Módulo 1 / **Programación estructurada**

1

Introducción a los algoritmos y a la programación de computadoras

Contenido

Objetivos del capítulo

- Entender los conceptos básicos sobre algoritmos y programación.
- Identificar los principales recursos lógicos y físicos que utilizan los programas.
- Estudiar el teorema de la programación estructurada y las estructuras de control que este teorema describe.
- Conocer las herramientas imprescindibles de programación: lenguaje de programación, compilador, entorno de desarrollo (IDE), etcétera.
- Desarrollar, compilar y ejecutar nuestro primer programa de computación.

Competencias específicas

- Dominar los conceptos básicos de la programación.
- Analizar problemas y representar su solución mediante algoritmos.
- Conocer las características principales del lenguaje C.
- Codificar algoritmos en el lenguaje de programación C.
- Compilar y ejecutar programas.

1.1 Introducción

Este capítulo introduce al lector en los conceptos iniciales sobre algoritmos y programación. Aquellos lectores que nunca han programado encontrarán aquí los conocimientos básicos para hacerlo.

Los contenidos son netamente introductorios ya que pretenden brindarle al lector las pautas, las expresiones y la jerga que necesitará conocer para poder leer y comprender los capítulos sucesivos.

Por lo anterior, la profundidad con la que se tratan los temas aquí expuestos es de un nivel inicial, ya que cada uno de estos temas será analizado en detalle llegado el momento, en los capítulos que así lo requieran.

1.2 Concepto de algoritmo

1.2.1 Definición de algoritmo y problema

Llamamos "algoritmo" al conjunto finito y ordenado de acciones con las que podemos resolver un determinado problema.

Llamamos "problema" a una situación que se nos presenta y que, mediante la aplicación de un algoritmo, pretendemos resolver.

Los algoritmos están presentes en nuestra vida cotidiana y, aún sin saberlo, aplicamos algoritmos cada vez que se nos presenta un problema sin importar cuál sea su grado de complejidad.

Para ejemplificar esto imaginemos que estamos dando un paseo por la ciudad y que, al llegar a una esquina, tenemos que cruzar la calle. Intuitivamente, aplicaremos el siguiente algoritmo:

- Esperar a que la luz del semáforo peatonal esté en verde (`esperarSemaforo`);
- Cruzar la calle (`cruzarCalle`);

Este algoritmo está compuesto por las acciones: `esperarSemaforo` y `cruzarCalle`, en ese orden, y es extremadamente simple gracias a que cada una de estas engloba a otro conjunto de acciones más puntuales y específicas. Por ejemplo:

La acción `esperarSemaforo` implica:

- Observar la luz del semáforo (`observarLuz`);
- **Si** la luz está en "rojo" o en "verde intermitente" **entonces**
  - esperar un momento (`esperar`);
  - **ir a**: `observarLuz`;

Por otro lado, la acción `cruzarCalle` implica:

- Bajar la calzada (`bajarCalzada`);
- Avanzar un paso (`avanzarPaso`);
- **Si** la distancia hacia la otra calzada es mayor que la distancia que podemos avanzar dando un nuevo paso **entonces**
  - **ir a**: `avanzarPaso`;
- Subir la calzada de la vereda de enfrente (`subirCalzada`);

Como vemos, cada acción puede descomponerse en acciones más puntuales y específicas. Por lo tanto, cada una de las acciones del algoritmo se resuelve con su propio algoritmo o secuencia finita y ordenada de acciones.

Cuando una acción se descompone en acciones más puntuales y específicas decimos que es un "módulo".

Podemos seguir profundizando cada acción tanto como queramos. Por ejemplo, analicemos la acción (o el módulo) `bajarCalzada`: esta acción se resuelve:

- levantando un pie,
- adelantándolo,
- inclinando el cuerpo hacia adelante,
- y apoyando el pie en la calle,
- luego levantando el otro pie,
- adelantándolo y
- apoyándolo en la calle.

Claro que "levantar un pie" implica tensionar un conjunto de músculos, etcétera.

Módulo
Cuando una acción se descompone en acciones más puntuales y específicas la llamamos "módulo".

1.2.2 Análisis del enunciado de un problema

Resulta evidente que, si vamos a diseñar un algoritmo para resolver un determinado problema, tenemos que tener totalmente estudiado y analizado el contexto de dicho problema. Esto implica:

- Comprender el alcance.
- Identificar los datos de entrada.
- Identificar los datos de salida o resultados.

El análisis anterior es fundamental para poder diseñar una estrategia de solución que será la piedra fundamental para el desarrollo del algoritmo.

1.2.2.1 Análisis del problema

Repasemos el contexto del problema de "cruzar la calle" que formulamos más arriba:

Caminando por la ciudad llegamos a una esquina y pretendemos cruzar la calle. Para poder cruzar con la mayor seguridad posible, tenemos que esperar que el semáforo habilite el paso peatonal. Hasta que esto ocurra nos encontraremos detenidos al lado del semáforo y una vez que este lo permita avanzaremos, paso a paso, hasta llegar a la vereda de enfrente. Se considera que el semáforo habilita el paso peatonal cuando emite una luz "verde fija". En cambio si el semáforo emite una luz "roja" o "verde intermitente" se recomienda esperar.

1.2.2.2 Datos de entrada

En este contexto podemos identificar los siguientes datos de entrada:

- Nuestra posición inicial - posición en donde nos detuvimos a esperar a que el semáforo habilite el paso peatonal. La llamaremos `posInicial`.
- Luz del semáforo - el color de la luz que el semáforo está emitiendo. Lo llamaremos `luz`.
- Distancia de avance de paso - que distancia avanzamos cada vez que damos un nuevo paso. La llamaremos `distPaso`.
- Posición de la vereda de enfrente. La llamaremos `posFinal` ya que es allí hacia donde nos dirigimos.

1.2.2.3 Datos de salida

Si bien en este problema no se identifican datos de salida resulta evidente que luego de ejecutar la secuencia de acciones del algoritmo nuestra posición actual deberá ser la misma que la posición de la vereda de enfrente. Es decir, si llamamos `p` a nuestra posición actual resultará que su valor al inicio del algoritmo será `posInicial` y al final del mismo deberá ser `posFinal`.

1.2.3 Memoria y operaciones aritméticas y lógicas

En la vida real, cuando observamos la luz del semáforo almacenamos este dato en algún lugar de nuestra memoria para poder evaluar si corresponde cruzar la calle o no.

Generalmente, los datos de entrada ingresan al algoritmo a través de una acción que llamamos "lectura" o "ingreso de datos" y permanecen en la memoria el tiempo durante el cual el algoritmo se está ejecutando.

Por otro lado, el hecho de evaluar si corresponde o no cruzar la calle implica realizar una operación lógica. Esto es: determinar si la proposición "el semáforo emite luz roja o verde intermitente" resulta ser verdadera o falsa.

También realizaremos una operación lógica cuando estemos cruzando la calle y tengamos que determinar si corresponde dar un nuevo paso o no. Recordemos que llamamos `p` a nuestra posición actual, `posFinal` a la posición de la vereda de enfrente y `distPaso` a la distancia que avanzamos dando un nuevo paso. Por lo tanto, corresponde dar otro paso si se verifica que `posFinal > p+distPaso`. Aquí además de la operación lógica estamos realizando una operación aritmética: la suma `p+distPaso`.

En resumen, estos son los recursos que tenemos disponibles para diseñar y desarrollar algoritmos:

- La memoria.
- La posibilidad de realizar operaciones aritméticas.
- La posibilidad de realizar operaciones lógicas.

A Corrado Böhm y Giuseppe Jacopini se les atribuye el teorema de la programación estructurada, debido a un artículo de 1966.
David Harel rastreó los orígenes de la programación estructurada hasta la descripción de 1946 de la arquitectura de von Neumann y el teorema de la forma normal de Kleene.
La carta escrita en 1968 por Dijkstra, titulada "*Go To Considered Harmful*" reavivó el debate.
En la Web de apoyo, encontrará el vínculo a la página Web personal de Corrado Böhm.

1.2.4 Teorema de la programación estructurada

Este teorema establece que todo algoritmo puede resolverse mediante el uso de tres estructuras básicas llamadas "estructuras de control":

- La "estructura secuencial" o "acción simple".
- La "estructura de decisión" o "acción condicional".
- La "estructura iterativa" o "acción de repetición".

Dos de estas estructuras las hemos utilizado en nuestro ejemplo de cruzar la calle. La estructura secuencial es la más sencilla y simplemente implica ejecutar una acción, luego otra y así sucesivamente.

La estructura de decisión la utilizamos para evaluar si correspondía dar un nuevo paso en función de nuestra ubicación actual y la ubicación de la vereda de enfrente.

Sin embargo, para resolver el problema hemos utilizado una estructura tabú: la estructura "**ir a**" o, en inglés, "*go to*" o "*goto*". Esta estructura quedó descartada luego de que el teorema de la programación estructurada demostrara que con una adecuada combinación de las tres estructuras de control antes mencionadas es posible resolver cualquier algoritmo sin tener que recurrir al "*goto*" (o estructura "ir a").

