Universidad Nacional de General Sarmiento
Primer Ciclo Universitario

Programación y Métodos Numéricos

Introducción al lenguaje C

Primer Semestre de 2008
Versión 1.07

Contenidos

Capítulo 1

Conceptos básicos

1.1 ¿Qué es una computadora?

Una computadora es una máquina automática que procesa información de acuerdo con un programa almacenado.

Analicemos en detalle el sentido de esta frase. En primer lugar, una computadora es una *máquina*, como pueden serlo un lavarropas o un avión. No es más que una herramienta que ejecuta un trabajo por medio de la energía eléctrica que recibe.

Además, se trata de una máquina *automática*. Una máquina de escribir, una calculadora o un automóvil no son máquinas automáticas, sino que exigen teclear, presionar botones, activar palancas e intervenir en todo momento para poder realizar el trabajo deseado. Si el operador se detiene, el trabajo no se hace o queda incompleto. En cambio, prácticamente no hay intervención humana en el trabajo de cálculo realizado por una computadora. Se proporciona la *información* que debe ser procesada, y a continuación la computadora se encarga de procesar dicha información según como haya sido programada, generando los resultados en el formato previsto.

¿A qué tipo de información nos referimos? En el sentido de la informática, la *información* es un hecho elemental, cuantificable y codificado; puede ser una información meteorológica, el pedido de un producto para un cliente, la cantidad de horas que un obrero dedica a un cierto trabajo, las mediciones de un dispositivo de laboratorio, etc. Estas informaciones no se seleccionan al azar, sino que se introducen en la computadora solamente aquellos datos que son necesarios para el procesamiento requerido. Por otra parte, la información tiene que estar *organizada* en un orden y formato predeterminado por el programa a ejecutar, para que la computadora pueda leerla convenientemente.

Por último, y ésta es la noción esencial, este procesamiento de información se realiza de acuerdo con un *programa almacenado*. Un *programa* es una serie de instrucciones que le indican a la computadora qué operaciones debe realizar frente a los datos que se le suministran como entrada. Se trata de una lista de instrucciones detallada y completa, que debe cubrir todos los casos posibles en su más mínimo detalle. Si el programa no tiene en cuenta todos los casos, puede suceder que al recibir una información para la cual no está preparado, la computadora se detenga con un error o bien ejecute de manera anárquica las instrucciones previstas para otros casos.

Este programa está *almacenado*, aunque en forma *provisoria*. Una computadora no es una máquina especializada: por complejo que resulte un programa, en pocos segundos se carga o descarga, y luego la computadora resolverá otros problemas por medio de los programas correspondientes. De esta forma, una misma computadora puede calcular las planillas de pago de los empleados de una universidad, calcular la trayectoria de un vehículo espacial, ordenar las

palabras de un diccionario o calcular el balance de una empresa, siempre que se cargue y ejecute el programa adecuado para realizar cada una de estas tareas.

▷ Adaptado de [2], Sección 1.2. Recomendamos a los alumnos bibliófilos buscar y leer este muy valioso libro!

1.1.1 Componentes de una computadora

Las primeras computadoras electrónicas tenían el tamaño de una habitación, mientras que las computadoras embebidas modernas (es decir, las computadoras incorporadas o construidas dentro de otros dispositivos, como electrodomésticos o teléfonos celulares) pueden tener un tamaño menor al de una tarjeta de crédito. Las computadoras más pequeñas diseñadas específicamente para el uso individual y multipropósito son conocidas como *computadoras personales* o *PCs*. La computadora personal es actualmente la herramienta de comunicación y de procesamiento de información por excelencia; en el uso coloquial el término "computadora" se refiere a esta categoría en particular.

Sin importar de qué tipo de computadora se trate, todas ellas comparten una estructura interna similar (ver Figura 1.1). Los componentes que usualmente varían son aquellos que determinan cómo se relaciona la computadora con el exterior. Por ejemplo, las computadoras embebidas no poseen teclado ni mouse, e incluso no tienen monitor ni display alguno.

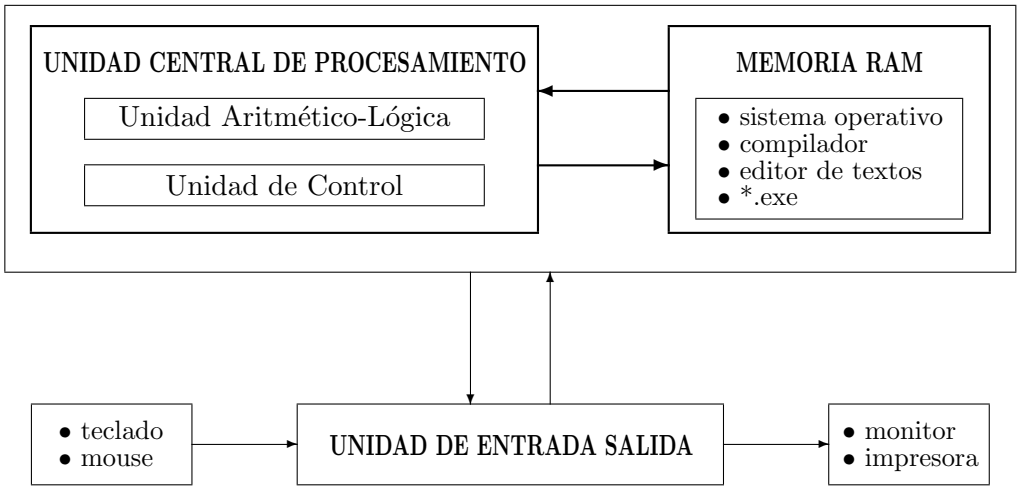

Figura 1.1: Esquema de una computadora.

Los siguientes son los componentes habituales que podemos encontrar en una computadora personal:

- *microprocesador* (también conocido con el nombre de *unidad central de procesamiento* o *CPU*), encargado de la ejecución del programa almacenado,
- *memoria RAM* de acceso rápido y almacenamiento volátil (es decir, se borra si no cuenta con energía eléctrica),
- *dispositivos de almacenamiento masivo* (habitualmente, discos rígidos), de acceso más lento que la memoria RAM, pero que no se borran al apagar la máquina,
- *periféricos*: dispositivos que se conectan a la computadora para permitir la interacción con el usuario (monitor, teclado, mouse, impresora, etc.),

- *dispositivos de comunicación* (placa de red, modem, etc.), que permiten la comunicación con otras computadoras,
- *buses* (cables o conectores), que interconectan los periféricos, la memoria y el microprocesador y transportan datos entre dichos dispositivos.

La información contenida en los medios de almacenamiento se organiza en *archivos*. Un *archivo* es un conjunto de información relacionada que se almacena bajo un mismo nombre lógico. En el disco rígido se suelen agrupar los archivos en una estructura jerárquica de *carpetas* o *directorios*.

1.2 Algoritmos y programas

Un *algoritmo* es una secuencia finita de instrucciones no ambiguas para realizar una tarea especifica. Por ejemplo, existen algoritmos para construir modelos de aviones a escala (expresados en forma de hojas de instrucciones), para operar un lavarropas (en forma de etiquetas pegadas en la tapa del aparato), para tocar un piano o algún otro instrumento musical (en forma de partituras), por sólo citar algunos ejemplos.

Si bien el concepto de algoritmo se suele comparar con el de una receta de cocina, en general los algoritmos son conjuntos de instrucciones más complejos: los algoritmos habitualmente requieren *repetición de pasos* (iteración) o necesitan tomar *decisiones* que involucren la aplicación de herramientas lógicas. Por otro lado, un algoritmo debe estar rigurosamente definido: se deben especificar todas las posibles circunstancias y cada caso particular se debe tratar sistemáticamente. Un algoritmo recibe normalmente *datos de entrada* y devuelve *resultados* o *datos de salida* como producto de su ejecución.

El estudio de los algoritmos comenzó como un tema propio de la matemática. La búsqueda de algoritmos era una actividad importante para los matemáticos mucho antes de que se inventaran las computadoras actuales, y el objetivo principal de dicha búsqueda era descubrir un conjunto de instrucciones para resolver un determinado problema. Uno de los resultados más conocidos de estas primeras búsquedas es el famoso algoritmo descubierto por el matemático griego Euclides para obtener el máximo común divisor de dos números enteros positivos (ver Figura 1.2).

ALGORITMO máximo común divisor

ENTRADA: dos números enteros positivos

SALIDA: el máximo común divisor

1. **asignar** a M el mayor y a N el menor de los dos valores
2. **asignar** a R el resto de la división $\frac{M}{N}$
3. ¿R es igual a cero?

Si →	el máximo común divisor es N
No →	**asignar** a M el valor de N **asignar** a N el valor de R volver al **paso 2**

DEVUELVE N.

Figura 1.2: Algoritmo euclidiano para hallar el máximo común divisor de dos enteros positivos.

En el ámbito de la informática, los algoritmos se encuentran contenidos dentro de programas que, además de los algoritmos propiamente dichos, contienen instrucciones para interactuar

con el usuario, almacenar información en medios de almacenamiento, realizar tareas de comunicación con otros programas, etc. A lo largo de la materia nos vamos a referir con el término "programa" a todas las instrucciones necesarias para la ejecución de una tarea, y con el término "algoritmo" exclusivamente a las instrucciones que resuelven un problema puntual. Habitualmente, las instrucciones que conforman el o los algoritmos incluidos en un programa solamente realizan cálculos y procesan información, sin interactuar con el usuario.

Muchos de los programas que utilizamos contienen millones de instrucciones. Las computadoras no obtienen sus extraordinarias capacidades mediante la ejecución de instrucciones complejas, sino que ejecutan una gran cantidad de instrucciones simples. Estas instrucciones se ejecutan secuencialmente al compás de un reloj interno —habitualmente relacionado con la capacidad de velocidad del microprocesador— hasta que el programa termina o hasta que una interrupción provoca que se alterne el programa actual en ejecución.

Hay que tener presente que para que un programa pueda ser ejecutado, primero debe ser cargado en la memoria RAM. A un programa en estado de ejecución —es decir, cargado en la memoria— lo denominamos *proceso*.

Por último, nos referimos con el término *software* al conjunto formado por un programa y todos los elementos adicionales que lo acompañan. Por ejemplo, un videojuego incluye, además del programa en sí, datos en forma de imágenes y sonidos que componen a la totalidad del juego.

▷ El contenido de esta sección fue adaptado de [5, 12].

1.3 Lenguajes de programación

Las instrucciones que conforman un programa de computadora se escriben en un lenguaje que la computadora puede entender. El único *lenguaje de programación* que una computadora puede entender —y por lo tanto, ejecutar— es el *lenguaje de máquina*, también conocido como *código de máquina*. Este lenguaje está compuesto por las instrucciones nativas del microprocesador particular instalado en la máquina, y se expresa en codificación binaria.

En un principio, los programadores trabajaban directamente con el lenguaje de máquina. Este lenguaje es muy complicado para programar, por lo que se desarrollaron diversos lenguajes para simplificar la tarea de la programación. Los distintos lenguajes de programación involucran distintos niveles de detalle en cuanto al control sobre el hardware por parte del programador. Cuando existe un mayor control sobre el hardware, hablamos de *lenguajes de bajo nivel*, y cuando existe un menor control sobre el hardware, hablamos de *lenguajes de alto nivel*. Al tener mayor control sobre el hardware se obtiene, en general, mayor performance, y también involucra mayor dificultad para programar.

Un lenguaje de programación de alto nivel es un conjunto de reglas sintácticas y semánticas que nos permite expresar instrucciones en un formato "entendible" para los seres humanos. La intención es que un lenguaje permita a un programador especificar de manera precisa sobre qué datos una computadora debe operar, cómo deben ser estos datos almacenados y transmitidos, y qué acciones se deben tomar en cada caso. Luego, mediante un proceso denominado *compilación* estas instrucciones se traducen al lenguaje de máquina, para que finalmente puedan ser ejecutadas por la computadora.

El programador escribe el programa utilizando un editor de textos. El archivo de texto resultante que contiene las instrucciones escritas en un lenguaje de programación es denominado *código fuente*. Luego, se utiliza un programa especial llamado *compilador* que traduce el código fuente en código ejecutable —es decir, en lenguaje de máquina— para que pueda ser ejecutado por la computadora (ver Figura 1.3).

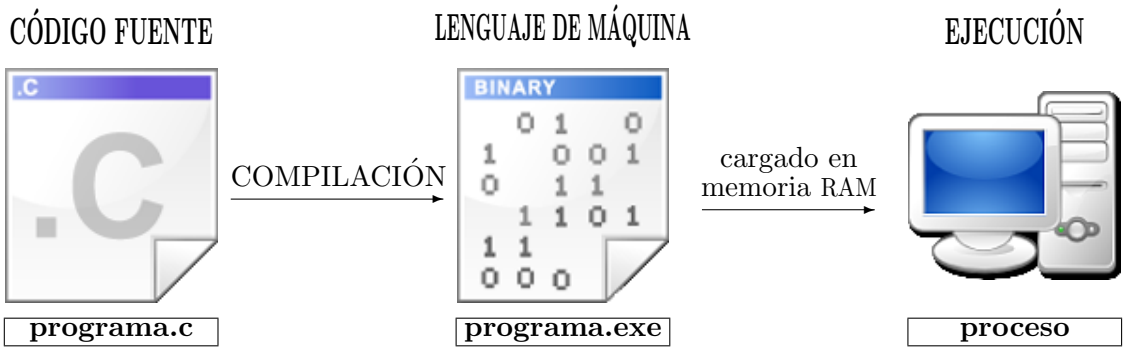

Figura 1.3: Esquema del proceso de compilación.

En nuestro caso, sólo vamos a ver el proceso de programación utilizando un *lenguaje compilado*. Sin embargo, existen otros lenguajes en los que los programas se ejecutan a través de un intérprete y que no necesitan ser compilados. Estos lenguajes se denominan *lenguajes interpretados*.

1.3.1 Programación estructurada

A finales de los años '60 surgió una nueva forma de programar que no solamente permite escribir programas más confiables y eficientes, sino que además los programas resultantes son más fáciles de comprender. Este tipo de programación, denominada *programación estructurada*, utiliza únicamente tres tipos de estructuras:

(i) estructuras secuenciales,

(ii) estructuras selectivas,

(iii) estructuras iterativas,

siendo innecesario el uso de instrucciones de salto incondicional (como por ejemplo, la instrucción `GOTO` de algunos lenguajes anteriores). Algunas de sus principales ventajas son:

- Los programas son más fáciles de entender. Un programa estructurado puede ser leído secuencialmente —de arriba hacia abajo— sin necesidad de estar saltando de un lugar a otro. La estructura del programa es más clara dado que las instrucciones están relacionadas entre sí, por lo que es más fácil entender qué hace cada sección del programa.

- El seguimiento de los errores o *debugging* se facilita debido a la lógica más clara, de tal forma que el programa puede corregirse con mayor facilidad.

- Aumento de la productividad del programador.

- Los programas quedan mejor documentados internamente.

El principal inconveniente de este método de programación es que se obtiene un único bloque de programa. Cuando el programa se hace demasiado grande, puede resultar problemático su mantenimiento. En este caso, es necesario emplear *programación modular*, definiendo módulos independientes programados y compilados por separado.

1.4 El Lenguaje de programación C

El *lenguaje de programación C* es un lenguaje de programación imperativo de propósito general desarrollado inicialmente para ser utilizado en conjunto con el sistema operativo Unix. Luego su uso se esparció a otros sistemas operativos, y en la actualidad es uno de los lenguajes de programación más utilizados.

El lenguaje C fue creado en 1969 por Dennis Ritchie y Ken Thompson en los Laboratorios Bell como evolución del lenguaje B, basado en BCPL. Al igual que B, el lenguaje C está orientado a la implementación de sistemas operativos y nació dentro del proyecto de desarrollo del sistema operativo Unix. C es apreciado por la eficiencia del código que produce y es el lenguaje de programación más popular para crear software de cálculo y aplicaciones científicas, aunque también se lo utiliza para crear aplicaciones de uso general y videojuegos.

Se trata de un lenguaje tipado de nivel intermedio pero con muchas características de bajo nivel. Dispone de las estructuras típicas de los lenguajes imperativos de alto nivel, pero a su vez dispone de construcciones que permiten un control a muy bajo nivel. Los compiladores suelen ofrecer extensiones al lenguaje que posibilitan mezclar código en lenguaje ensamblador con código C o acceder directamente a memoria o dispositivos periféricos.

Figura 1.4: Ken Thompson y Dennis Ritchie en 1970.

La primera estandarización del lenguaje C fue el ANSI C, con el estándar X3.159-1989. Posteriormente, en 1990 fue ratificado como estándar ISO (ISO/IEC 9899:1990). La adopción de este estándar es muy amplia; si los programas creados lo siguen, el código es portable entre plataformas y/o arquitecturas. En la práctica, los programadores suelen usar elementos no portables, es decir, dependientes del compilador o del sistema operativo utilizado.

1.4.1 Características

C es un lenguaje de programación relativamente minimalista. Uno de los objetivos de diseño de este lenguaje era que sólo fueran necesarias unas pocas instrucciones en lenguaje de máquina para traducir cada elemento del lenguaje, sin que hiciera falta un soporte intenso en tiempo de ejecución.

Se pueden desarrollar compiladores de C fácilmente, en parte a causa de ser de relativamente bajo nivel y de tener un conjunto de características modestas. En consecuencia, el lenguaje C está disponible para una amplia gama de plataformas —seguramente más que cualquier otro lenguaje. Además, y a pesar de su naturaleza de bajo nivel, el lenguaje se desarrolló para incentivar la programación independiente de la máquina. Un programa escrito cumpliendo con los estándares e intentando que sea portable puede compilarse en muchas computadoras de distintas arquitecturas.

