2

Estructuras básicas de control y lógica algorítmica

Contenido

Objetivos del capítulo

- Analizar y resolver algoritmos utilizando las estructuras de control básicas: secuencial, condicional e iterativa.
- Conocer, identificar y aplicar recursos algorítmicos: contadores, acumuladores, máximos, mínimos, etcétera.
- Hacer un seguimiento "paso a paso" del algoritmo y aprender a usar la herramienta de depuración: el *debugger*.
- Analizar y resolver problemas contextualizados.

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.
- Construir programas utilizando estructuras condicionales y repetitivas para aumentar su funcionalidad.

2.1 Introducción

En el capítulo anterior, estudiamos el teorema de la programación estructurada que describe tres estructuras básicas de control con las que demuestra que es posible resolver cualquier problema computacional.

En este capítulo, estudiaremos las diferentes implementaciones que provee C para estas estructuras y, además, analizaremos problemas con mayor grado de complejidad que nos permitirán integrar todos los conceptos que incorporamos a lo largo del Capítulo 1.

2.2 Estructura secuencial

En este caso, simplemente, haremos un breve repaso para recordar que la estructura secuencial consiste en ejecutar, secuencialmente, una acción simple detrás de otra.

Recordemos, también, que se considera acción simple a las acciones de leer, escribir, asignar valor a una variable e invocar a un módulo o función.

2.3 Estructura de decisión

La estructura de decisión permite decidir entre ejecutar uno u otro conjunto de acciones en función de que se cumpla o no una determinada condición lógica.

En el capítulo anterior, explicamos el uso del `if`. Más adelante, en este mismo capítulo, veremos que existen otras estructuras selectivas como por ejemplo la decisión múltiple (`switch`) y el *if-inline*.

Comencemos por analizar un problema extremadamente simple.

Problema 2.1

Leer dos valores numéricos enteros e indicar cuál es el mayor y cuál es el menor. Considerar que ambos valores son diferentes.

Análisis

Los datos de entrada para este problema son los dos valores que ingresará el usuario. Nuestra tarea será compararlos para determinar cuál es mayor y cuál es menor.

El enunciado dice que debemos considerar que los valores serán diferentes. Por lo tanto, para nuestro análisis la posibilidad de que los valores sean iguales no será tomada en cuenta.

Llamaremos `a` al primer valor y `b` al segundo. Para compararlos utilizaremos una estructura de decisión que nos permitirá determinar si `a` es mayor que `b`. Si esto resulta verdadero, entonces `a` será el mayor y `b` será el menor. De lo contrario, si la condición anterior resulta falsa, `b` será el mayor y `a` el menor ya que, como mencionamos más arriba, `a` y `b` no serán iguales.

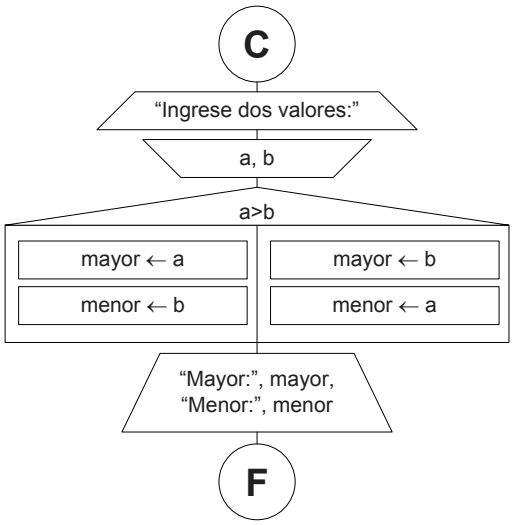


Fig. 2.1 Compara dos valores numéricos.

En el algoritmo utilizamos las variables `mayor` y `menor`. Una vez que comparamos `a` con `b` y determinamos cuál es el mayor y cuál es el menor, asignamos sus valores a estas variables y luego, mostramos sus contenidos.

El código fuente es el siguiente:

```
#include <stdio.h>

int main()
{
   int a,b;
   int mayor,menor;

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

   if( a>b )
   {
      mayor=a;
      menor=b;
   }
   else
   {
      mayor=b;
      menor=a;
   }

   printf("Mayor: %d\n",mayor);
   printf("Menor: %d\n",menor);

   return 0;
}
```

Para resolver este problema, utilizamos una estructura de decisión que nos permitió determinar cuál es el mayor valor. Luego, por descarte, el otro es el menor.

2.3.1 Estructuras de decisión anidadas

Cuando en una estructura de decisión utilizamos otra estructura de decisión, decimos que ambas son estructuras anidadas. En el siguiente problema, utilizaremos estructuras de decisión anidadas para determinar, entre tres valores numéricos, cuál es el mayor, cuál es el del medio y cuál es el menor.

Problema 2.2

Leer tres valores numéricos enteros, indicar cuál es el mayor, cuál es el del medio y cuál, el menor. Considerar que los tres valores serán diferentes.

Veamos primero el algoritmo y luego lo analizaremos.

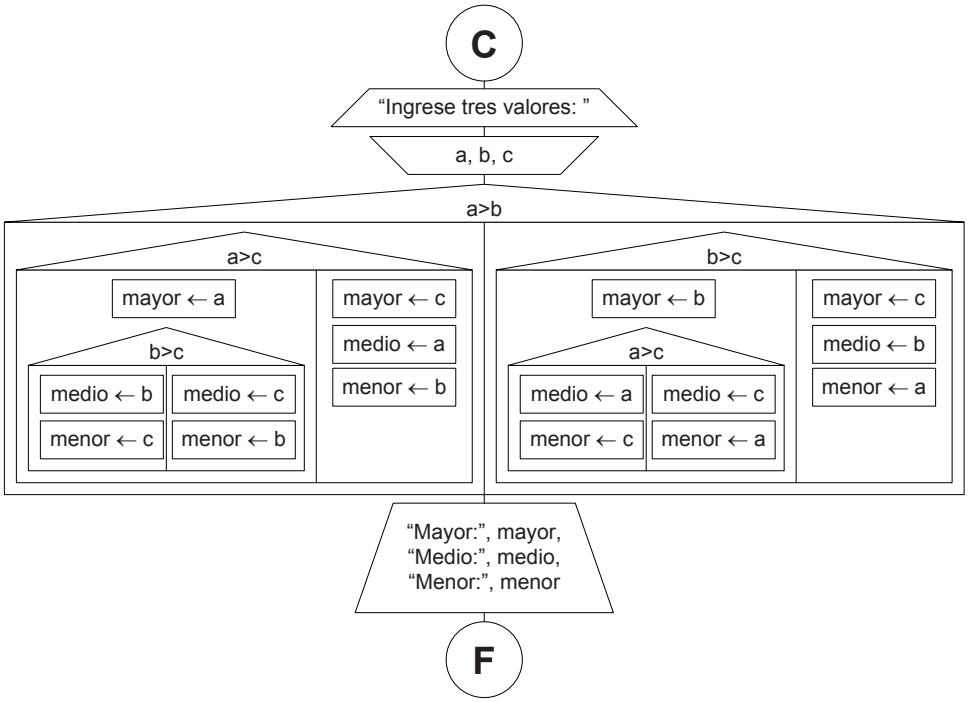


Fig. 2.2 Compara tres valores y determina cuál es mayor, medio y menor.

