Tema 3: Estructuras de control.

1 Introducción

En ocasiones es necesario que nuestros programas tomen determinadas **decisiones**. Por ejemplo, imaginemos un programa que nos pregunte dos números enteros a y b y devuelva el resultado de realizar la división a/b. Puesto que la operación de división por cero no está permitida (desbordamiento), tendremos que comprobar si b es o no cero. En caso de no ser cero, realizaremos la operación. En caso de ser cero tendremos que dar un mensaje de error al usuario.

De igual forma, en algunas ocasiones tendremos que hacer que determinadas operaciones se **repitan** un determinado número de veces. Por ejemplo, imaginemos un programa que escriba por pantalla los números del 0 al 10000. Para ello podríamos escribir un programa con 10001 instrucciones de salida por pantalla, pero como podemos imaginar sería sumamente engorroso.

En este tema estudiaremos las distintas estructuras de decisión y repetición que existen en el lenguaje C. Se trata, como veremos, de herramientas muy comunes en la programación debido a su gran utilidad.

1.1 Bloques

Para definir la sintaxis de cada una de las estructuras de este tema nos apoyaremos en la definición de **bloque de instrucciones**. Un bloque de instrucciones puede ser una sola instrucción o bien una lista de dos o más instrucciones enmarcadas entre llaves {}.

Por ejemplo, en el siguiente extracto de código las tres instrucciones que aparecen en las líneas tres y cuatro formarían un bloque y la instrucción que aparece en la línea seis otro bloque distinto:

```
int a,b,c;
{
      b=3; c=4;
      a=b*c+5;
}
b=c*a;
```

En ocasiones también es posible que podamos encontrar unos bloques dentro de otros, como por ejemplo:

```
int a,b,c;
{
      b=3; c=4;
      {
            a=b*c+5;
            c++;
      }
}
```

2 Estructura secuencial

Hasta ahora, nuestros programas siguen lo que se denomina **estructura secuencial**. En este tipo de estructuras, el orden de ejecución de las instrucciones es siempre el mismo, esto es, en el mismo orden en el que se escriben. Por ejemplo:

```
int a,b,c;
a=5; a=a/3;
b=10;
c=a+b*5+7;
```

En el ejemplo, la ejecución seguiría el orden de lectura, esto es, de izquierda a derecha y de arriba a abajo. No es posible, por tanto, que una misma instrucción se ejecute dos veces o que simplemente no se ejecute en una determinada circunstancia.

En los siguientes puntos veremos varias estructuras que permiten variar el orden de la ejecución de las instrucciones en ciertas condiciones.

3 Estructuras de decisión

Las estructuras de decisión nos sirven para ejecutar o no un conjunto de instrucciones dependiendo de una determinada condición. Distinguiremos entre estructuras de decisión **simple** y estructuras de decisión **múltiple**.

3.1 Decisión simple

Una estructura de decisión simple consta de una condición y dos bloques de instrucciones. El primer bloque se ejecutará si dicha condición es cierta. En caso contrario se ejecutará el segundo bloque. La sintaxis de esta estructura es la siguiente:

```
if( condicion )
    [BLOQUE SI CIERTO]
else
    [BLOQUE SI FALSO]
```

El funcionamiento es muy simple: en la condición se coloca una expresión (normalmente relacional o lógica) cuyo resultado sea un valor lógico (cierto o falso). Si la condición es cierta se ejecutan secuencialmente las instrucciones del primer bloque y si no las del segundo, pero nunca ambas.

Adicionalmente podemos excluir el segundo bloque, ya que éste es opcional. De esta forma, ejecutaríamos el primer bloque si la condición es cierta. En caso contrario no haríamos nada:

```
if( condicion )
    [BLOQUE SI CIERTO]
```

Volviendo al ejemplo del principio del tema, vamos a escribir un programa que solicite dos números a y b al usuario y muestre la división a/b siempre que b sea distinto de cero:

```
int a,b;
printf(“numero a:"); scanf("%d",&a);
printf(“numero b:"); scanf("%d",&b);
if(b!=0){
    printf("El resultado de a/b es: %d\n",a/b);
}
```

De esta forma, cuando b coincide con cero la operación no se realiza. Podemos ampliar nuestro programa utilizando la parte del else para informar del error al usuario:

```
int a,b;
printf(“numero a:"); scanf("%d",&a);
printf(“numero b:"); scanf("%d",&b);
if(b!=0){
    printf("El resultado de a/b es: %d\n",a/b);
}else{
    printf("Error. b no puede ser cero\n");
}
```