El lenguaje C se desarrolló originalmente —junto con el sistema operativo Unix, con el que ha estado asociado desde el comienzo— "por programadores y para programadores". Sin embargo, alcanzó una popularidad enorme, y se usa en contextos muy alejados de la programación de sistemas, para lo que fue diseñado en un principio.

C tiene las siguientes características de importancia:

√ El núcleo del lenguaje es simple, con importantes funcionalidades añadidas —como funciones matemáticas y de manejo de archivos— proporcionadas por las bibliotecas.

√ Es un lenguaje muy flexible que permite programar con múltiples estilos.

√ Posee un sistema de tipos que impide operaciones sin sentido, aunque admite conversiones implícitas entre valores numéricos y lógicos.

√ Usa un lenguaje de preprocesado —el preprocesador de C— para tareas como definir macros e incluir múltiples archivos de código fuente.

√ Permite el acceso a memoria de bajo nivel mediante el uso de punteros.

√ Cuenta con un conjunto reducido de palabras reservadas.

√ Los parámetros se pasan por valor. El paso por referencia se puede simular pasando explícitamente el valor de los punteros.

√ Soporta punteros a funciones y variables estáticas, que permiten una forma rudimentaria de encapsulado y polimorfismo.

√ Cuenta con tipos de datos agregados —`structs`— que permiten que datos relacionados se combinen y se manipulen como un todo.

Algunas características de las que C carece, pero que se encuentran presentes en otros lenguajes de programación, están dadas en el siguiente listado:

× Soporte para programación orientada a objetos, aunque estas características están incluidas en el lenguaje C++.

× Encapsulado y abstracción por medio de construcciones del lenguaje.

× Funciones anidadas, aunque el compilador `gcc` [9] ofrece esta característica como extensión.

× Polimorfismo en tiempo de código en forma de sobrecarga, sobrecarga de operadores y sólo dispone de un soporte rudimentario para la programación genérica.

× Soporte nativo para programación multi-threading y de redes de computadoras.

Aunque la lista de las características útiles de las que C carece es larga, este factor ha sido importante para su aceptación, porque de esta manera es mucho más fácil escribir rápidamente nuevos compiladores para nuevas plataformas. Asimismo, ésta es la causa de que a menudo C sea más eficiente que otros lenguajes. Sólo la programación cuidadosa en lenguaje ensamblador típicamente produce un código más rápido, debido al control total sobre las características del hardware, aunque los avances en los compiladores de C y la velocidad creciente de los procesadores modernos han reducido gradualmente esta diferencia.

1.4.2 Especificaciones del lenguaje C

En 1978, Brian Kernighan y Dennis Ritchie [7] publicaron la primera edición de *El lenguaje de programación C*. Este libro fue durante varios años la especificación informal del lenguaje. El lenguaje descripto en este libro recibe habitualmente el nombre de "el C de Kernighan y Ritchie" o simplemente "K&R C" (la segunda edición del libro cubre el estándar ANSI C, descripto más abajo).

ANSI C e ISO C

En los '80 se empezó a utilizar C en las IBM PC's, lo que incrementó su popularidad significativamente. En 1983, el American National Standards Institute (ANSI) organizó un comité para establecer una especificación estándar de C. Tras un proceso largo y arduo, se completó el estándar en 1989 y se ratificó como el Lenguaje de Programación C ANSI X3.159-1989. Esta versión del lenguaje se conoce a menudo como ANSI C, o a veces como C89 (para distinguirla de C99).

En 1990, el estándar ANSI —con algunas modificaciones menores— fue adoptado por la International Organization for Standardization (ISO) en el estándar ISO/IEC 9899:1990. Esta versión se conoce a veces como C90. En 1995 se publicó una enmienda al estándar de 1990. Esta enmienda se conoce comúnmente como AMD1 y al estándar que la incluye como C95.

Uno de los objetivos del proceso de estandarización del ANSI C era producir una extensión al C de Kernighan y Ritchie, incorporando muchas funcionalidades no oficiales. Sin embargo, el comité de estandarización incluyó también muchas funcionalidades nuevas, como prototipos de funciones, y un preprocesador mejorado. También se cambió la sintaxis de la declaración de parámetros para hacerla semejante a la empleada habitualmente en C++.

ANSI C está soportado hoy en día por casi la totalidad de los compiladores. La mayoría del código C que se escribe actualmente está basado en ANSI C. Cualquier programa escrito sólo en C estándar sin código que dependa de un hardware determinado funciona correctamente en cualquier plataforma que disponga de una implementación de C compatible. Sin embargo, muchos programas han sido escritos de forma que sólo pueden compilarse en una cierta plataforma, o con un compilador en particular, ya sea debido a la utilización de bibliotecas no estándares, o debido a que algunos compiladores no cumplen con las especificaciones del estándar ANSI C.

C99

Tras el proceso de estandarización de ANSI, la especificación del lenguaje C permaneció relativamente estable durante algún tiempo, mientras que C++ siguió evolucionando. Sin embargo, el estándar continuó bajo revisión hasta finales de los 90's, lo que llevó a la publicación del estándar ISO 9899:1999 en 1999. Este estándar se denomina habitualmente C99. Se adoptó como estándar ANSI en marzo de 2000.

El compilador `gcc` [9], entre muchos otros, soporta hoy en día la mayoría de las nuevas características de C99. Sin embargo, este nuevo estándar ha tenido poca aceptación entre algunas

empresas desarrolladoras de compiladores, como Microsoft y Borland, que han centrado su interés en C++.

▷ El contenido de este capítulo fue adaptado de [12], exceptuando las secciones donde se especificaron otras fuentes.

Capítulo 2

Introducción al lenguaje C

En éste y en los siguientes capítulos presentamos el lenguaje de programación C. Aunque en muchos casos presentamos algoritmos como ejemplo, estos capítulos no tienen contenidos de programación, sino que su objetivo es introducir los elementos del lenguaje.

2.1 Consideraciones generales

Al igual que en todos los lenguajes de programación imperativos, la programación en el lenguaje C requiere normalmente de los siguientes pasos:

Escribir y/o editar el programa: el programa se tipea directamente en un editor de textos y se guarda el *código fuente* en un archivo de texto. Por convención, los archivos de código fuente de C se guardan con extensión `.c` para poder identificarlos con facilidad. Actualmente los *entornos de desarrollo* (IDE) incluyen un editor de textos para realizar éstas y otras tareas dentro de un mismo entorno de trabajo.

Compilar el programa: para poder ejecutar un programa, el código fuente debe ser *compilado* a código de máquina. El resultado de la compilación genera un archivo con extensión `.exe` —es decir, el *archivo ejecutable*.

Ejecutar el programa: los archivos generados en la compilación se ejecutan como cualquier otra aplicación. Puede suceder que el programa no funcione como esperábamos y que tengamos que volver al código fuente para realizar cambios y correcciones, volviendo al comienzo y repitiendo este ciclo.

2.1.1 Símbolos especiales

Durante la escritura del archivo de código fuente nos vamos a encontrar con diversos símbolos. Además de letras —tanto en minúsculas como en mayúsculas— y de números, algunos de los símbolos que vamos a utilizar son: `# { } ( ) [ ] ; " % & = + - / * < >`. Veamos aquellos que en este momento merecen algunas observaciones.

Las llaves `{ }` se usan para delimitar las funciones y los bloques de instrucciones. Un *bloque* es un grupo de instrucciones que están asociadas y que se ejecutan como una unidad. El comienzo de un bloque se denota con una llave `{` y el fin de un bloque se denota con una llave `}`. Los bloques pueden estar vacíos: un bloque vacío se escribe `{}`. También pueden existir bloques dentro de otros bloques, que se conocen como *bloques anidados*.

Los paréntesis `( )` se suelen utilizar junto con los nombres de las funciones, como en `cos(x)`.

Cada instrucción o línea en C hasta el punto y coma se llama *sentencia*. El punto y coma `;` se usa para terminar una sentencia. Un error muy común que se suele cometer cuando se está aprendiendo C es omitir el punto y coma al final de una instrucción. El punto y coma le informa al compilador que se llegó al final de una sentencia y, si se lo omite, el compilador no sabe donde terminó la sentencia en cuestión. No es necesario un punto y coma después de la llave que cierra un bloque de instrucciones.

2.1.2 El preprocesador

El preprocesador de C es el primer componente que el compilador invoca en el momento de la compilación. Se encarga de interpretar las directivas de compilación `#include` y `#define`, entre otras. Todas las directivas del preprocesador comienzan con el símbolo `#` y no terminan en punto y coma.

En C existen bibliotecas de funciones estándar que se pueden incluir en nuestros programas mediante las directivas del preprocesador. Todos los programas que escribimos necesitan comunicarse con el usuario (como mínimo, el programa necesita dar una respuesta). Por lo tanto, todos los programas escritos en C necesitan de al menos una biblioteca estándar que se encargue de la entrada y salida de datos. Esta biblioteca estándar se llama `<stdio.h>` y se declara en el programa mediante la directiva del preprocesador

```
#include <stdio.h>
```

al principio de nuestro archivo de código fuente.

2.1.3 Palabras reservadas

Las *palabras reservadas* son palabras especiales que se usan en un lenguaje de programación para especificar instrucciones, construcciones y elementros propios del lenguaje. En el lenguaje C se utilizan solamente 32 palabras reservadas (ver Tabla 2.1).

Es importante notar que las palabras reservadas se escriben en minúsculas. El lenguaje C distingue entre mayúsculas y minúsculas, con lo cual no es lo mismo escribir un texto en mayúsculas que en minúsculas. Por este motivo, decimos que C es *case sensitive*.

`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatil`
`do`	`if`	`static`	`while`

Tabla 2.1: Lista de palabras reservadas del lenguaje C.

Una palabra reservada no se puede usar para otro propósito. Por ejemplo, no se puede declarar una variable llamada `while`.

2.2 El programa `Hola, Mundo!`

El programa `Hola, Mundo!` (ver Código 2.1) es el primer ejemplo que veremos de un programa escrito en lenguaje C. Este programa imprime en la salida estándar (normalmente se trata de la

pantalla del monitor) el mensaje “Hola, Mundo!”.

Código 2.1 El programa `Hola, Mundo!`.

```
#include <stdio.h>

int main()
{
    printf("Hola, Mundo!\n");
    return 0;
}
```

A continuación, analizamos línea por línea este primer ejemplo.

```
#include <stdio.h>
```

En la primera línea del programa vemos la directiva del preprocesador `#include`. El preprocesador sustituye esta directiva por el contenido del archivo al que se refiere. En este caso, el *header* `<stdio.h>` reemplazará a la presente línea.

Un *header* es un archivo especial de código fuente que, por convención, tiene extensión `.h`. El archivo `<stdio.h>` contiene las definiciones de las *funciones de entrada y salida*. Si necesitamos que nuestro programa interactúe con el usuario, tendremos que incluir este header. Los símbolos `< >` que rodean a `<stdio.h>` indican que el archivo `<stdio.h>` es un header suministrado por la implementación del lenguaje.

```
int main()
```

Esta línea indica que se definirá la función llamada `main()`. La función `main()` tiene un propósito especial en C: es la primera función que se invoca cuando se ejecuta un programa, y controla a las demás funciones. Salvo este hecho, la función `main()` es igual a todas las funciones dentro de un programa. La palabra `int` indica que el valor que devuelve la función es de tipo entero.

```
{
```

La llave que abre indica el comienzo de la definición de la función `main()`.

```
printf("Hola, Mundo!\n");
```

En esta línea se invoca a —es decir, se ejecuta el código de— una función llamada `printf()`, que está declarada en el header `<stdio.h>` incluido más arriba. La función `printf()` imprime en pantalla un texto según los argumentos que se especifican entre los paréntesis. En este caso, la función `printf()` recibe un sólo argumento, la cadena de caracteres `"Hola, Mundo!\n"`. Una cadena de caracteres se escribe encerrada entre comillas dobles `" "`. El par de caracteres `\n` es una secuencia de escape que genera una nueva línea en la pantalla.

```
return 0;
```

Esta línea indica el término de la ejecución de la función `main()` y provoca que la misma devuelva el valor entero cero, el cual, por convención, es el código de terminación que indica condiciones de terminación exitosas o no erróneas.

```
}
```

La llave que cierra indica el fin del código de la función `main()`. Cuando se compila el Código 2.1 y se ejecuta el programa obtenido, el programa realiza las siguientes acciones:

- imprime el mensaje "Hola, Mundo!" en la pantalla,
- mueve el cursor de la posición actual —detrás de la frase "Hola, Mundo!"— al comienzo de la siguiente línea, y
- devuelve un valor de salida cero al proceso invocador, en este caso el sistema operativo.

2.3 Comentarios

Siempre es útil poner comentarios en el código de nuestros programas. Los comentarios se utilizan principalmente para documentar el significado y propósito de una sección determinada de código fuente, de manera que podamos recordar luego la utilidad de dicha sección de código. También se pueden usar para eliminar temporalmente líneas de código, en especial para buscar errores en nuestros programas. Todos los comentarios son ignorados por el compilador.

Existen dos formas de introducir comentarios en el código de un programa en C. Por una parte, se pueden escribir varias líneas de comentarios comenzando con `/*` y terminando con `*/`. Por ejemplo:

```
/* Existen 10 tipos de personas en el mundo.
   Los que entienden numeración binaria y los que no. */
```

es un comentario válido en C. Por otra parte, si solamente se quiere escribir una línea de comentarios, se pueden comenzar los comentarios con `//`. Todo el texto a partir del símbolo `//` y hasta el final de la línea se considera como un comentario y es ignorado por el compilador. Por ejemplo, podríamos comentar el programa `Hola, Mundo!` como se ve en el Código 2.2.

Código 2.2 El programa `Hola Mundo` con comentarios varios.

```
/*****************************************************************
* Este es nuestro primer programa y se llama Hola Mundo. El mismo *
* muestra el mensaje "Hola Mundo" en pantalla.                    *
*****************************************************************/

#include <stdio.h>

int main()
{
    printf("Hola, Mundo!\n"); // imprime el mensaje en pantalla
    return 0; // retorna el valor 0 como resultado
}
```

▷ El contenido de este capítulo fue realizado recopilando información de [3, 7, 11, 12].

Capítulo 3

Tipos de datos, variables y constantes

Las variables y las constantes son los objetos básicos que se manipulan en un programa. Las declaraciones indican las variables que se usarán y establecen el tipo de datos al que pertenecen. Los operadores especifican lo que se hará con las variables. Las expresiones combinan variables y constantes para producir nuevos valores. El tipo de un objeto determina el conjunto de los posibles valores que éste puede tomar y qué operaciones se pueden realizar con dicho objeto.

3.1 Nombres válidos de variables

Los nombres de variables pueden contener sólo letras minúsculas y mayúsculas, el símbolo `_` (*underscore*) y números. Deben comenzar con una letra o con un underscore `_` y no deben contener espacios. Sólo los primeros 31 caracteres de la variable son significativos.

Se le puede dar cualquier nombre a una variable, aunque los nombres significativos pueden ser una valiosa ayuda para identificar el propósito de cada variable. Por ejemplo, nombres como `suma`, `total`, `promedio`, etc., nos dan una idea clara del uso de estas variables.

Es importante tener presente que el compilador distingue las letras minúsculas de las mayúsculas: las variables `hola`, `Hola`, `HOLA` y `HoLa` son diferentes. Por convención se usan nombres en minúsculas para variables locales y nombres en mayúsculas para constantes simbólicas.

3.2 Tipos de datos

Existen cinco tipos de datos básicos —`int`, `float`, `double`, `char` y `void`— que se pueden asociar a las variables. Todos los demás tipos de datos se basan en alguno de estos cinco. El tamaño y el rango de los tipos de datos varían con cada arquitectura de hardware y entre diferentes implementaciones de C. El estándar ANSI sólo estipula el rango mínimo para cada tipo. En la Tabla 3.1 se muestran los tipos de datos disponibles en C.

Existen también ciertos modificadores que se pueden aplicar a los tipos de datos básicos, de los cuales el que nos interesara aquí es el calificador `long` que amplia el tamaño del tipo de datos y que se puede aplicar tanto a `int` como a `double`.

3.3 Declaración de variables

Una *variable* es una porción de memoria RAM con nombre asociada a un tipo de datos. Se usa para almacenar un valor que puede ser modificado por el programa si se lo solicita explícitamente.

Tipo de dato	Tamaño	Comentario
`int`	16 bits	números enteros desde -32768 hasta 32767.
`long int`	32 bits	números enteros desde -2147483648 hasta 2147483647.
`long long`	64 bits	números enteros desde -9×10^{18} hasta 9×10^{18}.
`float`	32 bits	números de punto flotante de precisión simple.
`double`	64 bits	números de punto flotante de doble precisión.
`long double`	96 bits	números de punto flotante de triple precisión.
`char`	8 bits	un carácter —cualquier símbolo del teclado.
`void`	—	es un tipo de uso especial sin valor.

Tabla 3.1: Los distintos tipos de datos disponibles en C.

El tipo de datos de dicho valor debe coincidir con el tipo de datos de la variable.

La sentencia para darle un nombre a una variable y asociarla un tipo de datos en particular se llama *declaración de la variable* y, normalmente es lo primero que aparece dentro de la definición de una función. Se deben declarar todas las variables antes de su utilización. Las declaraciones especifican a qué tipos de datos pertenecen las variables. Una declaración consta de una lista de una ó más variables del tipo especificado, como por ejemplo:

```
int num, suma, total;
float delta, epsilon;
```

Las variables se pueden distribuir entre las declaraciones de cualquier forma. Por ejemplo, la lista de arriba es equivalente a las siguientes declaraciones:

```
int num;
int sum;
int total;
float delta;
float epsilon;
```

Esta ultima forma ocupa más espacio, pero es conveniente para agregar comentarios o para facilitar la lectura del código.

