosos entienden y evitan esto al asegurarse que entienden los requerimientos del problema. Éste es el primer paso en la creación de un programa y el más importante,

porque en él se determinan las especificaciones para la solución final del programa. Si los requerimientos no son entendidos por completo antes que comience la programación, los resultados casi siempre son desastrosos.

Por ejemplo, imagine diseñar y construir una casa sin entender por completo las especificaciones del propietario. Después que se ha terminado, el propietario le dice que se requería un baño en el primer piso, donde usted ha construido una pared entre la cocina y el comedor. Además, esa pared en particular es una de las paredes de soporte principales para la casa y contiene numerosas tuberías y cables eléctricos. En este caso, agregar un baño requiere una modificación bastante importante a la estructura básica de la casa.

Los programadores experimentados entienden la importancia de analizar y comprender los requerimientos del programa antes de codificarlo, en especial si también han elaborado programas que después han tenido que desmantelarse y rehacerse por completo. La clave del éxito aquí, la cual a fin de cuentas determina el éxito del programa final, es determinar el propósito principal del sistema visto por la persona que hace la solicitud. Para sistemas grandes, el análisis por lo general es realizado por un analista de sistemas. Para sistemas más pequeños o programas individuales, el análisis de manera típica se lleva a cabo en forma directa por el programador.

Sin tener en cuenta cómo se hizo el análisis, o por quién, al concluirlo deberá haber una comprensión clara de:

- Qué debe hacer el sistema o programa
- Qué salidas debe producir
- Qué entradas se requieren para crear las salidas deseadas

Paso 2 Desarrollar una solución

En este paso, se selecciona el conjunto exacto de pasos, llamado algoritmo, que se usará para resolver el problema. La solución se obtiene de manera típica por una serie de refinamientos, comenzando con el algoritmo inicial encontrado en el paso de análisis, hasta que se obtenga un algoritmo aceptable y completo. Este algoritmo debe verificarse, si no se ha hecho en el paso de análisis, para asegurar que produce en forma correcta las salidas deseadas. Por lo general la verificación se hace realizando uno o más cálculos manuales que no se han hecho.

Para programas pequeños el algoritmo seleccionado puede ser en extremo simple y consistir de sólo uno o más cálculos. De manera más general, la solución inicial debe refinarse y organizarse en subsistemas más pequeños, con especificaciones sobre la forma en que los subsistemas interactuarán entre sí. Para lograr este objetivo, la descripción del algoritmo comienza desde el requerimiento de nivel más alto (superior) y procede en forma descendente a las partes que deben elaborarse para lograr este requerimiento. Para hacer esto más significativo, considere un programa de computadora para dar seguimiento al número de partes en un inventario. La salida requerida para este programa es una descripción de todas las partes que se llevan en el inventario y el número de unidades de cada artículo en existencia; las entradas dadas son la cantidad inicial en inventario de cada parte, el número de artículos vendidos, el número de artículos devueltos y el número de artículos comprados.

Para estas especificaciones, el diseñador podría organizar al principio los requerimientos para el programa en las tres secciones ilustradas en la figura 1.8. Esto se llama ***diagrama de estructura de primer nivel*** porque representa la primera estructura general del programa seleccionado por el diseñador.

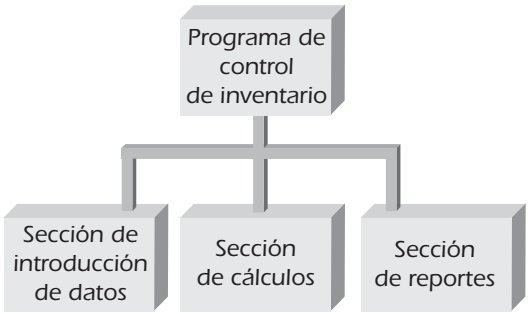


Figura 1.8 *Diagrama de estructura de primer nivel.*

Una vez que se ha desarrollado una estructura inicial, se refina hasta que las tareas indicadas en los cuadros están definidas por completo. Por ejemplo, tanto los módulos de introducción de datos como de reportes que se muestran en la figura 1.8 deberían refinarse más. El módulo de introducción de datos por supuesto debe incluir provisiones para introducir los datos. Debido a que es responsabilidad del diseñador del sistema planear las contingencias y el error humano, también deben tomarse provisiones para cambiar datos incorrectos después que se ha hecho una entrada y para eliminar por completo un valor introducido con anterioridad. También pueden hacerse subdivisiones similares para el módulo de reportes. La figura 1.9 ilustra un diagrama de estructura de segundo nivel para un sistema de seguimiento de inventario que incluye estos refinamientos adicionales.

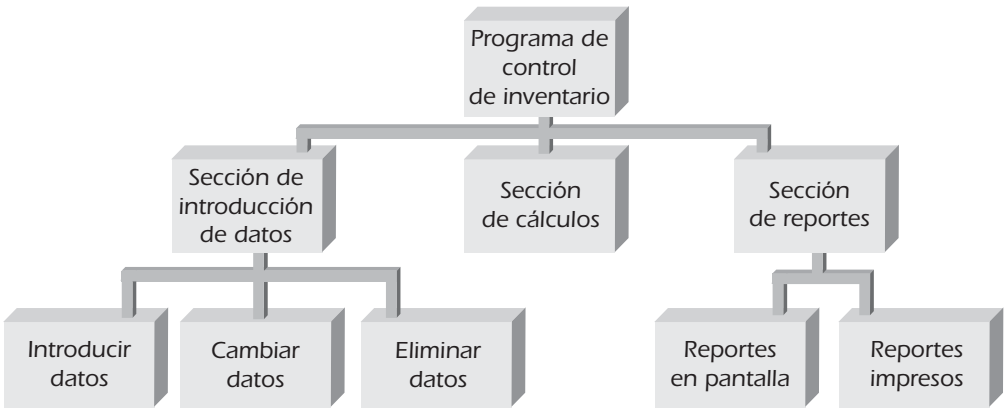


Figura 1.9 *Diagrama de una estructura refinada de segundo nivel.*

El proceso de refinar una solución continúa hasta que el requerimiento más pequeño se ha incluido dentro de la solución. Note que el diseño produce una estructura en forma de árbol, donde los niveles se ramifican conforme pasamos de la parte superior de la estructura a la parte inferior. Cuando el diseño está completo cada tarea designada en un cuadro es codificada, por lo general, en conjuntos separados de instrucciones que se ejecutan cuando son llamadas por tareas superiores en la estructura.