Análisis

Leemos los tres valores y comenzamos a comparar preguntando si `a>b`. Si esto se verifica entonces preguntamos si `a>c`. Si esto también se verifica entonces, como `a>b` y `a>c`, no hay dudas de que `a` es el mayor. Luego tenemos que comparar `b` y `c` para ver cuál está en segundo y en tercer lugar.

Si resulta que `a>b` pero no se verifica que `a>c` (es decir que `c` es mayor que `a`) será porque `c` es el mayor, `a` el medio y `b` el menor.

Por otro lado, si no se verifica que `a>b` preguntamos si `b>c`. Si esto es así, entonces el mayor será `b` (ya que `b` es mayor que `a` y `b` es mayor que `c`). Preguntamos si `a>c` y podremos deducir cuál está en segundo y tercer lugar.

Para finalizar, si es falso que `b>c` entonces el mayor será `c`, medio `b` y menor `a`.
El código fuente es el siguiente:

```
#include <stdio.h>

int main()
{
   int a,b,c;
   int mayor,medio,menor;

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

   if( a>b )
   {
      if( a>c )
      {
         mayor=a;
         if( b>c )
         {
            medio=b;
            menor=c;
         }
         else
         {
            medio=c;
            menor=b;
         }
      }
      else
      {
         mayor=c;
         medio=a;
         menor=b;
      }
   }
   else
   {
      if( b>c )
      {
         mayor=b;
         if( a>c )
         {
            medio=a;
            menor=c;
         }
         else
         {
            medio=c;
            menor=a;
         }
      }
      else
      {
```

```
            mayor=c;
            medio=b;
            menor=a;
        }
    }

    printf("Mayor: %d\n",mayor);
    printf("Medio: %d\n",medio);
    printf("Menor: %d\n",menor);

    return 0;
}
```

Para resolver este ejercicio recurrimos al uso de estructuras de decisión anidadas. El lector habrá notado que, a medida que anidamos más estructuras de decisión, resulta más complicado de seguir el código fuente.
Quizás un mejor análisis del problema hubiera sido el siguiente:
Si `a>b && a>c` entonces el mayor es `a`, el del medio será el máximo valor entre `b` y `c` y el menor será el mínimo valor entre estos.
Si la condición anterior no se verifica será porque `a` no es el mayor valor. Supongamos entonces que `b>a && b>c`. En este caso, el mayor será `b` y los valores medio y menor serán el máximo entre `a` y `c` y el mínimo entre `a` y `c` respectivamente.
Siguiendo este análisis, el algoritmo podría plantearse de la siguiente manera:

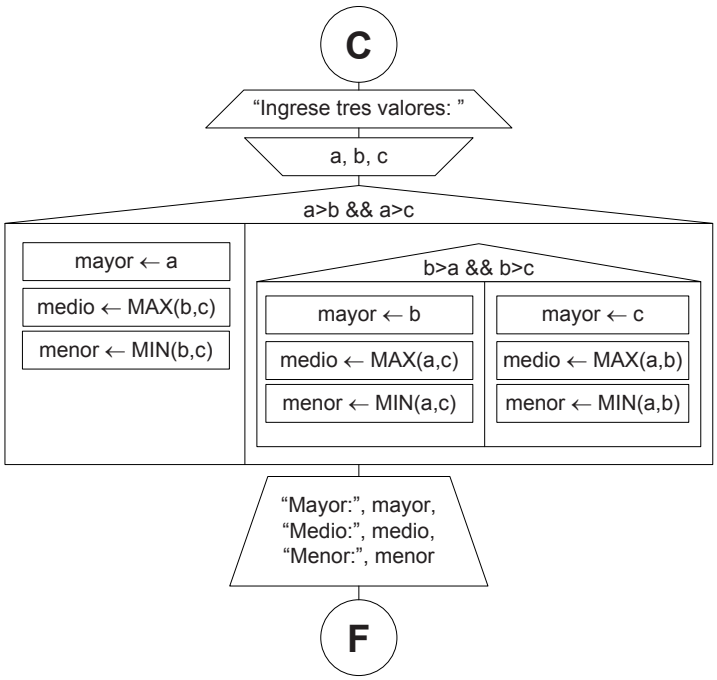


Fig. 2.3 Compara tres valores y determina cuál es mayor, medio y menor.

Para implementar esta solución, utilizaremos la estructura de selección en línea o *if-inline*.

2.3.2 Selección en línea o if-inline

Esta estructura implementa un `if` en una única línea de código y funciona así:

```
resultado = condicion ? expresion1:expresion2;
```

Si `condicion` resulta verdadera el *if-inline* retorna la expresión ubicada entre el signo de interrogación y los dos puntos. Si `condicion` resulta falsa entonces el valor de retorno será la expresión ubicada inmediatamente después de los dos puntos.

Por ejemplo: sean las variables `a` y `b` enteras y cada una con un valor numérico, y la variable `mayor` también entera, entonces:

```
mayor = a>b?a:b; // asigno a mayor el maximo valor entre a y b
```

Esta línea debe interpretarse de la siguiente manera: si se verifica que `a>b` entonces el *if-inline* retorna `a` (expresión ubicada entre el signo de interrogación y los dos puntos). Si la expresión lógica no se verifica entonces el valor de retorno del *if-inline* será `b` (expresión ubicada luego de los dos puntos). El resultado del *if-inline* lo asignamos a la variable `mayor`.

Utilizando el *if-inline* podemos codificar la segunda versión del problema 2.2 haciendo que sea más legible al reducir la cantidad de "ifes" anidados.

```
#include <stdio.h>

int main()
{
   int a,b,c;
   int mayor,medio,menor;

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

   if( a>b && a>c )
   {
      mayor=a;
      medio=b>c?b:c; // el mayor entre b y c
      menor=b<c?b:c; // el menor entre b y c
   }
   else
   {
      if( b>a && b>c )
      {
         mayor=b;
         medio=a>c?a:c; // el mayor entre a y c
         menor=a<c?a:c; // el menor entre a y c
      }
      else
      {
         mayor=c;
         medio=a>b?a:b; // el mayor entre a y b
         menor=a<b?a:b; // el menor entre a y c
      }
   }
```

```
   printf("Mayor: %d\n",mayor);
   printf("Medio: %d\n",medio);
   printf("Menor: %d\n",menor);

   return 0;
}
```

2.3.3 Macros

Las macros son directivas de preprocesador con las que podemos relacionar un nombre con una expresión.

