

UNEFA EXT-MARACAY  
PROGRAMACIÓN V-ING. DE SISTEMAS  
AUXILIAR DOCENTE NORKIS PÉREZ P. 03/05/2009  
**LIBRERÍAS**

Junto con los compiladores de C y C++, se incluyen ciertos ficheros llamados librerías. Las librerías contienen el código objeto de muchos programas que permiten hacer cosas comunes, como leer el teclado, escribir en la pantalla, manejar números, realizar funciones matemáticas, etc. Las librerías están clasificadas por el tipo de trabajos que hacen, hay librerías de entrada y salida, matemáticas, de manejo de memoria, de manejo de textos, etc.

### **LIBRERÍA STDIO.H**

#### **printf**

Función: Escribe en la salida estándar con formato.

Sintaxis: printf(formato , arg1 , ...);

#### **scanf**

Función: Lee de la salida estándar con formato.

Sintaxis: scanf(formato , arg1 , ...);

#### **puts**

Función: Escribe una cadena y salto de línea.

Sintaxis: puts(cadena);

#### **gets**

Función: Lee y guarda una cadena introducida por teclado.

Sintaxis: gets(cadena);

#### **fopen**

Función: Abre un fichero en el modo indicado.

Sintaxis: pf=fopen(fichero , modo);

#### **fclose**

Función: Cierra un fichero cuyo puntero le indicamos.

Sintaxis: fclose(pf);

### **fprintf**

Función: Escribe con formato en un fichero.

Sintaxis: fprintf(pf , formato , arg1 , ...);

### **fgets**

Función: Lee una cadena de un fichero.

Sintaxis: fgets(cadena , longitud , pf);

## LIBRERÍA STD.LH

### **atof**

Función: Convierte una cadena en float.

Sintaxis: numflo=atof(cadena);

### **atoi**

Función: Convierte una cadena en entero.

Sintaxis: nument=atoi(cadena);

### **itoa**

Función: Convierte un entero a cadena. La base, generalmente será 10.

Sintaxis: itoa(número , cadena , base);

### **exit**

Función: Termina la ejecución y abandona el programa.

Sintaxis: exit(estado); /\* Normalmente el estado será 0 \*/

## LIBRERÍA CONIO.H

### **clrscr**

Función: Borra la pantalla.

Sintaxis: clrscr( );

### **clreol**

Función: Borra desde la posición del cursor hasta el final de la línea.

Sintaxis: clreol( );

### **gotoxy**

Función: Cambia la posición del cursor a las coordenadas indicadas.

Sintaxis: gotoxy(columna , fila);

### **textcolor**

Función: Selecciona el color de texto (0 - 15).

Sintaxis: textcolor(color);

### **textbackground**

Función: Selecciona el color de fondo (0 - 7).

Sintaxis: textbackground(color);

### **wherex**

Función: Retorna la columna en la que se encuentra el cursor.

Sintaxis: col=wherex( );

### **wherey**

Función: Retorna la fila en la que se encuentra el cursor.

Sintaxis: fila=wherey( );

### **getch**

Función: Lee un carácter pulsado por el usuario sin mostrarlo por pantalla.

Sintaxis: letra=getch( );

### **getche**

Función: Lee un carácter pulsado por el usuario mostrándolo por pantalla.

Sintaxis: letra=getche( );

## **LIBRERÍA STRING.H**

### **strlen**

Función: Calcula la longitud de una cadena.

Sintaxis: longitud=strlen(cadena);

### **strcpy**

Función: Copia el contenido de una cadena sobre otra.

Sintaxis: strcpy(copia , original);

### **strcat**

Función: Concatena dos cadenas.

Sintaxis: strcat(cadena1 , cadena2);

### **strcmp**

Función: Compara el contenido de dos cadenas. Si cadena1 < cadena2 retorna un número negativo. Si cadena1 > cadena2, un número positivo, y si cadena1 es igual que cadena2 retorna 0 (o NULL ).

Sintaxis: valor=strcmp(cadena1 , cadena2);

## **FUNCIONES INTERESANTES**

### **fflush(stdin)**

Función: Limpia el buffer de teclado.

Sintaxis: fflush(stdin);

Prototipo: stdio.h

### **sizeof**

Función: Operador que retorna el tamaño en bytes de una variable.

Sintaxis: tamaño=sizeof(variable);

### **cprintf**

Función: Funciona como el printf pero escribe en el color que hayamos activado con la función textcolor sobre el color activado con textbackground.

Sintaxis: cprintf(formato , arg1 , ...);

Prototipo: conio.h

### **lkbhit**

Función: Espera la pulsación de una tecla para continuar la ejecución.