Paso 3 Codificar la solución

Este paso, el cual también se conoce como escribir el programa y poner en práctica la solución, consiste en traducir la solución de diseño elegida en un programa de computadora. Si los pasos de análisis y solución se han realizado en forma correcta, el paso de codificación se vuelve bastante mecánico. En un programa bien diseñado, los planteamientos que forman el programa se conformarán, sin embargo, con ciertos patrones o estructuras bien definidos en el paso de solución. Estas estructuras controlan la forma en que el programa se ejecuta y consiste en los siguientes tipos:

1. Secuencia
2. Selección
3. Iteración
4. Invocación

La **secuencia** define el orden en que son ejecutadas las instrucciones por el programa. La especificación de cuál instrucción entra primero, cuál en segundo lugar, etc., es esencial si el programa ha de lograr un propósito bien definido.

La **selección** proporciona la capacidad para hacer una elección entre diferentes operaciones, dependiendo del resultado de alguna condición. Por ejemplo, el valor de un número puede comprobarse antes que una divisións sea realizada. Si el número no es cero, puede usarse como el denominador de una operación de división; de lo contrario, la división no se ejecutará y se mostrará al usuario un mensaje de advertencia.

La **iteración**, la cual también se denomina *bucle*, *ciclo* o *repetición*, proporciona la capacidad para que la misma operación se repita con base en el valor de una condición. Por ejemplo, podrían introducirse y sumarse grados de manera repetida hasta que un grado negativo sea introducido. Ésta sería la condición que significa el fin de la entrada y adición repetitiva de grados. En ese punto podría ejecutarse el cálculo de un promedio para todos los grados introducidos.

La **invocación** implica invocar, o solicitar, un conjunto de instrucciones cuando sea necesario. Por ejemplo, el cálculo del pago neto de una persona implica las tareas de obtener las tarifas de salario y las horas trabajadas, calcular el pago neto y proporcionar un reporte o cheque por la cantidad requerida. Por lo general una de estas tareas individuales se codificarían como unidades separadas que son llamadas a ejecución, o invocadas, según se necesiten.

Paso 4 Probar y corregir el programa

El propósito de probar es verificar que el programa funciona en forma correcta y en realidad cumple con sus requerimientos. En teoría, las pruebas revelarían todos los errores del programa. (En terminología de computación, un error de programa se conoce como **bug**.[4]) En la práctica, esto requeriría comprobar todas las combinaciones posibles de ejecución de las instrucciones. Debido al tiempo y al esfuerzo requeridos, esto por lo general es una meta imposible, excepto para programas simples en extremo. (En la sección 4.8 se ilustra por qué por lo general ésta es una meta imposible.)

[4]La derivación de este término es bastante interesante. Cuando un programa dejó de ejecutarse en la Mark I, en la Universidad de Harvard, en septiembre de 1945, Grace Hopper rastreó el mal funcionamiento hasta llega a un insecto muerto que había entrado en los circuitos eléctricos. Registró el incidente en su bitácora a las 15:45 horas como "Interruptor #70. . . (polilla) en el interruptor. Primer caso real de bug (insecto) encontrado".

Debido a que no es posible realizar pruebas exhaustivas para la mayor parte de los problemas, han evolucionado diferentes filosofías y métodos de prueba. En su nivel más básico, sin embargo, la prueba requiere de un esfuerzo consciente para asegurarse que un programa funciona en forma correcta y produce resultados significativos. Esto quiere decir que debe meditarse con cuidado lo que se pretende lograr con la prueba y los datos que se usarán en la misma. Si la prueba revela un error (bug), puede iniciarse el proceso de depurar, el cual incluye localizar, corregir y verificar la corrección. Es importante percatarse que *aunque la prueba puede revelar la presencia de un error, no necesariamente indica la ausencia de uno*. Por tanto, *el hecho que una prueba revele un error no indica que otro no esté al acecho en algún otro lugar del programa*.

Para detectar y corregir errores en un programa es importante desarrollar un conjunto de datos de prueba por medio de los cuales determinar si el programa proporciona respuestas correctas. De hecho, un paso comúnmente aceptado en el desarrollo de software muchas veces incluye planear los procedimientos de prueba y crear datos de prueba significativos antes de escribir el código. Esto ayuda a ser más objetivo respecto a lo que debe hacer el programa debido a que en esencia elude cualquier tentación subconsciente después de codificar datos de prueba que no funcionarán. Los procedimientos para probar un programa deberán examinar toda las situaciónes posibles bajo las que se usará el programa. El programa deberá probarse con datos en un rango razonable, al igual que dentro de los límites y en áreas donde el programa debería indicar al usuario que los datos son inválidos. Desarrollar buenos procedimientos y datos de prueba para problemas complejos puede ser más difícil que escribir el código del programa en sí.

La tabla 1.1. enumera la cantidad relativa de esfuerzo que por lo general se dedica en cada uno de estos cuatro pasos de desarrollo y diseño en proyectos de programación comercial grandes. Como muestra este listado, la codificación no es el mayor esfuerzo en esta fase. Muchos programadores novatos tienen problemas debido a que dedicaron la mayor parte de su tiempo a escribir el programa, sin entender por completo el problema o diseñar una solución apropiada. En este aspecto, vale la pena recordar el proverbio de programación, "*Es imposible escribir un programa exitoso para un problema o aplicación que no se ha comprendido por completo*". Un proverbio equivalente e igual de valioso es "*Entre más pronto se comienza a codificar un programa, por lo general tomará más tiempo completarlo*".

