



Unidad 4

Sentencias de Control

Principal material bibliográfico utilizado

- www.jorgesanchez.net
- Fundamentos de programación C/C++ 4ta Edición – Ernesto Peñaloza Romero - Alfaomega

Expresiones de relación y lógicas

Operadores de relación

operador	significado
>	Mayor que
>=	Mayor o igual que
<	Menor que
<=	Menor o igual que
==	Igual que
!=	Distinto de

Ejemplos:

Total > 1000 letra == 'a'

Los resultados solo pueden tener dos valores posibles , verdadero o falso.

En C un valor falso es representado por el 0, el valor verdadero es representado por cualquier valor distintos de 0 (especialmente el 1)

Expresiones de relación y lógicas

Existen tres operadores (lógicos) que manipulan los valores de verdad.

operador	significado
&&	Y (AND)
	O (OR)
!	NO (NOT)

A su vez, estos operadores funcionan según la siguiente tabla de verdad

A	!A	A	B	A B	A	B	A && B
F	V	F	F	F	F	F	F
V	F	F	V	V	F	V	F
		V	F	V	V	F	F
		V	V	V	V	V	V
Negación (No)		Unión (o)			Conjunción (Y)		

Expresiones de relación y lógicas

La tabla de precedencia para los operadores que vimos es:

!	
> >= < <=	
== !=	
&&	

Mayor

↓

Menor

Ejemplo:

(iva + ispt) != max_gravable

iva + ispt != max_gravable

Son lo mismo.

CUIDADO CON LA PRECEDENCIA DE LAS OPERACIONES

Precedencias

- (1) `() [] . ->`
- (2) Lo forman los siguientes:
 - ◆ NOT de expresiones lógicas: `!`
 - ◆ NOT de bits: `~`
 - ◆ Operadores de punteros: `* &`
 - ◆ Cambio de signo: `-`
 - ◆ `sizeof`
 - ◆ `(cast)`
 - ◆ Decremento e incremento: `++ --`
- (3) Aritméticos prioritarios: `/ * %`
- (4) Aritméticos no prioritarios (suma y resta): `+ -`
- (5) Desplazamientos: `>> <<`
- (6) Relacionales sin igualdad: `> < >= <=`
- (7) Relacionales de igualdad: `== !=`
- (8) `&`
- (9) `^`
- (10) `|`
- (11) `&&`
- (12) `||`
- (13) `?:`
- (14) `= *= /= += -= %= >>= <<= |= &= ^=`
- (15) `,` (coma)

Sentencia **if**

Se trata de una sentencia que, tras evaluar una expresión lógica, ejecuta una serie de sentencias en caso de que la expresión lógica sea verdadera. Su sintaxis es:

```
if(expresión lógica)  
{  
  sentencias  
}
```

Si sólo se va a ejecutar una sentencia, no hace falta usar las llaves:

```
if(expresión lógica) sentencia;
```

Sentencia **if**

Ejemplo:

```
if(nota>=5)  
{  
    printf("Aprobado");  
    aprobados++;  
}
```


Sentencia condicional compuesta

Instrucciones que se ejecutarán si la expresión evaluada por el **if** es falsa.

Sintaxis:

```
if(expresión lógica){  
  sentencias  
}  
else {  
  sentencias  
}
```

Sentencia **if**

Las llaves son necesarias sólo si se ejecuta más de una sentencia. Ejemplo:

```
if(nota>=5){  
    printf("Aprobado");  
    aprobados++;  
}  
else {  
    printf("Suspensos");  
    suspensos++;  
}
```

Anidación

Dentro de una sentencia **if** se puede colocar otra sentencia **if**. A esto se le llama **anidación** y permite crear programas donde se valoren expresiones complejas.

Por ejemplo en un programa donde se realice una determinada operación dependiendo de los valores de una variable, el código podría quedar:

Anidación

```
if (x==1) {  
    sentencias  
    ...  
}  
else {  
    if(x==2) {  
        sentencias  
        ...  
    }  
    else {  
        if(x==3) {  
            sentencias  
            ...  
        }  
    }  
}
```

Anidación

Pero si cada **else** tiene dentro sólo una instrucción **if** entonces se podría escribir

de esta forma (que es más legible), llamada ***if-else-if***:

```
if (x==1) {  
  instrucciones  
  ...  
}  
else if (x==2) {  
  instrucciones  
  ...  
}  
else if (x==3) {  
  instrucciones  
  ...  
}
```

Sentencia **switch**

Se trata de una sentencia que permite construir alternativas múltiples. Pero que en el lenguaje C está muy limitada. Sólo sirve para evaluar el valor de una variable entera (o de carácter, ***char***).

