Programas = Datos + Algoritmos

Por Mario Macías

Observe el siguiente programa en C. No se preocupe si no puede comprenderlo todo, al acabar de leer este capítulo seguro que sí podrá:

```
1.    #include <stdio.h>
2.
3.    int main(void)
4.    {
5.          int a,b,c,i;
6.          char nombre[15];
7.
8.          printf("Introduce tu nombre: ");
9.          scanf("%s",nombre);
10.         printf("Introduce el valor de A: ");
11.         scanf("%d",&a);
12.         printf("Introduce el valor de B: ");
13.         scanf("%d",&b);
14.         printf("Hola %s, los resultados son\n",nombre);
15.         printf("A + B = %d\n", a+b );
16.         printf("A - B = %d\n", a-b );
17.         printf("A * B = %d\n", a*b );
18.         printf("A / B = %d\n", a/b );
19.
20.         /* Calculo de la potencia */
21.         c=a;
22.         for(i=0 ; i<b ; i++)
23.         {
24.               c=c*a;
25.         }
26.         printf("A ^ B = %d", c );
27.         return 0;
28.   }
```

Este programa daría una salida en pantalla como la siguiente:

```
> Introduce tu nombre: Pedro
> Introduce el valor de A: 34
> Introduce el valor de B: 2
> Hola Pedro, las operaciones entre A y B son:
> A + B = 36
> A -  B = 32
> A *  B = 68
> A /  B = 17
```

```
> A ^ B = 1156
```

Observar que en la línea 20 hay un texto que no indica nada al programa. Este es un texto que pone el programador en cualquier parte del programa para hacer un **comentario**. Para crear un comentario hay que escribir un texto cualquiera entre los símbolos /* y */, y el compilador ignorará ese texto. Es bueno poner muchos comentarios en los programas, para facilitar la lectura y saber con sólo leer el comentario qué realiza la parte comentada.

Podemos observar que este programa, como cualquier otro, consta de dos partes diferenciadas:

- **Estructuras de datos**: Son aquellos elementos que utilizamos para guardar datos en memoria. En el programa de ejemplo son las variables a,b,c e i, que las utilizamos para guardar números enteros, y la variable nombre, que la guardamos como una cadena de texto, para guardar nombres, frases, etc...
- **Algoritmos**: Son aquellos elementos utilizados para describir el flujo del programa; es decir, los comandos que describen cómo debe ejecutarse el programa paso a paso. En el programa de ejemplo es todo el texto que va desde el primer símbolo '{' hasta el último '}'.

Los algoritmos utilizan estructuras de datos para describir programas. A continuación pasamos a describir los diferentes tipos de datos y las estructuras de control de flujo básicas para definir algoritmos.

Tipos de datos

Hay dos subgrupos dentro de los tipos de datos: los datos simples y los datos compuestos, formados como agrupaciones de datos simples. Al principio puede parecer un poco complicado elegir el tipo de dato más adecuado para nuestro programa, pero a medida que vaya cogiendo experiencia como programador se dará cuenta que no es una tarea complicada, e incluso agradecerá a menudo poder definir sus propios tipos de datos.

Los tipos de datos definen el formato y el tamaño que tienen las **variables** a que acompañan. Una **variable** es un espacio de memoria en el que guardaremos los datos.

Para comprenderlo mejor, una variable es como una caja en la que guardar una sola cosa que vayamos a utilizar. Hay cajas de todos los tamaños y formatos, dependiendo de lo que se va a guardar en ellas. Por ejemplo, un televisor sólo se podrá guardar en cajas grandes, ya que en cajas pequeñas no cabe. En cambio, un anillo se puede guardar en una caja muy grande, pero sería un desperdicio de espacio, ya que en una caja pequeña cabe perfectamente. De la misma manera, si se quiere guardar un número que sabemos que será entre 0 y 100, sería mejor coger el tipo de variable más pequeña que tiene el lenguaje C: un *char*, que guarda números de -128 a 127. Eso no quiere decir que no pudiéramos guardarlo en un tipo de dato grande, por ejemplo un *double*, que permite cientos de miles de valores distintos; pero estaríamos desperdiciando espacio, ya que un *double* ocupa 8 veces más que un *char*.

Se podría comparar la memoria del ordenador con el espacio de un armario: cuantas más cajas metamos menos espacio libre iremos teniendo, por lo que hay que meter las cosas en cajas lo más pequeñas posibles para no desperdiciar el espacio.