En el siguiente ejemplo solicitamos dos números enteros al usuario y mediante una estructura de decisión simple calculamos cuál de ellos es mayor:

```
int a,b;
printf(“Numero a:”); scanf(“%d”,&a);
printf(“Numero b:”); scanf(“%d”,&b);
if(a>b){
    printf(“El mayor es a\n”);
}else{
    printf(“El mayor es b\n”);
}
```

Es también posible anidar estructuras de decisión. En el ejemplo anterior, si los dos números son iguales, el programa contestaría diciendo que el segundo es el mayor, lo cual no es del todo correcto. Para arreglarlo, podemos utilizar una estructura condicional anidada:

```
int a,b;
printf("Numero a:"); scanf("%d",&a);
printf("Numero b:"); scanf("%d",&b);
if(a>b){
    printf("El mayor es a\n");
}else{
    if(a<b){
        printf("El mayor es b\n");
    }else{
        printf("a y b son iguales\n");
    }
}
```

Como vemos, si *a* es mayor a *b* estrictamente podemos decir directamente que *a* es el mayor, pero si no ocurre esto tendremos que comprobar si *b* es el mayor. Si esto no se cumple, sabemos que *a* no es mayor a *b* y que *b* no es mayor a *a*, luego son iguales.

Veamos otro ejemplo similar pero para determinar si un número es positivo, negativo o cero:

```
int num;
printf("Numero:"); scanf("%d",&num);
if( num == 0){
    printf("El numero es cero\n");
}else{
    if( num > 0){
        printf("El numero es positivo \n");
    }else{
        printf("El numero es negativo\n");
    }
}
```

Como hemos dicho, en la condición se puede imponer cualquier expresión que pueda evaluarse a cierto o falso. Esto significa que podemos incluir cualquier expresión por complicada que sea cuyo resultado final sea verdadero o falso. Por ejemplo, el siguiente fragmento de código calcula si un número es positivo y par:

```
int num;
printf("Numero:"); scanf("%d",&num);
if( num >0 && (num/2)*2==num){
    printf("El numero es positivo y par\n");
}else{
    printf("El numero es negativo o impar\n");
}
```

Como vemos, hemos utilizado la conjunción para señalar que la expresión será cierta solamente cuando se cumplan las dos condiciones. Para calcular si es par o no, utilizamos la división entera y el producto. Si el número es par, el resultado es el mismo número, mientras que si es impar el resultado será una unidad menor. Esta condición

también se puede escribir utilizando el operador resto para comprobar si al dividir por dos resulta cero (par): num%2==0

3.2 Decisión múltiple

Las estructuras de decisión múltiple nos sirven para tomar más de una decisión. Constan de un dato y un conjunto de bloques. Cada bloque está ligado a un posible valor del dato de forma que, dependiendo del contenido de éste se ejecutará el bloque asociado al mismo.

La sintaxis es la siguiente:

```
switch (variable) {
    case valor0:
            // acciones si variable==valor0
            break;
    case valor1:
            // acciones si variable==valor1
            break;
    ...
    case valorN:
            // acciones si variable==valorN
            break;
    default:
            // acciones en cualquier otro caso
            break;
}
```

El funcionamiento es sencillo: se consulta el valor de la variable y se busca dicho valor en la lista, ejecutando todas las instrucciones ligadas al mismo hasta la siguiente instrucción *break*. Si no encuentra el valor, se ejecutarán las instrucciones ligadas al caso por defecto (default).

En el siguiente ejemplo solicitamos un número entero al usuario y determinamos a qué día de la semana se refiere, siendo el 0 Lunes, el 1 Martes, etc…

```
int dia; printf("Dia:"); scanf("%d",&dia);
printf("El dia %d se corresponede con : ",dia);
switch (dia) {
    case 0:
        printf("Lunes");
        break;
    case 1:
        printf("Martes");
        break;
    case 2:
        printf("Miercoles");
        break;
```

```
    case 3:
        printf("Jueves");
        break;
    case 4:
        printf("Viernes");
        break;
    case 5:
        printf("Sabado");
        break;
    case 6:
        printf("Domingo");
        break;
    default:
        printf("Ningun dia");
        break;
}
printf("\n");
```

Si en un caso no aparece la instrucción de corte *break*, se seguirán ejecutando las instrucciones del siguiente caso hasta que aparezca. Por tanto, se pueden agrupar distintos casos en uno solo. En el siguiente ejemplo, solicitamos un número entero al usuario y determinamos si se encuentre entre uno de los siguientes casos:

- Si se trata del número cero
- Si se trata del número uno
- Si se trata del número dos o del tres (indistintamente)
- Si no se encuentra entre el cero y el tres

```
char cadena[255 ];
int id;
printf("Numero:"); scanf("%d",&id);
switch (id) {
    case 0: strcpy(cadena, "Cero");
            break;
    case 1: strcpy(cadena, "Uno");
            break;
    case 2:
    case 3: strcpy(cadena, "Dos o Tres");
            break;
    default: strcpy(cadena, "No está entre cero y tres");
            break;
}
printf("Solucion : %s\n",cadena);
```

Como vemos, si se trata del número dos, se ejecutan las instrucciones asociadas al caso 3 puesto que el caso 2 no contiene un *break*.