Para ejemplificarlo vamos a replantear el desarrollo de los algoritmos anteriores y reemplazaremos la estructura "**ir a**" (*goto*) por una estructura iterativa: la estructura "**repetir mientras que**".

Veamos el desarrollo del módulo `esperarSemaforo`:

Con "**ir a**"	Sin "~~**ir a**~~" (solución correcta)
• `observarLuz`; • **Si** la luz está en "rojo" o en "verde intermitente" **entonces** - `esperar`; - **ir a**: `observarLuz`;	• `observarLuz`; • **Repetir mientras que** la luz esté en "rojo" o en "verde intermitente" **hacer** - `esperar`; - `observarLuz`;

Veamos ahora el desarrollo del módulo `cruzarCalle`:

Con "**ir a**"	Sin "~~**ir a**~~" (solución correcta)
• `bajarCalzada`; • `avanzarPaso`; • **Si** la distancia hacia la otra calzada es mayor que la distancia que podemos avanzar dando un nuevo paso **entonces** - **ir a**: `avanzarPaso`; • `subirCalzada`;	• `bajarCalzada`; • `avanzarPaso`; • **Repetir mientras que** la distancia hacia la otra calzada sea mayor que la distancia que podemos avanzar dando un nuevo paso **hacer** - `avanzarPaso`; • `subirCalzada`;

Como vemos, la combinación "acción condicional + *goto*" se puede reemplazar por una estructura de repetición. Si bien ambos desarrollos son equivalentes la nueva versión es mejor porque evita el uso del *goto*, estructura que quedó en desuso porque trae grandes problemas de mantenibilidad.

1.3 Conceptos de programación

En general, estudiamos algoritmos para aplicarlos a la resolución de problemas mediante el uso de la computadora. Las computadoras tienen memoria y tienen la capacidad de resolver operaciones aritméticas y lógicas. Por lo tanto, son la herramienta fundamental para ejecutar los algoritmos que vamos a desarrollar.

Para que una computadora pueda ejecutar las acciones que componen un algoritmo tendremos que especificarlas o describirlas de forma tal que las pueda comprender.

Todos escuchamos alguna vez que "las computadoras solo entienden 1 (unos) y 0 (ceros)" y, efectivamente, esto es así, pero para nosotros (simples humanos) especificar las acciones de nuestros algoritmos como diferentes combinaciones de unos y ceros sería una tarea verdaderamente difícil. Afortunadamente, existen los lenguajes de programación que proveen una solución a este problema.

1.3.1 Lenguajes de programación

Las computadoras entienden el lenguaje binario (unos y ceros) y nosotros, los humanos, entendemos lenguajes naturales (español, inglés, portugués, etc.).

Los lenguajes de programación son lenguajes formales que se componen de un conjunto de palabras, generalmente en inglés, y reglas sintácticas y semánticas.

Podemos utilizar un lenguaje de programación para escribir o codificar nuestro algoritmo y luego, con un programa especial llamado "compilador", podremos generar los "unos y ceros" que representan sus acciones. De esta manera, la computadora será capaz de comprender y convertir al algoritmo en un programa de computación.

Los lenguajes naturales son los que hablamos y escribimos los seres humanos: inglés, español, italiano, etc. Son dinámicos ya que, constantemente, incorporan nuevas variaciones, palabras y significados. Por el contrario, los lenguajes formales son rígidos y se construyen a partir de un conjunto de símbolos (alfabeto) unidos por un conjunto de reglas (gramática). Los lenguajes de programación son lenguajes formales.

El lenguaje C++ fue creado a mediados de los años ochenta por Bjarne Stroustrup, con el objetivo de extender al lenguaje de programación C con mecanismos que permitan la manipulación de objetos.

Java es un lenguaje de programación orientado a objetos, creado en la década del noventa por Sun Microsystems (actualmente adquirida por Oracle). Utiliza gran parte de la sintaxis de C y C++, pero su modelo de objetos es más simple.

El lenguaje de programación C fue creado por Dennis Ritchie (1941-2011). En 1967 Ritchie comenzó a trabajar para los laboratorios Bell, donde se ocupó, entre otros, del desarrollo del lenguaje B. Creó el lenguaje de programación C en 1972, junto con Ken Thompson.
Ritchie también participó en el desarrollo del sistema operativo Unix.

Existen muchos lenguajes de programación: Pascal, C, Java, COBOL, Basic, Smalltalk, etc., y también existen muchos lenguajes derivados de los anteriores: Delphi (derivado de Pascal), C++ (derivado de C), C# (derivado de C++), Visual Basic (derivado de Basic), etcétera.

En este libro utilizaremos el lenguaje de programación C y, para los capítulos de encapsulamiento y programación orientada a objetos, C++ y Java.

1.3.2 Codificación de un algoritmo

Cuando escribimos las acciones de un algoritmo en algún lenguaje de programación decimos que lo estamos "codificando". Generalmente, cada acción se codifica en una línea de código.

Al conjunto de líneas de código, que obtenemos luego de codificar el algoritmo, lo llamamos "código fuente".

El código fuente debe estar contenido en un archivo de texto cuyo nombre debe tener una extensión determinada que dependerá del lenguaje de programación que hayamos utilizado. Por ejemplo, si codificamos en Pascal entonces el nombre del archivo debe tener la extensión ".pas". Si codificamos en Java entonces la extensión del nombre del archivo deberá ser ".java" y si el algoritmo fue codificado en C entonces el nombre del archivo deberá tener la extensión ".c".

1.3.3 Bibliotecas de funciones

Los lenguajes de programación proveen bibliotecas o conjuntos de funciones a través de las cuales se ofrece cierta funcionalidad básica que permite, por ejemplo, leer y escribir datos sobre cualquier dispositivo de entrada/salida como podría ser la consola.

Es decir, gracias a que los lenguajes proveen estos conjuntos de funciones los programadores estamos exentos de programarlas y simplemente nos limitaremos a utilizarlas.

Programando en C, por ejemplo, cuando necesitemos mostrar un mensaje en la pantalla utilizaremos la función `printf` y cuando queramos leer datos a través del teclado utilizaremos la función `scanf`. Ambas funciones forman parte de la biblioteca estándar de entrada/salida de C, también conocida como "stdio.h".

1.3.4 Programas de computación

Un programa es un algoritmo que ha sido codificado y compilado y que, por lo tanto, puede ser ejecutado en una computadora.

El algoritmo constituye la lógica del programa. El programa se limita a ejecutar cada una de las acciones del algoritmo. Por esto, si el algoritmo tiene algún error entonces el programa también lo tendrá y si el algoritmo es lógicamente perfecto entonces el programa también lo será.

El proceso para crear un nuevo programa es el siguiente:

- Diseñar y desarrollar su algoritmo.
- Codificar el algoritmo utilizando un lenguaje de programación.
- Compilarlo para obtener el código de máquina o archivo ejecutable (los "unos y ceros").

Dado que el algoritmo es la parte lógica del programa muchas veces utilizaremos ambos términos como sinónimos ya que para hacer un programa primero necesitaremos diseñar su algoritmo. Por otro lado, si desarrollamos un algoritmo seguramente será para, luego, codificarlo y compilarlo, es decir, programarlo.

1.3.5 Consola

Llamamos “consola” al conjunto compuesto por el teclado y la pantalla de la computadora en modo texto. Cuando hablemos de ingreso de datos por consola nos estaremos refiriendo al teclado y cuando hablemos de mostrar datos por consola estaremos hablando de la pantalla, siempre en modo texto.

1.3.6 Entrada y salida de datos

Llamamos “entrada” al conjunto de datos externos que ingresan al algoritmo. Por ejemplo, el ingreso de datos por teclado, la información que se lee a través de algún dispositivo como podría ser un lector de código de barras, un *scanner* de huellas digitales, etcétera.

Llamamos “salida” a la información que el algoritmo emite sobre algún dispositivo como ser la consola, una impresora, un archivo, etcétera.

La consola es el dispositivo de entrada y salida por omisión. Es decir, si hablamos de ingreso de datos y no especificamos nada más será porque el ingreso de datos lo haremos a través del teclado. Análogamente, si hablamos de mostrar cierta información y no especificamos más detalles nos estaremos refiriendo a mostrarla por la pantalla de la computadora, en modo texto.

1.3.7 Lenguajes algorítmicos

Llamamos “lenguaje algorítmico” a todo recurso que permita describir con mayor o menor nivel de detalle los pasos que componen un algoritmo.

Los lenguajes de programación, por ejemplo, son lenguajes algorítmicos ya que, como veremos más adelante, la codificación del algoritmo es en si misma una descripción detallada de los pasos que lo componen. Sin embargo, para describir un algoritmo no siempre será necesario llegar al nivel de detalle que implica codificarlo. Una opción válida es utilizar un “pseudocódigo”.