Tipos de datos simples

Para utilizar tipos de datos simples debe escribir uno de los tipos de datos descritos a continuación y a continuación el nombre de la variable a que acompañan. También se puede escribir el tipo de dato y varias variables, separadas entre comillas.

- **char**: Este dato ocupa un *byte* de memoria, es decir, puede albergar 256 valores distintos: los números que van de -128 a 127 o un carácter. Por ejemplo, decir que un carácter vale 'A' es lo mismo que decir que vale 65, ya que 'A' tiene valor 65 en el código ASCII (ver apéndice).
- **unsigned char**. Igual que *char*, pero éste guarda 256 números sin signo, es decir, de 0 a 255.
- **int**: puede contener un numero entero. Dependiendo del lenguaje y el compilador que se utilice el tamaño y el número de valores posible variará. Suele ocupar 4 bytes (el mismo tamaño que 4 *char*). El rango de números irá de -2.147.483.648 hasta 2.147.483.647.
- **unsigned int**: Igual que *int*, pero los números serán sin signo. Por lo que el rango será de 0 hasta 4.294.967.295.
- **short int**: es un entero corto que sólo ocupa 2 bytes, por lo que el rango va de -32.768 a 32.767. Si le añadimos al principio **unsigned** el rango irá de 0 a 65.535.
- **long**: para la mayoría de los compiladores es sinónimo de *int*. Puede llevar delante el **unsigned**.
- **float**: número en coma flotante que ocupan 4 bytes. Permite representar números con decimales, obteniendo rangos mucho mayores que *int*. El rango va aproximadamente de -3.4×10^{38} hasta 3.4×10^{38}.
- **double**: número en coma flotante de 8 bytes. Como *float* pero el rango va de -1.7×10^{308} hasta 1.7×10^{308}.
- **long double:** número en coma flotante de 10 bytes. El rango va de -3.4×10^{4932} hasta 3.4×10^{4932}.

Tipos de datos complejos

- **Vector**: un vector es una agrupación de un numero determinado de datos de un mismo tipo. Su definición es la siguiente:

  <tipo de variable> <nombre de variable>[<número de elementos>]

  Ejemplos:

  ```
  int a[15];
  ```

  define un vector de 15 enteros llamado *a*.

  ```
  char matrix[3][4];
  ```

  define una matriz de *char*, de tamaño 4x3 (12 elementos), llamada *matrix*.

  Hemos explicado cómo definir un vector. Ahora nos hace falta saber cómo acceder y modificar los elementos de éste: añadiremos al nombre del vector la posición entre los símbolos '[' y ']'

  Ejemplos:

  ```
  a=mtx[1][2]
  ```

  asigna a la variable *'a*' el valor de la posición (1,2) de la matriz '*mtx*'.

```
vct[34]=0
```

asigna a la posición 34 del vector '*vct*' el valor 0.

Claro está que si queremos acceder a la posición 34 del vector, este vector tiene que tener al menos 35 elementos (del 0 al 34), ya que si no estaríamos accediendo a una zona de memoria que no hemos definido. Si definimos el vector:

```
int a[15];
```

Podremos acceder a las posiciones del vector a[0] hasta a[14];

Otro utilización esencial de los vectores es utilizar vectores de caracteres para guardar cadenas de texto. Por ejemplo, en la definición de la cadena

```
char nombre[]="Jose";
```

estamos indicando que la cadena es un vector de tantos *char* como letras hay entre las comillas (en este caso 4). Donde los valores del vector son nombre[0]='J', nombre[1]='o', nombre[2]='s', nombre[3]='e'.

Mientras los caracteres se definen como un número entero de 1 byte o como un carácter entre comillas simples. Una cadena de texto se define como un conjunto de caracteres entre comillas dobles. Por tanto **no hay que confundir**

```
char letra[]="A";
```

con

```
char letra='A';
```

ya que el primero es un vector de un sólo elemento y el segundo es un carácter que contiene la letra A mayúscula. En cambio, sí es lo mismo escribir

```
char letra='A';
```

que

```
char letra=65;
```

- **Estructura**: es una agrupación de datos de distintos tipos definidas por el programador, con el fin de crear sus propios tipos de datos. Por ejemplo, imagínese que va a hacer un programa para una tienda en el que guarde información de diferentes productos, tales como el nombre, el código de referencia, el precio, etc... Podría usted crear un entero para cada precio, una cadena para cada nombre de producto... o puede crear una estructura en la que englobe todos los datos necesarios para cada producto, como por ejemplo:

```
struct producto {
     char nombre[25];
     int precio;
     char referencia[10];
}
```