Sintaxis: while (!lkbhit( )) /\* Mientras no pulsemos una tecla... \*/

Prototipo: conio.h

### **random**

Función: Retorna un valor aleatorio entre 0 y num-1.

Sintaxis: valor=random(num); /\* También necesitamos la función randomize \*/

Prototipo: stdl.h

### **randomize**

Función: Inicializa el generador de números aleatorios. Debemos llamarlo al inicio de la función en que utilizemos el random. También deberemos utilizar el include time.h, ya que randomize hace una llamada a la función time, incluida en este último fichero.

Sintaxis: randomize( );

Prototipo: stdio.h

### **system**

Función: Ejecuta el comando indicado. Esto incluye tanto los comandos del sistema operativo, como cualquier programa que nosotros le indiquemos. Al acabar la ejecución del comando, volverá a la línea de código situada a continuación de la sentencia system.

Sintaxis: system(comando); /\* p.ej: system("arj a programa"); \*/

Prototipo: stdl.h

## **CADENAS DE FORMATO**

d, i	entero decimal con signo
o	entero octal sin signo
u	entero decimal sin signo
x	entero hexadecimal sin signo (en minúsculas)
X	entero hexadecimal sin signo (en mayúsculas)
f	Coma flotante en la forma [-]dddd.dddd

e	Coma flotante en la forma [-]d.dddd e[+/-]ddd
g	Coma flotante según el valor
E	Como e pero en mayúsculas
G	Como g pero en mayúsculas
c	un carácter
s	cadena de caracteres terminada en '\0'
%	imprime el carácter %
p	puntero

## LOS OPERADORES

Los operadores relacionar son aquellos que nos permiten comparar dos valores, resultando un valor lógico. Ellos son:

Operador Nombre

> mayor que

< menor que

<= menor o igual que

>= mayor o igual que

!= distinta que

= = igual que

Observación: No hay que confundir el operador ==, con el operador =, dado que el primero significa comparación, mientras el segundo asignación.

Vamos a ver como trabajan a través de un ejemplo. Supongamos que tenemos tres variables A=5, B=5 y C=7.

### Expresión Resultado

A>B 0

A>=B 1

A<=C 1

A!=B 0

A!=C 1  
A==B 1  
A==C 0

## ESTRUCTURA CONDICIONAL

### Estructura condicional “if”

Es una estructura simple que permite ejecutar una instrucción o un bloque de instrucciones sólo si se cumple una expresión lógica, es decir, se ejecuta sólo si el resultado de la expresión lógica es verdadero.

```
if (expresión_lógica)
{
acción;
}
```

Si la expresión lógica es verdadera la acción se ejecuta, si es falsa se ignora la acción y se continua con la instrucción siguiente a la estructura condicional. Si se quiere ejecutar una sola acción el uso de las llaves es opcional.

Esta estructura también permite ejecutar una acción si no se cumple (else) la expresión lógica. Su estructura sería:

```
if (expresión_lógica)
{acción_1;}
else
{acción_2;};
```

En este caso, si la expresión lógica es verdadera se ejecuta la acción\_1, si es falsa se ejecuta la acción\_2.

### Ejemplificando:

```
if (A > B)
C = A - B;
else
C = A + B;
```

## Estructura condicional “switch”

La instrucción switch se podría considerar como una extensión de la instrucción if. Permitiendo ejecutar múltiples acciones evaluando una sola variable de control a la cual llamaremos selector, que de acuerdo a su valor en el instante que se evalúa corresponderá la acción a ejecutar. Su sintaxis es:

```
switch (selector) {  
case valor_1: {accion_1; break;}  
case valor_2: {accion_2; break;}  
...  
case valor_n: {accion_n; break;}  
default:  
{accion_por_defecto;}  
}
```

El selector debe ser una variable ordinal, es decir, una variable que posea una secuencia definida (ordenada) y acotada (finitas posibilidades), como puede ser una variable del tipo int, bool, cualquier tipo definido por el usuario, o el resultado de una expresión; siempre y cuando cumplan con la condición.

Vamos a ver un ejemplo de su uso. Queremos analizar la paridad de un número ingresado por el usuario, almacenando en una variable lógica (bool) llamada par:

```
switch (num%2) {  
case 0: {par = true; break;}  
case 1: {par = false; break;}  
};
```

Se analizan solamente los casos 0 y 1 dado a que el resto de la división por 2 sólo puede tomar estos valores.