4 Estructuras de repetición

Las estructuras de repetición nos permiten ejecutar un conjunto de instrucciones un determinado número de veces. Distinguiremos entre estructuras de repetición con condición inicial, con condición final o con contador.

4.1 Repetición con condición inicial

Las estructuras de repetición con condición inicial constan de una condición y un bloque. El bloque se ejecuta mientras que la condición sea cierta. La comprobación de la condición se realiza antes de la primera ejecución del bloque.

La sintaxis de esta estructura es la siguiente:

```
while(condición)
    [BLOQUE]
```

Veamos un par de ejemplos simbólicos:

```
while(1){
    printf("Bucle infinito\n");
}

while(0){
    printf("Bucle vacio\n");
}
```

En el primer caso, la condición es verdadera al principio y además no hay posibilidad de que cambie, por lo que la instrucción que contiene se ejecutará infinitas veces. En el segundo bucle pasa algo parecido, o sea, depende de una condición que es falsa al principio y no hay posibilidad de que cambie. La instrucción que contiene no se ejecutará nunca.

Veamos un ejemplo más realista en el que calculamos el factorial de un número entero:

```
int num, soluc = 1;
printf("Numero:"); scanf("%d",&num);
while(num>0){
    soluc = soluc * num;
    num= num-1;
}
printf("Factorial = %d\n",soluc);
```

Como vemos, utilizamos un acumulador en el que vamos almacenando el resultado parcial. En cada vuelta de bucle multiplicamos el acumulador y restamos una unidad. Si el número fuese negativo o cero antes de empezar, el bucle no se ejecutaría ninguna vez y el resultado sería 1.

4.2 Repetición con condición final

Esta estructura es similar a la anterior. También consta de una condición y un bloque que se ejecutará repetidas veces mientras se cumpla la condición. La diferencia es que en este caso la condición se comprueba al final del bucle.

La sintaxis de esta estructura es la siguiente:

```
do
    [BLOQUE]
while(condición);
```

En este caso, puesto que la condición se comprueba al final, el bloque al menos se ejecutará la primera vez aunque sea falsa. Una vez superada la primera iteración, el resto se ejecutará mientras la condición sea verdadera.

Vamos a observar los casos simbólicos mostrados en el apartado anterior reescritos con esta estructura:

```
do{
    printf(“Bucle infinito\n”);
} while(1);

do{
    printf(“Bucle vacio\n”);
} while(0);
```

En el primer caso, el bucle seguiría siendo infinito puesto que la condición siempre es cierta. En el segundo caso, puesto que la condición es siempre falsa, el bucle realizaría la primera iteración ejecutando el bloque una vez. Después comprobaría la condición y terminaría.

En el siguiente ejemplo pediremos un número al usuario entre 1 y 10. Si el número no es correcto mostraremos un mensaje de error y volveremos a pedir el número hasta que el usuario introduzca un valor en dicho intervalo:

```
int valor;
do{
    printf("Introduzca numero entre 1 y 10:"); scanf("%d",&valor);
    if(valor<1 || valor>10){
            printf("Error. El numero no es valido.\n");
    }
}while(valor<1 || valor>10);
```

Como vemos, no tendría sentido utilizar la estructura de repetición con condición inicial puesto que, al menos, tenemos que preguntar el número la primera vez para para continuar o no el bucle.

4.3 Repetición con contador

La estructura de repetición con contador es, sin duda, la más utilizada por su versatilidad. Utilizaremos esta estructura cuando tengamos que repetir una secuencia de instrucciones una cantidad fija de veces. Normalmente se apoya en una variable entera a la que llamaremos **contador** que contiene el número de ejecuciones del bloque que se han realizado hasta el momento.

La sintaxis de esta estructura es la siguiente:

```
for( inicio ; condición ; incremento)
    [ BLOQUE ]
```

En la instrucción de **inicio** se da el valor inicial a la variable contador. El bucle se ejecutará mientras la **condición** sea cierta. En cada paso se ejecuta el **incremento** para actualizar el contador.