Se puede usar una variable sólo después de que ésta haya sido declarada, y nunca antes de su declaración. Si se intenta usar una variable que no fue declarada previamente el compilador mostrará un mensaje de error.

3.4 Instrucción de asignación

La instrucción de asignación es una instrucción fundamental de los lenguajes de programación imperativos como el lenguaje C. Esta instrucción modifica el estado actual de las variables durante la ejecución del programa. El operador de asignación en este lenguaje es el símbolo igual `=`.

Se puede asignar un valor en una variable usando la instrucción de asignación:

```
nombre_de_la_variable = expresión;
```

donde el valor de la expresión debe corresponderse con la declaración de la variable. Por ejemplo, si `nombre_de_la_variable` fue declarada como entera, entonces el valor de `expresión` debe ser entero. Es importante destacar que `expresión` se refiere a valores numéricos como 6 ó 2.71, a caracteres como `L` ó `k`, y a expresiones como `x`, `2*b+y/3` ó `9>2*(largo+ancho)`. En el Capitulo 5 se verán en detalle las expresiones válidas en C.

3.4.1 Variables de tipo entero: `int` y `long int`

Una variable del tipo `int` *(INTeger)* puede almacenar un valor entero de 16 bits, es decir, en el rango comprendido entre -32768 y $+32767$. Sin embargo, la mayoría de los compiladores modernos proveen enteros `int` de 32 bits, es decir, en el rango desde -2147483648 hasta 2147483647.

Para declarar una variable `int` debemos utilizar una instrucción de la forma de:

```
int a;
```

Esta instrucción reserva una porción de 16 bits de memoria RAM para almacenar datos del tipo `int` y le otorga el nombre `a`. Para asignarle un valor a la variable recién creada:

```
a = 10;
```

Es importante recordar que el símbolo `=` se usa para la instrucción de asignación. Esta instrucción debe interpretarse como: "tomar el valor numérico 10 y guardarlo en la posición de memoria asociada con la variable entera llamada `a`". De ninguna forma debe confundirse con una igualdad, dado que en ese caso una instrucción como:

```
a = a+10;
```

sería una expresión contradictoria. Esta ultima instrucción se interpreta como: "tomar el valor actual guardado en la posición de memoria asociada a la variable entera `a`, luego sumarle el valor numérico 10, y por último, tomar el resultado obtenido y guardarlo en la posición de memoria asociada con la variable entera `a`" —el valor previo de `a` se pierde.

Observar que en la instrucción de asignación se evalúa primero la expresión que se encuentra en la derecha del operador de asignación `=` y luego se almacena el valor en cuestión en la variable indicada a la izquierda de dicho operador.

También se puede usar el calificador `long` con el tipo de datos `int`, como se ve a continuación:

```
long int a;
```

Esta instrucción reserva una porción de 32 bits de memoria RAM para guardar datos del tipo `long` y le otorga el nombre `a`. Al declarar variables `long int`, se puede omitir la palabra reservada `int`. La asignación de un valor a este tipo de variable es igual al caso anterior.

El propósito es proveer distintos tamaños de enteros donde esto sea práctico. Cada compilador tiene la libertad de elegir el tamaño adecuado para el hardware donde ejecuta, sujeto a la restricción de que el tamaño del tipo `int` sea de al menos 16 bits, el de `long` de al menos 32 bits y de que el tamaño de `int` no sea mayor que el de `long`.

3.4.2 Variables de tipo real: `float` y `double`

En C existen dos palabras reservadas para declarar variables reales: `float` y `double`, que ofrecen distintos niveles de precisión:

float *(FLOATing point)* es un tipo de datos de un tamaño de 32 bits y tiene un rango que va desde 1.17549×10^{-38} hasta $3.40282 \times 10^{+38}$.

double *(DOUBLE precision)* es un tipo de datos de un tamaño de 64 *bits* y tiene un rango que va desde 2.22507×10^{-308} hasta $1.79769 \times 10^{+308}$.

Por ejemplo, si necesitamos declarar variables de este tipo, las siguientes instrucciones:

```
float   suma;
double total;
```

declaran dos variables llamadas `suma` y `total`, de tipo `float` y `double`, respectivamente. Para asignarles valores a estas dos variables usamos las instrucciones de asignación:

```
suma  = 3.141592;
total = 2.0;
```

de forma análoga que para el caso de variables enteras.

Si es necesario, podemos usar el calificador `long` junto con el tipo de datos `double`. El tipo de datos `long double` tiene un tamaño de 96 bits. Para declarar variables de este tipo, lo hacemos de la manera usual:

```
long double epsilon;
```

y la forma de la asignación es la misma de los ejemplos anteriores:

```
epsilon = 1.17550e-38;
```

En este caso utilizamos la notación exponencial, donde la expresión `1.17550e-38` es equivalente a 1.7550×10^{-38}.

3.4.3 Variables de caracteres: `char`

Las variables de tipo `char` *(CHARacter)* comprenden a cualquier símbolo que podemos encontrar en el teclado de la computadora. Las variables `char` son variables enteras de 8 bits —recordar que en el código ASCII cada caracter ocupa 8 bits y que cada valor corresponde a un símbolo. Para declarar una variable de caracter se usa la palabra reservada `char`. Por ejemplo, con la declaración:

```
char c;
```

se crea una variable del tipo `char` denominada `c`. Para asignarle un caracter a esta variable encerramos cualquier símbolo del teclado entre *comillas simples* `' '` de la siguiente manera:

```
c = 'A';
```

o, si sabemos cual es el valor que representa al caracter `A` en el código ASCII, también es válido escribir:

```
c = 65;
```

dado que al caracter `A` le corresponde el valor 65 (en un editor de textos se puede probar la combinación de teclas `ALT+65`). Notar que sólo se guarda un único carácter por cada variable de este tipo.

3.5 Asignaciones múltiples

El lenguaje C permite asignar el mismo valor a diferentes variables utilizando una asignación multiple en una única instrucción. Por ejemplo, el siguiente código asigna el valor entero 1 a las variables `a`, `b`, `c` y `d` simultaneamente:

```
a = b = c = d = 1;
```

Es habitual que los programadores usen este método para asignar valores comunes a las variables.

3.6 Inicializacion de variables

Es posible asignar un valor inicial cuando se declara una variable. Por ejemplo, la sentencia

```
int a = 1;
```

inicializa la variable de tipo `int` con el valor 1 inmediatamente después de haberla creado. Esta sentencia es equivalente a:

```
int a;
a = 1;
```

La diferencia radica en que el compilador puede trabajar más rápidamente al hacerlo en una sola instrucción. Cabe aclarar que no se puede usar este método de declaración combinado con la asignación multiple de la Sección 3.5.

Una ultima observación sobre la inicializacion de variables: No debemos asumir de ninguna manera que una variable no inicializada contiene un valor en particular —como por ejemplo, el valor cero— ya que esto no es parte de la especificación estándar del lenguaje C.

3.7 Constantes simbólicas

Puede suceder que tengamos que hacer cálculos con el mismo valor en distintas partes del código de nuestro programa. Por ejemplo si estamos trabajando en cálculos que involucren al IVA puede ser que el número 21 aparezca en muchos lugares dentro del programa. ¿Que pasaría si cambia el porcentaje del IVA?

Sería muy engorroso tener que modificar un valor repetido varias veces, además de correr el riesgo de omitir alguna de las modificaciones, generando errores muy dificiles de encontrar. Para evitar estos inconvenientes se usan las *constantes simbólicas*.

Una constante simbólica es un valor fijo que no se puede alterar durante la ejecución del programa. Las constantes se definen luego de la directiva `#include` y antes de la función `main()` o de cualquier otra función. Todos los tipos de datos vistos previamente se pueden definir como constantes. La directiva para definir una constante es:

```
#define NOMBRE_DE_LA_CONSTANTE valor_de_la_constante
```

Cabe aclarar que esta directiva no termina con un punto y coma, al igual que todas las directivas del preprocesador (que comienzan con el símbolo `#`). Las reglas que vimos para los nombres válidos de variables se aplican también a las constantes simbólicas, aunque por convención los nombres de las constantes se escriben en mayúsculas.

Por ejemplo, la directiva:

```
#define IVA 21.0
```

revolvería nuestro problema de tener repetidas muchas veces el valor 21 en el código fuente, y en el caso de que se presente la necesidad de modificarlo sólo tendríamos que cambiar esta única definición.

▷ El contenido de este capítulo fue realizado recopilando información de [3, 7, 8, 9, 11, 12].

Capítulo 4

Entrada y salida de datos

4.1 Funciones de entrada y salida

Existen varias funciones que se encargan de la entrada de datos en C, en este capítulo veremos la función `scanf()`. Para leer un valor entero en la variable `a` (de tipo `int`) usamos:

```
scanf("%d", &a);
```

Cuando el programa llega a la sentencia en la que se invoca a la función `scanf()`, éste se detiene y espera que el usuario tipee algún valor entero en el teclado. Una vez que usuario ingresó el número deseado, debe presionar la tecla `ENTER` para avisar que ya finalizó el ingreso de datos. Luego, el programa continúa con el nuevo valor guardado en la variable `a`. Cada vez que el programa se ejecute, el usuario debe ingresar un valor y, por lo tanto, el programa tiene la oportunidad de mostrar resultados distintos dependiendo de los datos de entrada.

Ahora centremos nuestra atención en la salida de los resultados del programa. Ya utilizamos en el programa `Hola Mundo` (Código 2.1) la función `printf()`, que imprime datos en pantalla. Si nos interesa mostrar el contenido guardado en la variable `a`, usamos una instrucción como:

```
printf("El contenido de la variable es %d", a);
```

El par de caracteres `%d` que aparece tanto en la función `printf()` como en la función `scanf()` de más arriba, le informa al compilador el tipo de valor que estamos manejando. En este caso, como `a` es entero, la secuencia `%d` *(Decimal integer)* le dice al compilador que se trata de un entero expresado en notación decimal.

Observar que la función `scanf()` no pide expresamente al usuario que ingrese datos. Por lo tanto, siempre antes de usar `scanf()` se debería usar la función `printf()` pidiendo al usuario el dato que debe ingresar:

```
printf("Ingresá un número entero: ");
scanf("%d", &a);
```

En las siguientes secciones veremos más en profundidad los detalles de estas dos funciones de entrada y salida. Las funciones `scanf()` y `printf()` tienen varias características en común, y difieren de otras funciones en que pueden tomar un número variable de argumentos.

4.1.1 La función de salida `printf()`

El primer argumento de la función `printf()` es siempre una cadena de caracteres. Luego de la misma se pueden incluir todos los argumentos que sean necesarios. El formato habitual de la función `printf()` es:

```
printf("cadena de control", var0, var1, var2, ...)
```

Los argumentos `var0`, `var1` y `var2` son generalmente variables, pero también pueden ser expresiones como `a+b`, `a%b`, `(-b+sqrt(b*b-4*a*c))/(2*a)`, etc.

La *cadena de control* es de suma importancia debido que la misma especifica el tipo de cada valor en la lista y cómo deben mostrarse estos valores en la pantalla. Debido a esto, a la cadena de control también se la conoce como *cadena de formato*.

La función `printf()` recorre la cadena e imprime en la pantalla todos los caracteres que encuentra, excepto cuando se trata de un símbolo `%`. El caracter `%` es una señal que indica que lo que sigue es la especificacion de cómo debe mostrarse el proximo argumento en la lista de argumentos. La función usa esta información para convertir y formatear el valor que se le pase mediante una variable o expresión. Por ejemplo, en

```
printf("Nos encanta Informática");
```

tenemos sólo la cadena de control —no necesitamos del símbolo `%` ya que lo unico que queremos es mostrar el mensaje "Nos encanta Informatica", para lo cual no se requieren valores.

Con el especificador de formato `%d` se convierte el próximo valor a entero decimal, de manera que:

```
printf("Total = %d", total);
```

imprime en pantalla el mensaje "Total = " seguido del valor guardado en la variable `total` como entero decimal. Por ejemplo, si la variable `total` tiene guardado el valor 45, el mensaje completo sería "Total = 45".

El caracter `%` no es sólo un especificador de formato, sino que también es un especificador de conversión. El mismo indica el tipo de datos de la variable y cómo estos datos deben convertirse a los caracteres que aparecerán en la pantalla. Por ejemplo, si por un descuido intentamos mostrar en pantalla una variable del tipo `float` usando el especificador `%d` para enteros, se mostrará un valor entero, aunque dicho valor no corresponderá con el valor correcto de la variable. Esto se debe a la forma en la que se representan los valores numéricos en una computadora.

4.1.2 Especificadores de formato

Los especificadores `%` que se pueden usar en ANSI C son los que se muestran en la Tabla 4.1. Se puede agregar una `L` ó `l` delante del especificador para indicar la forma `long` del tipo de datos. Por ejemplo, `%ld` se refiere a una variable `long int` y `%Lf` a una variable del tipo `long double`. Observar que para la función `printf()` no existe distinción entre `float` y `double`.

Especificador	Tipo de dato	Muestra en pantalla
`%c`	`char`	un caracter simple
`%d`	`int`	enteros con signo
`%e`	`float` ó `double`	floats en formato exponencial
`%f`	`float` ó `double`	floats con 6 decimales de precisión
`%g`	`float` ó `double`	usa `%f` ó `%e`, segun se requeriera
`%s`	arreglo de caracteres	cadenas de caracteres

Tabla 4.1: Especificadores de formato.

El especificador de formato sólo convierte un patrón de bits a una secuencia de caracteres. Si se desea dar algún formato especial a los caracteres, entonces se deben usar los símbolos de

Símbolo	Significado
`-`	justificación izquierda
`+`	mostrar siempre el signo
`space`	mostrar espacio si no hay signo

Tabla 4.2: Modificadores para los especificadores de formato.

la Tabla 4.2. Cada especificador puede ser precedido por un modificador que determina cómo se muestra un valor en pantalla.

La forma de uso es la siguiente:

```
printf("El saldo es %+f", total);
```

donde el `+` también puede ser un `-` o un espacio. Por ejemplo, si suponemos que el valor almacenado en la variable `total` es 1747.55, entonces el mensaje mostrado en la pantalla sería "El saldo es +1747.550000".

Adicionalmente, se pueden usar números para especificar el total de caracteres empleados al mostrar el valor y para indicar la precisión (número de posiciones después del punto decimal). Por ejemplo, el especificador `%10.3f` mostrará en pantalla un valor del tipo `float` utilizando 10 caracteres con 3 digitos despues del punto decimal. Notar que los 10 caracteres incluyen al punto decimal y al signo si lo hubiere. Si el valor necesita más espacio que el especificado, entonces se usará todo el espacio necesario.

El especificador `%-10d` mostrará un `int` justificado a la izquierda con 10 caracteres de ancho. Por último, el especificador `%+0.4f` mostrará un `float` con 4 lugares despues del punto decimal y con el signo correspondiente ($-$ ó $+$).

Secuencia	Significado
`\b`	retrocede un caracter
`\n`	nueva línea
`\r`	retorno de carro
`\t`	tabulación
`\'`	comilla simple
`\"`	comilla doble
`\0`	null

Tabla 4.3: Secuencias de escape.

Si se usan los códigos de la Tabla 4.3 en la cadena de control, entonces el código ASCII correspondiente será enviado a la pantalla y producirá el efecto que se lista en dicha tabla. El más importante de estos códigos de escape es `\n`, que especifica una nueva línea. Por ejemplo, la siguiente línea de código:

```
printf("\n\nHola Mundo,\nEstamos en clase de Informática.\n\n");
```

imprimirá en pantalla el mensaje:

```
-
-
Hola Mundo,
Estamos en clase de Informática.
-
-
```

4.1.3 La función de entrada `scanf()`

Luego de haber visto la función de salida `printf()`, encontraremos el uso de la función de entrada `scanf()` mucho más fácil, ya que ambas tienen muchas características en común. La forma general de la función `scanf()` es la siguiente:

```
scanf("cadena de control", &var0, &var1, &var2, ...)
```

En este caso, la cadena de control especifica cómo se convertirán los caracteres introducidos mediante el teclado en los valores correspondientes para luego guardarlos en la lista de variables. La función `scanf()` cambia los valores de las variables, asignandoles los valores que haya introducido el usuario desde el teclado.

Todos los especificadores de formato de la Tabla 4.1 se pueden usar también con la función `scanf()`. La cadena de control se procesa de izquierda a derecha y, cada vez que aparece un especificador de formato, la función `scanf()` trata de interpretar lo que se tipee como un valor.

Se puede usar una única invocación de la función `scanf()` para leer varios valores desde el teclado. Si se introducen varios valores, se asume que los mismos están separados por espacios —presionado la barra espaciadora o presionado la tecla `ENTER` todas las veces que deseemos. Por ejemplo, la instrucción

```
scanf("%f %d %d", &x, &i, &j);
```

lee desde el teclado un valor de tipo `float` y dos valores de tipo `int`, respectivamente, y los guarda en las variables `x`, `i` y `j` —el primero en `x`, el segundo en `i` y el tercero en `j`.

4.2 El programa `Nota de Taller de Utilitarios`

Veamos un programa un poco más complejo que `Hola Mundo`. En el programa que vemos en el Código 4.1 se introducen tres notas desde el teclado y se calcula la calificación definitiva de un estudiante. La nota se determina de la siguiente manera: la nota del examen parcial representa el 30% del total, la nota del trabajo práctico representa el 20% y la nota del final el 50%.