Tabla 1.1 Esfuerzo dedicado a la fase I

Paso	Esfuerzo
Analizar el problema	10%
Desarrollar una solución	20%
Codificar la solución	20%
Probar el programa	50%

Fase II. Documentación

Una gran cantidad de trabajo se vuelve inútil o se pierde y deben repetirse demasiadas tareas debido a una documentación inadecuada, por lo que se puede concluir que documentar el trabajo es uno de los pasos más importantes en la solución de problemas. En realidad, durante los pasos de análisis, diseño, codificación y prueba se crean muchos documentos esen-

ciales. Completar la documentación requiere recopilar estos documentos, agregar material práctico para el usuario y presentarlo en una forma que sea de la mayor utilidad.

Aunque no es unánime la clasificación, en esencia existen cinco documentos para toda solución de problema:

1. Descripción del programa
2. Desarrollo y cambios del algoritmo
3. Listado del programa bien comentado
4. Muestras de las pruebas efectuadas
5. Manual del usuario

"Ponerse en los zapatos" de un integrante del equipo de una empresa grande que podría ser el usuario de su trabajo, desde la secretaria hasta el programador, analistas y la gerencia, debería ayudarle a hacer claro el contenido de la documentación importante. La fase de documentación comienza de manera formal en la fase de desarrollo y diseño y continúa hasta la fase de mantenimiento.

Fase III. Mantenimiento

Esta fase tiene que ver con la corrección continua de problemas, revisiones para satisfacer necesidades cambiantes y la adición de características nuevas. El mantenimiento con frecuencia es el esfuerzo mayor, la fuente principal de ingresos y la más duradera de las fases de ingeniería. Mientras el desarrollo puede tomar días o meses, el mantenimiento puede continuar por años o décadas. Entre más completa es la documentación, el mantenimiento podrá efectuarse de manera más eficiente y el cliente y el usuario serán más felices.

Respaldo

Aunque no es parte del proceso de diseño formal, es esencial hacer y conservar copias de respaldo del programa en cada paso del proceso de programación y depuración. Es fácil eliminar o cambiar la versión de trabajo actual de un programa más allá del reconocimiento. Las copias de respaldo permiten la recuperación de la última etapa de trabajo con un esfuerzo mínimo. La versión de trabajo final de un programa útil deberá respaldarse al menos dos veces. A este respecto, otro proverbio de programación útil es "*El respaldo no es importante si no le importa empezar todo de nuevo*".

Muchas empresas conservan al menos un respaldo en el sitio, donde pueda recuperarse con facilidad, y otra copia de respaldo ya sea en una caja fuerte a prueba de fuego o en una ubicación remota.

Ejercicios 1.2

1. **a.** Enumere y describa los cuatro pasos requeridos en la etapa de diseño y desarrollo de un programa.
   b. Además de la etapa de diseño y desarrollo, ¿cuáles son las otras dos etapas requeridas para producir un programa y por qué son necesarias?

2. Una nota de su supervisor, el señor J. Bosworth, dice:

   Solucione nuestros problemas de iluminación.

   —J. Bosworth

   a. ¿Cuál debería ser su primera tarea?
   b. Cómo se llevaría a cabo esta tarea?
   c. ¿Cuánto tiempo espera que tome esta tarea, suponiendo que todos cooperen?

3. El desarrollo del programa es sólo una fase en el procedimiento de desarrollo de software general. Asumiendo que la documentación y el mantenimiento requieren 60% del esfuerzo de software total en el diseño de un sistema, y usando la tabla 1.1, determine la cantidad de esfuerzo requerido para la codificación del programa inicial como un porcentaje del esfuerzo de software total.

4. Muchas personas que solicitan un programa o sistema por primera vez consideran que la codificación es el aspecto más importante del desarrollo del programa. Sienten que saben lo que necesitan y piensan que el programador puede comenzar a codificar con un tiempo mínimo dedicado al análisis. Como programador, ¿qué dificultades puede prever al trabajar en esas condiciones?

5. Muchos usuarios novatos tratan de contratar a los programadores por una cuota fija (la cantidad total que se va a pagar se fija con anticipación). ¿Cuál es la ventaja para el usuario al hacer este arreglo? ¿Cuál es la ventaja para el programador al hacer este arreglo? ¿Cuáles son algunas desventajas tanto para el usuario como para el programador en este arreglo?

6. Muchos programadores prefieren trabajar con una tarifa por hora. ¿Por qué piensa que esto es así? ¿Bajo qué condiciones sería ventajoso para un programador darle a un cliente un precio fijo por el esfuerzo de programación?

7. Los usuarios experimentados por lo general desean una descripción redactada con claridad del trabajo de programación que se hará, incluyendo una descripción completa de lo que hará el programa, fechas de entrega, calendarios de pago y requerimientos de prueba. ¿Cuál es la ventaja para el usuario al requerir esto? ¿Cuál es la ventaja para el programador al trabajar bajo este acuerdo? ¿Qué desventajas tiene este acuerdo tanto para el usuario como para el programador?

1.3 ALGORITMOS

Antes que se escriba un programa, el programador debe entender con claridad qué datos van a usarse, el resultado deseado y el procedimiento que va a utilizarse para producir este resultado. El procedimiento, o solución, seleccionado se conoce como algoritmo. Con más precisión, un **algoritmo** se define como una secuencia paso a paso de instrucciones que deben realizarse y describe cómo han de procesarse los datos para producir las salidas deseadas. En esencia, un algoritmo responde la pregunta: "¿Qué método se usará para resolver este problema?".