1.3.8 Pseudocódigo

El pseudocódigo surge de mezclar un lenguaje natural (por ejemplo, el español) con ciertas convenciones sintácticas y semánticas propias de un lenguaje de programación. Justamente, el algoritmo que resuelve el problema de “cruzar la calle” fue desarrollado utilizando un pseudocódigo.

Los diagramas también permiten detallar los pasos que componen un algoritmo; por lo tanto, son lenguajes algorítmicos. En este libro profundizaremos en el uso de diagramas para luego codificarlos con el lenguaje de programación C.

Los diagramas Nassi-Shneiderman, también conocidos como "diagramas de Chapin", fueron publicados en 1973 por Ben Shneiderman e Isaac Nassi en el artículo llamado "A short history of strutctured flowcharts", con el objetivo de eliminar las líneas de los diagramas tradicionales y así reforzar la idea de estructuras de “única entrada y única salida”.
Si en la programación estructurada se elimina la sentencia GOTO entonces se deben eliminar las líneas en los diagramas que la representan.

1.4 Representación gráfica de algoritmos

Los algoritmos pueden representarse mediante el uso de diagramas. Los diagramas proveen una visión simplificada de la lógica del algoritmo y son una herramienta importantísima que utilizaremos para analizar y documentar los algoritmos o programas que vamos a desarrollar.

En este libro, utilizaremos una versión modificada de los diagramas de *Nassi-Shneiderman*, también conocidos como diagramas *Chapin*. Estos diagramas se componen de un conjunto de símbolos que permiten representar cada una de las estructuras de control que describe el teorema de la programación estructurada: la estructura secuencial, la estructura de decisión y la estructura de repetición.

1.4.1 Representación gráfica de la estructura secuencial o acción simple

La estructura secuencial se representa en un recuadro o, si se trata de un ingreso o egreso de datos se utiliza un trapecio como observamos a continuación:

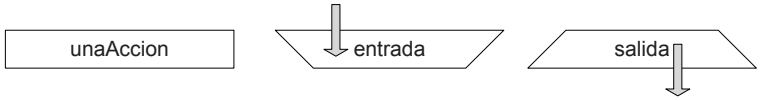

Fig. 1.1 Representación de estructuras secuenciales.

Nota: las flechas grises no son parte de los símbolos de entrada y salida, simplemente son ilustrativas.

1.4.2 Representación gráfica de la estructura de decisión

Esta estructura decide entre ejecutar un conjunto de acciones u otro en función de que se cumpla o no una determinada condición o expresión lógica.

Informalmente, diremos que una "expresión lógica" es un enunciado susceptible de ser verdadero o falso. Por ejemplo, "2 es par" o "5 es mayor que 8". Ambas expresiones tienen valor de verdad, la primera es verdadera y la segunda es falsa.

La estructura se representa dentro de una "casa con dos habitaciones y techo a dos aguas". En "el altillo" indicamos la expresión lógica que la estructura debe evaluar. Si esta expresión resulta ser verdadera entonces se ejecutarán las acciones ubicadas en la sección izquierda. En cambio, si la expresión resulta ser falsa se ejecutarán las acciones que estén ubicadas en la sección derecha.

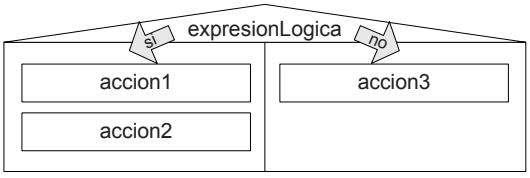

Fig. 1.2 Representación gráfica de la estructura condicional.

En este caso, si se verifica `expresionLogica` se ejecutarán las acciones `accion1` y `accion2`. En cambio, si `expresionLogica` resulta ser falsa se ejecutará únicamente la acción `accion3`. Las flechas grises son ilustrativas y no son parte del diagrama.

1.4.3 Representación gráfica de la estructura de repetición

La estructura de repetición se representa en una "caja" con una cabecera que indica la expresión lógica que se debe cumplir para seguir ejecutando las acciones contenidas en la sección principal.

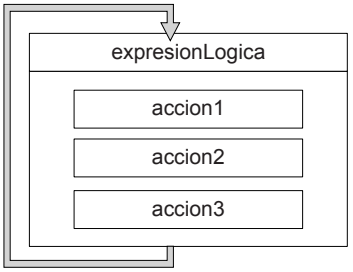

Fig. 1.3 Representación gráfica de la estructura de repetición.

En este caso, mientras `expresionLogica` resulte ser verdadera se repetirán una y otra vez las acciones `accion1`, `accion2` y `accion3`. Por lo tanto, dentro de alguna de estas acciones deberá suceder algo que haga que la condición del ciclo se deje de cumplir. De lo contrario, el algoritmo se quedará iterando dentro de este ciclo de repeticiones.

Con lo estudiado hasta aquí, podemos representar gráficamente el algoritmo que resuelve el problema de cruzar la calle.

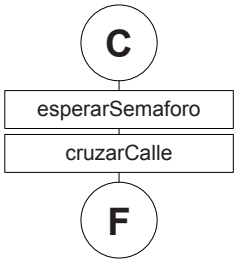

Fig. 1.4 Diagrama del algoritmo principal.

El diagrama principal comienza con una **C** y finaliza con una **F**, iniciales de "Comienzo" y "Fin", cada una encerrada dentro de un círculo. Luego, las acciones simples `esperarSemaforo` y `cruzarCalle` se representan dentro de rectángulos, una detrás de la otra. Como cada una de estas acciones corresponde a un módulo tendremos que proveer también sus propios diagramas.

1.4.4 Representación gráfica de módulos o funciones

Como estudiamos anteriormente, un módulo representa un algoritmo que resuelve un problema específico y puntual. Para representar, gráficamente, las acciones que componen a un módulo, utilizaremos un paralelogramo con el nombre del módulo como encabezado y una **R** (inicial de "Retorno") encerrada dentro de un círculo para indicar que el módulo finalizó y que se retornará el control al algoritmo o programa principal.

Fig. 1.5 Encabezado y finalización de un módulo.

Con esto, podemos desarrollar los diagramas de los módulos `esperarSemaforo` y `cruzarCalle`.

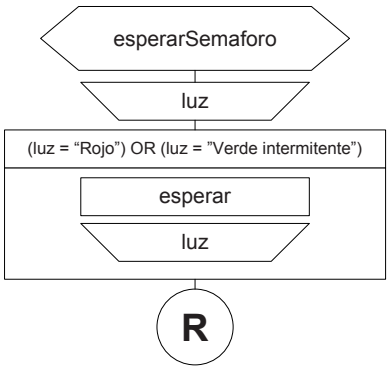

Fig. 1.6 Módulo `esperarSemaforo`.

En el diagrama del módulo `esperarSemaforo`, leemos el color de la luz que actualmente está emitiendo el semáforo y lo "mantenemos en memoria" asociándolo al identificador `luz`. Esto lo hacemos indicando el nombre del identificador dentro del símbolo de lectura o ingreso de datos.

Luego ingresamos a una estructura de repetición que iterará mientras que la expresión lógica `(luz = "Rojo") OR (luz = "Verde intermitente")` resulte verdadera. Las acciones encerradas dentro de esta estructura se repetirán una y otra vez mientras la condición anterior continúe verificándose. Estas acciones son: `esperar` y volver a leer la luz que emite el semáforo.

Evidentemente, en algún momento, el semáforo emitirá la luz "verde fija", la condición de la cabecera ya no se verificará y el ciclo de repeticiones dejará de iterar.

Veamos ahora el diagrama del módulo `cruzarCalle`.

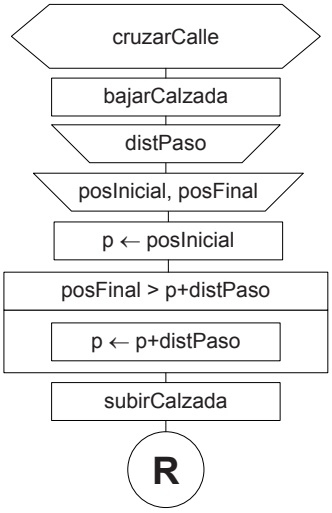

Fig. 1.7 Módulo `cruzarCalle`.

El análisis de este módulo es similar al anterior. Aquí primero invocamos al módulo `bajarCalzada` y luego, ingresamos los datos `distPaso` (distancia que avanzamos al dar un nuevo paso), `posInicial` y `posFinal` (la posición donde estamos parados y la posición de la vereda de enfrente respectivamente). A continuación, "asignamos" al identificador `p` el valor de `posInicial` y luego, ingresamos a una estructura de repetición que iterará mientras que se verifique la expresión lógica:

```
posFinal > p+distPaso
```

Dentro de la estructura de repetición, tenemos que avanzar un paso. Si bien en el primer análisis del algoritmo habíamos definido un módulo `avanzarPaso`, en este caso, preferí o reemplazarlo directamente por la acción:

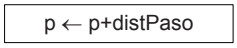

Fig. 1.8 Asignación y acumulador.