Una vez definida la estructura hemos definido el tipo de dato, no el dato en sí. Para utilizar ahora un dato de el tipo producto, lo definimos como cualquier otro dato simple, pero indicando primero que se trata de una estructura, luego el nombre de la estructura definida y luego el nombre de la variable. Por ejemplo

```
struct producto prod1;
```

definirá la variable prod1 del tipo producto.

Cuando queramos acceder a un elemento de la estructura lo haremos de la siguiente manera:

<nombre variable>.<nombre miembro>

```
prod1.precio=234;
```

asigna al precio de la variable prod1 el valor 234;

Otros tipos de datos

- **Enumerados**: no siempre será necesario trabajar con datos que representen números o cadenas de caracteres. Pongamos que quiere representar datos que simbolicen el día de la semana; se podrá representar como datos numéricos (días 1,2,3,4,5,6, y 7), aunque es mucho más cómodo representar directamente los días (lunes, martes...). Para esta función fueron creados el tipo de datos enumerados. Su sintaxis es la siguiente:

  enum <tipo de dato> {

  <lista de valores separados por coma>

  };

  Ejemplos:

  ```
  enum mes { enero, febrero, marzo, abril, mayo, junio, julio, agosto, septiembre, octubre, noviembre, diciembre };
  enum palo_baraja { bastos, copas, oros, espadas };
  ```

  Las ventajas de definir datos enumerados, aparte de la comodidad de no tener que representar con números objetos que no lo son, es que evitamos que se asignen valores que no están dentro del grupo definido (por ejemplo, no podremos asignar a una variable del tipo mes el valor 'Treciembre', mientras que si trabajásemos con números sí podríamos asignar el valor 13 a un mes, cosa que en la realidad es errónea).

  Podemos combinar también los tipos de datos complejos, de manera que podremos incluir vectores y enumerados dentro de nuestras estructuras, o crear vectores de estructuras y enumerados. Mire el siguiente ejemplo:

```
enum genero { hombre,mujer };
struct persona {
      char nombre[25];
      char telefono[10];
      unsigned char edad; /* la edad será entre 0 y 255 años */
      enum genero sexo;
      /* struct producto ya lo definimos antes */
      struct producto compras[10];
}
```

Supongamos que queremos crear una pequeño programa que guarde información de los clientes de una tienda. Una buena opción es crear primero una estructura con la información deseada de los clientes. Para definir el sexo se podría haber hecho con un número de manera que (0=> hombre, 1=> mujer) con lo que es menos visible a la hora de programar; además podríamos caer en el error de asignar otro número y el sexo quedaría indeterminado. Por eso es bueno definir 'genero' como un tipo de datos

enumerado. Si quisiéramos guardar también la información de los últimos 10 productos comprados, añadimos como campo de la estructura un vector de 10 productos.

Algoritmos

Una vez conocemos los tipos de datos básicos que van a contener nuestros programas, necesitamos conocer las órdenes que daremos al computador para utilizarlos (a partir de ahora 'instrucciones'). De esto se encarga la algorítmica. El orden en que se ejecutan las líneas escritas en el programa (de ahora en adelante le llamaremos 'flujo del programa') va desde la primera línea a la última y, en caso de colocar dos instrucciones en una misma línea, de izquierda a derecha.

Aunque este es el orden básico, muy a menudo necesitaremos volver a ejecutar instrucciones que ya se ejecutaron, o repetir muchas veces una misma instrucción. También a veces querremos que una orden se ejecute algunas veces y saltárnosla otras veces. Para ello están las 'estructuras de control de flujo'. Otras veces querremos ejecutar unas mismas líneas de código desde diferentes puntos del programa, y que el código ejecutado tenga diferentes efectos según el valor que tengan algunas determinadas variables; para ello están las 'funciones' y los 'procedimientos'.

