PROGRAMACIÓN ESTRUCTURADA

Guía Teórica

Introducción al lenguaje C

Índice de contenido

El preprocesador

Transforma el programa fuente, convirtiéndolo en otro archivo fuente con ciertas modificaciones.
Las transformaciones incluyen:

- Eliminar los comentarios.
- Incluir en el fuente el contenido de los ficheros declarados con `#include <fichero>` (a estos ficheros se les suele llamar cabeceras).
- Sustituir en el fuente las macros declaradas con #define (ej. `#define CIEN 100`)

El compilador

Convierte el código fuente entregado por el preprocesador en un archivo codificado en lenguaje máquina denominado fichero objeto. Algunos compiladores pasan por una fase intermedia en lenguaje ensamblador.

El enlazador

Un fichero objeto es código máquina, pero no se puede ejecutar, porque le falta código que se encuentra en otros archivos binarios.
El enlazador genera el ejecutable binario, a partir del contenido de los ficheros objetos y de las bibliotecas.

Bibliotecas estándares

La **biblioteca estándar de C** (también conocida como **libc**) es una recopilación de ficheros cabecera y bibliotecas con rutinas, estandarizadas por el comité de la Organización Internacional para la Estandarización (ISO), que implementan operaciones comunes tales como las de entrada y salida o el manejo de cadenas, entre muchas más.
Las interfaces de estos servicios vienen definidas en ficheros denominados cabeceras (*header files*). El nombre de estos ficheros suelen terminar en **`.h`**
Algunos de los servicios proporcionados por las bibliotecas estándares son:

- entrada y salida de datos `<stdio.h>`
- manejo de cadenas `<string.h>`
- memoria dinámica `<stdlib.h>`
- rutinas matemáticas `<math.h>`

Se encuentran en un programa antes de la palabra reservada main, invocadas con la directiva #include, de la siguiente forma:

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

```
#include <stdlib.h>
```

Comentarios

El comentario es texto que se puede escribir entre indicadores **el cual no formará parte del código de programación**. Los indicadores son `/*  */` para colocar entre ellos el texto comentarios o el `//` donde se puede colocar texto sólo en una línea:
ejemplos:

```
/*
Esto es un comentario
 multilinea
*/

//este es un comentario en una linea
```

Constante

La constante es un objeto básico el cual representa una espacio en memoria con un valor dado prefijado al inicio del programa y que no se puede modificar en tiempo de ejecución del programa. Su definición se realiza a través de directiva `#define`. El estilo de escritura es : *todo en mayúscula separando las palabras con guiones bajos.*

```
#define DIAS_SEMANA 7
#define MESES_ANIO 12
```

Variables

Una variable es uno de los objetos básicos que se manipulan en un programa. Presenta una espacio de memoria de un tamaño determinado por el tipo de dato. Se definen por lo general dentro de la función correspondiente y se le puede cambiar su valor asignándole el deseado en tiempo de ejecución. El estilo de escritura es (*camel case*): *todo en minúscula; si la variable está formada por varias palabras, la primera letra de cada palabra debe estar en mayúscula a excepción de la primer letra de la primer palabra.*

Sintaxis:

```
tipoDeDato nombreVariable;
```

Ejemplo:

```
int num;
double real;
char caracter;
int diasLaborables;
```

printf

Es la función del lenguaje que permite escribir texto (letras, números o símbolos) en la salida estándar (Pantalla o monitor de la computadora).

Sintaxis:

```
printf ( “cadena de formato y/o el texto en si”, arg1, arg2, ... argN );
```

En la `cadena de formato`: se escribe el texto que se desea imprimir, caracteres especiales, secuencias de escape, indicadores del formato de los argumentos.
Los argumentos `(arg1..)`: son expresiones válidas para el lenguaje (como variables; o funciones que retornan algún valor utilizable. Para usar `printf`, hay que escribir al principio del programa la directiva `#include <stdio.h>`

Ejemplos:

```
printf (“hola mundo”);
printf (“la cantidad a pagar es: %d pesos”, precio ); /*precio es una variable integer*/
```

scanf

Es la función del lenguaje que permite leer texto (letras, números o símbolos) de la entrada estándar (Generalmente teclado). Simplemente permite capturar lo que se escribe por teclado.

Sintaxis:

```
scanf ( “cadena de formato”, & arg1, & arg2, ... );
```

En `Cadena de formato`: se especifica qué tipo de datos se quieren leer. Se utiliza la misma descripción de formato que en `printf`. Además hay que incluir la cabecera `<stdio.h>`
Los argumentos `(arg1..)`: son expresiones válidas para el lenguaje (variables) que se utilizan para guardar la información ingresada por el teclado.