El orden de ejecución es el siguiente:

- El inicio se ejecuta antes de comenzar el bucle
- La condición se comprueba antes de cada vuelta
- El incremento se realiza al final de cada vuelta

El esquema de ejecución del bucle sería:

1) Ejecutar **inicio**
2) Comprobar **condición**. Si es falsa saltar a (5)
3) Ejecutar el **BLOQUE**
4) Realizar **incremento** y saltar a (2)
5) Fin del bucle.

Veamos un ejemplo sencillo para imprimir diez mensajes por pantalla:

```
int i;
for(i=0; i<10;i++){
    printf("El numero es %d\n",i);
}
```

En el ejemplo, el primer mensaje sería "El numero es 0" y el último "El numero es 9" puesto que en la inicialización del contador lo ponemos a 0 y la condición de finalización es estrictamente menor a 10.

En este otro ejemplo vamos a solicitar un número *num* al usuario y calcularemos su factorial y el sumatorio de todos los enteros positivos menores o iguales:

```
int num, factorial=1, sumatorio=0, i;
printf("Numero:");
scanf("%d",&num);
for(i=1; i<=num; i++){
    factorial=factorial * i;
    sumatorio=sumatorio + i;
```

```
}
printf("Factorial = %d\n",factorial);
printf("Sumatorio = %d\n",sumatorio);
```

Por tanto, hemos realizado los cálculos:

$$factorial = \prod_{i=1}^{num} i \qquad sumatorio = \sum_{i=1}^{num} i \,.$$

Como vemos, se han utilizado dos variables en las que se han ido acumulando los resultados. Es necesario que la inicialización de dichas variables sea coherente con su uso.

En ocasiones puede ser necesario anidar estructuras de repetición dentro de otras. En el siguiente ejemplo hemos anidado dos estructuras de repetición con contador para mostrar las tablas de multiplicar:

```
int i,j;
for(i=0; i<=10; i++){
    printf("Tabla del %d\n",i);
    for(j=0; j<=10; j++){
        printf("%d X %d = %d\n",i,j,i*j);
    }
}
```

Como vemos, primer bucle se repite diez veces y el interior se repite diez veces por cada vez que se repite el primero. Por tanto, el bloque asociado al bucle más interior se repetirá cien veces.

Veamos un ejemplo más complicado en el que tenemos que determinar si un número es o no primo:

```
#define Verdadero 1
#define Falso     0

void main(void)
{
    int num, I, EsPrimo = Verdadero;
    printf("Introduzca numero: ");
    scanf("%d",&num);
    for( I = 2; I<num ; I++ ){
        if( num % I == 0 ){
            EsPrimo = Falso;
        }
    }
    if(EsPrimo==Verdadero)    printf("El numero es primo\n");
    else                      printf("El numero no es primo\n");
}
```

Como vemos, hemos creado una variable booleana, que al no existir en el lenguaje, hemos tenido que simularla con un número entero y con las constantes Verdadero y Falso. Al principio suponemos que el número es primo. Si nos encontramos con un entero inferior a él por el que sea divisible (resto igual a cero) descartamos que sea primo.

5 Equivalencias entre estructuras

El conjunto de estructuras proporcionado resulta redundante. Esto significa que algunas estructuras pueden rescribirse utilizando otras sin perder funcionalidad. Por ejemplo, una estructura condicional múltiple como la siguiente:

```
switch(id){
    case v1:
        //acciones1
        break;
    case v2:
        //acciones2
        break;
    default:
        //accionesDefault
        break;
}
```

Podría rescribirse utilizando estructuras condicionales simples anidadas:

```
if(id == v1){
    //acciones1
}else
if(id == v2){
    //acciones 2
}else{
    //accionesDefault
}
```

Para expresar toda la funcionalidad del condicional múltiple, por ejemplo agrupando varios casos sin *break*, sería necesaria una estructura más compleja, pero se podría hacer de igual forma.

Como vemos, podríamos llegar a la conclusión de que con un conjunto de condicionales simples se puede escribir un condicional múltiple, por lo que el condicional múltiple no serviría para nada. Sin embargo, como vemos resulta más cómodo utilizarlo en determinadas circunstancias.

De igual forma, una estructura de repetición con contador sencilla como la que sigue:

```
int i, n=20;
for(i=0;i<n;i++){
    //acciones
}
```

Podría rescribirse con una estructura de repetición con condición inicial, controlando el número de iteraciones manualmente:

```
int i, n=20;
i=0;
while(i<n){
    //acciones
    i++;
}
```

Como vemos, aunque las estructuras realizan la misma función, la primera resulta más clara.

Además de los casos de redundancia citados, existen muchos otros. Esto se debe a que el lenguaje C es muy flexible y contiene una gran potencia expresiva. En general, aunque el lenguaje nos permita resolver un problema con distintas estructuras, elegiremos aquélla que resuelva el problema de forma más clara y efectiva.