Estructuras de control

- **if** (<condición>)

  {

  <instrucciones a ejecutar si se cumple la condición>

  [} **else** {

  <instrucciones a ejecutar si no se cumple la condición>]

  }

  Si se cumple la condición, se ejecutan ciertas instrucciones. Opcionalmente, se pueden escribir instrucciones alternativas, que se ejecutarán en caso que la condición no se cumpla. Ejemplo:

```
if (a<0)
{
     printf("El numero es negativo");
} else {
     printf("El numero es positivo");
}
```

  Fijémonos que si un numero es menor que 0 (negativo), sólo se ejecutará la instrucción entre el **if** y el **else**. Si el número no es negativo (positivo) entonces sólo se ejecutará la instrucción entre el **else** y el último signo '**}**'.

```
if (a<0)
{
     a=-a
```

}

Este código sirve para obtener siempre el valor positivo de *a* (valor absoluto). Si *a* es negativa, hay que cambiarle el signo. En cambio, si es positiva, se saltará el cambio de signo y no se ejecutará ninguna otra operación alternativa.

- **for(**<variable>=<valor inicial>;condición de permanencia>;<incremento>)

  {

  <instrucciones a ejecutar>

  }

  Realmente esta estructura permite muchas más formas de usarse, pero como son formas no necesarias y algo complicadas de leer, trataremos el **for** de la manera simple y clásica: se repite el conjunto de instrucciones que contiene incrementando el valor de una variable desde su valor inicial hasta el valor final indicado. Los intervalos desde el valor inicial hasta el final pueden ser saltos de uno en uno, o de varios números en varios números.

  Ejemplos:

```
for(a=0;a<32;a=a+1)
{
     vector[a]=0;
}
```

  Esta secuencia inicializa a 0 los elementos de un vector que van del 0 al 31 (observar que la condición de permanencia es a<32, a menor que 32).

```
for(a=0;a<=32;a=a+1)
{
     vector[a]=0;
}
```

  Esta secuencia inicializa a 0 los elementos de un vector que van del 0 al 32 (observar que la condición de permanencia es a<=32, a menor o igual que 32).

```
for(a:=0;a<32;a=a+3)
{
     vector[a]:=0;
}
```

  Esta secuencia inicializa a 0 los 32 primeros elementos múltiplos de 3 de un vector; es decir, los elementos {0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30}. Esto es así por que el incremento no es de 1 en 1 sino de 3 en 3.

- **while(**<condición>)

  {

<instrucciones a ejecutar>
}

Ejecuta las instrucciones indicadas mientras se cumpla la condición indicada. Ejemplo:

```
i=0
while(vector[i]==0)
{
      i=i+1;
}
```

Esta secuencia de instrucciones recorre un vector de 32 posiciones, hasta que se encuentra un valor diferente de 0 como uno de sus valores.

- **do {**
  <instrucciones a ejecutar>
  } while (<condición>**)**

Es similar a la estructura anterior, pero en vez de comprobar si se cumple la condición al inicio de todo, lo hace al final. La diferencia básica es que si en la estructura anterior, al llegar al 'while' la condición no se cumple, las instrucciones de la estructura de control no se ejecutarán nunca. En este while, al menos se ejecutarán una vez.

Ejemplo:

```
a=3;
do {
      a=a+1;
} while (a==7);  /* Mientras a sea igual a 7 */
```

Como vemos, a=a+1 se ejecutará una sola vez, ya que al no cumplirse la comprobación "a es igual a 7", se continuará ejecutando el programa por la siguiente instrucción.

- **switch** (<variable>)
  {
  case <opción 1>:
  <instrucciones si se cumple la opción 1>
  break;
  case <opción 2>:
  <instrucciones si se cumple la opción 2>
  break;

```
        ...
        ...
        case <opción n>:
                <instrucciones si se cumple la opción n>
                break;
        default:
                <instrucciones si no se cumple ninguna opción anterior>
}
```

Según el valor de la variable indicada, ejecutará la secuencia de instrucciones que corresponda al valor. Si no coincide ninguna, se ejecutará la secuencia indicada en **default**. Ejemplo:

```
switch(dia)
{
     case lunes:
          printf("Esta tarde clases de danza");
          break;
     case martes:
          printf("Esta tarde clases de pintura");
          break;
     case viernes:
          printf("Esta tarde clases de solfeo");
          break;
     default:
          printf("Esta tarde no hay clases");
}
```

Esta secuencia selecciona el día de la semana que se introduce y mira si para esa tarde hay programada alguna actividad. Para los días en los que no hay nada programado se escribe el mensaje por defecto.