Para el segundo ejemplo queremos determinar si un carácter es un vocal o no, y si es una vocal determinar cual. Vamos a analizar una variable caracter del tipo de dato llamado char que corresponde a un caracter, y devolveremos el resultado en una cadena de caracteres del tipo de datos AnsiString llamada Resultado:

```
switch (caracter) {  
case 'a': {Resultado = “es la vocal a”; break;}  
case 'e': {Resultado = “es la vocal e”; break;}  
}
```

```
case 'i': {Resultado = "es la vocal i"; break;}  
case 'o': {Resultado = "es la vocal o"; break;}  
case 'u': {Resultado = "es la vocal u"; break;}  
default:  
{Resultado = "no es vocal";} }  
}
```

## ESTRUCTURAS REPETITIVAS

Normalmente dentro de un programa, es necesario realizar acciones de forma repetida, por ejemplo, imaginemos que queremos ejecutar 100 veces una acción, podríamos escribir 100 veces la misma línea o bien indicar que ejecute 100 veces la misma línea. Para ello, se creó en el lenguaje de programación, las estructuras for, while, y do while, que son las que nos permitirán codificar ciclos o bucles según sea necesario. Primero, debemos saber que todo ciclo, tiene una condición inicial, que inicia el ciclo, una condición final, que, cuando se cumple, el bucle finaliza, y un cuerpo o bloque de código que el ciclo realizará.

El cuerpo contiene la instrucción que se ejecuta cada vez por medio del ciclo y puede incluir cualquier código válido en C++.

### Ciclo for

La estructura for ("para"), se utiliza para realizar, generalmente, una acción cierta cantidad determinada y definida de veces. Para ello, consta de tres parámetros que debemos definir:

- Inicialización
- Condición de salida
- Incremento

```
for (inicialización; condición de salida; incremento)  
{acción};
```

### Ciclo while

El ciclo while ("mientras") difiere del ciclo for en que sólo contiene una condición

de prueba, que se verifica al principio de cada iteración. Mientras la condición sea true el ciclo continúa funcionando. La sintaxis correspondiente es:

```
while (expresión_lógica) {acción_1;  
acción_2;  
.  
.  
acción_n};
```

En la misma, la acción se realiza mientras la expresión lógica sea verdadera (valor distinto de cero). Es de vital importancia que la acción tenga alguna forma de modificar el valor de la expresión lógica, para que, en algún momento, sea falsa y el ciclo finalice. Si de entrada la expresión lógica da falsa, la acción del ciclo nunca se realiza.

### **Ciclo do-while.**

Este ciclo es prácticamente igual al while. Sin embargo, la diferencia entre los dos es importante, ya que el ciclo while evalúa la expresión condicional al principio del ciclo; en el caso de do-while ("hacer – mientras"), la expresión se evalúa al final del propio ciclo. La sintaxis de este bucle es:

```
do {  
acción;  
} while (expr.lógica);
```

Debido a la manera como funciona el ciclo do-while, el código en el cuerpo del ciclo se ejecuta al menos una vez, sin importar el valor de la condición de prueba (ya que se evalúa al final del ciclo). En el caso del while, la condición se evalúa al principio, por lo que es posible que nunca se ejecute el cuerpo del ciclo.

También en este caso, la modificación de la expresión lógica debe ser explícita en el bloque de código, para que el ciclo finalice en algún momento, cuando la expresión condicional resulte falsa.

## **FUNCIONES I: DECLARACIÓN Y DEFINICIÓN**

Las funciones son un conjunto de instrucciones que realizan una tarea específica. En general toman unos valores de entrada, llamados parámetros y proporcionan un valor de salida o valor de retorno; aunque tanto unos como el otro pueden no existir. Tal vez sorprenda que las introduzca tan pronto, pero como son una herramienta muy valiosa, y

se usan en todos los programas C++, creo que debemos tener, al menos, una primera noción de su uso.

Al igual que con las variables, las funciones pueden declararse y definirse. Una declaración es simplemente una presentación, una definición contiene las instrucciones con las que realizará su trabajo la función. En general, la definición de una función se compone de las siguientes secciones, aunque pueden complicarse en ciertos casos:

Opcionalmente, una palabra que especifique el tipo de almacenamiento, puede ser "extern" o "static". Si no se especifica es "extern". No te preocupes de esto todavía, de momento sólo usaremos funciones externas, sólo lo menciono porque es parte de la declaración. Una pista: las funciones declaradas como extern están disponibles para todo el programa, las funciones static pueden no estarlo.