La primera instrucción declara cuatro variables reales: `parcial`, `tp`, `final` y `cal`. Los tres pares de `printf()`'s y `scanf()`'s se encargan de pedirle al usuario que ingrese los datos y los guardan en las variables respectivas. El procesamiento de datos del programa está centrado en las multiplicaciones y sumas para determinar la nota definitiva, que se realizan en la asignación a la variable `cal`. Por último, se muestra el resultado en pantalla.

▷ El contenido de este capítulo fue realizado recopilando información de [3, 7, 9, 11, 12].

Código 4.1 Determinado la nota definitiva de Taller de Utilitarios.

```
#include <stdio.h>

int main()
{
    float parcial, tp, final, cal;

    /* entrada de datos */
    printf("Ingresá la nota del parcial: ");
    scanf("%f", &parcial);
    printf("Ingresá la nota del TP: ");
    scanf("%f", &tp);
    printf("Ingresá la nota del final: ");
    scanf("%f", &final);

    /* cálculo de la calificación definitiva */
    cal = parcial*0.3 + tp*0.2 + final*0.5;

    /* mostramos el resultado */
    printf("La calificación definitiva es %0.2f.\n", cal);

    return 0;
}
```

Capítulo 5

Operadores y expresiones

Los operadores, las variables y las constantes son las unidades elementales de las expresiones. Una expresión en C es cualquier combinación válida de estos objetos. Como la mayoría de las expresiones sigue las reglas generales algebraicas, en general no es necesario dar una presentación detallada de estos elementos del lenguaje. Sin embargo, existen algunos aspectos relevantes de las expresiones que son específicos del lenguaje C.

5.1 Operadores y expresiones aritméticas

Las operaciones matemáticas de adición, sustracción, multiplicación y división se realizan en C usando los operadores aritméticos que se listan en la Tabla 5.1.

Operador	Operación	Ejemplo
`+`	adición	`c = a+b;`
`-`	sustracción	`c = a-b;`
`*`	multiplicación	`c = a*b;`
`/`	división	`c = a/b;`
`%`	resto de la división entera	`c = a%b;`

Tabla 5.1: Lista de operadores aritméticos.

Se pueden combinar varios operadores aritméticos en una misma expresión, como por ejemplo:

```
e = ( (a+b)/(7*x) ) - 3*x/y + d%2
```

El operador `%` representa al resto de la división entera —también conocido como módulo de la división entera— y se utiliza de la siguiente manera:

```
c = 29%3;
```

Luego de la ejecución de esta instrucción, `c` contendrá el valor entero 2. Otro ejemplo:

```
r = 10%5;
```

En este caso, la variable `r` contendrá el valor entero 0 después de la asignación. El operador `%` unicamente se puede usar con valores de tipo entero, de lo contrario el compilador mostrará un mensaje de error y abortará la compilacion.

La división en C usa aritmética real o entera de acuerdo con sus operandos. Por ejemplo, consideremos el siguiente bloque de instrucciones:

```
a = 10;
b = 3;
c = a/b;
```

El valor de la variable `c` depende de cómo se hayan declarado las variables. Si las variables `a`, `b` y `c` fueron declaradas como `float`, entonces la variable `c` tiene el valor 3.333333. En cambio, si `a`, `b` y `c` fueron declaradas como `int`, entonces la variable `c` tiene el valor 3. Este comportamiento sigue de las siguientes reglas generales:

- En las divisiones de valores enteros el resultado también es entero. Por ejemplo, la división entera 29/3 da como resultado el valor entero 9.
- En las divisiones de valores reales el resultado también es real. Por ejemplo, la división 29.0/3.0 da como resultado 9.666666.
- En las divisiones de valores enteros con valores reales, el resultado es real. Por ejemplo, la división 29.0/3 da como resultado 9.666666.

Se pueden mezclar libremente tipos de datos `int`, `long`, `float` y `double` en una expresión aritmética. En todos los casos, los valores de menor precisión se convierten al tipo de los valores de mayor precisión que se usen en la expresión considerada. Por ejemplo, en la expresión `f*i`, donde `f` es un `float` e `i` es un `int`, la variable `i` se convierte a `float` para luego realizar la multiplicación. El resultado final es un `float`, pero este valor puede ser asignado a otro tipo de datos y la conversión pertinente se realizará en forma automática. Si se asigna un valor a una variable de menor precisión entonces el valor es truncado.

5.1.1 Operadores de incremento y decremento

Existen dos operadores muy útiles en el lenguaje C para incrementar y decrementar variables. El operador `++` incrementa en uno a su operando, mientras que el operador `--` decrementa en uno a su operando. Por ejemplo, la instrucción

```
++x;
```

le suma 1 al valor previo de la variable `x`. Esta instrucción es equivalente a:

```
x = x+1;
```

Veamos ahora un ejemplo para el decremento. La instrucción

```
--x;
```

le resta uno al valor previo de la variable `x`. Es decir, esta instrucción es equivalente a:

```
x = x-1;
```

Los operadores de incremento y de decremento se pueden usar tanto como *prefijos*, es decir, antes de la variable, como en:

```
--x;
++y;
```

o como *postfijos*, es decir, después de la variable, como en:

```
x--;
y++;
```

El efecto es el mismo en ambos casos. Sin embargo, existe una diferencia cuando se los utiliza en una expresión. En la forma prefija del operador de incremento o decremento, la operación de incremento o decremento se lleva a cabo *antes* de utilizar el valor de la variable. En cambio, en la forma postfija de dichos operadores, se incrementa o decrementa el valor de la variable *después* de utilizar el valor de la variable. Por ejemplo, en el siguiente código:

```
x = 5;
y = ++x; /* a y se le asigna 6 */
```

a la variable `x` se le asigna el valor 5, luego se incrementa `x` en uno y, por último, a `y` se le asigna el valor actual de `x`, es decir, el valor 6. Sin embargo, si se escribe este código como:

```
x = 5;
y = x++; /* a y se le asigna 5 */
```

a la variable `x` se le asigna el valor 5, luego a `y` se le asigna el valor actual de `x`, es decir, el valor 5, y por último, se incrementa `x` en uno. En ambos casos la variable `x` termina conteniendo el valor 6, la diferencia radica en el momento en el que se cambia el valor.

Es importante notar que los operadores de incremento y decremento sólo pueden aplicarse a variables. Una expresión como:

```
(i+j)++; /* ESTO ES INCORRECTO! */
```

es ilegal.

5.1.2 Abreviaciones de asignaciones

Además del operador de asignación `=` que vimos previamente, se incluyen operadores de asignación abreviados que representan variaciones del operador de asignación tradicional (ver Tabla 5.2). Por ejemplo, el operador `+=` suma y asigna, como en:

```
x += 2.5;
```

Esta instrucción puede entenderse como "sumar `x` a `2.5` y luego asignar el resultado a `x`", con lo cual es equivalente a:

```
x = x+2.5;
```

En algunos casos esta notación abreviada puede ser un poco más rápida para el compilador. Además, es útil cuando trabajamos con expresiones largas.

Operador	Significado	Ejemplo
`+=`	suma y asigna	`suma_parcial+=h;`
`-=`	resta y asigna	`totales-=0.21;`
`*=`	multiplica y asigna	`producto*=3+x/2;`
`/=`	divide y asigna	`den/=3;`
`%=`	calcula el resto de la división entera y asigna	`num%=4;`

Tabla 5.2: Operadores de asignación.

Observar que una expresión como:

```
x *= 3+y;
```

es equivalente a:

```
x = x * (3+y);
```

y no a:

```
x = x*3 + y;
```

5.2 Operadores y expresiones relacionales y de equivalencia

Los símbolos utilizados para hacer comparaciones difieren de los símbolos tradicionales en el lenguaje C. Para comparar dos valores, variables o expresiones se usan los símbolos que se muestran en la Tabla 5.3. Notar los operadores utilizados para las comparaciones por igualdad y desigualdad.

Operadores	Significado
`<`	menor que
`>`	mayor que
`<=`	menor o igual a
`>=`	mayor o igual a
`==`	igual a
`!=`	distinto a

Tabla 5.3: Operadores relacionales y de equivalencia.

Por ejemplo, la expresión:

```
a < 0
```

es verdadera si la variable `a` contiene un valor menor que cero; en caso contrario (es decir, si `a` contiene un valor negativo o igual a cero) es falsa. Para verificar la condición de igualdad usamos la expresión:

```
a == 0
```

Esta expresión es verdadera si y sólo si la variable `a` contiene el valor cero. Es de suma importancia no confundir el operador de equivalencia `==` con el símbolo de asignación `=` que vimos en la Sección 3.4.

5.3 Operadores y expresiones lógicas

Los bloques básicos de las expresiones lógicas son las *proposiciones*, y las expresiones se forman combinando proposiciones por medio de operadores adecuados. Las expresiones relacionales y expresiones lógicas devuelven el valor `true` en caso de verdadero y `false` en caso de falso (ver Tabla 5.4).

Operador	Significado	Ejemplo	Comentario
`&&`	y lógico	`p&&q`	`true` si ambos son verdaderos, `false` si alguno de los dos es falso.
`||`	ó lógico	`p||q`	`true` si cualquiera de los dos es verdadero, `false` si ambos son falsos.
`!`	negación	`!p`	`true` si `p` es falso, `false` si `p` es verdadero.

Tabla 5.4: Operadores booleanos.

Una *proposición* es una afirmación indivisible, de la cual se puede decir que es verdadera o es falsa. Si estamos hablando sobre los números naturales, entonces la afirmación "el 1 es menor que el 2" es una proposición —que además es verdadera. Por su parte, la afirmación "el 3 es menor que el 1" es una proposición —y que obviamente es falsa. Se pueden combinar varias operaciones en una expresión:

```
27 > 5+13 && !(9<8) || 3<=4
```

En este caso, el resultado es verdadero.

Cabe aclarar que en el lenguaje C se usa *short circuit evaluation* para las expresiones lógicas: una vez que se pudo determinar la verdad o la falsedad de una expresión lógica, se detiene la evaluación del resto de la expresión. Por ejemplo, supongamos que la variable `d` tiene el valor 2 en la siguiente expresión lógica:

```
( d==2 || (a && b) || (c <= d) );
```

Como la primera proposición es verdadera, se determina inmediatamente que la totalidad de la expresión es verdadera sin evaluar el resto de la misma.

5.3.1 Los valores lógicos `true` y `false`

Si queremos trabajar con variables que van a guardar el resultado de una expresión lógica, debemos declararlas para tal fin. Si observamos la Tabla 3.1, veremos que no existe ningún tipo de datos lógico. En la especificación original del lenguaje C no se incluía el tipo de datos lógico.

El tipo de datos lógico `bool` es una referencia a un tipo de datos entero de sólo dos valores, ya que los valores lógicos `false` y `true` son simplemente "sinónimos" de 0 y 1 respectivamente. No es estrictamente necesario utilizar el tipo de datos `bool` —aunque es mucho más practico y se recomienda su uso— dado que cualquier valor distinto de cero es tomado como verdadero, y sólo el cero es considerado falso. Para declarar variables del tipo `bool` escribimos:

```
bool b, esta;
```

similar a las declaraciones anteriores. Y para asignarles valores:

```
b = true;
esta = (a==b) || (b<0);
```

también del mismo modo que con las variables de otros tipos de datos.

Debido a que la especificación original del lenguaje C no incluía el tipo de datos lógico, la inclusión del mismo se hizo posteriormente mediante el agregado de la biblioteca `<stdbool.h>`, parte del estándar C99. Por ende, para poder utilizar el tipo de datos `bool` previamente hay que incluir en el archivo de código el header `<stdbool.h>`, mediante la directiva del preprocesador `#include <stdbool.h>`.

Podemos valernos de la interpretación de los valores lógicos por parte del lenguaje C —distinto de cero es verdadero y cero es falso— para escribir código más conciso.

Código 5.1 Ejemplo del uso de valores lógicos.

```
#include <stdio.h>
#include <stdbool.h>

int main()
{
    int a,b;
    bool cero;

    printf("Ingresá a y b: ");
    scanf("%d %d", &a, &b);

    cero = a+b; /* recordar que todo valor distinto de cero es true */
    printf("cero = %d\n", cero);

    return 0;
}
```

En el Código 5.1 vemos un programa que muestra un 1 en pantalla si el resultado de la suma `a+b` es distinto de cero y muestra un 0 si el resultado es cero.

5.4 Orden de precedencia en las expresiones

En la Tabla 5.5 se muestra la precedencia de los operadores aritméticos. Se pueden utilizar paréntesis para alterar el orden de evaluación. Los paréntesis fuerzan a que una operación o conjunto de operaciones tenga un nivel de precedencia mayor. El uso de paréntesis redundantes o adicionales en una expresión no produce errores ni disminuye la velocidad de ejecución de la expresión. Conviene utilizar paréntesis para hacer más claro el orden en que se producen las evaluaciones, en especial cuando la expresión es muy complicada.

	Operadores	Asociatividad
$>$	`++ --`	der. a izq.
	`-`	der. a izq.
	`* / %`	izq. a der.
$<$	`+ -`	izq. a der.

Tabla 5.5: Precedencia de los operadores aritméticos.

Tanto los operadores relacionales como los lógicos tienen un nivel de precedencia menor que los operadores aritméticos. Esto significa que una expresión como $7 > 1 + 12$ se evalúa como si se hubiera escrito $7 > (1 + 12)$.

En la Tabla 5.6 se muestra la precedencia relativa entre los operadores relacionales y los operadores lógicos.

	Operadores	Asociatividad
>	`!`	der. a izq.
	`< <= > >=`	izq. a der.
	`== !=`	izq. a der.
	`&&`	izq. a der.
<	`||`	izq. a der.

Tabla 5.6: Precedencia de los operadores relacionales y lógicos.

Como en el caso de las expresiones aritméticas, en las expresiones relacionales y lógicas se pueden usar paréntesis para alterar el orden natural de evaluación. Por ejemplo:

```
!false && false || false
```

es falso. Sin embargo, si se agregan paréntesis en esta misma expresión como se muestra a continuación, el resultado es verdadero:

```
!(false && false) || false
```

A modo de resumen del contenido de esta sección, en la Tabla 5.7 se muestran las precedencias de todos los operadores.

	Operadores	Asociatividad
mayor	`( ) [ ] -> .`	izq. a der.
	`! ~ ++ -- + - (tipo) * & sizeof`	der. a izq.
	`* / %`	izq. a der.
	`+ -`	izq. a der.
	`<< >>`	izq. a der.
	`< <= > >=`	izq. a der.
	`== !=`	izq. a der.
	`&`	izq. a der.
	`^`	izq. a der.
	`|`	izq. a der.
	`&&`	izq. a der.
	`||`	izq. a der.
menor	`?:`	der. a izq.
	`= += -= *= /= %= &= ^= |= <<= >>=`	der. a izq.
	`,`	izq. a der.

Tabla 5.7: Precedencia de los distintos operadores.

▷ El contenido de este capítulo fue realizado recopilando información de [3, 7, 8, 9, 11, 12].

Capítulo 6

Estructuras de control

Las *estructuras de control* en un lenguaje de programación imperativo especifican en qué orden y cuántas veces se ejecutarán las instrucciones. Las estructuras de control incluyen principalmente dos tipos de estructuras. Por un lado, las estructuras condicionales controlan el flujo del programa, haciendo que un bloque específico de instrucciones se ejecute si se cumple una determinada situación. Por otro lado, las estructuras iterativas provocan la ejecución repetida de un bloque de instrucciones.

6.1 La estructura condicional `if-else`

La primera estructura de control que veremos es la *estructura condicional*, tambien conocida como *instrucción condicional*. Esta instrucción permite elegir entre dos caminos a seguir dependiendo del estado de la ejecución.

Supongamos que nos interesa saber si el contenido de la variable `total` es negativo. Para esto, debemos verificar si la expresión $total < 0$ es verdadera o falsa. Si la expresión $total < 0$ es verdadera decimos "Sí es negativo", y en caso contrario no decimos nada. Realizamos esta acción por medio de la siguiente instrucción alternativa:

```
if ( total<0 )
    printf("Si es negativo.\n");
```

La expresión $total < 0$ se denomina la *guarda* de la alternativa, y la instrucción a continuación se denomina la *rama afirmativa* o *rama positiva*. Cuando se ejecuta esta instrucción, se evalúa el valor de la guarda en el estado actual (es decir, con los valores actuales de las variables). Si la guarda evalúa a `true`, entonces se ejecuta la rama afirmativa. En caso contrario (si la guarda evalúa a `false`) no se realiza ninguna acción y se contiúa con la instrucción que se encuentre a continuación de la alternativa.

Supongamos ahora que el contenido de la variable `total` puede ser solamente positivo o negativo. Si $total < 0$ es verdadero decimos "Es negativo", y en caso contrario decimos "Es positivo". La forma de expresar esta decisión en C es:

```
if ( total<0 )
    printf("Es negativo.\n");
else
    printf("Es positivo.\n");
```

En este caso, la instrucción que muestra el mensaje "Es positivo" se denomina la *rama negativa* de la alternativa. Cuando una instrucción alternativa tiene ramas afirmativa y negativa, se

evalúa el valor de la guarda, y se ejecuta la rama positiva si la guarda evalúa a `true` y la rama negativa si la guarda evalúa a `false`.

¿Qué sucede si la variable `total` puede ser igual a cero? Para decidir entre más de dos opciones podemos "anidar" `if`'s uno dentro de otro, como se ve a continuación:

```
if ( total<0 )
    printf("Es negativo.\n");
else
    if ( total==0 )
        printf("Es igual a cero.\n");
    else
        printf("Es positivo.\n");
```

En este caso, la rama negativa de la *alternativa exterior* es a su vez otra alternativa (la *alternativa interior*). Cuando la guarda de la alternativa exterior evalúa a `false`, se ejecuta la alternativa interior, que involucra evaluar la guarda $total == 0$ y ejecutar la rama correspondiente de esta alternativa. Podemos especificar un `if` dentro de otro y así sucesivamente todas las veces que sean necesarias.