Sólo después de entender con claridad los datos que se usarán y seleccionar un algoritmo (los pasos específicos requeridos para producir el resultado deseado) podemos codificar

el programa. Vista bajo esta luz, la programación es la traducción de un algoritmo seleccionado a un lenguaje que pueda usar la computadora.

Para ilustrar un algoritmo, se considerará un problema simple. Suponga que un programa debe calcular la suma de todos los números enteros del 1 al 100. La figura 1.10 ilustra tres métodos que podrían usarse para encontrar la suma requerida. Cada método constituye un algoritmo.

Es evidente que la mayoría de las personas no se molestaría en enumerar las posibles alternativas en una manera paso por paso detallada, como lo hemos hecho aquí, y luego seleccionar uno de los algoritmos para solucionar el problema. Pero claro, la mayoría de las personas no piensa en forma algorítmica; tiende a pensar de manera heurística.

Método 1 - Columnas: *Ordenar los números del 1 al 100 en una columna y sumarlos*

```
   1
   2
   3
   4
   .
   .
   .
  98
  99
+100
----
5050
```

Método 2 - Grupos: *Ordenar los números en grupos que sumen 101 y multiplicar el número de grupos por 101*

```
1+100=101
 2+99=101
 3+98=101
 4+97=101   } 50 grupos
   .  .     }        ↓
   .  .     }   (50x101=5050)
49+52=101   }
50+51=101   }
```

Método 3 - Fórmula: *Usar la fórmula*

$$\text{suma} = \frac{n(a+b)}{2}$$

donde

n = número de términos que se van a sumar (100)
a = primer número que será sumado (1)
b = último número que será sumado (100)

$$\text{suma} = \frac{100(1+100)}{2} = 5050$$

Figura 1.10 Sumar los números del 1 al 100.

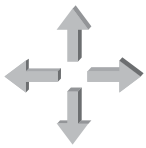
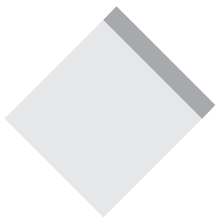
Símbolo	Nombre	Descripción
	Terminal	Indica el principio o fin de un programa
	Entrada/Salida	Indica una operación de entrada o salida
	Proceso	Indica cálculo o manipulación de datos
	Líneas de flujo	Usadas para conectar los otros símbolos del diagrama de flujo e indica el flujo lógico
	Decisión	Indica un punto de ramificación del programa
	Iteración	Indica los valores inicial, límite y de incremento de una iteración
	Proceso predefinido	Indica un proceso predefinido, como llamar a una función
	Conector	Indica una entrada a, o salida de, otra parte de un diagrama de flujo o un punto de conexión
	Reporte	Indica un reporte de salida escrito

Figura 1.11 *Símbolos de diagrama de flujo.*

Por ejemplo, si tuviera que cambiar una llanta desinflada en su automóvil, no pensaría en todos los pasos requeridos, tan sólo cambiaría la llanta o llamaría a alguien que hiciera el trabajo. Éste es un ejemplo de pensamiento heurístico.

Por desgracia, las computadoras no responden a comandos heurísticos. Una instrucción general como "sumar los números del 1 al 100" no significa nada para una computadora porque sólo puede responder a comandos algorítmicos escritos en un lenguaje aceptable como C++. Para programar una computadora con éxito, debe entender con claridad esta diferencia entre comandos algorítmicos y heurísticos. Una computadora es una máquina "que responde a algoritmos"; no es una máquina "que responda a la heurística". No se le puede decir a una computadora que cambie una llanta o sume los números del 1 al 100. En cambio, debe dársele a la computadora un conjunto de instrucciones paso por paso detallado que, de manera colectiva, forma un algoritmo. Por ejemplo, el siguiente conjunto de instrucciones forma un método detallado, o algoritmo, para determinar la suma de los números del 1 al 100:

Establezca que n es igual a 100
Establezca a = 1
Establezca que b es igual a 100
Calcule la suma $= \dfrac{n(a + b)}{2}$
Imprima la suma

Note que estas instrucciones no son un programa de computadora. A diferencia de un programa, el cual debe escribirse en un lenguaje al que pueda responder la computadora, un algoritmo puede escribirse o describirse en varias formas. Cuando se utilizan enunciados en español o en inglés para describir el algoritmo (los pasos de procesamiento), como en este ejemplo, la descripción se llama **seudocódigo**. Cuando se usan ecuaciones matemáticas, la descripción se llama **fórmula**. Cuando se usan diagramas que emplean los símbolos mostrados en la figura 1.11, la descripción se conoce como un **diagrama de flujo**. La figura 1.12 ilustra el uso de estos símbolos para describir un algoritmo para determinar el promedio de tres números.

Debido a que los diagramas de flujo son engorrosos para revisar y pueden soportar con facilidad prácticas de programación poco estructuradas, han perdido el favor de los programadores profesionales, mientras el uso de seudocódigo para expresar la lógica de los algoritmos ha ganado una aceptación creciente. Al describir un algoritmo usando seudocódigo, se usan enunciados cortos en español. Por ejemplo, un seudocódigo aceptable para describir los pasos necesarios para calcular el promedio de tres números es:

Introducir los tres números en la memoria de la computadora
Calcular el promedio sumando los números y dividiendo la suma entre tres
Mostrar el promedio

Sólo después que se ha seleccionado un algoritmo y el programador entiende los pasos requeridos puede escribirse el algoritmo usando instrucciones en lenguaje de computadora. La redacción de un algoritmo usando instrucciones en lenguaje de computadora se llama codificar el algoritmo, lo cual es el tercer paso en nuestro procedimiento de desarrollo del programa (véase la figura 1.13). La mayor parte de la primera parte de este texto está dedicada a mostrarle cómo desarrollar y codificar algoritmos en C++.