Esta acción indica que a `p` (identificador que contiene nuestra posición) le asignamos la suma de su valor actual más el valor de `distPaso` (distancia que avanzamos dando un nuevo paso). Luego de esto estaremos más cerca de `posFinal`.

Para finalizar invocamos al módulo `subirCalzada` y retornamos al diagrama del programa principal.

1.5 Nuestro primer programa

Vamos a analizar, diseñar y programar nuestro primer algoritmo. Se trata de un programa trivial que simplemente emite en la consola la frase “Hola Mundo”.

La representación gráfica de este algoritmo es la siguiente:

El algoritmo comienza, emite un mensaje en la consola diciendo “Hola Mundo” y finaliza.

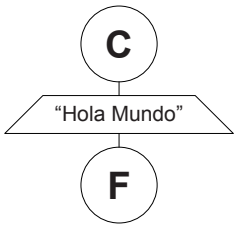

Fig. 1.9 Representación gráfica del algoritmo “Hola Mundo”.

1.5.1 Codificación del algoritmo utilizando el lenguaje C

El siguiente paso es codificar el algoritmo. Con el diagrama resuelto, la codificación se limita a transcribirlo en el `lenguaje` de programación elegido que, en este caso, es C.

Veamos el código fuente y luego lo analizaremos:

```
#include <stdio.h>

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

Todos los programas C se codifican dentro de la función `main`. La palabra “*main*” (que significa “principal”) es una palabra reservada y se utiliza como encabezado del programa principal.

Para emitir el mensaje “Hola Mundo” en la consola, utilizamos la función `printf`. Como comentamos anteriormente esta función es parte de la biblioteca estándar de entrada/salida de C llamada “stdio.h”.

La función `printf` recibe como argumento una cadena de caracteres y la imprime en la consola. Notemos que al texto que queremos imprimir le agregamos el carácter especial `\n` (léase “barra ene”). Este carácter representa un salto de línea.

Prácticamente, todos los programas C comienzan con la directiva (de preprocesador):

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

Esta directiva “incluye” las definiciones de las funciones de entrada y salida de la biblioteca estándar.

El preprocesador es un programa que es invocado por el compilador antes de comenzar su trabajo para, por ejemplo, eliminar los comentarios y otros elementos no necesarios para la compilación.

El valor de retorno de la función `main`, también llamado *exit code*, indica como finalizó el programa. Un valor igual a 0 representa una finalización exitosa mientras que un valor diferente indica algún grado de anormalidad.
El *exit code* se utiliza en la programación de *scripts* ya que en función de este se puede determinar si el programa concretó su tarea o no.

Los bloques de código se delimitan entre `{` y `}` ("llave de apertura" y "llave de cierre"). En este caso, el único bloque de código es el bloque que delimita las líneas que forman parte de la función `main` o programa principal.
La sintaxis del lenguaje de programación C exige que todas las líneas de código (salvo las directivas de preprocesador como `include` y los delimitadores de bloques) finalicen con `;` (punto y coma).

Por último, con la sentencia `return` hacemos que la función retorne (o devuelva) un valor a quién la invocó. Este valor es conocido como "el valor de retorno de la función" y lo analizaremos en detalle en el Capítulo 3.

1.5.2 El archivo de código fuente

La codificación del algoritmo constituye el "código fuente". Estas líneas de código deben estar contenidas en un archivo de texto cuyo nombre tiene que tener la extensión ".c". En este caso, el nombre del archivo de código fuente podría ser: `HolaMundo.c`.

1.5.3 Comentarios en el código fuente

A medida que la complejidad del algoritmo se incremente, el código fuente que tendremos que escribir para codificarlo será más complejo y, por lo tanto, más difícil de entender. Por este motivo, los lenguajes de programación permiten agregar comentarios de forma tal que el programador pueda anotar acotaciones que ayuden a hacer más legible el programa que desarrolló.
En C podemos utilizar dos tipos de comentarios:

- Comentarios en línea: cualquier línea de código que comience con "doble barra" será considerada como un comentario.

```
// esto es un comentario en linea
```

- Comentarios en varias líneas: son aquellos que están encerrados entre un "barra asterisco" y un "asterisco barra" como se muestra a continuación:

```
/* Esto es un
comentario en varias
lineas de codigo */
```

A continuación, agregaremos algunos comentarios al programa "HolaMundo".

```
/*
   Programa: HolaMundo.c
   Autor: Pablo Sznajdleder
   Fecha: 24/junio/2012
*/

// incluye las definiciones de las funciones de la bibloteca estandar
#include <stdio.h>

// programa o funcion principal
int main()
{
   // escribe un mensaje en la consola
   printf("Hola Mundo\n");
   return 0;
}
```

Los comentarios dentro del código fuente ayudan a que el programa sea más legible para el programador y, obviamente, no son tomados en cuenta por el compilador.

1.5.4 La compilación y el programa ejecutable

El próximo paso será compilar el código fuente para obtener el programa ejecutable y poder ejecutarlo (correrlo) en la computadora.

La compilación se realiza con un programa especial llamado "compilador". Existen diversos compiladores que permiten compilar programas escritos en C. En este libro utilizaremos el compilador *GCC* o *MinGW* (http://www.mingw.org/).

Para compilar el programa `HolaMundo.c` con *GCC* escribimos el siguiente comando en la línea de comandos:

```
gcc HolaMundo.c -o HolaMundo.exe
```

Luego de esto se generará, dentro de la misma carpeta en la que compilamos el programa, un nuevo archivo llamado `HolaMundo.exe` que al ejecutarlo mostrará el texto "Hola Mundo" en la consola.

1.5.5 El entorno integrado de desarrollo (IDE)

Cuando los algoritmos adquieren mayor nivel de complejidad, su codificación y compilación pueden convertirse en verdaderos problemas.

Una IDE (por sus siglas en inglés, *Integrated Development Environment*) es una herramienta que integra todos los recursos que necesita un programador para codificar, compilar, depurar, ejecutar y documentar sus programas.

Para programar en C existen diferentes IDEs. Algunas de estas son: Visual Studio (de Microsoft), C++ Builder (de Borland), Dev C (*open source*), Eclipse (*open source*), etcétera.

En este libro utilizaremos Eclipse. Su instalación, configuración y funcionamiento lo analizaremos en los videotutoriales que lo acompañan. Sin embargo, y a modo de ejemplo, a continuación haremos un breve repaso por la herramienta.

Instalación y uso de Eclipse para C

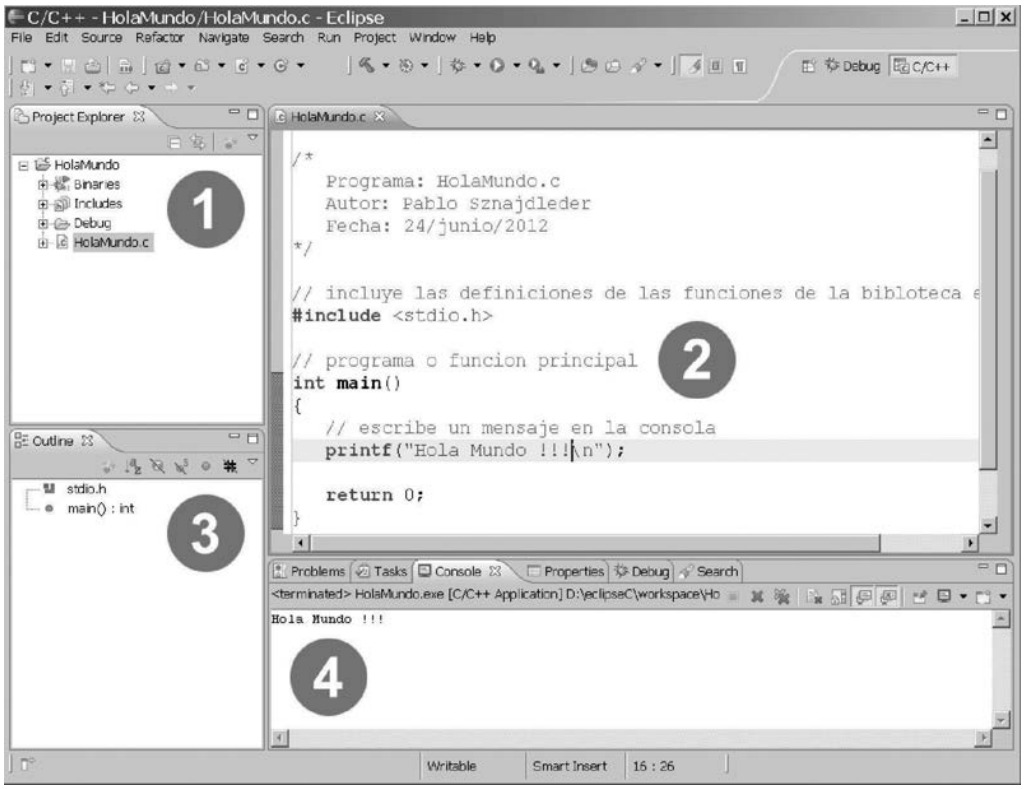

Fig. 1.10 El entorno integrado de desarrollo Eclipse.

En la imagen podemos identificar 4 secciones:

- El **Project Explorer** (1) – Aquí vemos todos los archivos de código fuente que integran nuestro proyecto. En este caso, el proyecto tiene un único archivo: `HolaMundo.c`.
- La **Ventana de Edición** (2) – Es el área principal en donde podemos ver y editar cada uno de los archivos de código fuente.
- El **Outline** (3) – Aquí se detalla la estructura del archivo de código fuente que estamos editando. Se muestra un resumen de todas sus funciones, variables, los `.h` que incluye, etcétera.
- La **Consola** (4) – Es el área en donde ingresaremos los datos a través del teclado y donde se mostrarán los resultados.

1.6 La memoria de la computadora

Más arriba hablamos de entrada y salida de datos. Los datos ingresan al algoritmo a través de cualquier dispositivo de entrada y es nuestra responsabilidad mantenerlos en memoria para tenerlos disponibles y, llegado el momento, poderlos utilizar.

Dependiendo del contenido del dato, necesitaremos utilizar una mayor o menor cantidad de memoria para almacenarlo. Por ejemplo, si el dato corresponde al nombre de una persona probablemente 20 caracteres sean más que suficiente. En cambio, si el dato corresponde a su dirección postal, seguramente, necesitemos utilizar una mayor cantidad de caracteres, quizás 150 o 200.

1.6.1 El byte

La memoria se mide en *bytes*. Un *byte* constituye la mínima unidad de información que podemos almacenar en la memoria. En la actualidad, las computadoras traen grandes cantidades de memoria expresadas en múltiplos del *byte*. Por ejemplo, 1 *gigabyte* representa 1024 *megabytes*. 1 *megabyte* representa 1024 *kilobytes* y un *kilobyte* representa 1024 *bytes*.

A su vez, un *byte* representa un conjunto de 8 bits (dígitos binarios). Por lo tanto, en un *byte* podemos almacenar un número binario de hasta 8 dígitos.

Obviamente, aunque la representación interna del número sea binaria, nosotros podemos pensarlo en base 10 como veremos a continuación.

Bit es el acrónimo de *binary digit*. (dígito binario). Corresponde a un dígito del sistema de numeración binaria que puede representar uno de dos valores, 0 o 1. Es la unidad mínima de información usada en informática.

1.6.2 Conversión numérica: de base 2 a base 10

Un número entero representado en base 2 puede ser fácilmente representado en base 10 realizando los siguientes pasos:

1. Considerar que cada dígito binario representa una potencia de 2, comenzando desde 2^0 y finalizando en 2^{n-1} siendo *n* la cantidad de dígitos binarios con los que el número está siendo representado.
2. El número en base 10 se obtiene sumando aquellas potencias de 2 cuyo dígito binario sea 1.

Analicemos un ejemplo:

Sea el número binario: 10110011 entonces:

1	0	**1**	**1**	0	0	**1**	**1**
$\mathbf{2^7}$	2^6	$\mathbf{2^5}$	$\mathbf{2^4}$	2^3	2^2	$\mathbf{2^1}$	$\mathbf{2^0}$
128	64	**32**	**16**	8	4	**2**	**1**

Vemos que cada dígito binario representa una potencia de 2, desde la potencia 0 hasta la potencia 7, comenzando desde la derecha.

Concretamente, la representación decimal del número binario 10110011 se obtiene sumando aquellas potencias de 2 cuyo dígito binario es 1.

En este caso será: $2^0+2^1+2^4+2^5+2^7 = 179$.

Según lo anterior, el mayor valor numérico que podremos almacenar en 1 *byte* de información será: 11111111 (8 bits todos en 1) y corresponde al número decimal 255 (que coincide con 2^8-1).

1.6.3 Dimensionamiento de los datos

El análisis anterior es fundamental para comprender que si, por ejemplo, el dato que va a ingresar a nuestro algoritmo corresponde a la edad de una persona podremos almacenarlo sin problemas en 1 *byte* de memoria ya que, salvo Matusalén, nadie llega a vivir 255 años.

En el Antiguo Testamento, se menciona a Matusalén como la persona más anciana. Vivió hasta los 969 años según Génesis 5:27.

También podríamos almacenar en 1 *byte* datos tales como:

- El día del mes (un número que puede estar entre 1 y 31)
- La nota obtenida en un examen (número entre 1 y 10 o entre 1 y 100)
- El número que salió en la ruleta (número entre 0 y 36)

En cambio, no podemos almacenar en 1 *byte*: el día del año ya que este valor será un número entre 1 y 365. Y no sería prudente almacenar en 1 *byte* la distancia (expresada en kilómetros) que existe entre dos ciudades ya que muy probablemente esta distancia sea superior a los 255 km.

Para estos casos necesitaremos utilizar 2 *bytes* de memoria. En 16 bits podremos almacenar valores numéricos hasta 2^{16}-1 = 65535.

1.6.4 Los números negativos

En el párrafo anterior, vimos que en 1 *byte* podemos representar valores numéricos enteros entre 0 y 255, en 2 *bytes* podemos representar valores numéricos enteros entre 0 y 65535 y, en general, en *n bytes* podremos representar valores numéricos enteros entre 0 y 2^n-1.

Para representar valores numéricos enteros negativos, se reserva el bit más significativo del conjunto de *bytes* para considerarlo como "bit de signo". Si este bit vale 1 entonces los restantes bits del conjunto de *bytes* representarán un valor numérico entero negativo. En cambio, si este bit vale 0 entonces los bits restantes estarán representando un valor numérico entero positivo.

Así, en un *byte* con bit de signo, el mayor valor que se podrá representar será 01111111 (un cero y siete unos) y corresponde al valor decimal 127 (es decir: 2^7-1) y el menor valor que se podrá almacenar será 10000000 (un 1 y siete 0) y corresponde al valor decimal: -127 (*).

Análogamente, en 2 *bytes* con bit de signo el mayor valor que podremos representar será: 0111111111111111 (un cero y 15 unos), que corresponde al valor decimal 32767 mientras que el menor valor será -32767 (*).

Ahora bien, si ***0***0000000 representa al valor decimal 0 (cero) entonces ¿la combinación ***1***0000000 representa a -0 ("menos cero")?

La respuesta a este planteamiento se fundamenta en la representación binaria interna que utilizan las computadoras para los números negativos y que se basa en el complemento a 2 del número positivo. En este esquema los valores negativos se representan invirtiendo los bits que representan a los números positivos y luego, sumando 1 al resultado.

La operación inversa nos permite obtener el valor absoluto de cualquier número binario que comience con 1. Así, el valor absoluto del número 10000000 (-0) será:
01111111+1 = 10000000 = **128**.

Por este motivo, los tipos de datos que representan números enteros signados admiten valores entre -2^{n-1} y $+2^{n-1}$-1 siendo *n* la cantidad de bits utilizados en dicha representación. 1 *byte* implica *n*=8, 2 *bytes* implica *n*=16, etcétera.

1.6.5 Los caracteres

Los caracteres se representan como valores numéricos enteros positivos. Cada carácter tiene asignado un valor numérico definido en la tabla ASCII. En esta tabla se define que el carácter 'A' se representa con el valor 65, el carácter 'B' con el 66 y así sucesivamente. Para representar al carácter 'a' se utiliza el valor 97, el 'b' se representa con el valor 98, el carácter '0' con el valor 48, el '1' con el 49, etcétera.

Por lo tanto, para representar cada carácter alcanzará con 1 *byte* de memoria y *n* caracteres requerirán *n bytes* de memoria.

1.7 Las variables

Las variables representan un espacio de la memoria de la computadora. A través de una variable, podemos almacenar temporalmente datos para tenerlos disponibles durante la ejecución del programa.

Para utilizar una variable, tenemos que especificar un nombre y un tipo de datos. Al nombre de la variable lo llamamos "identificador". En general, un identificador debe comenzar con una letra y no puede contener espacios en blanco, signos de puntuación u operadores.

Nombres de variables o identificadores **válidos** son:
- `fecha`
- `fec`
- `fechaNacimiento`
- `fechaNac`
- `fec1`
- `fec2`
- `iFechaNac`

Nombres de variables o identificadores **incorrectos** son:
- `2fecha` (no puede comenzar con un número)
- `-fecha` (no puede comenzar con un signo "menos")
- `fecha nacimiento` (no puede tener espacios en blanco)
- `fecha-nacimiento` (no puede tener el signo "menos")
- `fecha+nacimiento` (no puede tener el signo "más")

Los valores que mantienen las variables pueden cambiar durante la ejecución del programa, justamente por eso, son "variables". Para que una variable adquiera un determinado valor se lo tendremos que asignar manualmente con el operador de asignación o bien leer un valor a través de algún dispositivo de entrada y almacenarlo en la variable. Esto lo representaremos gráficamente de la siguiente manera:

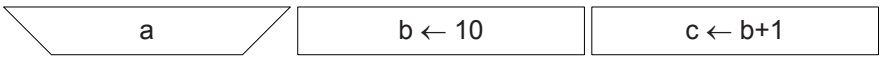

Fig. 1.11 Representación gráfica de lectura y asignación.

En las figuras vemos representadas, de izquierda a derecha, las siguientes acciones:
- Leo un valor por consola y lo asigno a la variable `a`.
- Asigno el valor 10 a la variable `b`.
- Asigno a la variable `c` el valor que contiene la variable `b` "más 1".

Recordemos que para definir una variable es necesario especificar su tipo de datos lo que significa que una variable solo podrá contener datos del mismo tipo. Es decir, la variable puede tomar diferentes valores durante la ejecución del programa, pero todos estos valores siempre serán del mismo tipo: el tipo de datos con el que la variable fue definida.

1.7.1 Convención de nomenclatura para variables

Si bien el lenguaje de programación solo exige que los nombres de las variables o identificadores respeten las restricciones mencionadas más arriba, en este libro, adoptaremos la siguiente convención de nomenclatura:

Nombres simples: deben escribirse completamente en minúscula. Por ejemplo: `fecha`, `nombre`, `edad`, `direccion`, etcétera.

Nombres compuestos: si el nombre de la variable está compuesto por dos o más palabras entonces cada palabra, a excepción de la primera, debe comenzar en mayúscula. Por ejemplo: `fechaNacimiento`, `nombreYApellido`, etcétera.

Nunca el nombre de una variable debería comenzar en mayúscula ni escribirse completamente en mayúscula.

1.7.2 Los tipos de datos

Dependiendo del tipo de datos que definamos para una variable, esta estará representando una mayor o menor cantidad de *bytes* de memoria y, por lo tanto, nos permitirá almacenar mayor o menor cantidad de información.

Según su contenido, los datos pueden clasificarse como numéricos, alfanuméricos o lógicos.

Por ejemplo: la dirección postal de una persona es un dato **alfanumérico**. Este tipo de dato representa una sucesión de caracteres alfabéticos y/o numéricos como podría ser "Av. Del Libertador Nro. 2345, piso 6".

En cambio, la edad de una persona es un dato numérico, entero positivo y acotado ya que, como analizamos más arriba, 255 es una edad absurda a la que un ser humano nunca podrá llegar. Es decir que este dato es un número **entero corto** y sin bit de signo o *unsigned* ya que ninguna persona podrá tener una edad negativa.

El kilometraje de un auto, por ejemplo, es un valor entero positivo y, potencialmente, muy grande. No podrá representarse en 1 *byte* (255), ni siquiera en 2 *bytes* (65535). Tendremos que utilizar una mayor cantidad de *bytes*. A este tipo de datos lo llamaremos **entero largo**. Este dato también es *unsigned* ya que un auto no puede tener un kilometraje negativo.

El ecuador divide al globo en el hemisferio norte y el hemisferio sur, es una línea imaginaria (un círculo máximo) que se encuentra, exactamente, a la misma distancia de los polos geográficos.

La distancia (expresada en kilómetros) que existe entre dos ciudades es un dato que puede representarse en 2 *bytes*. Esto se desprende del siguiente razonamiento: Como el diámetro del ecuador es de aproximadamente 12710 km entonces el perímetro de la Tierra se puede calcular como π*diámetro = 40074 km, también aproximado. Es decir que, en el peor de los casos, una ciudad puede estar en un punto de la Tierra y la otra puede estar justo al otro lado del mundo y aun así, la distancia nunca será mayor que la mitad de perímetro de la Tierra: 20037 km.

El saldo de una cuenta bancaria es un valor numérico real. Es decir, probablemente tenga decimales y podrá ser positivo o negativo. A este tipo de datos lo llamamos **flotante**. La representación interna de los números con punto flotante la analizaremos más adelante.

Según sea el grado de precisión que demande el valor real que tenemos que representar, su tipo podrá ser simplemente "flotante" o bien flotante de **doble precisión**.

Por último, los datos lógicos o **booleanos** son aquellos que tienen valor de verdad. Este valor puede ser verdadero o falso. En general, utilizamos este tipo de dato para almacenar el resultado de una operación lógica.

1.7.3 Los tipos de datos provistos por el lenguaje C

Los lenguajes de programación proveen tipos de datos con los cuales se puede definir (o declarar) variables.

Como vimos más arriba, para utilizar una variable será necesario definir su identificador (nombre de la variable) y su tipo de datos. En C disponemos de los siguientes tipos:

Naturaleza	Tipo	Bytes	Descripción
Tipos numéricos enteros	*short*	1	entero corto
	int	2	entero
	long	4	entero largo
	char	1	carácter (entero corto)
Tipos numéricos flotantes (o reales)	*float*	4	flotante
	double	8	flotante doble precisión

Nota muy importante: Las cantidades de *bytes* no siempre serán las mencionadas en la tabla ya que estas dependerán del compilador y de la arquitectura del *hardware* en donde estemos compilando y ejecutando nuestro programa. Sin embargo, en este libro, por cuestiones didácticas, consideraremos que esta será la cantidad de *bytes* de memoria reservada para cada tipo de dato.

A cada uno de los tipos de datos enteros se les puede aplicar el modificador `unsigned` con lo que podremos indicar si queremos o no que se deje sin efecto el bit de signo. Con esto, aumentamos el rango de valores positivos que admite el tipo de datos a costa de sacrificar la posibilidad de almacenar valores negativos.

Los datos lógicos o booleanos se manejan como tipos enteros, considerando que el valor 0 es "falso" y cualquier otro valor distinto de 0 es "verdadero". En otros lenguajes, como Pascal o Java, se provee el tipo de datos `boolean`, pero en C los datos lógicos simplemente se trabajan como `int`.

Por último, los datos alfanuméricos o "cadenas de caracteres" se representan como "conjuntos de variables" de tipo `char`. A estos "conjuntos" se los denomina *arrays*, un tema que estudiaremos en detalle más adelante.

1.7.3.1 Notación húngara

La notación húngara es una convención de nomenclatura que propone anteponer al nombre de la variable un prefijo que, a simple vista, permita identificar su tipo de datos.

Por ejemplo: `iContador` ("i" de `int`), `sNombre` ("s" de *string*), `bFin` ("b" de *boolean*), `dPrecioVenta` ("d" de `double`), etcétera.

Si bien la notación húngara cuenta con una importante comunidad de detractores que aseguran que, a la larga, complica la legibilidad y la mantenibilidad del código fuente, personalmente considero que, en ciertas ocasiones, se puede utilizar.

1.7.4 La función de biblioteca `printf`

Como ya sabemos la función `printf` permite mostrar datos en la consola. La salida puede estar compuesta por un texto fijo (texto literal) o por una combinación de texto literal y contenido variable como veremos en el siguiente programa.