La forma general de la instrucción condicional `if-else` es la que se muestra en el Código 6.1. Si la expresión lógica es verdadera, entonces se ejecuta el primer bloque de instrucciones. En caso contrario, se ejecuta el segundo bloque de instrucciones. Cuando una rama consiste de más de una instrucción, se deben encerrar estas instrucciones entre llaves. Las llaves son opcionales cuando una rama está compuesta por una sola instrucción.

Código 6.1 Estructura genérica de la instrucción condicional `if-else`. El termino `boolop` es uno de los operadores lógicos `&&` y `||` (ver Sección 5.3).

```
if (condicion0 boolop condicion1 boolop condicion2 boolop ... )
{
   instruccion0;
   instruccion1;
   instruccion2;
   ...
}
else
{
   instruccion0;
   instruccion1;
   instruccion2;
   ...
}
```

En el Código 6.2 vemos un ejemplo del uso de una alternativa en un programa completo. Este programa pide al usuario dos números y determina si el primero es múltiplo del segundo. Si el segundo número es cero, el programa muestra un mensaje pertinente.

6.1.1 Igualdad vs. asignación

En el lenguaje C se usan dos signos igual `==` para verificar la condición de igualdad, dado que el signo igual `=` se usa siempre como operador de asignación (ver Sección 3.4). Esto suele causar muchos problemas cuando se está aprendiendo C, ya que es común escribir por error:

Código 6.2 Determinando si un número es multiplo de otro.

```
#include <stdio.h>

int main()
{
    int a,b;

    printf("Ingresá un número: ");
    scanf("%d", &a);
    printf("Ingresá otro número: ");
    scanf("%d", &b);

    if ( b!=0 )
        if ( a%b==0 )
            printf("%d SI es multiplo de %d.\n", a, b);
        else
            printf("%d NO es multiplo de %d.\n", a, b);
    else
        printf("Segundo número igual a cero. Probá de nuevo.\n");

    return 0;
}
```

```
if ( a=10 )
    instruccion0;
else
    instruccion1;
```

en lugar de:

```
if ( a==10 )
    instruccion0;
else
    instruccion1;
```

Es importante tener en cuenta que la sentencia `if ( a=10 )` es válida y, por lo tanto, no produce mensajes de error en el momento de la compilación. En este caso, se asigna el valor 10 a la variable `a`, y la expresión toma el valor asignado (que, como es distinto de cero se interpreta como verdadero).

Analicemos el siguiente fragmento de código:

```
if ( (y=sin(x)) != 0 )
    z = x/y;
else
    printf("El seno de %f es cero.\n", x);
```

Si la variable `y` es igual a cero, la instrucción `z = x/y` no es válida, ya que no se puede hacer la división. La instrucción `y=sin(x)` almacena el resultado de `sin(x)` en la variable `y`. A continuación, con la expresión lógica `( (y=sin(x)) != 0 )` preguntamos si el valor que se le asignó a la variable `y` es distinto de cero.

El código anterior es válido porque el operador de asignación devuelve el valor asignado, además de asignar el valor en cuestión, de la misma forma en la una función devuelve un valor. Además, una asignación se puede anidar dentro de otras expresiones. Si la intención del programador es comparar dos valores en una sentencia `if` pero usa el operador de asignación `=` en lugar del operador de comparación `==`, la asignación devolverá un valor que puede ser interpretado como `true` (en caso de asignarse un valor distinto de cero) o como `false` (en caso de asignarse el valor cero), causando que la instrucción `if` aparentemente funcione bien, cuando en realidad no se están comparando los valores.

6.2 Estructuras de control iterativas

Las construcciones que permiten ejecutar un bloque de instrucciones repetidamente se llaman *estructuras de control iterativas*, también conocidas como *ciclos*.

6.2.1 El ciclo `for`

Supongamos que necesitamos mostrar 1000 veces el mensaje "Hola Mundo" en pantalla. Podemos resolver este problema con un ciclo `for`, que repita esta cantidad de veces una instrucción `printf` con el mensaje. Para esto, introducimos una variable `i` que cuenta el número de veces que el ciclo se repetirá, y la condición de terminación `i<1000` para establecer la finalización del ciclo. El problema de escribir 1000 veces el texto "Hola Mundo" queda resuelto con el siguiente ciclo:

```
for (int i=0; i<1000; ++i)
    printf("%d Hola Mundo\n", i);
```

La instrucción `for` consta de tres partes separadas por punto y coma. La primera parte se denomina la *inicialización* del ciclo `for`, y en este caso consta de la instrucción `int i=0` que fija el valor inicial de la variable `i` en cero. La variable `i` se denomina la *variable de control* del ciclo, y cuenta la cantidad de iteraciones realizadas hasta el momento. Por este motivo, se inicializa en cero y se le suma una unidad luego de cada iteración. Este incremento se realiza en el *paso del ciclo* `++i` especificado en la tercera parte del ciclo `for`. Por último, la segunda parte especifica la *guarda* del ciclo, que indica la condición que se debe cumplir para que el ciclo siga iterando. En este caso, se especifica que el ciclo debe seguir iterando mientras el valor de `i` sea menor a 1000, con lo cual se realizan exactamente 1000 repeticiones de la instrucción `printf` que se encuentra a continuación.

La instrucción `printf` se denomina el *cuerpo del ciclo*, y es la instrucción que se ejecuta repetidamente, con los valores de la variable `i` según especifique la declaración del ciclo. Del mismo modo que en las ramas de una instrucción alternativa, el cuerpo del ciclo se debe encerrar entre llaves si está formado por más de una instrucción, y las llaves son optativas si el cuerpo del ciclo está compuesto por una sola instrucción. Es importante observar que no hay restricciones al tipo de instrucciones que se pueden ubicar en el cuerpo del ciclo: puede tratarse de asignaciones, alternativas, u otros ciclos.

En el Código 6.3 podemos ver un ejemplo de un ciclo `for` con una estructura similar al ejemplo anterior. En este programa se suman los primeros 36 números naturales y se muestra por pantalla el resultado. Observar que este programa no le pide datos al usuario, sino que siempre suma los primeros 36 números naturales. La variable de control `i` se inicializa en el valor 1, y el paso `i++` incrementa esta variable para que las sucesivas *iteraciones* (ejecuciones del cuerpo del ciclo) se realicen con la variable `i` tomando todos los valores enteros entre 1 y 36.

En el Código 6.4 se puede ver la estructura genérica de un ciclo `for`. La instrucción de inicialización puede ser cualquier instrucción, aunque normalmente consiste de la asignación

Código 6.3 Programa que calcula la suma de los primeros 36 números naturales.

```
#include <stdio.h>

int main()
{
   int sum=0;
   for (int i=1; i<=36; i++)
      sum+=i;

   printf("La suma de los primeros 36 números naturales es %d.\n", sum);
   return 0;
}
```

de un valor inicial a la variable de control. El paso del ciclo también puede ser cualquier instrucción. Normalmente consiste de una actualización a la variable de control, pero puede ser una instrucción arbitraria e incluso ser una instrucción vacía (dejando un espacio en blanco entre el punto y coma `;` y el paréntesis que cierra la declaración del ciclo `for`). En particular, es importante notar que un ciclo `for` no está limitado a incrementar o decrementar la variable de control en una unidad, sino que se pueden realizar actualizaciones más sofisticadas en caso de que sea necesario.

Código 6.4 Estructura genérica del ciclo `for`.

```
for (inicializacion; guarda; paso)
{
   instruccion0;
   instruccion1;
   instruccion2;
   ...
}
```

6.2.2 El ciclo `while`

El ciclo `while` es una estructura de control iterativa cuyo propósito es repetir la ejecución de un bloque de instrucciones mientras una condición se mantenga verdadera. En el lenguaje C, este tipo de ciclos es un caso particular de los ciclos `for`, pero se cuenta con una notación especial dada su importancia como estructura de control de la programación estructurada.

El Código 6.5 muestra la estructura general de un ciclo `while`. La expresión lógica del ciclo (las condiciones encerradas entre paréntesis en la primera línea) es la *guarda* del ciclo. En cada iteración se evalúa la guarda. Si resultado es verdadero, se ejecuta una vez más el bloque de instrucciones, repitiendo el procedimiento. Cuando la guarda se hace falsa —las variables que participan en la guarda deben modificarse en algún momento mediante alguna instrucción dentro del cuerpo del ciclo— se termina la ejecución del bloque de instrucciones y se pasa a ejecutar la próxima instrucción que se encuentra inmediatamente debajo del ciclo. Del mismo modo que en las restantes estructuras de control, el cuerpo del ciclo debe encerrarse entre llaves si está compuesto por más de una instrucción, y estas llaves son optativas si el cuerpo del ciclo consta de una sola instrucción.

Código 6.5 Estructura genérica del ciclo `while`. El termino `boolop` representa a cualquiera de los operadores booleanos de la Tabla 5.4.

```
while (condicion0 boolop condicion1 boolop condicion2 boolop ... )
{
   instruccion0;
   instruccion1;
   instruccion2;
   ...
}
```

Código 6.6 Ejemplo de un ciclo `while` que suma los primeros `n` números naturales.

```
#include<stdio.h>

int main()
{
    int n, i, sum;

    printf("Ingrese un entero positivo:\n > ");
    scanf("%d", &n);

    i = 0;
    sum = 0;
    while ( i <= n )
    {
        sum = sum + i;
        ++i;
    }

    printf("La suma es %d.\n", sum);
    return 0;
}
```

En el Código 6.6 se puede ver un programa que suma los primeros `n` números naturales, pidiendo al usuario el valor de esta variable. Es importante notar que en un ciclo `while` la variable de control del ciclo se debe modificar "manualmente" dentro del cuerpo del ciclo (a diferencia de un ciclo `for`, en el cual el incremento o decremento de la variable de control se realiza en la cabecera del ciclo). Por este motivo, antes de comenzar el ciclo se asigna el valor 0 a la variable de control `i`, que además se incrementa en 1 en cada iteración.

▷ El contenido de este capítulo fue realizado recopilando información de [3, 7, 11, 12].

Capítulo 7

Funciones en el lenguaje C

Todo programa en C consta de un conjunto de *funciones* que se invocan entre sí. La única función que debe estar siempre presente es la función llamada `main()`. Sin embargo, es usual que en un mismo programa se incluyan varias funciones. En el Código 7.1 se muestra forma general de un programa en lenguaje C.

7.1 Las funciones

Una función es un conjunto encapsulado de sentencias que se agrupan bajo un nombre determinado. Las funciones en el lenguaje C son el equivalente de lo que otros lenguajes llaman subrutinas o procedimientos.

Una función permite usar una misma sección de código repetidas veces sólo invocando su nombre. Sin embargo, las funciones tienen un propósito aun más importante. En el desarrollo de un gran proyecto de software las funciones permiten a los programadores dividir el código en pequeñas unidades para poder trabajar independientemente.

Existe cierta libertad para escoger el lugar donde puede ir la definición de una función. El único requisito es que debe conocerse el tipo de la función antes de poder utilizarla. Normalmente, se declara la función al comienzo del programa separada de la definición de la misma, como se puede ver en el Código 7.2.

7.1.1 Nombres válidos para las funciones

Los nombres de variables, constantes simbólicas, funciones y otros objetos definidos por el usuario se conocen como identificadores. Los primeros identificadores que vimos fueron los nombres de las variables en la Sección 3.1.

Las reglas para especificar nombres válidos para las funciones son las mismas que para las variables. Todos los identificadores consisten de letras (recordemos que el compilador distingue las letras minúsculas de las mayúsculas), números y underscores _ siempre que cumplan con la regla de que el primer caracter no puede ser un número. En C los nombres que comienzan con un underscore _ se usan generalmente para diferenciar identificadores especiales del sistema.

7.1.2 Variables locales

Una función es una subunidad de programa —un miniprograma dentro del programa principal. Las variables pertenecientes a una función se declaran de la misma manera en que se declaran en el programa principal.

Código 7.1 Estructura genérica de un programa escrito en lenguaje C.

```
/* directivas del preprocesador */

#include <header0.h>    /* inclusión de bibliotecas estándar */
#include <header1.h>
#include <header2.h>
...
#define CONST0 valor0   /* definición de constantes simbólicas */
#define CONST1 valor1
#define CONST2 valor2
...

/* declaraciones globales */

tipo funcion0(tipo0, tipo1, tipo2, ... );   /* declaraciones de funciones */
tipo funcion1(tipo0, tipo1, tipo2, ... );
tipo funcion2(tipo0, tipo1, tipo2, ... );
...

/* definiciones de funciones */

tipo main(tipo0, tipo1, tipo2, ... )
{
   tipo0 var0,var1,var2, ... ;
   tipo1 var0,var1,var2, ... ;
   tipo2 var0,var1,var2, ... ;
   ...
   instruccion0;
   instruccion1;
   instruccion2;
   ...
   return expresion;
}

tipo funcion0(tipo0, tipo1, tipo2, ... )
{
   /* ídem función main() */
}
tipo funcion1(tipo0, tipo1, tipo2, ... )
{
   /* ídem anterior */
}
tipo funcion2(tipo0, tipo1, tipo2, ... )
{
   /* ídem anterior */
}
...
```

Código 7.2 Esquema de la utilizacion de una función típica en C.

```
#include <stdio.h>
#include <math.h>

/* declaración de la funcion_ejemplo() */
int funcion_ejemplo(float parametro0, char parametro1);

int main()
{
   float entrada0;
   char entrada1;
   int resultado;

   /* entrada de datos */
   /* inicializacion de variables */
   /* todas las instrucciones que fueran necesarias */

   /* se le pasa a funcion_ejemplo los argumentos entrada0 y entrada1 */
   resultado = funcion_ejemplo(entrada0,entrada1);

   /* todas las instrucciones que fueran necesarias */

   return 0;
}

/* definición de la funcion_ejemplo() */
int funcion_ejemplo(float parametro0, char parametro1)
{
   /* todo lo que se necesite que haga la función */
}
```

Una *variable local* es una variable que pertenece *sólo* a la función donde fue declarada. Se puede usar una variable local solamente en la función donde fue declarada. Una variable local a una función no existe en las otras funciones del programa, y no está relacionada con las variables del mismo nombre declaradas en otras funciones. Las variables locales son creadas cuando se invoca la función y desaparecen cuando la función termina. La funciones están aisladas del resto del programa: ninguna variable local sobrevive cuando la función termina. Debido a este hecho se dice que las funciones son segmentos aislados y encapsulados de código.

7.1.3 Parámetros de la función

El código que invoca a una función puede "pasarle" datos y recibir los resultados que "devuelve" la función. Los paréntesis alrededor del nombre de un función cumplen este primer rol de tomar valores.

Se pueden definir variables especiales llamadas *parámetros* que se usan para pasarle valores a la función. Estos parámetros se listan y declaran entre los paréntesis en la definición de la función. Por ejemplo, en el siguiente código:

```
int sumar(int a, int b)
{
    int resultado;
    resultado=a+b;
    return resultado;
}
```

se define una función llamada `sumar()` que recibe dos parámetros enteros `a` y `b`.

Es importante observar que la variable `resultado` se declara de la misma manera como lo hacíamos usualmente en la función `main()`. Los parámetros `a` y `b` se usan dentro de la función del mismo modo que una variable local. La única diferencia entre un parámetro y una variable local consiste en que los parámetros reciben valores iniciales cuando la función se invoca. Para esto, se escriben argumentos entre los paréntesis cuando invocamos a la función:

```
a = sumar(28,17);
```

donde los argumentos 28 y 17 se pasan como parámetros a la función. Esta instrucción ejecuta el código de la función `sumar()` que se encarga de sumar dichos valores, y el resultado (en este caso, 45) es asignado a la variable `a`.

Los argumentos también pueden ser expresiones cuyos resultados se pasan como parámetros:

```
a = sumar(x+2, 3*y);
```

donde el primer parámetro de la función recibirá el valor al cual evalúe `x+2` y el segundo parámetro recibirá el valor al cual evalúe `3*y`. Generalmente usaremos el termino *parámetro* para una variable nombrada en la lista entre paréntesis en la definición de una función, y *argumento* para el valor empleado al invocar a la función.

7.1.4 La instrucción `return`

Veamos ahora como obtener los resultados de las mismas. La forma más directa para obtener valores de una función es usar la instrucción `return`. El valor que la función devolverá se especifica con la instrucción:

```
return expresion;
```

donde `expresion` puede ser cualquier expresión válida en C —por ejemplo, `y-(x+7)/3` ó `x==2*y`. La expresión también puede ser simplemente una variable o un valor constante.

La instrucción `return` se puede escribir en cualquier parte dentro de la función, no solamente al final de la misma. Sin embargo, esta instrucción siempre indica la terminacion de la función y regresa el control al proceso invocante.

Una función devuelve un valor "por medio de su nombre", como si el nombre de la función fuera una variable que contiene un valor. Por este motivo, se debe asociar el nombre de la función a un tipo de datos. Para esto se escribe un tipo de datos antes del nombre de la función. Con la sentencia:

```
int sumar(int a, int b);
```

especificamos el tipo de datos de retorno de la función —en este caso, `int`. Se pueden crear funciones con la cantidad de parámetros que queramos pero las funciones siempre devuelven un único valor. Cuando queremos usar la función escribimos, por ejemplo, la instrucción:

```
a = sumar(5,7);
```

que sumará $5 + 7$ y guardará el resultado obtenido en la variable `a`. Es importante observar que la variable `a` que vemos aquí no está relacionada con el parámetro de la función llamado `a` de más arriba.