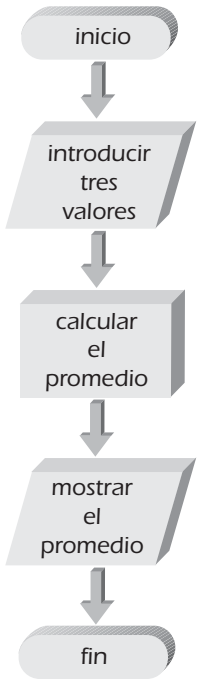


Figura 1.12 Diagrama de flujo para calcular el promedio de tres números.

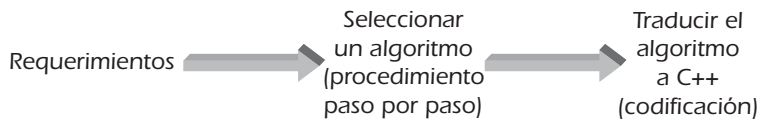


Figura 1.13 Codificación de un algoritmo.

Ejercicios 1.3

1. Determine un procedimiento paso a paso (lista de pasos) para hacer las siguientes tareas. (*Nota: No hay una sola respuesta correcta para cada una de estas tareas. El ejercicio está diseñado para brindarle práctica en convertir comandos tipo heurístico en algoritmos equivalentes y hacer el cambio entre los procesos de pensamiento implicados en los dos tipos de pensamiento.)*
   - **a.** Arreglar una llanta desinflada
   - **b.** Hacer una llamada telefónica
   - **c.** Iniciar sesión en una computadora
   - **d.** Asar un pavo
2. ¿Los procedimientos que desarrolló en el ejercicio 1 son algoritmos? Discuta por qué sí o por qué no.

3. Determine y describa un algoritmo (lista de pasos) para intercambiar los contenidos de dos tazas de líquido. Suponga que dispone de una tercera taza para conservar el contenido de cualquier taza de manera temporal. Cada taza deberá enjuagarse antes que cualquier líquido nuevo se vierta en ella.

4. Escriba un conjunto de instrucciones detallado, en español, para calcular la resistencia de los siguientes resistores conectados en serie: *n* resistores, cada uno con una resistencia de 56 ohmios, *m* resistores, cada uno con una resistencia de 33 ohmios, y *p* resistores, cada uno con una resistencia de 15 ohmios. Note que la resistencia total de los resistores conectados en serie es la suma de todas las resistencias individuales.

5. Escriba un conjunto de instrucciones detalladas paso a paso, para encontrar el número más pequeño en un grupo de tres números enteros.

6. **a.** Escriba un conjunto de instrucciones detalladas paso a paso, para calcular el número menor de billetes en dólares necesarios para pagar una factura de una cantidad TOTAL. Por ejemplo, si TOTAL fuera $97, los billetes consistirían en uno de $50, dos de $20, uno de $5 y dos de $1. (Para este ejercicio, suponga que sólo están disponibles billetes de $100, $50, $20, $10, $5 y $1.)
   b. Repita el ejercicio 6a, pero suponga que la factura debe pagarse sólo con billetes de $1.

7. **a.** Escriba un algoritmo para localizar la primera ocurrencia del nombre JEANS en una lista de nombres ordenada al azar.
   b. Discuta cómo podría mejorar su algoritmo para el ejercicio 7a si la lista de nombres estuviera en orden alfabético.

8. Escriba un algoritmo para determinar las ocurrencias totales de la letra *e* en cualquier enunciado.

9. Determine y escriba un algoritmo para clasificar cuatro números en orden ascendente (de menor a mayor).

1.4 ERRORES COMUNES DE PROGRAMACIÓN

Los errores más comunes asociados con el material presentado en este capítulo son los siguientes:

1. Un error de programación importante cometido por la mayoría de los programadores principiantes es apresurarse a escribir y correr un programa antes de entender por completo qué se requiere, incluyendo los algoritmos que se usarán para producir el resultado deseado. Un síntoma de esta prisa por introducir un programa en la computadora es la falta de cualquier documentación, o incluso un bosquejo de un programa o programa escrito en sí. Pueden detectarse muchos problemas con sólo revisar una copia del programa o incluso una descripción del algoritmo escrito en seudocódigo.

2. Un segundo error importante es no respaldar un programa. Casi todos los programadores nuevos cometen este error hasta que pierden un programa que les ha tomado un tiempo considerable codificar.

3. El tercer error cometido por muchos programadores novatos es la falta de comprensión de que las computadoras sólo responden a algoritmos definidos de manera explícita. Pedirle a una computadora que sume un grupo de números es bastante diferente que decirle a un amigo que sume los números. A la computadora deben dársele las instrucciones precisas para hacer la adición en un lenguaje de programación.

1.5 RESUMEN DEL CAPÍTULO

1. Los programas usados para operar una computadora se denominan *software*.

2. Los lenguajes de programación se presentan en una variedad de formas y tipos. Los programas en *lenguaje de máquina*, también conocidos como *programas ejecutables*, contienen los códigos binarios que pueden ser ejecutados por una computadora. Los *lenguajes ensambladores* permiten el uso de nombres simbólicos para operaciones
matemáticas y direcciones de memoria. Los programas escritos en lenguajes ensambladores deben ser convertidos en lenguajes de máquina, usando programas traductores llamados *ensambladores*, antes que los programas puedan ser ejecutados. Los lenguajes ensambladores y de máquina se denominan *lenguajes de nivel bajo*.

Los *lenguajes compilados* e *interpretados* se denominan *lenguajes de alto nivel*. Esto significa que son escritos usando instrucciones que se parecen a un lenguaje escrito, como el inglés, y pueden ejecutarse en una variedad de tipos de computadora. Los lenguajes compilados requieren un *compilador* para traducir el programa en una forma de lenguaje binario, mientras los lenguajes interpretados requieren un *intérprete* para hacer la traducción.