Por ejemplo:

```
#define MAX(x,y) x>y?x:y
#define MIN(x,y) x<y?x:y
```

El preprocesador de C reemplazará cada macro por la expresión que representa. Así, podemos invocar a la macro `MAX` de la siguiente manera:

```
mayor = MAX(a,b);
```

donde `a` y `b` son argumentos que le pasamos a la macro `MAX`. El preprocesador reemplazará la línea anterior por la siguiente línea:

```
mayor = a>b?a:b
```

Utilizando estas macros podemos mejorar aún más la legibilidad del código del problema 2.2.

```
#include <stdio.h>

// definimos las macros
#define MAX(x,y) x>y?x:y
#define MIN(x,y) x<y?x:y

int main()
{
   int a,b,c;
   int mayor,medio,menor;

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

   if( a>b && a>c )
   {
      mayor=a;
      medio=MAX(b,c);
      menor=MIN(b,c);
   }
   else
   {
      if( b>a && b>c )
      {
         mayor=b;
         medio=MAX(a,c);
         menor=MIN(a,c);
```

```
      }
      else
      {
         mayor=c;
         medio=MAX(a,b);
         menor=MIN(a,b);
      }
   }

   printf("Mayor: %d\n",mayor);
   printf("Medio: %d\n",medio);
   printf("Menor: %d\n",menor);

   return 0;
}
```

2.3.4 Selección múltiple (switch)

La estructura de selección múltiple permite tomar una decisión en función de que el valor de una variable o el resultado de una expresión numérica entera coincidan o no con alguno de los valores indicados en diferentes casos o con ninguno de estos. Gráficamente, la representaremos así:

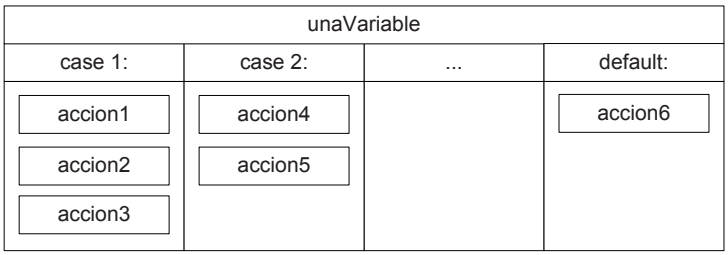


Fig. 2.4 Representación gráfica de la estructura de selección múltiple.

Este gráfico debe interpretarse de la siguiente manera: si el valor de `unaVariable` es 1 entonces se ejecutarán las acciones `accion1`, `accion2` y `accion3`. En cambio, si `unaVariable` vale 2 se ejecutarán las acciones `accion4` y `accion5`. Podemos agregar tantos casos como necesitemos. Por último, podemos indicar la opción `default` que representa a todos los otros casos que no fueron indicados explícitamente. A continuación, vemos la sintaxis genérica de esta estructura.

```
switch(expresion)
{
   case expresion_cte_1:
      sentencia_1;
      break;
   case expresion_cte_2:
      sentencia_2;
      break;
    :
   case expresion_cte_n:
      sentencia_n;
      break;
   [default:
      sentencia;]
}
```

Los casos deben representarse con valores numéricos literales o constantes. El caso `default` es opcional.

Problema 2.3

Leer un valor numérico que representa un día de la semana. Se pide mostrar por pantalla el nombre del día considerando que el lunes es el día 1, el martes es el día 2 y así, sucesivamente.

Análisis

Este problema se puede resolver fácilmente utilizando una estructura de selección múltiple como veremos a continuación:

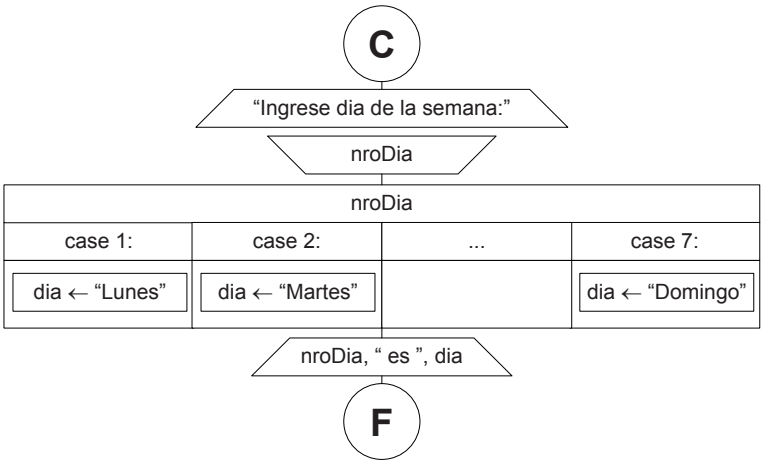


Fig. 2.5 Muestra el día de la semana según el número de día ingresado.

En el diagrama leemos el número de día en la variable `nroDia` y luego, utilizamos una estructura de selección múltiple (`switch`) con los casos 1, 2, 3, 4, 5, 6 y 7. Si el usuario ingresó el número de día 1, entonces la estructura ingresará por el caso `case 1` donde le asignamos la cadena "Lunes" a la variable `dia`. Análogamente, si el usuario ingresa el valor 2, entraremos por `case 2` y le asignaremos a `dia` la cadena "Martes" y así, sucesivamente.

La codificación es la siguiente:

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

int main()
{
   int nroDia;
   char dia[10];

   printf("Ingrese dia de la semana: ");
   scanf("%d",&nroDia);
```

```
   switch( nroDia )
   {
      case 1:
         strcpy(dia,"Lunes");  // asigno a dia la cadena "Lunes"
         break;
      case 2:
         strcpy(dia,"Martes"); // asigno a dia la cadena "Martes"
         break;
      case 3:
         strcpy(dia,"Miercoles");
         break;
      case 4:
         strcpy(dia,"Jueves");
         break;
      case 5:
         strcpy(dia,"Viernes");
         break;
      case 6:
         strcpy(dia,"Sabado");
         break;
      case 7:
         strcpy(dia,"Domingo");
         break;
   }

   printf("%d es %s\n",nroDia,dia);

   return 0;
}
```

Es muy importante poner la sentencia `break` al finalizar el conjunto de acciones que se ejecutan dentro de cada caso ya que si la omitimos se ejecutarán secuencialmente todas las acciones de todos los casos subsiguientes.

Es decir, supongamos que el usuario ingresa el día número 5 y omitimos poner los *breaks* entonces el programa ingresará por `case 5`, asignará "Viernes" a `dia`, luego le asignará "Sábado" y luego "Domingo". Por lo tanto, la salida será:

```
5 es Domingo
```

Para asignar el nombre de cada día a la variable `dia`, no utilizamos el operador de asignación, lo hacemos a través de la función `strcpy`. Esto lo explicaremos a continuación.

2.3.5 Asignación de valores alfanuméricos (función `strcpy`)

Como comentamos anteriormente las cadenas de caracteres tienen un tratamiento especial ya que en C se implementan sobre *arrays* (conjuntos) de caracteres. Por este motivo, no podemos utilizar el operador de asignación `=` para asignarles valor. Tenemos que hacerlo a través de la función de biblioteca `strcpy` (definida en el archivo "string.h") como vemos en las siguientes líneas de código:

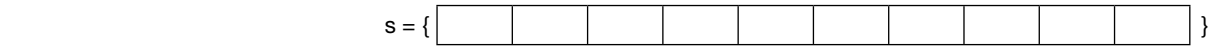
```
// defino un "conjunto" de 10 variables de tipo char
char s[10];