Funciones

A menudo es necesario ejecutar desde diferentes partes del programa unas mismas líneas de código, pero en las que los datos pueden tomar valores diferentes. Para no tener que escribir varias veces el mismo código están las funciones. Por ejemplo, imagine que necesita calcular varias veces qué numero de entre tres es el mayor. Puede repetir varias veces el mismo código en todos los puntos del programa, o puede llamar a una función a la cual sólo hay que pasarle los 3 números y esta se encargará de devolvernos el mayor:

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

```
/* a continuación viene la función
   que retorna el máximo de 3 números */
int max(int a, int b, int c)
{
    int max;  /*variable de visibilidad local*/

    if(a<b)
    {
        max=b;
    } else {
        max=a;
    }

    if(max<c)
    {
        max=c;
    }
}

int main(void)
{
    int n1=4567;
    int n2=6221;
    int n3=-5434;

    printf("El numero mayor de %d, %d y %d es %d.\n",
                                    n1,n2,n3,max(n1,n2,n3));

    return 0;
}
```

Aquí se llama desde el programa principal a una función llamada *max*, a la cual se le indican los tres números a evaluar como **parámetros**, y retorna el valor del número máximo.

La definición de una función es la siguiente:

<tipo de retorno> <nombre>(<lista de parámetros> ó <void>)

{

<cuerpo de la funcion>

return <valor de retorno>

}

Observar que la función ha de retornar un valor, el cual se especifica en la cabecera. Si no queremos que retorne ningún valor, y sólo nos interesa que realice determinadas instrucciones, debemos especificar como tipo de retorno **void**. De la

misma manera, si queremos especificar parámetros, hay que enumerarlos entre paréntesis después del nombre de la función, con el formato <tipo de parámetro> <nombre del parámetro>, cada uno de los cuales va separado del resto por una coma. Si no se quieren especificar parámetros habrá que indicarlo entre los paréntesis con **void**.

Cuando se llama a una función con parámetros, estos pueden ser tanto variables como constantes. Es decir, podemos indicar un valor directamente, o podemos indicar la variable que contiene dicho valor.

A continuación vienen unos ejemplos del uso de funciones:

- Función que saca por pantalla un mensaje de error. No hace falta pasarle ningún parámetro ni que retorne ningún valor.

```
void error(void)
{
      printf("Ha ocurrido un error!\n");
}
```

- Función que saca por pantalla un error especificado por el programador. No será necesario que retorne nada, pero sí hay que pasarle un número de error por parámetro.

```
void error(int errno)
{
      printf("Ha sucedido el error numero %d.\n",errno);
}
```

- Función que eleva al cuadrado un número entero. Se necesita el número a elevar como parámetro y debe retornar el resultado.

```
int eleva(int num)
{
      return num*num;  /*multiplica num por sí mismo*/
}
```

Ahora veremos el uso de funciones en el contexto del programa mediante un ejemplo. Imagine que quiere crear una aplicación que calcule alguna formula física. Además el programa tiene que tener una presentación inicial que diga si se quieren hacer los cálculos o si se quiere salir del programa. Para calcular la fórmula física sería interesante crear una función, la pantalla de presentación la podemos colocar en la parte principal del programa, y la pantalla que muestre los cálculos hacerla mediante otra función.

```
#include <stdio.h>

/* a continuación viene la función
   que retorna la velocidad conseguida
   a partir de la acelaración y el tiempo */
float velocidad(float aceleracion, float tiempo)
{
    return aceleracion*tiempo;
```

```
}

/* a continuación viene la función
   que retorna la distancia recorrida
   a partir de la aceleración y el tiempo */
float distancia(float aceleracion,float tiempo)
{
    /* ya que distancia=velocidad*tiempo llamamos
       a la función velocidad para calcular esta */

    return velocidad(aceleracion,tiempo)*tiempo;
}

void calculos(void)
{
    float accel,tiempo;
    printf("Introducir aceleración: ");
    scanf("%f",&accel);
    printf("Introducir tiempo: ");
    scanf("%f",&tiempo);

    printf("La distancia recorrida es %f.\n",distancia(accel,tiempo));
    printf("La velocidad adquirida es %f.\n",velocidad(accel,tiempo));
}

int main(void)
{
    char opcion;

    do {
        printf("1.- Realizar cálculos.\n");
        printf("2.- Salir.\n");
        printf("Elija opción: ");

        scanf("%d",&opcion);

        if(opcion==1)
        {
            calculos();
        }
    } while(opcion!=2); /* mientras opcion es diferente de 2 */

    return 0;
}
```