3. Como una disciplina, la *ingeniería de software* se ocupa de crear programas y sistemas legibles, eficientes, confiables y mantenibles.

4. El procedimiento de desarrollo de software consta de tres fases:

- Desarrollo y diseño del programa
- Documentación
- Mantenimiento

5. La fase de desarrollo y diseño del programa consta de cuatro pasos bien definidos:

- Analizar el problema
- Desarrollar una solución
- Codificar la solución
- Prueba y corrección de la solución

6. Un *algoritmo* es un procedimiento paso por paso que deben realizarse y describe cómo ha de ejecutarse un cálculo o tarea.

7. Un *programa de computadora* es una unidad independiente de instrucciones y datos usados para operar una computadora y producir un resultado específico.
8. Las cuatro estructuras de control fundamentales usadas en la codificación de un algoritmo son
   - Secuencia
   - Selección
   - Iteración
   - Invocación

1.6 APÉNDICE DEL CAPÍTULO: HARDWARE DE COMPUTACIÓN Y CONCEPTOS DE ALMACENAMIENTO

Todas las computadoras, desde las grandes supercomputadoras que cuestan millones de dólares hasta las computadoras personales de escritorio más pequeñas, deben realizar un conjunto mínimo de funciones y proporcionar la capacidad para:

1. Aceptar entradas
2. Mostrar salidas
3. Almacenar información en un formato lógico consistente (tradicionalmente binario)
4. Ejecutar operaciones aritméticas y lógicas en los datos de entrada o en los almacenados
5. Supervisar, controlar y dirigir la operación y secuencia general del sistema

La figura 1.14 ilustra los componentes de la computadora que respaldan estas capacidades y que de manera colectiva forman el **hardware** de la computadora.

La **unidad de aritmética y lógica (ALU)** ejecuta todas las funciones aritméticas y lógicas como adición y sustracción, y las proporcionadas por la computadora.

La **unidad de control** dirige y supervisa la operación general de la computadora. Rastrea el lugar de la memoria donde reside la siguiente instrucción, emite las señales necesarias para leer datos y escribir datos en otras unidades en el sistema y controla la ejecución de todas las instrucciones.

La **unidad de memoria** almacena información en un formato lógico consistente. De manera típica, tanto instrucciones como datos se almacenan en la memoria, por lo general en áreas separadas y distintas.

La **unidad de entrada y salida (I/O o E/S)** proporciona la interfaz a la que se conectan dispositivos periféricos como teclados, monitores, impresoras y lectores de tarjetas.

Almacenamiento secundario: Debido a que la memoria principal en cantidades muy grandes aún es relativamente cara y volátil (lo cual significa que la información se pierde cuando se suspende la energía), no es práctica como un área de almacenamiento permanente para programas y datos. Para este propósito se usan dispositivos de almacenamiento secundario o auxiliar. Aunque los datos se han almacenado en tarjetas perforadas, cinta de papel y otros medios en el pasado, casi todo el almacenamiento secundario se hace ahora en cintas magnéticas, discos magnéticos y CD-ROM.

En las primeras computadoras disponibles en forma comercial en la década de los años 50, todas las unidades de hardware se construían usando relés y tubos catódicos, y el almacenamiento secundario consistía en tarjetas perforadas. Las computadoras resultantes eran piezas de equipo grandes en extremo, capaces de hacer miles de cálculos por segundo que costaban millones de dólares.

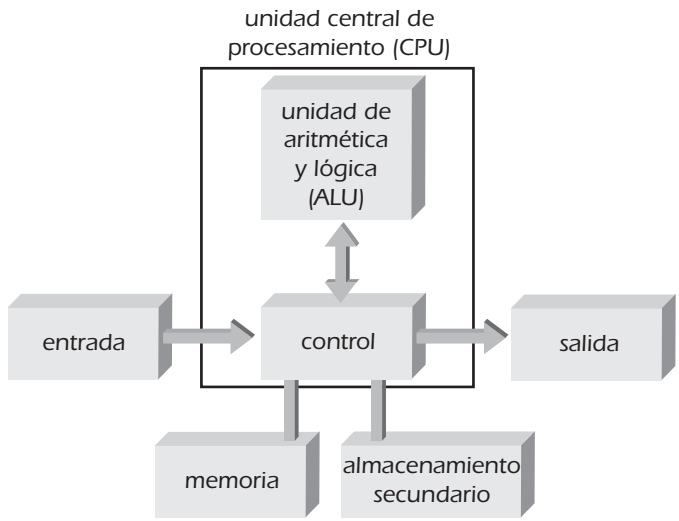


Figura 1.14 *Unidades básicas de hardware de una computadora.*

En la década de los años 60 con la introducción de los transistores, se redujeron tanto el tamaño como el costo del hardware de la computadora. El transistor era aproximadamente una vigésima parte de su contraparte, el tubo catódico. El tamaño pequeño del transistor permitió a los fabricantes combinar la unidad de aritmética y lógica con la unidad de control en una sola unidad. Esta unidad combinada se llama **unidad central de procesamiento (CPU)**. La combinación de la ALU y la unidad de control en una CPU tiene sentido porque la mayoría de las señales de control generadas por un programa están dirigidas a la ALU en respuesta a instrucciones aritméticas y lógicas dentro del programa. Combinar la ALU con la unidad de control simplificó la interfaz entre estas dos unidades y proporcionó una velocidad de procesamiento mejorada.

A mediados de la década de los años 60 se implementó la introducción de circuitos integrados (IC), los cuales produjeron otra reducción significativa en el espacio requerido para producir una CPU. Al principio, los circuitos integrados se fabricaban hasta con 100 transistores en un solo chip de silicio de 1 cm^2. Tales dispositivos se conocen como circuitos integrados de pequeña escala (SSI). Las versiones actuales de estos chips contienen cientos de miles a más de un millón de transistores y se conocen como chips integrados a gran escala (VLSI).