// strcpy asigna cada uno de los caracteres de "Hola"
// a cada una de las variables del conjunto s
strcpy(s,"Hola");
```

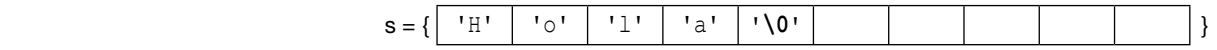
Esta función toma cada uno de los caracteres de la cadena “Hola” y los asigna uno a uno a los elementos del conjunto `s`.

Gráficamente, podemos verlo así:

1. Definimos un *array* de 10 caracteres (o un conjunto de 10 variables de tipo `char`)
   `char s[10];`



2. Asignamos uno a uno los caracteres de la cadena “Hola” a los caracteres de `s`.
   `strcpy(s, "Hola");`



Como vemos `strcpy` asigna el *i-ésimo* carácter de la cadena “Hola” al *i-ésimo* carácter del conjunto o *array* `s`.

Además `strcpy` agrega el carácter especial `'\0'` (léase “barra cero”) que delimita el final de la cadena. Es decir que si bien `s` tiene espacio para almacenar 10 caracteres nosotros solo estamos utilizando 5. Cuatro para la cadena “Hola” más 1 para el `'\0'`.

Vale decir entonces que en un *array* de *n* caracteres podremos almacenar cadenas de, a lo sumo, *n*-1 caracteres ya que siempre se necesitará incluir el carácter de fin de cadena `'\0'` al final.

En otros lenguajes como Java o Pascal, las cadenas de caracteres tienen su propio tipo de datos, pero lamentablemente en C el manejo de cadenas es bastante más complicado. Por este motivo, lo estudiaremos en detalle más adelante, pero considero conveniente mencionar una cosa más: en el Capítulo 1, se expuso el siguiente ejemplo:

```
#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;
}
```

Aquí utilizamos el operador de asignación `=` para asignar el valor “Pablo” a la variable `nombre`. Esto solo se puede hacer al momento de definir la variable. Incluso, como no hemos dimensionado la cantidad de caracteres que el *array* `nombre` puede contener, C dimensionará el conjunto de caracteres con tantos elementos como sea necesario para mantener la cadena “Pablo” más 1 para contener el `'\0'`.

Recomiendo al lector modificar el programa anterior de la siguiente manera:

```
#include <stdio.h>

int main()
{
   char nombre[20];

   nombre = "Pablo";

   // :
   // todo lo demas...
   // :

   return 0;
}
```

Aquí estamos intentando asignar “Pablo” a `nombre`, pero en una línea posterior a la declaración de la variable. Al compilar obtendremos el siguiente error:

```
datospersona.c: In function 'main':
datospersona.c:8: error: incompatible types in assignment
```

2.4 Estructura de repetición

La estructura de repetición, también llamada “estructura iterativa” o “ciclo de repetición”, permite ejecutar una y otra vez un conjunto de acciones en función de que se verifique una determinada condición lógica.

Según sea exacta o inexacta la cantidad de veces que el ciclo iterará podemos clasificar a la estructura de repetición de la siguiente manera:

- Estructura de repetición inexacta que itera entre 0 y *n* veces
- Estructura de repetición inexacta que itera entre 1 y *n* veces
- Estructura de repetición exacta que itera entre *i* y *n* veces, siendo *i* <= *n*

2.4.1 Estructuras de repetición inexactas

Llamamos así a las estructuras que iteran una cantidad variable de veces. En C estas estructuras son el ciclo `while` y el ciclo `do-while` y las representamos así:

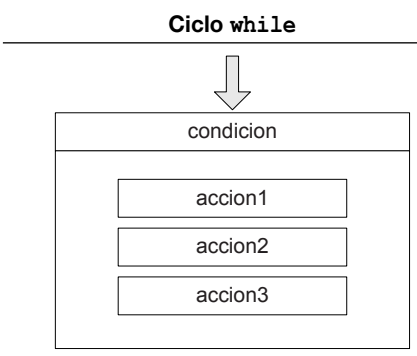


Fig. 2.6 Ciclo `while`.

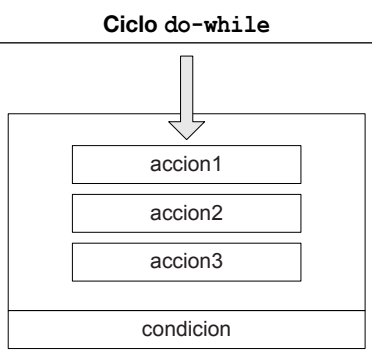


Fig. 2.7 Ciclo `do-while`.

El ciclo `while` itera mientras que se cumpla la condición lógica indicada en su cabecera. Si al llegar a esta estructura la condición resulta ser falsa entonces el ciclo no iterará ni siquiera una vez. Por eso, decimos que se trata de un ciclo de repeticiones inexacto que puede iterar entre 0 y *n* veces. Además, como la condición se evalúa antes de ingresar al ciclo decimos que el `while` es un ciclo con "precondición".

En cambio, la entrada al ciclo `do-while` no está condicionada, por lo tanto, las acciones encerradas dentro de esta estructura se realizarán al menos una vez. Luego de esto se evaluará la condición lógica ubicada al pie de la estructura, que continuará iterando mientras que esta condición resulte verdadera. En este caso, decimos que se trata de un ciclo de repeticiones inexacto que iterará entre 1 y *n* veces. El `do-while` es un ciclo con "poscondición".

Problema 2.4

Se ingresa por teclado un conjunto de valores numéricos enteros positivos, se pide informar, por cada uno, si el valor ingresado es par o impar. Para indicar el final se ingresará un valor cero o negativo.

Análisis

Los datos de entrada de este problema son los números que ingresará el usuario. No sabemos cuántos números va a ingresar, solo sabemos que el ingreso de datos finalizará con la llegada de un valor cero o negativo. Por esto, mientras que el número ingresado sea mayor que cero tenemos que "procesarlo" para indicar si es par o impar.

Para resolver este problema, utilizaremos un ciclo de repetición que iterará mientras que el número ingresado sea mayor que cero. Luego, dentro de la estructura lo procesaremos para determinar e informar si el número es par o impar.

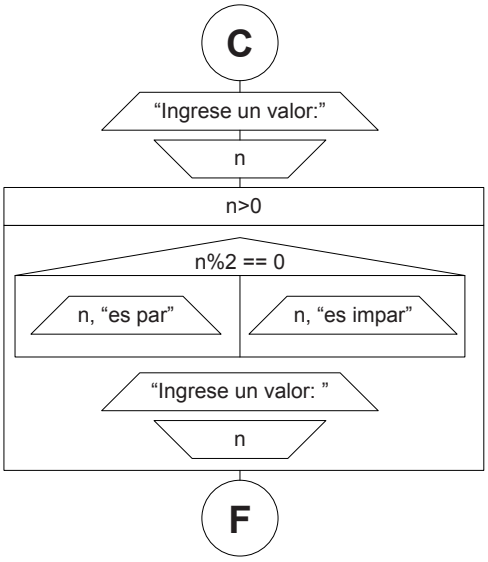


Fig. 2.8 Lee un conjunto de números e indica cuál es par y cuál es impar.

Como comentamos en el capítulo anterior, dentro del ciclo de repeticiones debe suceder "algo" que haga que, en algún momento, la condición lógica se deje de cumplir. En este caso, por cada iteración volvemos a leer en la variable `n` el siguiente número del conjunto. Por lo tanto, cuando el usuario ingrese un valor cero o negativo la condición lógica resultará falsa y el ciclo dejará de iterar.

Notemos que si el conjunto de datos está vacío el usuario solo ingresará un valor cero o negativo con el que la condición del ciclo resultará falsa y, directamente, no ingresará.

Veamos la codificación:

```
#include <stdio.h>

int main()
{
   int n;

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

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

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

   return 0;
}
```

2.4.2 Estructuras de repetición exactas

A diferencia de las estructuras inexactas, las estructuras exactas permiten controlar la cantidad de veces que van a iterar. En general, se llaman "ciclos *for*" y definen una variable de control que toma valores sucesivos comenzando desde un valor inicial v_i y finalizando en un valor final v_n. Así, el `for` itera exactamente $v_n - v_i + 1$ veces y en cada iteración la variable de control tomará los valores v_i, v_i+1, v_i+2, ...,v_n.

En C el ciclo `for` se implementa como una mezcla entre el "*for* tradicional" y el ciclo `while`. Gráficamente, lo representaremos así:

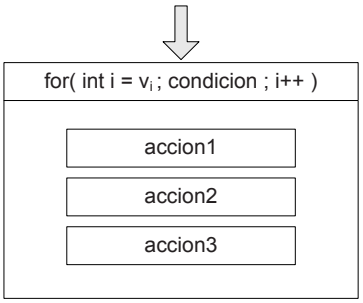


Fig. 2.9 Representación gráfica del ciclo `for`.

La cabecera del ciclo se compone de tres secciones separadas por `;` (punto y coma). En la primera sección, definimos la variable de control como una variable entera (en este caso es la variable `i`) y le asignamos un valor inicial. En la tercera sección, definimos la modalidad de incremento que se le aplicará a la variable de control. Con `i++` le indicamos al `for` que en cada iteración incremente en 1 el valor de `i`.

En la segunda sección, definimos una condición lógica. El for iterará incrementando el valor de la variable `i` de uno en uno y mientras que se verifique dicha condición. Es decir, si queremos que el ciclo itere exactamente *n* veces con una variable `i` variando entre 0 y *n*-1 tendremos que definir la cabecera del for de la siguiente manera:

```
for( int i=0; i<n; i++ )
```

Problema 2.5

Desarrollar un algoritmo que muestre por pantalla los primeros *n* números naturales considerando al 0 (cero) como primer número natural.

Análisis

El único dato de entrada que recibe el algoritmo es el valor *n* que el usuario deberá ingresar por teclado. Luego, la salida será: 0, 1, 2, ..., *n*-2, *n*-1. Es decir, si el usuario ingresa el valor 3 entonces la salida del algoritmo debe ser 0,1,2. Y si el usuario ingresa el valor 5 la salida será 0,1,2,3,4.

Para resolver el algoritmo utilizaremos una variable `i` cuyo valor inicial será 0. Luego mostramos su valor y lo incrementamos para que pase a ser 1. Si repetimos esta operación mientras que `i` sea menor que *n* tendremos resuelto el problema.

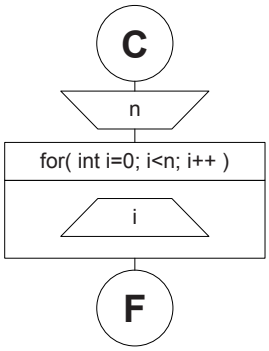


Fig. 2.10 Muestra los primeros *n* números naturales incluyendo el 0.

Para simplificar la lectura de los diagramas, en algunos casos comenzaré a omitir los mensajes con indicaciones para el usuario. Por ejemplo, aquí decidí no incluir el mensaje “Ingrese un valor numérico”. Sin embargo, en la codificación sí los incluiré.

Como vemos, el `for` prácticamente resuelve todo el problema ya este ciclo define la variable `i`, le asigna el valor inicial 0, la incrementa de uno en uno e itera mientras que su valor sea menor que `n`. Dentro del `for` todo lo que queda por hacer es mostrar el valor de la variable `i`.

Veamos la codificación del algoritmo:

```
#include <stdio.h>

int main()
{
   int n;

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

   for(int i=0; i<n; i++)
   {
      printf("%d\n",i);
   }

   return 0;
}
```

Este problema también podemos resolverlo utilizando un ciclo `while` pero, entonces, la responsabilidad de incrementar la variable será nuestra. Para esto, en cada iteración luego de mostrar el valor de `i` incrementaremos su valor asignándole su valor actual "más 1". En este caso, decimos que `i` es un contador.

2.4.3 Contadores

Llamamos "contador" a una variable cuyo valor iremos incrementando manualmente, de uno en uno, dentro de un ciclo de repetición. Para incrementar el valor de una variable numérica, tenemos que asignarle "su valor actual más 1" de la siguiente manera:

```
// incrementamos el valor de la variable x
x = x+1;
```

La línea anterior debe leerse así: "a `x` le asigno `x` más 1". Si el valor actual de `x` es 2, luego de incrementarlo su valor será 3. Los contadores siempre deben tener un valor inicial.

Resolveremos el problema anterior reemplazando el ciclo `for` por un ciclo `while` e incrementando manualmente un contador.

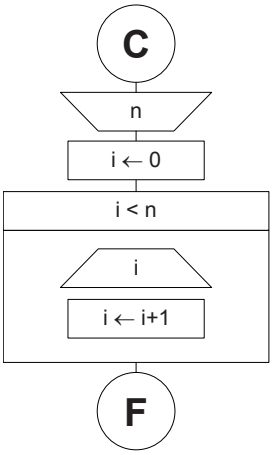


Fig. 2.11 Muestra los primeros *n* números naturales utilizando un contador.

Es muy importante asignarle un valor inicial a la variable que implementa el contador.

En el diagrama le asignamos el valor inicial 0 a la variable `i`. De no haberlo hecho entonces la condición del `while` no tendría sentido. Luego, al mostrar `i` estaríamos mostrando algo incierto. Tampoco podríamos asignarle a `i` "su valor actual más 1" porque `i` no tendría ningún valor actual.

Llamamos "inicializar" a la acción de asignarle valor inicial a una variable. Una variable que no está inicializada no tiene asignado ningún valor concreto. Es decir, no tiene valor.

Volviendo al diagrama inicializamos la variable `i`, pero no inicializamos la variable `n`. Esto se debe a que `n` tomará su valor inicial luego de que el usuario lo ingrese por teclado.

Veamos la codificación:

```
#include <stdio.h>

int main()
{
   int i, n;

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

   i = 0;
   while( i<n )
   {
      printf("%d\n",i);
      i = i+1;
   }

   return 0;
}
```

2.4.4 Acumuladores

Llamamos así a una variable cuyo valor iremos incrementando en cantidades variables dentro de un ciclo de repetición. Esto lo logramos de la siguiente manera:

```
x = x+n;
```

La diferencia entre un acumulador y un contador es que el contador se incrementa de a 1 en cambio el acumulador se incrementa de a *n* siendo *n* una variable. Obviamente, un contador es un caso particular de acumulador.

Problema 2.6

Determinar la sumatoria de los elementos de un conjunto de valores numéricos. Los números se ingresarán por teclado. Se ingresará un cero para finalizar.

Análisis

Para resolver este problema, utilizaremos un ciclo `while` dentro del cual acumularemos en la variable `suma` cada uno de los valores que ingrese el usuario. El ciclo de repetición iterará mientras que el valor ingresado sea distinto de cero.

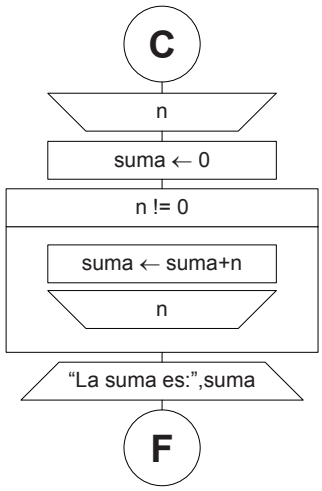


Fig. 2.12 Suma un conjunto de valores numéricos.

La variable `suma` es un acumulador porque dentro del ciclo `while` su valor se incrementa de a `n` unidades siendo `n` un valor potencialmente distinto en cada iteración ya que toma cada uno de los valores que ingresa el usuario.

2.4.5 Seguimiento del algoritmo y “prueba de escritorio”

La prueba de escritorio es una herramienta que nos permite comprobar si el algoritmo realmente funciona como esperamos. Básicamente, consiste en definir un conjunto de datos arbitrarios que llamaremos “lote de datos” y utilizarlos en un seguimiento, paso a paso, de cada una de las acciones del algoritmo controlando los valores que irán tomando las variables, las expresiones lógicas y la salida que se va a emitir.

Analizaremos una prueba de escritorio para el diagrama del problema anterior, considerando el siguiente lote de datos {5, 2, 3, 0}.

Comenzamos leyendo `n` que, según nuestro lote de datos, tomará el valor 5. Inicializamos `suma` en 0 y luego entramos a un ciclo de repetición para iterar mientras que `n` sea distinto de 0 o, dicho de otro modo, mientras que la expresión “`n` es distinto de cero” sea verdadera. Como `n` vale 5 se verifica la condición lógica e ingresamos al `while`. Dentro del `while` asignamos a `suma` su valor actual (cero) más el valor de `n` (cinco) por lo que ahora `suma` vale 5. Luego volvemos a leer `n` que tomará el valor 2 y volvemos a la cabecera del ciclo de repetición para evaluar si corresponde iterar una vez más. Como `n` (que vale 2) es distinto de cero volvemos a ingresar al `while`, asignamos a `suma` su valor actual (cinco) más el valor de `n` (dos) por lo que `suma` ahora vale 7. Luego leemos `n` que tomará el valor 3. La condición del ciclo se sigue verificando porque `n` (que vale 3) es distinto de cero. Entonces asignamos a `suma` su valor actual (que es 7) más el valor de `n` (que es 3). Ahora `suma` vale 10. Volvemos a leer `n` que tomará el valor cero y evaluamos la condición del ciclo. Como la expresión “`n` es distinto de cero” resulta falsa el ciclo no volverá a iterar, salimos del `while` y mostramos el valor de la variable `suma`: 10.

Todo este análisis puede resumirse en la siguiente tabla que iremos llenando paso a paso, siguiendo el algoritmo y considerando que ingresan uno a uno los valores del lote de datos.

n	suma					
5	0	n!=0	Acción	suma = suma+n	n	Salida
		true	entro al while	5	2	
		true	entro al while	7	3	
		true	entro al while	10	0	
		false	salgo del while			La suma es: 10

La tabla debe leerse de arriba hacia abajo y de izquierda a derecha. El lector puede acceder al videotutorial que muestra el desarrollo de esta misma prueba de escritorio.

Por último, veamos la codificación del algoritmo:

```
#include <stdio.h>

int main()
{
   int n,suma;

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

   suma=0;
   while( n!=0 )
   {
      suma = suma+n;

      printf("Ingrese el siguiente valor: ");
      scanf("%d",&n);
   }

   printf("La suma es: %d\n",suma);

   return 0;
}
```

Uso del *debugger* para depurar un programa

2.4.6 El debugger, la herramienta de depuración

El *debugger* es una herramienta que permite seguir paso a paso la ejecución del código fuente del programa. También permite monitorear los valores que toman las variables y evaluar las expresiones.

Las IDE integran el *debugger* con el editor de código fuente de forma tal que, dentro del mismo editor, el programador pueda seguir línea por línea la ejecución del programa y así, depurarlo de errores de lógica.

En los videos tutoriales se explica como *debuggear* (depurar) un programa utilizando el *debugger* de Eclipse.

Continuemos con más ejemplos.

Problema 2.7

Dado un conjunto de valores numéricos que se ingresan por teclado determinar el valor promedio. El fin de datos se indicará ingresando un valor igual a cero.

Análisis

Sabemos que para obtener el promedio de un conjunto de números tenemos que sumarlos y luego dividir la suma por la cantidad de elementos del conjunto. Por esto necesitaremos un acumulador para obtener la suma (lo llamaremos `suma`) y un contador para saber cuántos números fueron sumados (lo llamaremos `cant`). El resultado estará disponible luego de que hayamos procesado todos los elementos del conjunto. Esto es: fuera del ciclo de repetición.

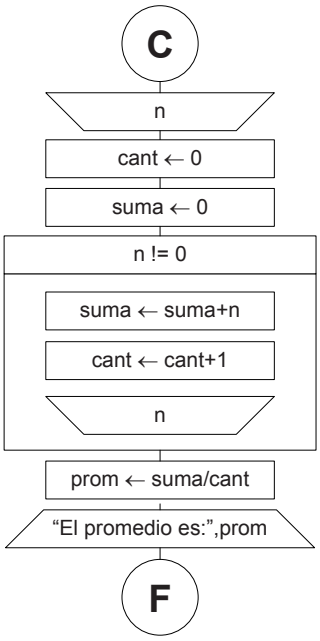


Fig. 2.13 Obtiene el promedio de un conjunto de valores numéricos.

Veamos la codificación:

```
#include <stdio.h>

int main()
{
   int n,cant,suma;
   double prom;

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

   cant=0;
   suma=0;
```

```
   while( n!=0 )
   {
      suma = suma+n;
      cant = cant+1;

      printf("Ingrese el siguiente valor: ");
      scanf("%d",&n);
   }

   prom = (double)suma/cant;

   printf("El promedio es: %lf\n",prom);

   return 0;
}
```

Problema 2.8

Se ingresa un valor numérico por consola, determinar e informar si se trata de un número primo o no.

Análisis

Los números primos son aquellos que solo son divisibles por sí mismos y por la unidad. Es decir que un número `n` es primo si para todo entero `i` tal que `i>1 && i<n` se verifica que `n%i` es distinto de cero.

La estrategia que utilizaremos para resolver este problema será la siguiente: consideraremos que el número `n` que ingresa el usuario es primo y luego vamos a ver si entre 2 y `n-1` existe algún número `i` tal que `i` sea divisor de `n`. Si existe algún valor de `i` que haga que `n%i` sea cero será porque `i` es divisor de `n` y, por lo tanto, `n` no será primo.

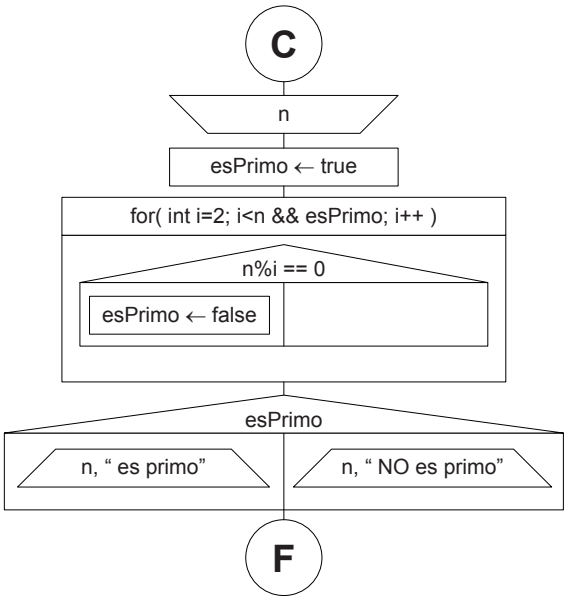


Fig. 2.14 Determina si un número es primo o no.

```
#include <stdio.h>

int main()
{
   int n,esPrimo;

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

   esPrimo=1;

   for( int i=2; i<n && esPrimo; i++ )
   {
      if( n%i==0 )
      {
         esPrimo=0;
      }
   }

   if( esPrimo )
   {
      printf("%d es primo\n",n);
   }
   else
   {
      printf("%d NO es primo\n",n);
   }

   return 0;
}
```

2.4.7 Estructuras de repetición anidadas

Cuando una estructura de repetición encierra a otra estructura de repetición decimos que son estructuras anidadas.

Problema 2.9

Desarrollar un algoritmo que muestre los primeros *n* números primos siendo *n* un valor que debe ingresar el usuario.

Análisis

En este algoritmo el usuario ingresa la cantidad `n` de números primos que quiere ver. Por ejemplo, si `n` es 5 entonces el programa debe mostrar los primeros cinco números primos, es decir, 1, 2, 3, 5, 7.

La estrategia aquí será mantener dos contadores que llamaremos `num` y `cont`. A `num` lo vamos a inicializar en 1 y lo utilizaremos para evaluar si su valor actual es primo, luego lo incrementaremos. Cada vez que determinemos que `num` es primo tenemos que mostrarlo e incrementar a `cont`. Por lo tanto, `cont` contará la cantidad de números primos que hemos mostrado hasta el momento.

Para implementar esta estrategia, utilizaremos dos ciclos de repetición anidados. El ciclo interno iterará entre 2 y `num-1` para determinar si el valor de `num` es primo o no utilizando el mismo algoritmo del problema anterior. El ciclo externo iterará mientras que `cont` sea menor que `n`. Este ciclo controlará que mostremos exactamente los primeros `n` números primos.

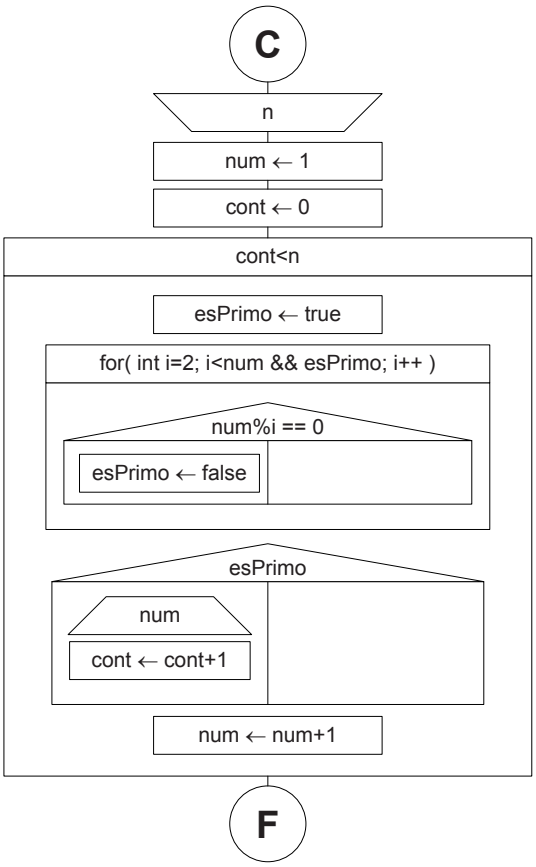


Fig. 2.15 Muestra los primeros n números primos.

Veamos la codificación:

```
#include <stdio.h>

int main()
{
   int n,esPrimo;
   int num,cont;

   printf("Cuantos primos quiere ver? ");
   scanf("%d",&n);


   num=1;
   cont=0;

   while( cont<n )
   {
      esPrimo=1;
```

```
      for( int i=2; i<num && esPrimo; i++ )
      {
         if( num%i==0 )
         {
            esPrimo=0;
         }
      }

      if( esPrimo )
      {
         printf("%d\n",num);
         cont=cont+1;
      }

      num = num+1;
   }

   return 0;
}
```

2.4.8 Manejo de valores booleanos

En el Capítulo 1, hablamos sobre los tipos de datos y comentamos que existe un tipo de datos capaz de contener los valores lógicos “verdadero” y “falso” o, en inglés, *true* y *false*. A este tipo de datos lo llamamos “tipo booleano” o simplemente *boolean*.

En C no existe el tipo *boolean* como tal. En su lugar se usan las variables de tipo `int` y se considera que su valor de verdad es *false* si contienen 0 y *true* si contienen cualquier otro valor numérico.

Esto nos permite asignar en una variable de tipo `int` el resultado de una expresión lógica como veremos a continuación:

```
#include <stdio.h>

int main()
{
   int n, esPar;

   scanf("%d",&n);