Ejemplos:

```
scanf ( “%d”, & numero1); /*numero 1 es una variable entera*/
scanf ( “%d,%d”, & numero1,& numero2 ); /*numero 1 y numero 2 son variables entera*/
```

***Nota**: Es obligatorio anteponer el ampersand (&) para variable tipo `int`, `char`, `double`, entre otras.*

getchar

`getchar()`: Es una función que lee el carácter ingresado por el teclado luego de presionar el enter. Si se ingresan varios caracteres sólo retornará el primero egresado..

Ejemplo:

```
c=getchar(); /* asigna en la variable char c el valor ingresado por el teclado*/
```

getche

`getche()`: Es una función que lee el carácter ingresado por el teclado. Se ejecuta inmediatamente después de apretar la tecla. Es ideal para ingresar caracteres (caracter a caracter) combinándolo con un ciclo. Requiere la librería `#include <conio.h>`
IMPORTANTE: Es sólo compatible con windows.

Ejemplo:

```
c=getche(); /* asigna en la variable char c el valor ingresado por el teclado*/
```

if

Es una expresión que evalúa una condición (pregunta lógica). Luego de evaluarse determina el valor de verdad de la condición (V= Verdadero o F=Falso). De acuerdo al resultado de la evaluación de la condición permite la ejecución del bloque de código V o del bloque de código F (ambos mutuamente excluyente).

elseif()

Sintaxis:

```
if (condición_lógica)
{

   bloques_de_sentencias_si_condición_verdadera;

}
else
{
   bloques_de_sentencias_si_condición_falsa;
}
```

Ejemplo:

```
if(numero>=0)
{
    printf("%d es un numero positivo o neutro", numero);
```

```
}
else
{
    printf(“%d es un numero positivo o neutro”, numero);
}
```

switch

Es una expresión que evalúa el valor de la `variable_de_condición`, luego compara ese valor con los distintos valores correspondientes a cada caso (*`case valor1:`…, `case valor2:`…etc*), de encontrar exactamente el valor de condición en algún caso (*match=coincidencia*), ejecutará el código correspondiente al bloque cuya condición realizó (*match*). En el caso de no encontrar coincidencia se ejecutará el bloque `default`, de estar el mismo escrito.

Sintaxis:

```
switch(variable_de_condición)
{
    case valor1:
         bloque_de_sentenciasA;
       break;
    case valor2:
         bloque_de_sentenciasB;
       break;
            .
            .
    case valorN:
          bloque_de_sentenciasN;
       break;

   defalut:
         bloque_de_sentenciasD;
       break;
}
```

Las sentencias A se ejecutarán si `variable_de_condición` adquiere el `valor1`.
Las sentencias B se ejecutarán si `variable_de_condición` adquiere el `valor2`.
Las sentencias N se ejecutarán si `variable_de_condición` adquiere el `valorN`.
Cualquier otro valor de expresión conduce a la ejecución del `bloque_de_sentenciasD`; eso viene indicado por la palabra reservada `default`.

Ejemplo:

```
switch ( numero1 ) /*numero1 es una variable integer*/
{
   case 1:
       printf(“El numero ingresado es UNO”);
       break;
   case 10:
       printf(“El numero ingresado es DIEZ”);
       break;
```

```
   case 100:
       printf(“El numero ingresado es CIEN”);
       break;
   default;
       printf(“El numero ingresado NO es CIEN NI DIEZ NI UNO”);
       break;
}
```

for

El ciclo de repetición `for` repite el `bloques_de_sentencias` que está dentro de él, hasta que la `condición_lógica` sea falsa. Para que `for` “cicle”, es condición necesaria que `condición_lógica` sea verdadero.
La `expresión_inicial` se ejecuta sólo la primera vez que se entra al ciclo.
La *`expresión_de_paso`* se ejecuta todas las veces que se realiza un ciclo y antes que se evalúe `condición_lógica;` pero no se ejecuta a la entrada del ciclo (la primera vez).
Se debe tener en cuenta la correcta confección de la `condición_lógica` y que exista una `expresión_de_paso` adecuada para no incurrir en un ciclo infinito.

Sintaxis:

```
for ( expresión_inicial; condición_lógica; expresión_de_paso )
{
   bloques_de_sentencias;
}
```

Ejemplo:

```
main()
{
   int i;
   for ( i=0; i<10; i++ )
   {
      printf (“%d \n“, i );
   }
}
```

while

El ciclo de repetición `while` repite el `bloques_de_sentencias` que está dentro de él, hasta que la `condición_lógica` sea falsa. Para qué `while` comience a “ciclar” es condición necesaria que `condición_lógica` sea verdadero.
Se debe tener en cuenta la correcta confección de la `condición_lógica` y el manejo de los cambios de estado de la `condición_lógica` dentro del bloque de ejecución del `while` para no incurrir en un ciclo infinito.