La tecnología de chips VLSI ha proporcionado los medios para transformar las computadoras gigantes de la década de los años 50 en las computadoras personales de escritorio y portátiles de la actualidad. Cada unidad individual requerida para formar una computadora (CPU, memoria e I/O) se fabrica ahora en un chip VLSI individual, y la CPU de un solo chip

se denomina **microprocesador**. La figura 1.15 ilustra cómo se conectan estos chips en forma interna dentro de las computadoras personales actuales, como las PC de IBM.

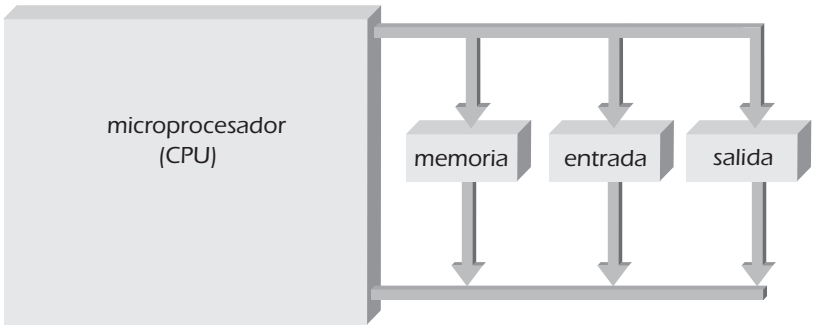


Figura 1.15 Conexiones de chips VLSI para una computadora de escritorio.

En forma concurrente con la reducción notable en el tamaño del hardware de la computadora ha habido una disminución igual de impresionante en el costo y un aumento en las velocidades de procesamiento. Hardware de computadora equivalente que costaba más de un millón de dólares en 1950 ahora puede comprarse por menos de quinientos dólares. Si las mismas reducciones ocurrieran en la industria automotriz, por ejemplo, ¡un Rolls Royce podría comprarse ahora por diez dólares! Las velocidades de procesamiento de las computadoras actuales también se han incrementado por un factor de miles sobre sus predecesores de la década de los años 50, con las velocidades de cómputo de las máquinas actuales midiéndose en millones de instrucciones por segundo (MIPS) y miles de millones de instrucciones por segundo (BIPS).

Almacenamiento de computadora

Los componentes físicos usados en la fabricación de una computadora requieren que los números y letras dentro de su unidad de memoria no se almacenen usando los mismos símbolos que la gente usa. El número 126, por ejemplo, no se almacena usando los números 1, 2 y 6. Ni la letra que reconocemos como A se almacena usando este símbolo. En esta sección veremos por qué es así y cómo almacenan números las computadoras. En el capítulo 2 se verá cómo se almacenan las letras.

La pieza más pequeña y básica de datos en una computadora se llama **bit**. Desde el punto de vista físico, un bit en realidad es un interruptor que puede abrirse o cerrarse. La convención que se seguirá es que las posiciones abierto y cerrado de cada interruptor se representan como un 0 y un 1, respectivamente.[5]

Un solo bit que puede representar los valores 0 y 1, por sí solo, tiene utilidad limitada. Todas las computadoras, por consiguiente, agrupan un número establecido de bits, tanto para su almacenamiento como para su transmisión. El agrupamiento de ocho bits para formar una unidad más grande es un estándar casi universal en la computación. Tales grupos se conocen como **bytes**. Un solo byte consistente en ocho bits, donde cada bit es 0 o 1, puede representar cualquiera de 256 patrones distintos. Éstos consisten del patrón 00000000 (los ocho interruptores abiertos) al patrón 11111111 (los ocho interruptores cerrados) y todas las

[5]Esta convención, por desgracia, es bastante arbitraria, y con frecuencia se encontrará la correspondencia inversa donde las posiciones abierto y cerrado son representadas como 1 y 0, respectivamente.

combinaciones intermedias posibles de 0 y 1. Cada uno de estos patrones puede usarse para representar ya sea una letra del alfabeto, otros caracteres individuales como un signo de dólar, una coma, etc., un solo dígito, o números que contienen más de un dígito. La colección de patrones consistentes en 0 y 1 que se usan para representar letras, dígitos individuales y otros caracteres individuales se llama **código de caracteres**. (Uno de estos códigos, llamado código ASCII, se presenta en la sección 2.3.) Los patrones usados para almacenar números se llaman **código de números**, uno de los cuales se presenta a continuación.

Números en complemento a dos

El código de números más común para almacenar valores enteros dentro de una computadora es la representación de **complemento a dos**. Usando este código, el equivalente entero de cualquier patrón de bits, como 10001101, es fácil de determinar y puede encontrarse para enteros positivos o negativos sin cambio en el método de conversión. Por conveniencia se asumirán patrones de bits del tamaño de un byte consistente en un conjunto de ocho bits cada uno, aunque el procedimiento puede utilizarse en patrones de bits de tamaño más grande.

La forma más fácil de determinar el entero representado por cada patrón de bits es construir primero un recurso simple llamado caja de valores. La figura 1.16 ilustra una de estas cajas para un solo byte. Desde el punto de vista matemático, cada valor en la caja ilustrada en la figura 1.16 representa un aumento en una potencia de dos. Ya que los números complementados a dos deben ser capaces de representar tanto enteros positivos como negativos, la posición en el extremo izquierdo, además de tener la magnitud absoluta más grande, también tiene un signo negativo.

```
-128| 64 | 32 | 16 |  8 |  4 |  2 |  1
----|----|----|----|----|----|----|---
    |    |    |    |    |    |    |
```

Figura 1.16 *Una caja de valores de ocho bits.*

La conversión de cualquier número binario, por ejemplo 10001101, tan sólo requiere insertar el patrón de bits en la caja de valores y sumar los valores que tienen 1 bajo ellos. Por tanto, como se ilustra en la figura 1.17, el patrón de bits 10001101 representa el número entero –115.

```
-128 | 64 | 32 | 16 |  8 |  4 |  2 |  1
 ----|----|----|----|----|----|----|---
   1 |  0 |  0 |  0 |  1 |  1 |  0 |  1
-128 +  0 +  0 +  0 +  8 +  4 +  0 +  1 = -115
```

Figura 1.17 *Conversión de 10001101 a un número en base 10.*

También puede usarse la caja de valores a la inversa para convertir un número entero en base 10 en su patrón de bits binario equivalente. Algunas conversiones, de hecho, pueden hacerse por inspección. Por ejemplo, el número en base 10 –125 se obtiene sumando 3 a –128. Por tanto, la representación binaria de –125 es 10000011, la cual es igual a –128 + 2 + 1. Del mismo modo, la representación en complemento a dos del número 40 es 00101000, la cual es 32 + 8.