   // asigno a esPar el resultado de la expresion n%2==0
   esPar = n%2==0;

   // si esPar es verdadero...
   if( esPar )
   {
      printf("%d es par\n",n);
   }

   return 0;
}
```

En este ejemplo asignamos a la variable `esPar` el resultado de la expresión lógica `n%2==0`. Luego preguntamos directamente por el valor de verdad de `esPar`.

Quizás pueda resultar confuso que un mismo tipo de datos se utilice para contener valores de naturaleza diferente. La siguiente tabla muestra como a una variable de tipo `int` se la puede tratar como tipo *boolean* y como tipo entero.

Tratamiento booleano	Tratamiento numérico

`// :`
`if( esPar )`
`{`
`  printf("%d es par\n",n);`
`}`
	`// :`
	`if( esPar!=0 )`
	`{`
	`  printf("%d es par\n",n);`
	`}`

Ambos tratamientos son correctos. Sin embargo, el más apropiado es el que se muestra en el cuadro de la izquierda porque estamos haciendo referencia al valor lógico de la variable `esPar`, no a su valor numérico.

2.4.9 Máximos y mínimos

En ocasiones necesitaremos encontrar el máximo o el mínimo valor dentro de un conjunto de valores numéricos. Los siguientes problemas nos ayudarán a estudiar este tema.

Problema 2.10

Dado un conjunto de valores numéricos indicar cuál es el mayor. El ingreso de datos finaliza con la llegada de un cero.

Análisis

Tenemos que leer los valores que el usuario ingresará por teclado y determinar cuál de estos es el mayor. Para encontrar un algoritmo que nos permita resolver el problema, primero pensemos como lo haríamos mentalmente.

Supongamos que trabajamos con el siguiente lote de datos {2, 6, 1, 7, 3, 0} y solo podemos acceder a un valor a la vez. Entonces leeremos un valor y lo consideraremos como "el mayor" hasta tanto no leamos otro que lo supere.

Acción	Valor que ingresa	Mayor hasta el momento
leo un valor	2	2
leo el siguiente valor	6	6
leo el siguiente valor	1	6
leo el siguiente valor	7	7
leo el siguiente valor	3	7
leo el siguiente valor	0	**7**

Como vemos, al finalizar el ingreso de datos podemos determinar que el mayor valor del conjunto, según nuestro lote, es el número 7.

Este algoritmo lo representamos de la siguiente manera:

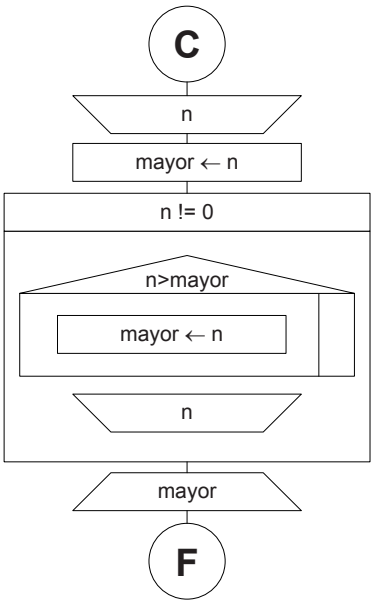


Fig. 2.16 Obtiene el mayor valor de un conjunto.

Veamos la codificación:

```
#include <stdio.h>

int main()
{
   int n,mayor;

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

   mayor = n;

   while( n!=0 )
   {
      if( n>mayor )
      {
         mayor = n;
      }

      printf("Ingrese el siguiente valor: ");
      scanf("%d",&n);
   }

   printf("EL mayor es: %d\n", mayor);

   return 0;
}
```

Problema 2.11

Determinar el menor valor de un conjunto de números e indicar también su posición relativa dentro del mismo. El ingreso de datos finaliza con la llegada de un cero.

Análisis

El algoritmo para obtener el menor valor de un conjunto es análogo al que utilizamos para obtener el mayor. Leeremos un valor y este será el menor mientras no leamos otro que resulte ser más pequeño.

En este problema, además tenemos que indicar la posición relativa. Para analizar esto utilizaremos el mismo lote de datos del problema anterior: {2, 6, 1, 7, 3, 0}.

Según el lote de datos, el menor valor del conjunto es 1 y su posición relativa es 3 ya que se encuentra ubicado en el tercer lugar. Recordemos que el cero no debe ser tenido en cuenta porque su función es indicar el fin del ingreso de datos.

Para identificar la posición relativa del menor valor del conjunto, tendremos que usar un contador y guardar su valor actual cada vez que encontremos un número más chico que el que, hasta el momento, consideramos como el menor.

Acción	**Valor que ingresa**	**posRel**	**menor**	**posRelMenor**
Leo un valor	2	1	2	1
Leo el siguiente	6	2	2	1
Leo el siguiente	1	3	1	3
Leo el siguiente	7	4	1	3
Leo el siguiente	3	5	1	3
Leo el siguiente	0		**1**	**3**

El algoritmo es el siguiente:

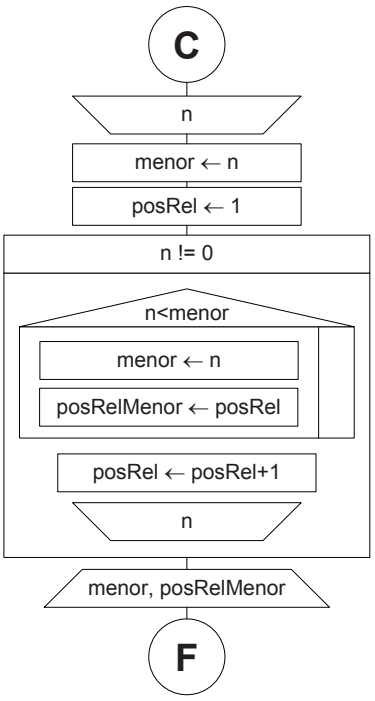


Fig. 2.17 Dado un conjunto de valores obtiene el menor y su posición relativa.

```
#include <stdio.h>

int main()
{
   int n,menor;
   int posRel,posRelMenor;

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

   menor = n;
   posRel = 1;

   while( n!=0 )
   {
      if( n<menor )
      {
         menor = n;
         posRelMenor = posRel;
      }


      posRel = posRel+1;

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

   printf("Menor valor: %d, posicion relativa: %d\n"
         ,menor
         ,posRelMenor);

   return 0;
}
```

2.5 Contextualización del problema

Hasta aquí analizamos problemas que simplemente hablaban de conjuntos de valores numéricos. El objetivo ahora es trabajar con enunciados que hagan referencia a contextos reales como pueden ser sueldos, personas, calificaciones en exámenes, fechas, etcétera.

Problema 2.12

Se tiene una tabla o planilla con los resultados de la última llamada a examen de una materia, con la siguiente información:

- matrícula (valor numérico entero de 8 dígitos)
- nota (valor numérico entero de 2 dígitos entre 1 y 10)
- nombre (valor alfanumérico de 10 caracteres)