La salida por pantalla de este programa sería la siguiente:

>1 – Realizar cálculos
>2 – Salir
>Elija opción: 3
>Opción incorrecta
>
>1 – Realizar cálculos
>2 – Salir
>Elija opción: 1
>Introducir aceleración y tiempo: 2.1
>3.2
>La distancia recorrida es 21.504
>La velocidad adquirida es 6.72
>
>1 – Realizar cálculos
>2 – Salir
>Elija opción: 2
>¡Adiós!

No se preocupe si todavía no comprende del todo cuando es necesario incluir procedimientos y funciones en sus programas. A medida que vaya viendo más ejemplos y vaya creando sus propios programas irá viendo las grandes ventajas de estos elementos.

Otros elementos de los programas

A continuación, para acabar de comprender la estructura y los elementos que componen un programa de ordenador, hay que introducir algunos conceptos básicos más, tales como la visibilidad de los datos (global o local), los operadores y las constantes.

Visibilidad de los datos: variables globales y variables locales

Observe el siguiente programa en C:

```
#include <stdio.h>


void proc(int a)
{
     a=3;
}


int main(void)
{
     int a=6;
```

```
     proc(); /* Ahora se ejecuta el procedimiento proc() */
     printf("El valor de la variable A es %d.\n",a)
}
```

Vemos que el programa se inicia asignando el valor 6 a la variable *a*, que se llama al procedimiento *proc*, donde se asigna el valor 3 a la variable *a*, y se saca el resultado por pantalla. Entonces... ¿Qué cree usted que sacará por pantalla? A lo mejor pensará que la respuesta correcta es "El valor de la variable A es 3", pero sin embargo es "El valor de la variable A es 6"

"*¿Y cómo es posible, si antes de sacar el resultado por pantalla se ha llamado a proc(), donde se le asigna el valor 3?*" Se preguntará usted. Pues esto es debido a que la variable *a* es una **variable local**. Es decir, está definida en el procedimiento principal y en el procedimiento *proc()*, y en cada uno de ellos se trata de una variable distinta, aunque tengan el mismo nombre. Y además desde el procedimiento principal no se puede ver ni modificar la variable *a* declarada en el procedimiento *proc()*, ni desde *proc()* se puede ver ni modificar la variable *a* declarada en el procedimiento principal.

Observe ahora el siguiente programa en C:

```
#include <stdio.h>

int a;

void proc()
{
     a=3;
}

int main(void)
{
     a=6;
     proc(); /* Ahora se ejecuta el procedimiento proc() */
     printf("El valor de la variable A es %d.\n",a)
}
```

Tal y como habrá podido intuir, el resultado de este programa es la salida por pantalla de "El valor de la variable A es 3". Esto es debido a que a es ahora una **variable global**. Esta vez no está definida dentro del cuerpo de la función principal ni como parámetro de la función *proc*. Está definida al principio del programa, fuera de toda función. Esto significará que *a* se puede leer y modificar desde cualquier parte del programa. La *a* que se utiliza en el procedimiento *proc()* y la que se utiliza en el procedimiento principal son la misma variable.

Para acabar de entender los conceptos de global y local, se podría hacer una comparación entre los tipos de variables y los objetos de la vida real: imagínese que

cada función es una casa, y las personas son instrucciones (salvando las distancias!). Una variable global sería el equivalente a un objeto que todo el mundo podría ver y tocar, como podría ser un buzón, un cartel propagandístico, o un banco del parque. Una variable local, en cambio, serían los objetos personales que están dentro de una casa, y sólo los que habitan en ésta pueden ver y tocar.

Operadores numéricos

Las variables en sí son espacios reservados en la memoria para ser usados por los programas. Pero para utilizar estas variables son necesarios los **operadores**, que son un conjunto de símbolos que indican la operación que hay que realizar con una o varias variables, como puede ser una suma, una asignación, etc...