```
#include <stdio.h>

int main()
{
   char nombre[] = "Pablo";
   int edad = 39;
   double altura = 1.70;
```

```
   printf("Mi nombre es %s, tengo %d anios y mido %lf metros.\n"
          ,nombre
          ,edad
          ,altura);

   return 0;
}
```

La salida de este programa es:

```
Mi nombre es Pablo, tengo 39 anios y mido 1.70 metros.
```

La función `printf` intercaló los valores de las variables `nombre`, `edad` y `altura` dentro del texto literal, en las posiciones indicadas con los `%` (marcadores de posición o *placeholders*).

La cadena que contiene el texto literal con los marcadores de posición se llama "máscara" y es el primer argumento que recibe `printf`. A continuación, le pasamos tantos argumentos como *placeholders* tenga la máscara.

```
printf("Mi nombre es %s, tengo %d anios y mido %lf metros.\n"
       ,nombre
       ,edad
       ,altura);
```

Cada marcador debe ir acompañado de un carácter que representa un tipo de datos. Por ejemplo, `%s` indica que allí se mostrará un valor alfanumérico, `%d` representa a un valor entero y `%lf` indica que se mostrará un valor flotante.

Por último, pasamos la lista de variables cuyos valores se intercalarán en el texto que queremos que `printf` escriba en la consola.

1.7.5 La función de biblioteca `scanf`

Esta función permite leer datos a través del teclado y los asigna en las variables que le pasemos como argumento.

Analicemos el código del siguiente programa donde se le pide al usuario que ingrese datos de diferentes tipos.