Sintaxis:

```
while (condición_lógica)
{
     bloque_de_sentencias;
}
```

Ejemplo:

```
main()
{
    int x=1;
    while ( x < 10 )
    {
        printf("Línea número %d\n",x);
        x++;
    }
}
```

do while

Sintaxis:

```
do{
    bloques_de_sentencias;
}while( condición_lógica );
```

Ejemplo:

```
main()
{
    int x=1;
    do{
        printf("Línea número %d\n",x);
        x++;
    }while ( x < 10);
}
```

Arreglos

Los arreglos o vectores es una zona de almacenamiento contiguo de elementos de un mismo tipo de dato. Se declara de la misma forma que una variable (*de hecho es una variable*) y se le agrega a la definición, la cantidad de elementos que contendrá.
Conceptualmente podemos modelar una fila con n elemento al cual se lo conoce como un vector de n elementos; o podemos modelar una matriz de n filas y m columnas a la cual se la conoce como matriz de n x m.

Sintaxis: de la declaración de un arreglo de números enteros de una dimensión (*osea un vector*).

```
int nombreDelArreglo[cantidadDeElementos];
```

Sintaxis: de la declaración de un arreglo de números enteros de dos dimensiones (*osea una matriz*).

```
int nombreDelArreglo[filas][columnas];
```

Ejemplo: declaración.

```
int nota[30]; /*declaración de un arreglo de una dimensión*/
double temperaturaDiaMes[31][7]; /*declaración de un arreglo de dos dimensiones*/
```

Como los arreglos son variables, su contenido podrá ser asignado (escribir) o invocado (leer). Para poder alcanzar el espacio de memoria donde se almacena la información deberemos indicar (dentro de los corchetes) el número de elemento de la variable al cual nos referimos para leer o escribir.

Ejemplo: asignación de un arreglo

```
nota[2]=10                      /*guardo en la posición 2 el número 10*/
nota[0]=9                       /*guardo en la posición 0 el número 9*/
temperaturaDiaMes[0][18]=26,3;  /*guardo en la columna 0 fila 18 el número 26*/
```

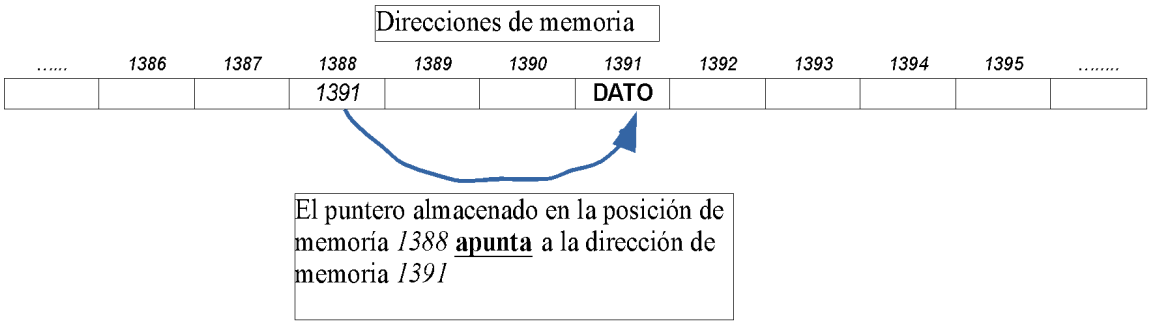
Ejemplo: lectura de un arreglo

```
/*Imprime el valor leído en la posición 2 de la variable nota*/
printf("la nota es: %d", nota[2]);

/*Imprime el valor leído en la columna 0 fila 18 de la variable temperaturaDiaMes*/
printf ("la temperatura es %lf", temperaturaDiaMes[0][18]);
```

Punteros

Un puntero es una variable que tiene la capacidad de almacenar una dirección de memoria. *Por ejemplo: La dirección de memoria de otra variable.*
Cuando un programa está en ejecución, cada dato de del mismo está almacenado en memoria la memoria principal, y dentro de esta en una ubicación específica que se denomina "dirección de memoria".
En una máquina típica, la memoria se puede expresar como una secuencia de celdas numeradas secuencialmente con un número el cual representa la dirección en memoria (ubicación). Dentro de cada celda se guarda el contenido deseado de acuerdo a su tipo.