El tipo del valor de retorno, que puede ser "void", si no necesitamos valor de retorno. En C, si no se establece, por defecto será "int", aunque en general se considera de mal gusto omitir el tipo de valor de retorno. En C++ es obligatorio indicar el tipo del valor de retorno.

Modificadores opcionales. Tienen un uso muy específico, de momento no entraremos en este particular, lo veremos en capítulos posteriores.

El nombre de la función. Es costumbre, muy útil y muy recomendable, poner nombres que indiquen, lo más claramente posible, qué es lo que hace la función, y que permitan interpretar qué hace el programa con sólo leerlo. Cuando se precisen varias palabras para conseguir este efecto existen varias reglas aplicables de uso común. Una consiste en separar cada palabra con un "\_", la otra, que yo prefiero, consiste en escribir la primera letra de cada palabra en mayúscula y el resto en minúsculas. Por ejemplo, si hacemos una función que busque el número de teléfono de una persona en una base de datos, podríamos llamarla "busca\_telefono" o "BuscaTelefono".

Una lista de declaraciones de parámetros entre paréntesis. Los parámetros de una función son los valores de entrada (y en ocasiones también de salida). Para la función se comportan exactamente igual que variables, y de hecho cada parámetro se declara igual que una variable. Una lista de parámetros es un conjunto de declaraciones de parámetros separados con comas. Puede tratarse de una lista vacía. En C es preferible usar la forma "func(void)" para listas de parámetros vacías. En C++ este procedimiento se considera obsoleto, se usa simplemente "func()".

Un cuerpo de función que representa el código que será ejecutado cuando se llame a la función. El cuerpo de la función se encierra entre llaves "{}". Una función muy especial

es la función "main". Se trata de la función de entrada, y debe existir siempre, será la que tome el control cuando se ejecute un programa en C.

ejemplo: int Mayor(int a, int b);

Sirve para indicar al compilador los tipos de retorno y los de los parámetros de una función, de modo que compruebe si son del tipo correcto cada vez que se use esta función dentro del programa, o para hacer las conversiones de tipo cuando sea necesario.

Los nombres de los parámetros son opcionales, y se incluyen como documentación y ayuda en la interpretación y comprensión del programa. El ejemplo de prototipo anterior sería igualmente válido y se podría poner como:

int Mayor(int,int);

Esto sólo indica que en algún lugar del programa se definirá una función "Mayor" que admite dos parámetros de tipo "int" y que devolverá un valor de tipo "int". No es necesario escribir nombres para los parámetros, ya que el prototipo no los usa. En otro lugar del programa habrá una definición completa de la función.

Una definición de la función "Mayor" podría ser la siguiente:

```
int Mayor(int a, int b)
{
if(a > b) return a; else return b;
}
```

## EJERCICIOS

**1 ejemplo: /\*MAYOR Y MENOR DE UN CONJUNTO DE NÚMEROS  
\*\*NORKIS PÉREZ \*\*/**

```
#include<conio.h>
#include<stdio.h>
main (void)
{
int n,may,men,x;
may=0;
men=10000;
for(x=1;x<=8;x++)
{
printf("\nINGRESE EL VALOR %d=> ",x);
```

```
scanf("%d",&n);
if (may<n)
{
may=n;
}
if (men>n)
{
men=n;
} }
printf("NUMERO MAYOR ES %d\n Y MENOR ES %d\n",may,men);
getch();
return 0;
}
```

**2 ejemplo: //MENU DE OPCIONES \*\*NORKIS PÉREZ\*\***

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

main()
{
char opcion;

printf("MENU DE OPCIONES\n");
printf ("1.- CREACION\n");
printf("2.- MODIFICACION\n");
printf ("3.- ELIMINACION\n");
printf ("0.- SALIDA\n");
printf ("SU OPCION ? \n");
opcion=getche();/*permite la captura de cualquier
caracter numerico o alfabetico*/
printf( "\n\n");
switch(opcion)
{
case '1': printf("RUTINA DE CREACION\n");break;
case '2': printf("RUTINA DE MODIFICACION\n");break;
case '3': printf("RUTINA DE ELIMINACION\n");break;
```

UNEFA EXT-MARACAY  
PROGRAMACIÓN V-ING. DE SISTEMAS  
AUXILIAR DOCENTE NORKIS PÉREZ P. 03/05/2009

```
case '0': printf ("SALIDA AL SISTEMA OPERATIVO\n");break;
```

```
default: printf("OPCION INVALIDA.....\n");
```

```
}
```

```
getch();
```

```
}
```

```
/*PROGRAMA QUE DEFINE SI EL ALUMNO APROBÓ O REPROBÓ LA  
MATERIA DE UNA LISTA DE ALUMNOS POR NORKIS PÉREZ*/
```

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