Aunque el método de conversión de la caja de valores es engañosamente simple, se relaciona de manera directa con la base matemática subyacente de los números binarios de com-

plemento a dos. El nombre original del código de complemento a dos era código de signo ponderado, el cual se correlaciona en forma directa con la caja de valores. Como implica el nombre **signo ponderado**, cada posición de bit tiene un peso, o valor, de dos elevado a una potencia y un signo. Los signos de todos los bits excepto el bit de la extrema izquierda son positivos y el de la extrema izquierda es negativo.

Al revisar la caja de valores, es evidente que cualquier número binario en complemento a dos con un 1 inicial representa un número negativo, y cualquier patrón de bits con un 0 inicial representa un número positivo. Usando la caja de valores, es fácil determinar los valores más positivos y negativos que pueden almacenarse. El valor más negativo que puede almacenarse en un solo byte es el número decimal –128, el cual tiene el patrón de bits 10000000. Cualquier otro bit diferente de cero tan sólo agregará una cantidad positiva al número. Además, es claro que un número positivo debe tener un 0 como su bit en la extrema izquierda. A partir de esto se puede ver que el número de complemento de dos de ocho bits positivo más grande es 01111111 o 127.

Palabras y direcciones

Uno o más bytes pueden agruparse en unidades más grandes, llamadas **palabras**, lo cual facilita un acceso más rápido y más extenso a los datos. Por ejemplo, recuperar una palabra consistente en cuatro bytes de la memoria de una computadora proporciona más información que la obtenida al recuperar una palabra consistente de un solo byte. Dicha recuperación también es considerablemente más rápida que cuatro recuperaciones de bytes individuales. Sin embargo, este incremento en la velocidad y la capacidad se logra por un aumento en el costo y complejidad de la computadora.

Las primeras computadoras personales, como las máquinas Apple IIe y Commodore, almacenaban y transmitían internamente palabras consistentes de bytes individuales. Las primeras PC de IBM usaban tamaños de palabra consistentes de dos bytes, mientras que las PC más actuales almacenan y procesan palabras consistentes de cuatro bytes cada una.

El número de bytes en una palabra determina los valores máximo y mínimo que pueden ser representados por la palabra. La tabla 1.2 enumera estos valores para palabras de 1, 2 y 4 bytes (cada uno de los valores enumerados puede derivarse usando cajas de valores de 8, 16 y 32 bits, respectivamente).

Tabla 1.2 Valores enteros y tamaño de una palabra

Tamaño de la palabra	Valor del número entero máximo	Valor del número entero mínimo
1 Byte	127	–128
2 Bytes	32 767	–32 768
4 Bytes	2 147 483 647	–2 147 483 648

Además de representar valores enteros, las computadoras también deben almacenar y transmitir números que contienen puntos decimales, los cuales se conocen en matemáticas como números reales. Los códigos usados para números reales, los cuales son más complejos que los usados para enteros, se presentan en el apéndice C.

Consideración de opciones de carrera

Ingeniería aeronáutica y aeroespacial

Entre las más jóvenes de las disciplinas de ingeniería, la ingeniería aeronáutica y aeroespacial se ocupa de todos los aspectos del diseño, producción, prueba y utilización de vehículos o dispositivos que vuelan en el aire (aeronáutica) o en el espacio (aeroespacial), desde alas delta hasta transbordadores espaciales. Debido a que los principios científicos y de ingeniería implicados son tan amplios, los aeroingenieros por lo general se especializan en una subárea que puede superponerse con otros campos de la ingeniería como la ingeniería mecánica, metalúrgica o de materiales, química, civil o eléctrica. Estas subáreas incluyen las siguientes:

1. Aerodinámica. Estudia las características de vuelo de varias estructuras o configuraciones. Las consideraciones típicas son el arrastre y elevación asociados con el diseño de aviones o la aparición del flujo turbulento. Es esencial tener un conocimiento de la dinámica de fluidos. El modelamiento y prueba de todas las formas de aeronaves es parte de esta disciplina.
2. Diseño estructural. El diseño, producción y prueba de aeronaves y naves espaciales para resistir la amplia gama de demandas de vuelo de estos vehículos, así como naves submarinas, son el territorio del ingeniero estructural.
3. Sistemas de propulsión. El diseño de motores de combustión interna, a reacción, y de combustible líquido y sólido para cohetes y su coordinación en el diseño general del vehículo. Los motores de cohete, en especial, requieren ingeniería innovadora para adecuarse a las temperaturas extremas de almacenamiento, mezcla e ignición de combustibles como el oxígeno líquido.
4. Instrumentación y conducción. La industria aeroespacial ha sido líder en el desarrollo y utilización de electrónica de estado sólido en forma de microprocesadores para vigilar y ajustar las operaciones de cientos de funciones de aviones y naves espaciales. Este campo usa la pericia de ingenieros eléctricos y aeroingenieros.
5. Navegación. El cálculo de órbitas dentro y fuera de la atmósfera, y la determinación de la orientación de un vehículo con respecto a puntos en la tierra o en el espacio.