Tras indicar la expresión que se evalúa, a continuación se compara con cada valor agrupado por una sentencia **case**. Cuando el programa encuentra un **case** que encaja con el valor de la expresión igualmente se ejecutan todos los **case** siguientes. Por eso se utiliza la sentencia **break** para hacer que el programa abandone el bloque **switch**.

Sentencia switch

```
switch(expresión ){  
case valor1:  
  sentencias  
break; /*Para que programa salte fuera del switch  
de otro modo atraviesa todos los demás  
case */  
case valor2:  
  sentencias  
  ...  
default:  
  sentencias  
}
```

Sentencia switch

```
switch (diasemana) {  
    case 1:  
        printf("Lunes");  
        break;  
    case 2:  
        printf("Martes");  
        break;  
    case 3:  
        printf("Miércoles");  
        break;  
    case 4:  
        printf("Jueves");  
        break;  
    case 5:  
        printf("Viernes");  
        break;  
    case 6:  
        printf("Sábado");  
        break;  
    case 7:  
        printf("Domingo");  
        break;  
    default:  
        Printf("Error");  
}
```


Sentencia **switch**

Sólo se pueden evaluar expresiones con valores concretos (no hay un **case** **>3** por ejemplo).

Aunque sí se pueden agrupar varias expresiones aprovechando el hecho de que al entrar en un case se ejecutan las expresiones de los siguientes.

Sentencia switch

```
switch (diasemana) {  
  case 1:  
  case 2:  
  case 3:  
  case 4:  
  case 5:  
    printf("Laborable");  
    break;  
  case 6:  
  case 7:  
    printf("Fin de semana");  
    break;  
  default:  
    printf("Error");  
}
```

Bucles

A continuación se presentan las instrucciones C que permiten realizar instrucciones repetitivas (bucles).

Sentencia **while**

Es una de las sentencias fundamentales para poder programar.

Se trata de una serie de instrucciones que se ejecutan continuamente mientras una expresión lógica sea cierta.

Sintaxis:

```
while (expresión lógica) {  
sentencias  
}
```

Sentencia **while**

El programa se ejecuta siguiendo estos pasos:

- (1) Se evalúa la expresión lógica.
- (2) Si la expresión es verdadera ejecuta las sentencias, sino el programa abandona la sentencia **while**.
- (3) Tras ejecutar las sentencias, volvemos al paso 1.

Sentencia **while**

Ejemplo (escribir números del 1 al 100):

```
int i=1;  
while (i<=100){  
    printf(“%d”,i);  
    i++;  
}
```

Sentencia do..while

La única diferencia respecto a la anterior está en que la expresión lógica se evalúa después de haber ejecutado las sentencias.

Es decir el bucle al menos se ejecuta una vez.
Los pasos son:

- (1) Ejecutar sentencias.
- (2) Evaluar expresión lógica.
- (3) Si la expresión es verdadera volver al paso 1, sino continuar fuera del while.

Sentencia do..while

Sintaxis:

```
do {  
  sentencias  
} while (expresión lógica)
```

Ejemplo (contar del 1 al 1000):

```
int i=0;  
do {  
  i++;  
  printf("%d",i);  
} while (i<=1000);
```


Sentencia **for**

Se trata de un bucle especialmente útil para utilizar contadores.

Su formato es:

```
for(inicialización;condición;incremento){  
  sentencias  
}
```

Sentencia **for**

Las sentencias se ejecutan mientras la condición sea verdadera. Además antes de entrar en el bucle se ejecuta la instrucción de inicialización y en cada vuelta se ejecuta el incremento. Es decir el funcionamiento es:

(1) Se ejecuta la instrucción de inicialización.

(2) Se comprueba la condición.

(3) Si la condición es cierta, entonces se ejecutan las sentencias. Si la condición es falsa, abandonamos el bloque **for**

(4) Tras ejecutar las sentencias, se ejecuta la instrucción de incremento y se vuelve al paso 2

Sentencia for

Ejemplo (contar números del 1 al 1000):

```
for(int i=1;i<=1000;i++){  
    printf("%d",i);  
}
```

La ventaja que tiene es que el código se reduce. La desventaja es que el código es menos comprensible. El bucle anterior es equivalente al siguiente bucle

```
while: i=1; /*sentencia de inicialización*/  
while(i<=1000) { /*condición*/  
    printf("%d",i);  
    i++; /*incremento*/  
}
```



FIN