Adicionalmente, se puede usar esta función para formar expresiones más complejas, como por ejemplo:

```
if ( resultado <= 3 + sumar( 5+x*y, sumar(17,-6)/sin(4) )/(n-13) )
...
```

Por último, es importante observar que siempre hay una proposición `return` al final de la función `main()`. Dado que `main()` es una función como cualquier otra, también puede regresar un valor a quien la invoca, que es en efecto el ambiente en el que el programa se ejecuta. Típicamente, el valor de regreso cero implica la terminación normal. Los valores distintos de cero indican condiciones de terminación no comunes o erróneas, estos valores van desde 1 hasta 255. Recordemos, entonces, que los programas deben regresar su estado final de ejecución a su entorno.

7.1.5 Funciones sin parámetros

No es necesario especificar parámetros en una función si no vamos a pasarle ningún argumento —aún así es necesario el par de paréntesis vacíos. Este hecho se evidencia, por ejemplo, en la definición de la función `main()`:

```
int main()
{
...
}
```

7.1.6 Funciones sin valores de retorno

Una función no tiene necesariamente que devolver un valor. Del mismo modo, el proceso que invoque a una función tampoco tiene que guardar un valor. Por ejemplo, es válido invocar a la función `sumar()` de la siguiente manera:

```
...
sumar(28,17);
...
```

donde el valor de retorno se descarta. El uso de una función de esta forma puede parecer un tanto desprolijo y confuso. Por este motivo se usa el tipo de datos `void` para indicar que la función no devuelve ningún valor. Por ejemplo, el siguiente código:

```
void hola()
{
    printf("Hola Mundo.\n");
}
```

define una función llamada `hola()` que no recibe ningún parámetro y que no devuelve ningún valor. No es necesario el uso de la instrucción `return` que vimos en la Subseccion 7.1.4.

7.1.7 El tipo de datos `void`

El tipo `void` es un "tipo de datos" que se usa como tipo de retorno en las funciones que no producen un resultado directo. Generalmente, a estas funciones no se las invoca para que realicen *cálculos*, sino que son invocadas para que realicen *acciones*. Este tipo especial también puede aparecer en la lista de parámetros del prototipo de una función para indicar que dicha función no toma ningún argumento. Cabe destacar que no esta permitido el uso del tipo `void` en la declaración de variables.

7.2 Las funciones de la biblioteca estándar

La *biblioteca estándar* del lenguaje C es una colección de archivos llamados *headers* y de rutinas que implementan operaciones comunes, tales como la entrada y salida de datos, entre muchas otras. Existe gran variedad de funciones estándar que habitualmente se incluyen con los compiladores de C. Las reglas para el uso de las funciones de la biblioteca estándar son las mismas que se trataron en la Sección 7.1.

Código 7.3 Ejemplo del uso de algunas funciones matemáticas.

```
#include <stdio.h>
#include <math.h>

int main()
{
    float x;

    printf("Valores que toma la función f(x)=|sin(x)+exp(x)|^(1/2):\n");

    for (x=0; x<100; ++x)
        printf("%f %f\n", x, sqrt(fabs(sin(x)+exp(x))) );

    return 0;
}
```

En el Código 7.3 se muestra un ejemplo con algunas de las funciones incluidas en el header `<math.h>` de la biblioteca estándar, y en la Tabla 7.1 se listan algunas de las más relevantes.

Función	Calcula
`atan(x)`	$\arctan(x)$
`sin(x)`	$\sin(x)$
`cos(x)`	$\cos(x)$
`exp(x)`	e^x
`pow(x,y)`	x^y
`sqrt(x)`	$\sqrt{x}$

Tabla 7.1: Algunas funciones matemáticas incluidas la biblioteca estándar, donde `x` e `y` son del tipo `double`.

7.3 Parámetros pasados por valor

En el lenguaje C, todos los parámetros de las funciones se pasan *por valor*. Esto significa que la función invocada recibe los valores de sus argumentos en variables temporales que se crean cuando se invoca la función y de ninguna manera afecta a los valores originales de la función que hizo la invocación.

Código 7.4 La función `potencia()` ejemplifica el concepto de parámetros pasados por valor.

```
#include <stdio.h>

int potencia(int, int);
int main()
{
    int base, n;
    int resultado;

    printf("Ingresá la base: ");
    scanf("%d", &base);
    printf("Ingresá el exponente: ");
    scanf("%d", &n);

    /* Antes de la invocación, n tiene el valor introducido por el usuario */
    resultado = potencia(base,n);
    /* Después de la invocación, n tiene el valor introducido por el usuario */

    printf("%d^%d = %d\n", base, n, resultado);
    return 0;
}

int potencia(int base, int n)
{
    int p;
    for (p=1; 1<=n; --n)
        p*=base;
    /* Al término del for, n tiene el valor 0 */
    return p;
}
```

Los parámetros de una función son variables locales inicializadas con los valores de los argumentos que se le pasen a la función. En el Código 7.4 se puede ver claramente el uso de esta propiedad. El parámetro `n` se usa como una variable auxiliar que va decrementándose hasta que llega al valor cero. Todo lo que se le haga al parámetro `n` dentro de la función no tiene ningún efecto en la variable `n` original de la función `main()` que se le pasó como argumento.

Existe otro método para pasar los argumentos a una función llamado *por dirección*, en el cual la función tiene acceso a los valores originales, sin hacer copias en variables locales a la función. Este método se tratará en la Sección 8.2.

▷ El contenido de este capítulo fue realizado recopilando información de [3, 7, 11, 12].

Capítulo 8

Punteros y arreglos

Los punteros son una muy poderosa herramienta del lenguaje C, que permiten manipular directamente posiciones de memoria RAM. Entender claramente el uso de punteros nos será de gran utilidad para terminar de ver un tema pendiente del Capitulo 7: cómo modificar los valores que se pasan a una función como argumento. Además, nos servirá para comprender mejor el manejo de arreglos dentro del lenguaje.

8.1 Variables y punteros

Como sabemos, una variable es un área de memoria RAM que tiene un nombre dado. Por ejemplo,

```
int total;
```

es una porción de memoria a la cual se le dio el nombre `total`, y que puede almacenar datos del tipo entero. La principal ventaja de este esquema es que se puede usar un nombre fácil de recordar para especificar que se guardarán datos en la memoria RAM, sin la necesidad de saber en qué sector especifico de la memoria se almacenarán dichos datos, y sin la necesidad de saber cuanta memoria se requiere para guardar datos del tipo `int`. Por ejemplo, la instrucción:

```
total = 2;
```

guarda el valor entero 2 en el área de memoria llamada `total` sin preocuparnos por cuanta memoria se necesita para guardar un valor entero.

El concepto de "variable" en los lenguajes de alto nivel está tan asimilado que generalmente nos olvidamos de lo que realmente está sucediendo con la memoria. Sin embargo, existe otra forma de trabajar con la memoria de la computadora. La computadora no accede a la memoria mediante el nombre de las variables, sino que lo hace mediante un mapa con cada posición de la memoria unívocamente referenciada por un número, conocido como *dirección de memoria*.

Un *puntero* es una variable que guarda la *dirección de memoria* de una variable. En términos informales, decimos que un puntero "apunta" a una variable. Un puntero es una variable y, por lo tanto, debe ser declarada como cualquier otra variable. Para declarar un puntero a un tipo de datos determinado se agrega un asterisco `*` antes del nombre de la variable en la declaración de la misma. Por ejemplo, la instrucción:

```
int *p;
```

declara la variable `p` como puntero a un entero. El asterisco se aplica sólo a la variable que lo tiene, de modo que:

Código 8.1 Simple ejemplo que muestra el uso de los punteros.

```
#include <stdio.h>

int main()
{
    int variable;
    int *puntero;
    int valor_apuntado;

    variable = 1232;
    puntero = &variable;
    valor_apuntado = *puntero;

    printf("Valor de la variable          : %d\n", variable);
    printf("Dirección de la variable      : %p\n", &variable);
    printf("Valor del puntero             : %p\n", puntero);
    printf("Valor apuntado por el puntero : %d\n", *puntero);

    return 0;
}
```

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

declara a la variable `p` como puntero a un entero y a la variable `q` como entero.

Luego de la declaración, los punteros se pueden usar como cualquier otra variable. Las operaciones disponibles sobre una variable de tipo puntero están dadas por los operadores `&` y `*`. El operador `&` devuelve la dirección de la variable cuando se lo agrega antes del nombre de una variable. Por ejemplo, con el siguiente código:

```
int *p, q;
p = &q;
```

declaramos la variable `p` como puntero y la variable `q` como entero, y luego asignamos la dirección de memoria de la variable `q` al puntero `p`. Esto último, se puede pensar como "el puntero `p` apunta a la variable `q`".

El operador `*` devuelve el valor guardado en la variable apuntada. Si se agrega el operador `*` antes del nombre de un puntero, entonces el resultado es el valor guardado en la variable apuntada. En otras palabras, si `p` es la dirección de otra variable, entonces `*p` es el valor de la variable apuntada por `p`. No confundir el operador `*` cuando se lo usa para devolver el valor guardado en la variable apuntada con el uso que se le da en la declaración de un puntero.

El Código 8.1 muestra un ejemplo que ilustra los conceptos esenciales sobre punteros. En primer lugar, se declaran tres variables, `variable` y `valor_apuntado` como enteros y `puntero` como puntero a un entero. Luego se guarda en `variable` el valor entero 1232, del modo usual.

En segundo lugar, con la instrucción

```
puntero = &variable;
```

se guarda la dirección de `variable` en `puntero`. Ahora `puntero` apunta a `variable`. Por último, la instrucción

```
valor_apuntado = *puntero;
```

guarda en `valor_apuntado` el valor de la variable apuntada por `puntero`. Como `puntero` apunta a `variable`, se guarda el valor 1232 en `valor_apuntado`. Finalmente, se muestran por pantalla los valores mencionados.

8.2 Funciones y punteros: parámetros por dirección

Consideremos el problema simple de escribir una función que intercambie los contenidos de dos variables enteras. Llamemos a esta función `cambiar()`, de manera que reciba dos argumentos enteros y que intercambie sus valores. En principio, se podría pensar que el siguiente código resuelve el problema:

```
void cambiar(int a, int b);
{
  int aux = a;
  a = b;
  b = aux;
}
```

Ahora bien, ¿es correcta la función `cambiar()`? Si se implementa la función `cambiar()` se encontrará que la misma no tiene ningún efecto. Se la puede usar como:

```
cambiar(a,b);
```

pero no intercambia los valores de `a` y `b` en la función que la invoca.

Recordemos de la Sección 7.3 que todos los parámetros de las funciones en C se pasan por valor. Cuando usamos la función `cambiar()`, los valores de `a` y `b` se pasan a la función mediante los parámetros y los cambios hechos a los parámetros no alteran a los valores de `a` y `b`. Notar que efectivamente se cambian los valores dentro de la función `cambiar()`, pero esto no afecta a los valores en la función que invoca a `cambiar()`.

La solución a este problema es pasar las direcciones de memoria de las variables en lugar de los valores guardados en las variables. La función para intercambiar los valores de dos variables usará punteros para obtener los valores y modificarlos en la función que hace la invocación. Llamemos `swap()` a esta función modificada que usa punteros. Entonces, el código de la función es:

```
void swap(int *a, int *b);
{
  int aux = *a;
  *a = *b;
  *b = aux;
}
```

Ahora, los dos parámetros de la función son punteros y en las asignaciones se usa el operador `*` para cambiar los valores guardados en las variables apuntadas. La instrucción:

```
*a = *b;
```

guarda el valor apuntado por `b` en el lugar donde está guardado el valor apuntado por `a`. El resto de la función se basa en esta misma idea.

Cabe hacer una ultima observación sobre la función `swap()`. Cuando se invoque la función no hay que olvidar que los parámetros de la función son punteros, con lo cual se deben pasar las direcciones de las variables a la función `swap()`. Es decir, la invocación:

```
swap(a, b); /* INCORRECTO */
```

es incorrecta. La forma correcta de pasar las direcciones de memoria de las variables `a` y `b` es la siguiente:

```
swap(&a, &b); /* CORRECTO */
```

Se tiene que ser cuidadoso con los valores o las direcciones que se le pasen a una función. Por ejemplo, si se invoca la función para intercambiar valores como:

```
swap(a, b); /* ERROR */
```

se intercambiarán dos posiciones arbitrarias de memoria, resultando en un error desastroso que podría provocar que el sistema completo quede en un estado inestable.

La necesidad de pasarle las direcciones a una función para que pueda modificar los valores explica la diferencia entre las funciones de entrada y salida que vinimos usando desde el comienzo de la cursada. La función `printf()` no debe cambiar los valores de sus argumentos, por eso es invocada como:

```
printf("%d", a);
```

En cambio, la función `scanf()` debe cambiar los valores de sus argumentos, por eso es invocada como:

```
scanf("%d", &a);
```

8.3 Arreglos

Un *arreglo* es un conjunto de datos homogeneos —es decir, un conjunto del mismo tipo de datos— cuyos elementos se localizan en memoria RAM en forma contigua.

A veces necesitamos trabajar con un conjunto de números o de items de otros tipos. Supongamos que queremos calcular el promedio de cinco números. Una primera aproximación a este problema podría ser crear la cantidad necesaria de variables individuales; creamos 5 variables, leemos 5 valores desde el teclado, calculamos el promedio de estos 5 valores y mostramos el resultado. Por ejemplo, la función para calcular dicho promedio sería la siguiente:

```
float promedio(float a0, float a1, float a2, float a3, float a4)
{
    return (a0+a1+a2+a3+a4)/5;
}
```

No es una función muy complicada. Sólo faltaria escribir la función `main()` para leer los 5 valores y mostrar el promedio calculado. Ahora bien, ¿qué sucede si queremos calcular el promedio de 100 números ó de 1000 números?

La solución es usar un conjunto de variables de la forma `a[i]`, donde `i` es una variable entera que especifica un elemento en particular del conjunto. Al valor entero entre corchetes `[ ]` que se usa para referirse a cada elemento de un arreglo se lo conoce como el *índice* al arreglo.

Para poder usar un arreglo primero debemos declararlo como cualquier otra variable. Para declarar un arreglo de floats de 5 elementos simplemente escribimos:

```
float a[5];
```

donde el primer elemento del arreglo es `a[0]` y el último es `a[4]` —en el lenguaje C los índices de los arreglos comienzan desde cero.

La forma general de la declaración de un arreglo es la siguiente:

```
tipo_de_dato nombre_del_arreglo[cantidad_de_elementos];
```

Es importante tener en cuenta que el primer elemento del arreglo es `nombre_del_arreglo[0]` y que el último elemento es `nombre_del_arreglo[cantidad_de_elementos-1]`.

La asignación de los valores del arreglo se hace uno por uno, es decir:

```
a[2]=0.5;
a[0]=3.49;
a[3]=7.5;
```

También se pueden leer los elementos uno por uno desde teclado con la función `scanf()` de la manera usual:

```
scanf("%f", &a[4]);
scanf("%f", &a[1]);
```

Esto último puede ser molesto si se desea leer la totalidad de los elementos del arreglo. Aquí es donde hacemos uso de los ciclos, en particular del ciclo `for` —podríamos decir que este ciclo fue creado especialmente para trabajar con arreglos. El ciclo `for` se suele usar para generar una secuencia de valores enteros para procesar cada elemento del arreglo, aunque también se puede usar un ciclo `while`. Por ejemplo, para leer los n elementos de un arreglo `a` de números enteros escribimos:

```
for (int i=0; i<n; ++i)
    scanf("%d", &a[i]);
```

Se puede escribir un código similar para mostrar los valores de un arreglo, lo cual queda a cargo del lector.

Utilizando arreglos, el problema de calcular el promedio de n elementos queda resuelto como se ve en el Código 8.2. La lectura de los datos del arreglo y el algoritmo para calcular el promedio se escribieron como funciones. Ambas funciones reciben como parámetros un arreglo de floats y un entero que corresponde a la cantidad de elementos del arreglo.

El lector suspicaz habrá notado que el primer parámetro de las funciones `leer_arreglo()` y `promedio()` son en realidad punteros y que el primer parámetro de la función `leer_arreglo()` se modificó. En el lenguaje C, los arreglos son en realidad punteros. Este tema se trata a continuación.

Código 8.2 Calculo del promedio de n números.

```
#include <stdio.h>

void leer_arreglo(float *arreglo, int n);
float promedio(float *arreglo, int n);

int main()
{
    int n;
    printf("Ingresá la cantidad de números que querés promediar: ");
    scanf("%d", &n);

    float a[n];
    printf("Ingresá los números:\n > ");
    leer_arreglo(a,n);

    printf("El promedio es %f.\n", promedio(a,n) );

    return 0;
}

void leer_arreglo(float *arreglo, int n)
{
    for (int i=0; i<n; ++i)
    scanf("%f", &arreglo[i]);
}

float promedio(float *arreglo, int n)
{
    float sum=0;
    for (int i=0; i<n; ++i)
        sum+=arreglo[i];
    return sum/n;
}
```

8.3.1 Arreglos y punteros

En el lenguaje C existe una conexión muy cercana entre punteros y arreglos. Cuando se declara un arreglo como:

```
int a[10];
```

en realidad se está declarando un puntero al primer elemento del arreglo. La única diferencia entre un arreglo y un puntero es que el arreglo es un puntero constante, es decir, no se puede cambiar la posición de memoria apuntada. Una expresión como `a[i]` se convierte automaticamente en una expresión de puntero que devuelve el valor del elemento apropiado.