```
#include<conio.h>
```

```
int main()
```

```
{
```

```
int cal,alum,i,cont1=0,cont2=0;
```

```
printf("Ingrese el numero de alumnos: \n");
```

```
scanf("%d",&alum);
```

```
for(i=0;i<=alum;i++)
```

```
{
```

```
printf("Ingrese su calificación: \n");
```

```
scanf("%d",&cal);
```

```
if (cal>=60)
```

```
{
```

```
printf("Alumno Aprobado\n");
```

```
cont1=cont1+1;
```

```
}
```

```
else
```

```
{
```

```
printf("Alumno Reprobado\n");
```

```
cont2=cont2+1;
```

```
}
```

```
}
```

```
printf("Cantidad de aprobados: %d\n",cont1);
```

```
printf("Cantidad de reprobados: %d\n",cont2);
```

```
if(cont2>=10)
```

```
{
```

```
printf("debe repetir la materia");
```

```
}  
getch();  
}  
  
//PROGRAMA CON EL MANEJO DE FUNCIONES  
  
#include<stdio.h>  
#include<conio.h>  
int mayor(int a);  
int main()  
{  
    int nota;  
    printf("Ingrese la nota ");  
    scanf("%d",&nota);  
    mayor(nota);  
    getch();  
}  
int mayor(int a)  
{  
    if(a>=10)  
    {  
        printf("Es aprobado");  
    }  
    else  
    {  
        printf("Es reprobado");  
    }  
}
```

## APUNTADORES

### Apuntadores en C

1.-Un Apuntador es una variable que contiene una dirección de memoria, la cual corresponderá a un dato o a una variable que contiene el dato. Los apuntadores también

deben de seguir las mismas reglas que se aplican a las demás variables, deben tener nombre únicos y deben de declararse antes de usarse.

Cada variable que se utiliza en una aplicación ocupa una o varias posiciones de memoria. Estas posiciones de memoria se accedan por medio de una dirección.

En la figura el texto Hello ésta guardado en memoria, comenzando en la dirección 1000. Cada carácter ocupa un espacio de dirección único en memoria. Los apuntadores proporcionan un método para conservar y llegar a estas direcciones en memoria. Los apuntadores facilitan el manejo de datos, debido a que conservan la dirección de otra variable o ubicación de datos.

1.2 Operadores de Indirección y Dirección. Hay 2 operadores que se usan cuando trabajan con direcciones en un programa C; el Operador de Indirección ( \* ) y el de Dirección (&). Estos operadores son diferentes de los tratados anteriormente.

El Operador de Dirección ( & ) regresa la dirección de una variable. Este operador está asociado con la variable a su derecha: &h; Esta línea regresa la dirección de la variable h.

El Operador de Indirección ( \* ) trabaja a la inversa del operador de Dirección. También esta asociado con la variable a su derecha, toma la dirección y regresa el dato que contiene esa dirección. Por ejemplo, la siguiente línea determina la dirección de la variable h y luego usa el operador de Indirección para acceder la variable y darle un valor de 42:

```
*(&h)=42;
```

**La declaración de un puntero de manera general es:**

```
tipo *nombre de apuntador;
```

Tipo : Especifica el tipo de objeto apuntado y puede ser cualquier tipo.

Nombre de apuntador: Es el identificador del apuntador.

El espacio de memoria requerido para un apuntador, es el número de bytes necesarios para especificar una dirección de memoria, debiendo apuntar siempre al tipo de dato correcto.

**EJEMPLO:**

```
#include<stdio.h>
#include<conio.h>
int main()
{
    int numero;
```

```
numero=43;  
printf("DIRECCION DE NUMERO= %p,valor de NUMERO= %i\n",numero,numero);  
getch();  
}
```

### EJEMPLO 2:

```
//puntero de dos variables q no son iguales  
#include<conio.h>  
#include<stdio.h>  
int main()  
{  
int a,b;  
int *put1,*put2;  
a=5,b=5;  
put1=&a;  
put2=&b;  
if (put1==put2)  
printf("Son iguales");  
else  
printf("No son iguales");  
getch();  
}
```

### EJEMPLO 3 /\* punteros a array\*/

```
#include<conio.h>  
#include<stdio.h>  
int main()  
{  
int i;  
int temp [10]={5,6,8,2,3,4,6,8,9,6};  
int *put;  
put=temp;  
for (i=0;i<10;i++)  
{  
printf("Elemento %d %d \n",i,*put);  
put++;  
}  
getch();}
```