```
#include <stdio.h>

int main()
{
   char nombre[20];
   int edad;
   double altura;

   printf("Ingrese su nombre: ");
   scanf("%s",nombre);

   printf("Ingrese su edad: ");
   scanf("%d",&edad);

   printf("Ingrese su altura: ");
   scanf("%lf",&altura);
```

```
   printf("Ud. es %s, tiene %d anios y una altura de %lf\n"
         ,nombre
         ,edad
         ,altura);

   return 0;
}
```

scanf recibe como primer argumento una máscara que indica el tipo de los datos que la función debe leer. Los marcadores coinciden con los que utiliza printf, por tanto con %s le indicamos que lea una cadena de caracteres y con %d y %lf le indicamos que lea un entero y un flotante respectivamente.

Para que una función pueda modificar el valor de un argumento tenemos que pasarle su referencia o su dirección de memoria. Esto lo obtenemos anteponiendo el operador & (léase "operador ampersand") al identificador de la variable cuyo valor queremos que la función pueda modificar. Veamos las siguientes líneas:

```
printf("Ingrese su edad: ");
scanf("%d",&edad);

printf("Ingrese su altura: ");
scanf("%lf",&altura);
```

Utilizamos scanf para leer un valor entero y asignarlo en la variable edad, luego usamos scanf para leer un valor flotante y asignarlo en la variable altura.

El caso de las cadenas de caracteres es diferente. Como comentamos más arriba, las cadenas se manejan como *arrays* (o conjuntos) de caracteres. Si bien este tema lo estudiaremos más adelante, es importante saber que el identificador de un *array* es en sí mismo su dirección de memoria, razón por la cual no fue necesario anteponer el operador & a la variable nombre para que scanf pueda modificar su valor.

```
printf("Ingrese su nombre: ");
scanf("%s",nombre);
```

Nota: en todos los casos scanf lee desde el teclado valores alfanuméricos. Luego, dependiendo de la máscara, convierte estos valores en los tipos de datos que corresponda y los asigna en sus respectivas variables.

1.7.6 El operador de dirección &

El operador & se llama "operador de dirección" y aplicado a una variable nos permite obtener su referencia o dirección de memoria.

Tendremos que utilizar este operador cada vez que necesitemos que una función pueda modificar el valor de alguna variable que le pasemos como argumento.

1.7.7 Las constantes

Los algoritmos resuelven situaciones problemáticas que surgen de la realidad, donde existen valores que nunca cambiarán. Dos ejemplos típicos son los números PI y E cuyas aproximaciones son: 3.141592654 y 2.718281828 respectivamente.

1.7.7.1 La directiva de preprocesador #define

Esta directiva permite definir valores constantes de la siguiente manera:

```
#define NUMERO_PI 3.1415169254
#define NUMERO_E 2.718281828
```

El preprocesador de C reemplazará cada aparición de `NUMERO_PI` y `NUMERO_E` por sus respectivos y correspondientes valores.

1.7.7.2 El modificador `const`

Si a la declaración de una variable le aplicamos el modificador `const` su valor ya no podrá ser modificado. Por ejemplo:

```
const int temperaturaMaxima = 45;
```

Luego, cualquier intento de asignar otro valor a la variable `temperaturaMaxima` en el código del programa generará un error de compilación.

1.7.8 Nomenclatura para las constantes

Adoptaremos la siguiente convención de nomenclatura para los nombres de las constantes definidas con la directiva `#define`:

Las constantes deben escribirse completamente en mayúscula. Luego, si se trata de un nombre compuesto por dos o más palabras, cada una debe separarse de la anterior mediante el carácter guión bajo o *underscore*.

En cambio, para las constantes declaradas con el modificador `const` respetaremos la convención de nomenclatura para variables estudiada más arriba.

1.8 Operadores aritméticos

Comenzamos el capítulo explicando que los recursos que tenemos para diseñar y desarrollar algoritmos son la memoria y la capacidad de ejecutar operaciones aritméticas y lógicas.

Todos los lenguajes de programación proveen un conjunto de operadores con los que podemos realizar operaciones aritméticas. En C estos operadores son los siguientes:

Operador	Descripción
+	suma
-	resta
*	multiplicación
/	división (cociente)
%	módulo, resto o valor residual

Veamos un ejemplo.

Problema 1.1

Leer dos valores enteros e informar su suma.

Análisis

En este problema, los datos de entrada son los dos valores numéricos enteros que ingresará el usuario. La salida del algoritmo será un mensaje en la consola informando la suma de estos valores. El proceso implica sumar los dos valores y mostrar el resultado en la pantalla.

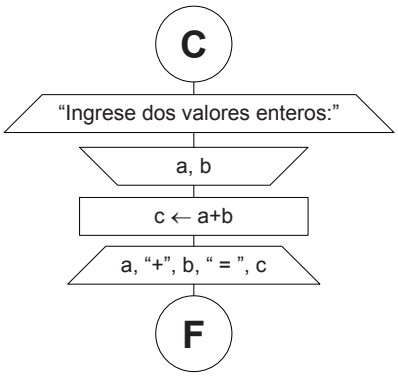

Fig. 1.12 Suma dos valores numéricos.

Comenzamos el algoritmo emitiendo un mensaje para informarle al usuario que debe ingresar dos valores numéricos enteros. A continuación, leemos los valores que el usuario va a ingresar, quedarán asignados en las variables `a` y `b`. Luego asignamos en la variable `c` la suma `a+b` y mostramos el resultado intercalando los valores de las variables `a`, `b` y `c` con las cadenas literales “+” e “=”.

La codificación del algoritmo es la siguiente:

```
#include <stdio.h>

int main()
{
   int a,b,c;

   printf("Ingrese dos valores enteros: ");
   scanf("%d %d",&a, &b);

   c = a+b;

   printf("%d + %d = %d\n",a,b,c);

   return 0;
}
```

El algoritmo hubiera sido, prácticamente, el mismo si en lugar de tener que mostrar la suma de los dos valores ingresados por el usuario nos hubiera pedido que mostremos su diferencia o su producto. Pero, ¿qué sucedería si nos pidieran que mostremos su cociente? Lo analizaremos a continuación:

Problema 1.2

Leer dos valores numéricos enteros e informar su cociente.

Análisis

En este problema, los datos de entrada son los dos valores enteros que ingresará el usuario a través del teclado (los llamaremos `a` y `b`) y la salida será su cociente (un número flotante).

Ahora bien, existe la posibilidad de que el usuario ingrese como segundo valor el número 0 (cero). En este caso, no podremos mostrar el cociente ya que la división por cero es una indeterminación, así que tendremos que emitir un mensaje informando las causas por las cuales no se podrá efectuar la operación.

Para resolver este algoritmo utilizaremos una estructura condicional que nos permitirá decidir, en función del valor de `b`, entre mostrar un mensaje de error y realizar la división e informar el cociente. Veamos el algoritmo:

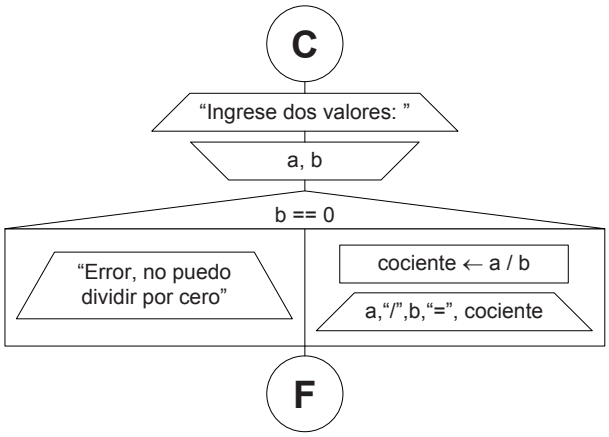

Fig. 1.13 Muestra el cociente de la división de dos números.

Leemos dos valores en las variables `a` y `b` y luego preguntamos si el valor de `b` es cero. Para esto, utilizamos el operador de comparación `==` (léase "igual igual"). Si efectivamente `b` vale cero entonces mostramos un mensaje de error informando lo sucedido, si no asignamos a la variable `cociente` el resultado de la división `a/b` y lo mostramos en la consola. El código fuente es el siguiente:

```
#include <stdio.h>

int main()
{
   int a,b;
   double cociente;

   printf("Ingrese dos valores: ");
   scanf("%d %d", &a, &b);

   // verifico si el denomicador es cero
   if( b == 0 )
   {
      printf("Error, no puedo dividir por cero\n");
   }
   else
   {
      cociente = (double)a/b;
      printf("%d / %d = %lf\n",a,b,cociente);
   }

   return 0;
}
```

Como vemos, la estructura de decisión se codifica con `if` ("si" en inglés). Si se verifica la condición expresada dentro de los paréntesis del `if` el programa ingresará por el primer bloque de código para mostrar el mensaje de error. Si la condición no se verifica (`else`) el programa ingresará al segundo bloque de código para efectuar la división y mostrar el resultado.

1.8.1 Conversión de tipos de datos (type casting)

C permite convertir datos de un tipo a otro. Esta operación se llama *type casting* o simplemente *casting*.

Si bien la conversión de datos de un tipo a otro es automática, en ocasiones necesitaremos realizarla explícitamente.

Observemos la línea de código donde calculamos la división y asignamos el resultado a la variable `cociente`:

```
cociente = (double)a/b;
```

El operador `/` (operador de división) convierte el resultado al mayor tipo de datos de sus operandos. Esto significa que si estamos dividiendo dos `int` entonces el resultado también será de tipo `int`, por lo tanto, el cociente que obtendremos será el de la división entera.

Para solucionar este problema y no perder los decimales, tenemos que convertir el tipo de alguno de los operandos a `double` (el tipo de datos de la variable `cociente`). Así, el mayor tipo de datos entre `int` y `double` es `double` por lo que el resultado de la división también lo será.

Le recomiendo al lector eliminar el *casteo* y luego recompilar y probar el programa para observar que resultados arroja.

1.8.2 El operador `%` ("módulo" o "resto")

El operador `%` (léase "operador módulo" u "operador resto") retorna el resto (o valor residual) que se obtiene luego de efectuar la división entera de sus operandos.

Por ejemplo:

```
int a = 5;
int b = 3
int r = a % b;
```

En este ejemplo estamos asignando a la variable `r` el valor 2 ya que este es el resto (o valor residual) que se origina al dividir 5 por 3.

Problema 1.3

Dado un valor numérico entero, informar si es par o impar.

Análisis

En este problema tenemos un único dato de entrada: un valor numérico entero que deberá ingresar el usuario. La salida del algoritmo será informar si el usuario ingresó un valor par o impar.

Sabemos que un número par es aquel que es divisible por 2 o, también, que un número es par si el valor residual que se obtiene al dividirlo por 2 es cero.

Según lo anterior, podremos informar que el número ingresado por el usuario es par si al dividirlo por 2 obtenemos un resto igual a cero. De lo contrario, informaremos que el número es impar.

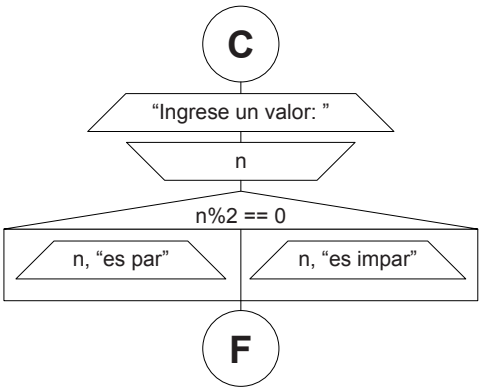

Fig. 1.14 Informa si un número es par o impar.

El código fuente es el siguiente:

```
#include <stdio.h>

int main()
{
   int n;

   printf("Ingrese un valor: ");
   scanf("%d",&n);

   if( n%2 == 0 )
   {
      printf("%d es par\n",n);
   }
   else
   {
      printf("%d es impar\n",n);
   }

   return 0;
}
```

Veamos otro ejemplo en el cual tendremos que utilizar los operadores aritméticos de módulo y división.

Problema 1.4

Se ingresa un valor numérico de 8 dígitos que representa una fecha con el siguiente formato: *aaaammdd*. Esto es: los 4 primeros dígitos representan el año, los siguientes 2 dígitos representan el mes y los 2 dígitos restantes representan el día. Se pide informar por separado el día, el mes y el año de la fecha ingresada.

Análisis

El dato que ingresará al algoritmo es un valor numérico de 8 dígitos como el siguiente: 20081015. Si este fuera el caso, entonces la salida deberá ser:

```
dia: 15
mes: 10
anio: 2008
```

Es decir, el problema consiste en desmenuzar el número ingresado por el usuario para separar los primeros 4 dígitos, después los siguientes 2 dígitos y luego, los 2 últimos.

Supongamos que efectivamente el valor ingresado es 20081015 entonces si lo dividimos por 10 obtendremos el siguiente resultado:

20081015 / 10 = 2008101,5

Pero si en lugar de dividirlo por 10 lo dividimos por 100 el resultado será:

20081015 / 100 = 200810,15

Siguiendo el mismo razonamiento, si al número lo dividimos por 10000 entonces el resultado que obtendremos será:

20081015 / 10000 = 2008,1015

Si de este valor tomamos solo la parte entera tendremos 2008 que coincide con el año representado en la fecha que ingresó el usuario.

Por otro lado, el resto obtenido en la división anterior será: 1015. Esto coincide con el mes y el día, por lo tanto, aplicando un razonamiento similar podremos separar y mostrar estos valores.

Para representar la "división entera" en los diagramas, utilizaremos el operador `div`. Este operador no existe en C, pero nos permitirá diferenciar, visualmente, entre una división real o flotante y una división entera.

Veamos el algoritmo:

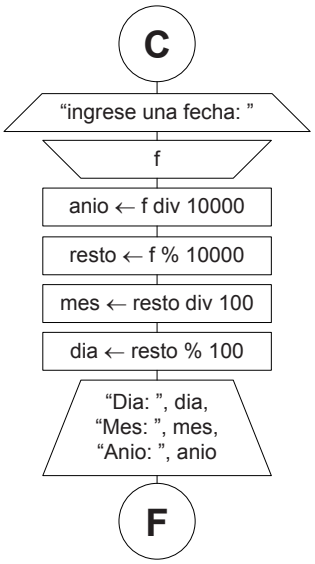

Fig. 1.15 Separa los dígitos de un número entero.

Para codificar este algoritmo debemos tener en cuenta que el valor que ingresará el usuario representa una fecha con formato *aaaammdd*. Es decir que, en el peor de los casos, este valor será 99991231 y no lo podemos almacenar en 2 *bytes* ya que excede por mucho su capacidad (32767 o 65535). Por este motivo, utilizaremos 4 *bytes* que nos permitirán almacenar números de hasta 2^{32}-1 que superan los 8 dígitos, es decir: trabajaremos con variables de tipo `long`.

Recordemos que, por cuestiones didácticas, consideramos que el tipo `int` representa 2 *bytes* y el tipo `long` representa 4 *bytes*.

El código fuente es el siguiente:

```
#include <stdio.h>

int main()
{
   long f;
   int dia, mes, anio;
   int resto;

   printf("Ingrese una fecha: ");
   scanf("%ld",&f);

   // recordemos que la division entre dos enteros tambien lo sera
   anio = f/10000;
   resto = f%10000;

   mes = resto/100;
   dia = resto%100;

   printf("Dia: %d\n",dia);
   printf("Mes: %d\n",mes);
   printf("Anio: %d\n",anio);

   return 0;
}
```

Recordemos que el operador de división `/` retorna un resultado del mismo tipo de datos que el mayor tipo de sus operandos. Como, en este caso, `f` es de tipo `long` entonces el cociente también lo será. Es decir que la división será entera.

1.8.3 Operadores relacionales

Estos operadores permiten evaluar la relación que existe entre dos valores numéricos. Ya hemos mencionado y utilizado al operador de comparación `==` (igual igual) para comparar si un número es igual a otro. Ahora veremos la lista completa que incluye a los operadores "mayor", "menor", "mayor o igual", "menor o igual" y "distinto".

Operador	Descripción
>	mayor que...
<	menor que...
>=	mayor o igual que...
<=	menor o igual que...
==	igual a...
!=	distinto de...

Como veremos más adelante, el operador `!` (signo de admiración) es el operador lógico de negación (operador "*not*"), por lo tanto, el operador `!=` puede leerse como "distinto", "no igual" o "*not equals*".

1.9 Expresiones lógicas

Llamamos expresión lógica a una proposición que es susceptible de ser verdadera o falsa. Es decir, una proposición que tiene valor de verdad.

Por ejemplo, las siguientes proposiciones son expresiones lógicas cuyo valor de verdad es verdadero:

- La Tierra gira alrededor del Sol.
- EE. UU. tiene dos costas.
- 2 “es menor que” 5 (o simplemente 2<5).
- 5+1 = 6

Y las siguientes proposiciones son expresiones lógicas cuyo valor de verdad es falso:

- La Tierra es el centro del universo.
- EE. UU. queda en Europa.
- 2 “es mayor que” 5 (o simplemente 2>5).
- 5+1 = 8

El Feng Shui explica como nos afectan los flujos energéticos de nuestro entorno, dicha energía es conocida como Chi, circula por los diferentes espacios afectando en su recorrido todo lo que toca.

En cambio, no son expresiones lógicas las siguientes proposiciones:

- Hoy hace frío.
- La pared está bastante sucia.
- Los números impares tienen mejor “Chi”.

Las expresiones lógicas pueden combinarse entre sí formando nuevas expresiones lógicas con su correspondiente valor de verdad. Para esto, se utilizan los operadores lógicos.

1.9.1 Operadores lógicos

Las expresiones lógicas pueden conectarse a través de los operadores lógicos y así se obtienen expresiones lógicas compuestas cuyo valor de verdad dependerá de los valores de verdad de las expresiones lógicas simples que las componen.

Los operadores lógicos son los siguientes:

Operador	Descripción
`&&`	“*and*” o producto lógico
`||`	“*or*” o suma lógica
!	“*not*” o negación

Con los operadores lógicos podemos conectar dos o más expresiones lógicas y así obtener una nueva expresión lógica con su correspondiente valor de verdad.

Sean las expresiones lógicas `p` y `q` cada una con su correspondiente valor de verdad entonces el valor de verdad de la expresión lógica `h = p && q` será verdadero o falso según la siguiente tabla:

p	**q**	**h = p && q**
verdadero	verdadero	verdadero
verdadero	falso	falso
falso	verdadero	falso
falso	falso	falso