- **Operador de asignación**: Su función es asignar el valor de una variable o de una constante a otra variable. Ejemplos:

  a=3; /* asigna a la variable *a* el valor de la constante 3 */
  b=a; /* asigna a la variable *b* el valor *a*. Es decir, ahora *b* vale 3 */
  a=a+1; /* asigna a *a* el valor de *a+1*. Es decir, ahora *a* vale 4 */
  /* Al finalizar esta secuencia, *b* vale 3 y *a* vale 4. */

- **Operadores aritméticos:** Básicamente son: suma (+), resta (-), multiplicación (*) y división (/). Ejemplos:

  a=2
  b=3
  a=b*3 /* *a* ahora vale 9 */
  b=a-3 /* b ahora vale 6 */
  a=a+3 /* a ahora vale 12 */
  b=a/4 /* b ahora vale 4 */

- **Operadores de comparación**: se utilizan para comparar valores entre variables, estos son igual (==), menor que (<), mayor que (>) y diferente (!=). Ejemplos:

```
if (a<5)
{
     /* aquí va el código que se ejecutaría si
        a fuera menor que 5 */
}

if (a!=5)
{
```

```
        /* aquí va el código que se ejecutaría si
           a fuera diferente de 5 */
}
```

- **Operadores lógicos**: estos son ***y (&&)***, ***no (!)***, y ***o (||)***. Se utilizan para evaluar varias condiciones dentro de una misma. Ejemplos:

```
if((a==3) && (b==4))
{
     /* aquí va el código que se ejecutaría si
        y sólo si a valiese 3 y b valiese 4 */
}

while ( !(a==0) )
{
     /* aquí va el código que se ejecutaría
        sólo mientras a no fuera igual a 0
        también se podría especificar como a!=0
        (a diferente de 0)   */
}

if((a==0) || (a==3))
{
     /* aquí va el código que se ejecutaría */
     /* si a valiese 3 ó 0 */
}
```

- **Operadores lógicos a nivel de bit**: Sirven para operar con las variables, pero bit a bit. Estos operadores no los comentaremos, ya que su utilización se sale de los objetivos de este libro. Tan sólo decir que son y (&), o (|), o exclusivo (^) y los desplazamientos hacia izquierda y derecha (<< y >>).

Constantes

Las constantes son variables a las cuales no se les puede cambiar el valor. Es decir, se declaran en el programa con un valor inicial y éste ya no se puede cambiar. Su finalidad es muy sencilla: imagínese que quiere crear un programa que opera con un vector de 25 elementos como el siguiente:

```
void main(void)
{
```

```
      int a[25];
      int i;

      for(i=0;i<25;i++)
      {
            a[i]=0
      }
}
```

Ahora imagine que ha cambiado de opinión y quiere que el vector no sea de 25 elementos, sino de 45. No le costará mucho cambiar los 25 que ha escrito por un par de 45. Pero si se tratara de un programa muy largo, donde el número 25 está escrito cientos de veces, podría ser bastante engorroso cambiarlos todos, y probablemente alguno se nos olvidaría. Este problema lo hubiéramos evitado si desde el principio se hubiera utilizado una constante para definir el número de elementos del vector:

```
      #define NUM_ELEMS 25

      void main(void)
      {
            int a[NUM_ELEMS];
            int i;

            for(i=0;i<NUM_ELEMS;i++)
            {
                  a[i]=0
            }
      }
```

Una constante es un número cuyo valor no se puede cambiar, ya que no es una variable. Poniendo al principio la directiva *#define* seguida del nombre de la constante y del valor, podremos escribir en el cuerpo del programa el nombre de la constante en vez del número al que representa. Así bastará con cambiar la constante para cambiar el valor en todo el programa. Otro ejemplo del uso de constantes es el usar constantes físicas y matemáticas, por ejemplo:

```
#define pi 3.141592653589
```

En este ejemplo, sabemos que nunca vamos a cambiar el valor de la constante (pi es siempre el mismo valor), pero nos resulta más cómodo en el programa escribir *pi* en vez de 3.1415... cada vez que queramos hacer referencia a éste.

No escatime en utilizar constantes, ya que puede facilitarle mucho trabajo, también ayuda a hacer el programa más legible por el programador, y **no ocupan memoria** ni se pierde velocidad.

Estructura de un programa

Ahora que conoce los elementos básicos de un programa, pasamos a describir su estructura. Todo programa que escriba tendrá la siguiente estructura:

#<inclusión de librerías>

#<definición de constantes>

<definición de variables globales>

<declaración de funciones>

<función principal del programa (main)>