8.3.2 Arreglos y funciones

Cuando se usa el nombre de un arreglo como argumento en una función, el valor que le pasamos a la misma es la dirección del primer elemento del arreglo. Por ejemplo, le pasamos un arreglo completo a la función `leer_arreglo()` (ver Código 8.2) de la siguiente forma:

```
leer_arreglo(a,n);
```

Esto nos permite escribir funciones que procesen la totalidad de un arreglo sin tener la necesidad de pasar los valores uno por uno —sólo se necesita pasarle a la función el puntero al primer elemento.

Dado que debemos pasarle a la función la dirección del primer elemento del arreglo, los parámetros del tipo arreglo se declaran de la siguiente forma:

```
tipo nombre_de_la_funcion(tipo *parametro, ... );
```

es decir, se declaran parámetros del tipo puntero. Obviamente se pueden usar todos los parámetros que hagan falta.

Como vimos en la Sección 8.2 al pasarle a la función como parámetros las direcciones de memoria, la función tiene acceso a los datos originales y no a copias locales, de este modo, la función puede alterar el contenido de los parámetros.

La declaración anterior también puede escribirse como:

```
tipo nombre_de_la_funcion(tipo parametro[], ... );
```

De esta forma, los parámetros de tipo arreglo pueden identificarse con mayor claridad.

En otras palabras, se pueden definir los parámetros de la función como punteros o como arreglos. En ambos casos se puede usar la notación de indexado de arreglos.

8.3.3 Inicializacion de arreglos

Para inicializar arreglos se pueden especificar en la declaración los valores de cada elemento entre llaves y los índices entre corchetes. Por ejemplo, la siguiente declaración:

```
int a[6] = { [4]=29, [2]=15 };
```

declara un arreglo de enteros de 6 elementos y especifica que el elemento `a[4]` contenga el valor entero `29` y que el elemento `a[2]` contenga el valor entero `15`. El resto de los elementos del arreglo reciben el valor cero. Es decir, la declaración de arriba es equivalente a:

```
int a[6];
a[0] = 0;
a[1] = 0;
a[2] = 15;
a[3] = 0;
a[4] = 29;
a[5] = 0;
```

Para inicializar un rango de elementos con el mismo valor se escribe el comienzo del rango seguido de tres puntos, luego el final del rango, y por último el valor. Por ejemplo:

```
int a[100] = { [0 ... 9] = 1, [10 ... 98] = 2, [99] = 3 };
```

Si queremos inicializar todos los elementos del arreglo a cero, escribimos:

```
int a[616] = {};
```

como no especificamos ningún valor, inicializa todos los elementos a cero.

Hay que se cuidadoso con los espacios entre los tres puntos y los rangos. Por ejemplo, debemos escribir

```
[1 ... 5]
```

en lugar de

```
[1...5]
```

ya que esto último provocaría error.

8.4 Arreglos de dos dimensiones: Matrices

Los arreglos de dos dimensiones o *bidimensionales* son arreglos rectangulares, cuyos elementos están indexados por fila y columna. Normalmente se identifica este tipo de arreglo con las matrices, dado que son útiles para representar matrices y para realizar cálculos sobre este tipo de objetos.

Para declarar una variable arreglo de dos dimensiones, escribimos:

```
int matrix[4][4];
```

La variable `matrix[]` es un arreglo que contiene 4×4 elementos, es decir, contiene 16 elementos. El primer elemento es `matrix[0][0]` y el último elemento es `matrix[3][3]`.

Para asignarle valores, usamos una instrucción del tipo:

```
matrix[2][1] = 3
```

El método para inicializar arreglos de una dimensión que vimos en la Subseccion 8.3.3 no es válido para arreglos de dos dimensiones.

▷ El contenido de este capítulo fue realizado recopilando información de [3, 7, 9, 11, 12].

Capítulo 9

Archivos de texto

Llamamos *archivo de texto plano* a un archivo que contiene un conjunto de caracteres sin formato, ocupando un byte por cada caracter. Cada caracter está representado por su código ASCII, y la longitud en bytes del archivo se corresponde con la cantidad de símbolos, tabulaciones y saltos de línea que contenga el texto. Normalmente este tipo de archivos se identifica con la extensión .txt, y puede ser editado con cualquier editor de textos.

9.1 Lectura de archivos

Las funciones del lenguaje C necesarias para leer datos desde un archivo de texto se encuentran declaradas en la biblioteca `<stdio.h>`. Físicamente, el archivo se encuentra en un medio de almacenamiento permanente, como el disco rígido o un diskette. Para trabajar con un archivo desde un programa en C, es necesario asociar al archivo una variable de un tipo de datos especial. Todos los requerimientos de lectura y escritura sobre el archivo se deben realizar sobre esta variable en memoria, y la biblioteca `<stdio.h>` se encarga de las solicitudes al sistema operativo para llevar a cabo las acciones solicitadas.

El primer paso para trabajar con un archivo consiste en *abrir* el archivo. Esta operación no tiene un correlato físico (el archivo en disco no se modifica), sino que tiene como propósito solicitar permiso al sistema operativo para acceder a los datos contenidos en el archivo. El sistema operativo coordina que no existan peticiones de lectura y escritura simultáneas sobre un mismo archivo, dado que este tipo de operaciones concurrentes podrían causar inconsistencias en los datos del archivo. La operación de apertura del archivo avisa al sistema operativo que nuestro programa tiene la intención de leer el archivo, y el sistema operativo rechaza esta solicitud si el archivo se encuentra actualmente abierto por otro progama para escritura.

Para abrir el archivo, es necesario declarar una variable de tipo `FILE *` (es decir, puntero al tipo de datos especial `FILE`, que representa un descriptor de archivo). La función `fopen` toma como parámetros el nombre del archivo y una cadena de caracteres con la descripción del tipo de operación que se pide realizar (que debe ser `"r"` si se necesita abrir el archivo para lectura), y retorna un puntero al descriptor del archivo en memoria. Esta función retorna el puntero NULL en caso de que la apertura haya fallado. Por ejemplo, el siguiente código declara una variable llamada `fp` de tipo `FILE *` y abre el archivo datos.txt para lectura:

```
FILE *fp = fopen("datos.txt", "r");
```

Se puede pasar como primer parámetro el nombre completo del archivo (unidad, ruta y nombre), en caso de que se encuentre en una carpeta distinta de la carpeta en la cual se está ejecutando el programa:

```
FILE *fp = fopen("a:\\datos.txt", "r");
FILE *gp = fopen("c:\\Mis documentos\\salida.txt", "r");
FILE *hp = fopen("c:\\Mis documentos\\Datos\\pepe.txt", "r");
```

Recordar que dentro de una cadena de caracteres, la barra invertida \ se escribe como la secuencia de escape \\.

Una vez que el archivo está abierto, la función `fscanf` se utiliza del mismo modo que la función `scanf` para tomar los datos del archivo, en este caso pasando como primer parámetro el puntero `FILE *` obtenido con la función `fopen`. Por ejemplo, si el archivo datos.txt abierto anteriormente tiene tres números enteros y un número real, se pueden obtener estos datos con las siguientes instrucciones:

```
int a,b,c;
float x;
fscanf(fp, "%d %d %d", &a, &b, &c);
fscanf(fp, "%f", &x);
```

El mecanismo para la lectura de datos desde un archivo de texto a través de la función `fscanf` es el mismo que se utiliza para la lectura de datos desde teclado con la función `scanf`. No hay diferencias conceptuales entre estas dos funciones, salvo el hecho de que para leer datos desde un archivo de texto se debe primero abrir el archivo, y no hay una operación equivalente para el teclado. Es importante observar que el programa debe conocer de antemano la cantidad y secuencia de los datos que se encuentran en el archivo.

Una vez que se obtuvieron los datos necesarios, se debe *cerrar* el archivo. El cierre del archivo le indica al sistema operativo que el programa ya no necesita leer el archivo, con lo cual el archivo queda disponible para otros programas que quieran acceder a él. La operación de cierre del archivo se especifica con:

```
fclose(fp);
```

Ejemplo 9.1.1 Supongamos que el archivo pepe.txt contiene una cierta cantidad de mediciones, y queremos escribir un archivo que obtenga estas mediciones y que muestre su promedio. La primera línea del archivo contiene un entero con la cantidad de mediciones, y las siguientes líneas contienen reales con las mediciones propiamente dichas. Por ejemplo, el contenido del archivo podría ser el siguiente:

```
5
1.22341
2.33435
5.2322
3.232
4.34343
```

El Código 9.1 muestra un programa que pregunta al usuario el nombre del archivo, lee estos datos del archivo y calcula el promedio de estas mediciones. Observar que se utiliza un arreglo de caracteres para guardar el nombre del archivo, y que se pasa un puntero a este arreglo en lugar de escribir una cadena de caracteres fija con el nombre del archivo. ♦

Código 9.1 Lectura de un archivo de texto.

```
#include <stdio.h>

int main()
{
    char archivo[20];
    int i,n;
    float x, prom;
    FILE *fp;

    printf("Ingrese el nombre del archivo: ");
    scanf("%s", archivo);

    fp = fopen(s, "r");
    fscanf(fp, "%d", &n);

    prom = 0;
    for(i=0; i<n; ++i)
    {
        fscanf(fp, "%f", &x);
        prom += x;
    }

    prom /= n;
    fclose(fp);

    printf("El promedio es: %f", prom);
    return 0;
}
```

9.2 Escritura de archivos

La creación y escritura de un archivo de texto se realiza del mismo modo que la lectura descripta en la sección anterior. A diferencia de la lectura, cuando se abre un archivo para escritura se debe especificar `"w"` como segundo parámetro de la función `fopen`. Por otra parte, la escritura de datos a un archivo de texto se realiza con la función `fprintf`, que tiene las mismas especificaciones que la función `printf` de escritura en consola (nuevamente, con la salvedad de que se debe pasar el puntero `FILE *` al descriptor del archivo como primer parámetro de la función). Por ejemplo, el Código 9.2 muestra un programa que escribe los números del 1 al 10 en un archivo de texto.

Código 9.2 Escritura de un archivo de texto.

```
#include <stdio.h>

int main()
{
    int i;
    FILE *fp = fopen("pepe.txt", "w");

    for(i=1; i<=10; ++i)
        fprintf(fp, "%f \n", i);

    fclose(fp);
    return 0;
}
```

9.3 Modos de apertura

El segundo parámetro que se pasa a la función `fopen` especifica el modo de apertura del archivo (cuyo nombre se pasa como segundo parámetro). En las secciones anteriores vimos que este parámetro debe ser `"r"` para lectura y `"w"` para escritura. En la Tabla 9.1 se encuentran todos los modos de apertura existentes, que se pueden utilizar para lograr un manejo más sofisticado.

La apertura de un archivo en modo de lectura `"r"` falla si el archivo no existe o no puede ser leído. La apertura para escritura al final `"a"` hace que todas las escrituras que se realizan al archivo se agreguen al final de los datos existentes.

Especificador	Descripción
`"r"`	abre el archivo para lectura
`"w"`	trunca a longitud cero o crea un archivo para escritura
`"a"`	abre para escritura al final del archivo
`"rb"`	abre el archivo para lectura en modo binario
`"wb"`	abre el archivo para escritura en modo binario
`"a"`	abre para escritura en modo binario al final del archivo
`"r+"`	abre el archivo para lectura y escritura
`"w+"`	crea o trunca el archivo para lectura y escritura
`"a+"`	abre para lectura y escritura al final del archivo
`"rb+"`	abre el archivo para lectura y escritura en modo binario
`"wb+"`	crea o trunca el archivo para lectura y escritura en modo binario
`"ab+"`	abre para lectura y escritura al final del archivo en modo binario

Tabla 9.1: Modos de apertura de archivos.

Capítulo 10

Algoritmos fundamentales

Cuando nos proponemos encontrar un algoritmo para resolver un problema, no estamos comenzando esta tarea "desde cero", sino que siempre podemos —y es muy útil— contar con la experiencia de algoritmos ya conocidos. De esta forma, el programador no está solo en la tarea de construir un algoritmo para resolver un problema, ya que se apoya en algoritmos conocidos y, sobre todo, en las ideas que previamente generaron estos algoritmos. Para ilustrar este punto, veremos en este capítulo algunos ejemplos de extensiones y variaciones sobre una misma idea de un algoritmo, que permiten resolver problemas muy distintos entre sí.

10.1 Consideraciones generales

La programación es un proceso inherentemente creativo, pero que debe respetar un cierto número de reglas estrictas. Estas reglas van desde elementos básicos del procesamiento de datos hasta temas más complejos como la terminación del algoritmo:

Completitud de los datos. Al construir un algoritmo se debe tener en cuenta que los datos de entrada estén completos —es decir, que no falte información para obtener la respuesta al problema en cuestión— y que los resultados que genere el algoritmo respondan efectivamente al problema. Por otra parte, es importante explicitar las condiciones que deben cumplir los datos de entrada (valores que deben ser positivos, relaciones entre los datos, etc.), e incluir en el programa las verificaciones necesarias para reaccionar ante los incumplimientos de estas reglas.

Manejo de los datos. La manipulación de los datos que se hace durante la ejecución de un algoritmo está sujeta a reglas estrictas. No se puede usar una variable si esta no fue inicializada antes, no se pueden ejecutar operaciones aritméticas libremente (por ejemplo, sólo se puede hacer una división si el divisor es distinto de cero), no se puede acceder a una posición inexistente en un arreglo, etc. Es de suma importancia tener en cuenta estas consideraciones durante el proceso de creación de un algoritmo para evitar problemas inesperados durante la ejecución del programa.

Corrección del algoritmo. Una característica fundamental de la programación es que se trata de una actividad *orientada a un objetivo*. No es un proceso creativo libre, sino que el algoritmo que se genere como producto debe resolver efectivamente el problema planteado. Un algoritmo es *correcto* cuando termina y, al terminar, proporciona una respuesta correcta al problema que pretende resolver. Un algoritmo que no termina —por ejemplo, un ciclo que "se cuelga"— no se considera correcto.

Detalles de implementación. Una vez completado el proceso de creación del algoritmo, el siguiente paso consiste en *implementar* el algoritmo por medio de la escritura de un programa en un lenguaje de programación adecuado. En este momento aparecen consideraciones de sintaxis y semántica del lenguaje, que debemos respetar. Un error de sintaxis o un comando mal interpretado no invalida la corrección del algoritmo, pero hará que la compilación o la ejecución no sean exitosas, con lo cual el resultado final no será aceptable.

En muchos aspectos, la construcción de un algoritmo es similar a la construcción de una demostración para un teorema matemático. El enunciado del teorema equivale al problema que queremos resolver: las hipótesis se corresponden con los datos de entrada del algoritmo, y la conclusión del teorema se corresponde con los resultados de salida del algoritmo. Las reglas de manipulación de datos de un algoritmo se corresponden con las reglas de derivación lógica válidas en el contexto de una demostración —modus ponens, modus tollens, tercero excluido, etc. Finalmente, el requerimiento de *corrección* de un algoritmo es el mismo que en una demostración: un algoritmo debe ser un cálculo correcto para obtener los resultados a partir de los datos de entrada, y una demostración tiene que ser una argumentación válida que permita establecer la verdad de la conclusión a partir de la verdad de las hipótesis.

Esta comparación entre un algoritmo y una demostración es muy precisa: la demostración *argumenta* cómo a partir de las hipótesis "se llega" a la conclusión, mientras que un algoritmo contiene una secuencia de pasos para *construir* los resultados deseados a partir de los datos de entrada. Más aún, muchas demostraciones son *constructivas*, puesto que efectivamente proporcionan un algoritmo para construir la conclusión a partir de las hipótesis. La afirmación recíproca no es cierta: una demostración no está obligada a proveer un método para *calcular* la conclusión, sino que en muchos casos puede limitarse a probar su validez por medios indirectos, pruebas de existencia, pruebas por reducción al absurdo, etc.

Por último, cabe notar que el proceso de implementación de un algoritmo en un lenguaje de programación no tiene un análogo en la construcción de una demostración. La demostración de un teorema termina cuando se escribe y se valida la demostración. Del mismo modo, cuando se escribe —en papel— y se valida un algoritmo, el proceso de creación del algoritmo está completo. Sin embargo, cuando se construye un algoritmo se lo hace para resolver algún problema específico, con lo cual el siguiente paso es la implementación efectiva del algoritmo en un lenguaje de programación para luego proceder con su ejecución y obtener así la respuesta al problema. Este paso involucra detalles de sintaxis y semántica del lenguaje de programación, agregando así un nivel de complejidad adicional —aunque debe notarse que este nivel adicional normalmente no es problemático una vez que se dominan las características del lenguaje de programación seleccionado.

10.1.1 Principios de la programación

Las consideraciones de la sección anterior muestran que la construcción de algoritmos y la escritura de programas que los implementan no es una actividad de creación libre. Por este motivo, es importante tener en cuenta los principios fundamentales sobre la programación que se enumeran en esta sección.

Principio: La programación es una actividad orientada a un objetivo.

Con este principio queremos decir que el resultado deseado —es decir, el objetivo— juega un papel más importante que las condiciones sobre datos de entrada. Los datos de entrada son importantes, pero todo el esfuerzo creativo debe estar puesto sobre el resultado del problema. La intención debe ser encontrar algoritmos inteligentes para obtener el resultado y, si estos

algoritmos no son correctos para todos los casos contemplados entre los datos de entrada, refinar los algoritmos propuestos o "salvar" los casos no previstos por medio de instrucciones especiales.

> **Principio:** Antes de intentar la resolución de un problema, hay que estar absolutamente seguro de cuál es el problema a resolver.

Al momento de pensar un algoritmo para resolver un problema, es fundamental tener en claro el problema que se debe resolver. Nos referiremos con el término *precondición* de un algoritmo a la especificación precisa de las condiciones que deben cumplir los datos de entrada, y con el término *postcondición* a una descripción de los resultados deseados —habitualmente en función de los datos de entrada. El par formado por la precondición y la postcondición de un algoritmo se denomina *especificación* del algoritmo. Es muy importante que la especificación sea precisa y esté bien entendida antes de comenzar la construcción del algoritmo para el problema, con lo cual el principio anterior se transforma en el siguiente principio:

> **Principio:** Antes de construir un algoritmo, escribir con precisión la precondición y la postcondición del mismo.

Suele suceder que un problema esté especificado de una manera que admita más de una interpretación. En estos casos, es fundamental destinar algo de tiempo a entender el problema y precisar la precondición y la postcondición del algoritmo. Esta especificación puede estar escrita utilizando notación matemática o en lenguaje informal, siempre que la descripción sea precisa y no contenga ambigüedades. La especificación debe indicar *qué* debe hacer el algoritmo, sin decir nada sobre *cómo* debe hacerlo. Cómo se debe realizar la tarea pedida por la especificación es el objetivo de la construcción del algoritmo, y aquí estamos en una etapa previa.

> **Principio:** Para construir un algoritmo se deben conocer las propiedades de los objetos que se van a manipular.

El proceso de construcción de un algoritmo habitualmente requiere de conocimiento de los objetos matemáticos con los que se va a trabajar. Por ejemplo, para determinar si un número entero es primo, es crucial conocer las propiedades de divisibilidad de los números enteros. Para escribir un algoritmo que aproxime la raíz cuadrada de un número real, se deben conocer elementos de análisis matemático. En general, cuanto más conocimiento tenga el programador sobre el dominio del problema a resolver, mayores serán sus posibilidades de construir un algoritmo correcto y eficiente.

▷ El contenido de esta subsección fue adaptado de [6], Capítulos 13 y 14.

10.2 Búsqueda lineal

El esquema de búsqueda lineal se aplica en las situaciones en las cuales se debe buscar en un conjunto finito un elemento que cumple cierta propiedad. Es esencial que esta propiedad pueda ser evaluada por medio de una expresión lógica, o por medio de un fragmento de código encapsulado en una función.

10.2.1 Determinando si un número es primo

Dado un entero $n \geq 2$, queremos escribir una función que determine si el número n es primo o no. La precondición del algoritmo es simplemente $n \geq 2$, y la postcondición del algoritmo pide que el valor de retorno de la función sea verdadero si y sólo si n es primo.

La forma más sencilla de decidir este hecho es *buscar* algún divisor de n en el conjunto $C = \{2, \ldots, n-1\}$, denominado el *conjunto de candidatos* para la búsqueda. El número n no es primo si y sólo si algún elemento del conjunto de candidatos es divisor de n. El algoritmo de búsqueda lineal consiste en recorrer uno por uno los elementos del conjunto de candidatos, verificando para cada uno la propiedad que nos interesa buscar —en este caso, la propiedad consiste en ser divisor de n. Realizamos esto con la función que se muestra en el Código 10.1.

Código 10.1 Determinando si un número es primo.

```
bool primo(int n)
{
    int i=2;
    while ( i<n && n%i != 0 )
        ++i;

    if ( i<n )
        return false;
    else
        return true;
}
```

El ciclo `while` recorre todos los elementos del conjunto de candidatos, y la segunda condición de la guarda verifica si el elemento actual cumple la propiedad que nos interesa. En caso de que no la cumpla, se continúa con la búsqueda. Es esencial notar que la segunda condición de la guarda del ciclo es la propiedad opuesta a la que estamos buscando: si el elemento actual no cumple la propiedad buscada, se continúa iterando (el cuerpo del ciclo incrementa la variable i en uno para pasar al siguiente candidato).

La iteración se detiene cuando se encuentra un elemento que cumple la propiedad buscada (es decir, cuando la segunda condición de la guarda no se cumple), o bien cuando se llega al final del conjunto de candidatos sin haber encontrado un elemento con la propiedad buscada (en este caso, se deja de cumplir la primera condición de la guarda del ciclo). La alternativa que se encuentra a la salida del ciclo determina el valor de retorno en función de esta situación: si $i < n$ entonces el ciclo se interrumpió al encontrar un divisor de n, con lo cual este número no es primo. Por el contrario, si $i = n$ entonces el ciclo recorrió todo el conjunto de candidatos sin haber encontrado ningún divisor, con lo cual n es primo.

Es importante observar que el ciclo se detiene cuando encuentra el primer divisor de n. Si el problema hubiera sido encontrar un divisor de n (en lugar de determinar si n es primo), entonces este ciclo siempre encontraría el primer divisor. En cambio, si nos interesa buscar el mayor divisor de n distinto de n, entonces debemos recorrer el conjunto de candidatos hacia atrás, y cuando el ciclo se detiene porque encontró un divisor, este divisor será el mayor divisor propio de n.

> **Principio de la búsqueda lineal:** Para encontrar el menor elemento que cumple una propiedad, la búsqueda lineal debe recorrer el conjunto de candidatos de menor a mayor. Para encontrar el mayor elemento que cumple una propiedad, la búsqueda lineal debe recorrer el conjunto de candidatos de mayor a menor.

10.2.2 Cálculo de la parte entera de la raíz cuadrada de un número

Dado un número entero $n \geq 0$, la parte entera de su raíz cuadrada es $\lfloor \sqrt{n} \rfloor$. Para encontrar este valor, el esquema más sencillo consiste en plantear una búsqueda lineal sobre el conjunto de candidatos $C = \{0, 1, \ldots, n\}$. Es fácil ver que $\lfloor \sqrt{n} \rfloor$ tiene que estar dentro de este conjunto de candidatos.

Ahora bien, ¿qué propiedad buscamos dentro de este conjunto de candidatos? Tenemos que $m = \lfloor \sqrt{n} \rfloor$ si y sólo si $m^2 \leq n < (m+1)^2$. Es decir, m debe ser el menor número que cumple $n < (m+1)^2$, y ésta es la propiedad que buscamos en el algoritmo. El Código 10.2 muestra una función que implementa este algoritmo.

Código 10.2 Calculando la parte entera de la raíz cuadrada de un entero.

```
int raiz_entera(int n)
{
    int i=0;
    while ( i<=n && n >= (i+1)*(i+1) )
        ++i;

    return i;
}
```

10.2.3 Búsqueda de un elemento en un arreglo

Sea b un arreglo de n enteros no necesariamente ordenado, y supongamos que estamos interesados en determinar si un cierto entero x se encuentra en el arreglo.

Nuevamente, es fácil ver que el esquema de búsqueda lineal se aplica tambien en este caso. El conjunto de candidatos son las celdas del arreglo (podemos representarlas por medio de sus índices), de modo tal que $C = \{0, \ldots, n-1\}$, y la propiedad a buscar es que el elemento del arreglo sea igual al valor x buscado (ver Código 10.3).

Código 10.3 Búsqueda de un elemento en un arreglo no ordenado.

```
bool pertenece(int *b, int n, int x)
{
    int i=0;
    while ( i<n && b[i] != x )
        ++i;

    if ( i<n )
        return true;
    else
        return false;
}
```

10.3 Búsqueda binaria

En el algoritmo de búsqueda binaria tenemos como datos de entrada un arreglo *ordenado* b de enteros (con n elementos), y queremos decidir si un cierto entero x se encuentra en el arreglo. La precondición del algoritmo básico de búsqueda binaria es

$$P \equiv b[0] \leq x \wedge x < b[n-1].$$

Esta precondición no es restrictiva: si $b[0] > x$ entonces x no está en el arreglo —porque el arreglo está ordenado— y si $x \geq b[n-1]$ entonces x solamente puede estar en la última posición del arreglo. Se pueden salvar estos dos casos especiales con una instrucción alternativa, dejando el tratamiento del caso general al algoritmo de búsqueda binaria. La postcondición del algoritmo pide que el valor de retorno de la función sea verdadero si y sólo si el entero x se encuentra en el arreglo.

El método de búsqueda binaria consiste en dividir el arreglo en dos mitades. Se comprueba en cuál de las dos mitades se puede encontrar x, y se procede a buscar sobre la mitad seleccionada, usando el mismo esquema de búsqueda: este subarreglo se divide a su vez en dos mitades, y se repite el procedimiento hasta llegar a un subarreglo formado por un sólo elemento, que es la celda del arreglo candidata a contener el valor x.

A lo largo de su ejecución, el algoritmo mantiene un intervalo dentro del cual puede estar el número x. Este subarreglo se representa por medio de dos variables enteras i y j. Inicialmente $i = 0$ y $j = n - 1$, especificando que el "subarreglo" candidato es todo el arreglo. En cada paso se obtiene la mitad $k = (i + j)/2$ del subarreglo candidato y se verifica en cuál de las dos

Código 10.4 Búsqueda binaria en un arreglo ordenado.

```
bool busqueda_binaria(int *b, int n, int x)
{
    assert( b[0] <= x && x < b[n-1] );

    int i = 0;
    int j = n-1;
    int k;

    while ( i+1<j )
    {
        k = (i+j)/2;

        if ( b[k] > x )
            j = k;
        else
            i = k;
    }

    if ( b[i] == x )
        return true;
    else
        return false;
}
```

mitades se puede encontrar x: si $b[k] > x$ entonces se continúa la búsqueda en la primera mitad, y, si $b[k] \leq x$ entonces se continúa la búsqueda en la segunda mitad.

Es muy instructivo analizar el invariante del ciclo en el algoritmo de búsqueda binaria. El invariante de este ciclo indica que el subarreglo comprendido entre las variables i y j es candidato a contener el valor x:

$$I \equiv b[i] \leq x \wedge x < b[j].$$

Es decir, el valor x está "encerrado" entre las posiciones $b[i]$ y $b[j]$. Notar que la inicialización $i = 0$ y $j = n - 1$ establece automáticamente la validez del invariante, dado que en ese caso el invariante es igual a la precondición (lo cual explica la elección de la precondición). Por otra parte, es fácil ver que la alternativa del cuerpo del ciclo mantiene la validez del invariante. Solamente se cambia el valor de la variable i cuando tenemos garantizado que el nuevo valor de i cumplirá $b[i] \leq x$, y solamente se cambia el valor de la variable j cuando tenemos garantizado que el nuevo valor de j cumplirá $b[j] < x$.

El ciclo termina con $i + 1 = j$. Cuando esto sucede, tenemos que $b[i] \leq x$ y $x < b[i + 1]$, con lo cual hemos acotado el subarreglo que puede contener el valor x a una sola celda. Por este motivo, a la salida del ciclo es suficiente con verificar si x se encuentra en la posición $b[i]$. Si esto sucede, entonces acabamos de encontrar a x en el arreglo. Por el contrario, si $b[i] \neq x$ implica que $b[i] < x < b[i + 1]$ y, por lo tanto, x no se encuentra en el arreglo.

El algoritmo de búsqueda binaria sobre arreglos ordenados es un caso particular de la búsqueda binaria sobre una estructura de datos ordenada. Más generalmente, este método de búsqueda forma parte del grupo de algoritmos de *dividir y conquistar*. En este tipo de algoritmos se divide el problema en dos partes, y se resuelven los dos subproblemas en forma recursiva, hasta obtener una respuesta al problema original. Recomendamos la lectura de [1], Sección 10.1, para conocer más sobre este tema.

10.4 Ordenamiento de arreglos

El objetivo de un método de ordenamiento de arreglos es permutar los elementos de un arreglo para que luego de aplicar el algoritmo el arreglo quede ordenado. Por ejemplo, supongamos que el arreglo inicial es:

4	1	8	3	11	12	2	4	7	9

Luego del proceso de ordenamiento, el arreglo ordenado tiene que ser:

1	2	3	4	4	7	8	9	11	12

Notar que el arreglo original tenía dos apariciones del número 4 que, en el arreglo ordenado, se encuentran en posiciones consecutivas. En un arreglo ordenado, todas las apariciones de un mismo número se encuentran en celdas consecutivas. La precondición de un algoritmo de ordenamiento no pide ninguna condición sobre el arreglo de entrada, y la postcondición solicita que el arreglo contenga una permutación ordenada de los valores iniciales del mismo arreglo.

En los algoritmos de ordenamiento que veremos en esta sección, adoptamos la convención de modificar los arreglos únicamente por medio de intercambios de elementos. De esta forma se garantiza trivialmente que el arreglo modificado por el algoritmo es una permutación del arreglo original. Dado que vamos a usar frecuentemente esta operación de intercambio, definimos en el Código 10.5 un procedimiento que encapsula esta operación.

Código 10.5 Intercambio de dos elementos de un arreglo.

```
void swap(int *b, int i, int j)
{
    int aux = b[i];
    b[i] = b[j];
    b[j] = aux;
}
```

10.4.1 Ordenamiento por burbujeo

La idea básica del método de *ordenamiento por burbujeo* proviene de imaginar el arreglo en forma vertical, suponiendo que los valores más grandes son más pesados y tienden a ir hacia abajo. Se recorre varias veces el arreglo y, al hacer esto, si hay dos elementos adyacentes que no están en orden (es decir, si el más pesado está arriba del más liviano), se intercambian estos elementos.

El efecto producido por esta recorrida es que el elemento más pesado de todos (o sea, el mayor elemento del arreglo) desciende hasta la parte inferior del arreglo (es decir, hacia la posición $n-1$ del arreglo). En el segundo recorrido, el segundo mayor valor del arreglo se ubica en la posición $n-2$. Observar que en este caso no es necesario recorrer el arreglo hasta la última posición, dado que allí ya se encuentra el máximo valor del arreglo. En general, el i-ésimo recorrido (con $0 \leq i \leq n-1$) debe recorrer hasta la posición $n-i-1$ del arreglo, y ubicará el i-ésimo mayor elemento en esa posición. Por este motivo, a lo sumo luego de $n-1$ recorridos el arreglo estará ordenado. El Código 10.6 muestra una implementación básica de este método.

Código 10.6 Algoritmo de ordenamiento por burbujeo.

```
void burbujeo(int *b, int n)
{
    for (int i=0; i<n-1; ++i)
        for (int j=0; j<n-i-1; ++j)
            if ( b[j] > b[j+1] )
                swap(b, j, j+1);
}
```

10.4.2 Ordenamiento por inserción

En el algoritmo de *ordenamiento por inserción* se realizan $n-1$ recorridos del arreglo. En el i-ésimo recorrido (con $0 \leq i \leq n-1$), se "inserta" el elemento `b[i]` en la posición correcta en el subarreglo `b[0..i]`, de modo tal que este subarreglo contenga los mismos elementos que se encontraban en estas posiciones en el arreglo original, pero ordenados. De esta forma, luego de $n-1$ recorridos el arreglo completo está ordenado. El Código 10.7 muestra una implementación posible de este algoritmo.

10.4.3 Ordenamiento por selección

En el algoritmo de *ordenamiento por selección* se realizan $n-1$ recorridos del arreglo. En el i-ésimo recorrido (con $0 \leq i \leq n-1$), se busca el menor elemento del subarreglo `b[i..n]` y se

Código 10.7 Algoritmo de ordenamiento por inserción.

```
void insercion(int *b, int n)
{
    for (int i=1; i<n; ++i)
    {
        int j = i;
        while ( j>0 && b[j] < b[j-1] )
        {
            swap(b, j, j-1);
            --j;
        }
    }
}
```

intercambia con la posición `b[i]`. De esta forma, luego del i-ésimo recorrido, el i-ésimo elemento (en orden) del arreglo se encuentra en la posición `b[i]`, y entonces el subarreglo `b[0..i]` se encuentra ordenado con los $i + 1$ menores valores del arreglo. Luego de $n - 1$ recorridos, el arreglo completo está ordenado. El nombre del algoritmo está dado por el hecho de que se "selecciona" el menor elemento del subarreglo que resta ordenar, y se ubica este elemento en la posición correcta en el arreglo definitivo.

▷ El contenido de esta sección fue adaptado de [1], Sección 8.2.

Bibliografía

[1] Aho, A., Hopcroft, J. y Ullman, J. (1988) *Estructuras de Datos y Algoritmos*, Wilmington: Addison-Wesley Iberoamericana.

[2] Bellavoine, C. (1984) *¿Qué es una Computadora?*, Buenos Aires: El Ateneo.

[3] Bloodshed Software (2005) *Documentación interna del compilador Dev-C++*. URL: http://www.bloodshed.net/devcpp.html

[4] Brey, B. (1993) *8086/8088, 80286, 80386, and 80486 Assembly Language Programming*, New York: Merril-Macmillan Publishing Company.

[5] Brookshear, J. G. (1995) *Introducción a las Ciencias de la Computación*, México: Addison-Wesley Iberoamericana.

[6] Gries, D. (1981) *The Science of Programming*, New York: Springer-Verlag.

[7] Kernighan, B. W. and Ritchie, D. M. (1988) *The C Programming Language*, New York: Prentice Hall.

[8] Schildt, H. (1996) *C Manual de Referencia*, México: Osborne/McGraw-Hill.

[9] Stallman R. M. et al. (2004) *Using the GNU Compiler Collection*, Boston: GNU Press. URL: http://www.gnu.org

[10] Torres, J. A. y Czitrom, V. (1980) *Métodos para la Solución de Problemas con Computadora Digital*, México: Representaciones y Servicios de Ingeniería.

[11] Wikibooks (2006) *Wikibooks, the open-content textbooks collection: C Programming*. URL: http://en.wikibooks.org/wiki/C_Programming

[12] Wikipedia (2006) *Wikipedia, the free encyclopedia*. URL: http://www.wikipedia.org