Sintaxis:

```
TipoDeDato* nombreVariable; /*Declaración de una variable puntero*/
```

Una variable puntero siempre contiene una dirección de memoria del tipo de dato del cual ha sido declarada.

Ejemplos:

```
char* m;    /* Esta es una variable puntero que "apunta" a una dirección que contiene un char */
int* n;     /* Esta es una variable puntero que "apunta" a una dirección que contiene un int  */
double* p;  /* Esta es una variable puntero que "apunta" a una dirección que contiene un double */
```

Operadores & y *

OPERADOR	DESCRIPCIÓN	SE APLICA A
&	*Dirección de*	*A cualquier variable que se quiera saber su dirección de memoria*
*	*Contenido de*	*Se aplica a cualquier variable **puntero** la cual se desea saber el contenido de la dirección que almacena el puntero.*

Ejemplo: &

```
dirA=&a;
```

El operador & actúa sobre una variable y retorna su dirección de memoria. *La variable* **dirA** *es del tipo puntero y se le asignó la dirección de memoria donde se encuentra alojada la variable* ***a.***

Ejemplo(1): *

```
b=*dirA;
```

El operador * actúa sobre variables que contienen direcciones de memoria (punteros) y retorna el contenido de la posición de memoria de la variable sobre la que actúa. Si **dirA** es una variable que almacena una dirección de memoria, la expresión almacena en la variable **b** el "contenido de la posición de memoria" apuntada por **dirA**.

Ejemplo(2): *

```
*dirA=80;
```

Se almacena el valor **80** en la "dirección de memoria apuntada" por **dirA.**

Ejemplo: General.

```
int main()
{
    int a=50,*dirA;
    dirA=&a;
    printf("El contenido de a es %d\n",a);
    printf("El contenido de a es %d\n",*dirA);
    printf("El contenido de a es %d\n",*(&a));
    printf("La direccion de a es %p\n",&a);
    printf("La direccion de a es %p\n",dirA);
    printf("La direccion de a es %p\n",&(*dirA));

    return 0;
}
```

Salida por consola::

```
El contenido de a es: 50
El contenido de a es: 50
El contenido de a es: 50
La dirección de a es: 0022FF44
La dirección de a es: 0022FF44
La dirección de a es: 0022FF44
```

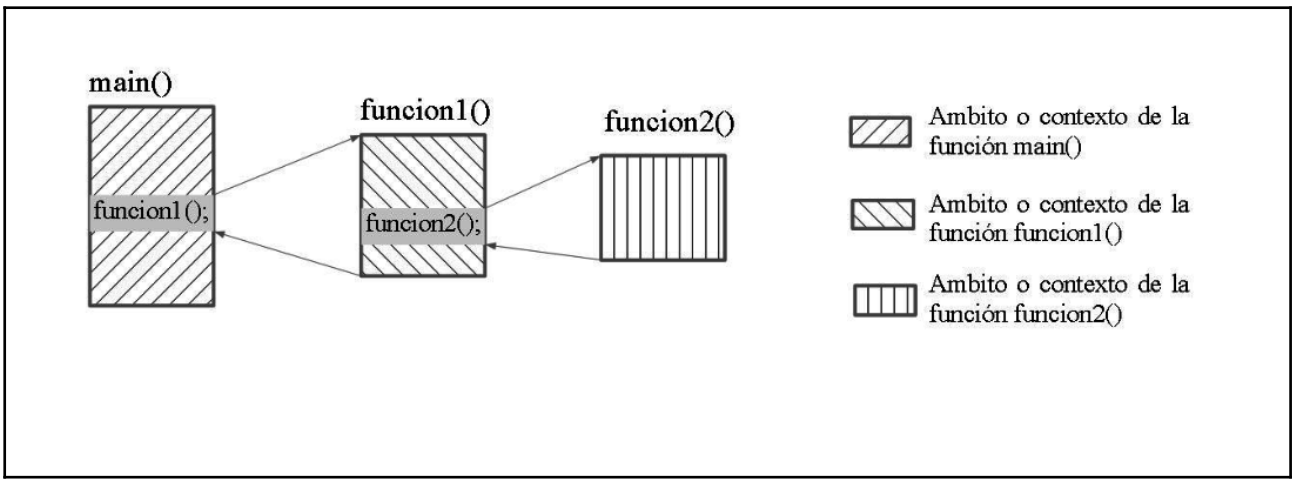
Funciones

Prototipo genérico de una función:

<tipo de dato de retorno> **nombeFuncion (** *<Argumentos>***)**

Una función o subrutina es un conjunto de líneas (sentencias) de código identificadas con un nombre (*nombre de la función*) la cual puede ser invocada a través de su nombre desde cualquier parte de un programa (*si está definida en el contexto del dicho programa*) y así ejecutarse todas las líneas de código que contiene.
Una función puede ser desarrollada para que retorne o no, un valor de un tipo de dato válido. Además, a una función se le puede enviar o pasar argumentos (o parámetros). Estos son variables de tipo de dato válido que se utilizan para hacer **"una comunicación**" entre una función (que es la llamada) y otra función (que es la llamadora); pasando valores de variables (1) o valores de direcciones de variables (2).
Hay dos formas bien conocidas de pasar valores a una función (Por Valor(1) y Por Dirección(2)):

Esquema General:



(1) <u>Por Valor</u>: Es cuando se pasa el valor de una variable de interés. Esto lleva a que la función llamada tenga el valor enviado desde afuera por la función llamadora. El valor recibido se guarda en una variable del mismo tipo del enviado (*dentro del contexto de la función llamada*). Toda modificación que la función llamada realice en la variable que contiene el valor enviado **NO se verá afectado** dicho valor original de la variable en el contexto de la función llamadora.

(2) <u>Por dirección</u>: Es cuando se pasa el valor de la dirección de memoria de la variable de interés. Esto lleva a que la función llamada tenga la dirección de memoria de la variable del contexto de la función llamadora (*e inclusive de otro contexto*). Toda modificación que la función llamada realice en el contenido de la dirección de memoria recibida **SI se verá afectado** el valor original de la variable en el contexto de la función llamadora (*e inclusive de otro contexto*).

Sobre el retorno de una función: Una función llamada puede o no retornar valor al contexto de la función que la llamó. Se especifica siempre en la definición de la función el <tipo de dato de retorno>. Con la palabra reservada **`void`** se indica que una función no retorna valor. Y con un <*tipo de dato válido*> se especifica cuando la función retorna un valor de dicho tipo de dato válido.

Sobre los argumentos de una función: Un función llamada puede o no poseer valores o argumentos. Dichos argumento especificaciones de definiciones de variables.

Ejemplos:

Función SIN valor de retorno y con pasaje de argumento por valor.	**Función CON valor de retorno y con pasaje de argumento por valor.**

`#include <stdio.h>`
`#include <stdlib.h>`
 	
`/*declaración de la cabecera de la función*/`
`void sumar( int numero1, int numero2);`
 	
`int main()`
`{`
`    int num1, num2;`
	`#include <stdio.h>`
	`#include <stdlib.h>`
	
	`/*declaración de la cabecera de la función*/`
	`int sumar( int numero1, int numero2);`
	
	`int main()`
	`{`
	`    int num1, num2, suma;`

```
    num1=13; /*asigno valor a num1*/
    num2=21; /*asigno valor a num2*/
    /*llamada a la función sumar*/
    sumar(num1,num2);
    return 0;
}

/*codigo de la función sumar*/
void sumar ( int numero1, int numero2)
{
    int suma;
    suma= numero1 + numero2;
    printf("El numero 1 es: %d\n", numero1);
    printf("El numero 2 es: %d\n", numero2);
    printf("La suma es    : %d\n", suma);
}
```

```
    num1=13;/*asigno valor a num1*/
    num2=21;/*asigno valor a num2*/
    /*llamada a la función sumar*/
    suma=sumar(num1,num2);
    printf("El num1 es    : %d\n", num1);
    printf("El num2 es    : %d\n", num2);
    printf("La suma es    : %d\n", suma);

    return 0;
}

/*codigo de la función sumar*/
int sumar ( int numero1, int numero2)
{
    int suma;
    suma= numero1 + numero2;

    return suma;
}
```

Ejemplo:

Función SIN valor de retorno y con pasaje de argumento por valor y por dirección.

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

/*declaración de la cabecera de la función*/
void sumar(int numero1, int numero2, int* pSuma);

int main()
{
    int num1, num2,suma;
    num1=13; /*asigno valor a num1*/
    num2=21; /*asigno valor a num2*/
    /*llamada a la función sumar*/
    sumar(num1,num2,&suma);

    printf("El num 1 es   : %d\n", num1);
    printf("El num 2 es   : %d\n", num2);
    printf("La suma es    : %d\n", suma);

    return 0;
}

/*codigo de la función sumar*/
void sumar ( int numero1, int numero2, int* pSuma)
{
    int suma;
    *pSuma= numero1 + numero2;
}
```

struct

La palabra reservada `struct` es utilizada para definir nuestras propias estructuras de datos. Cuando definimos una estructura de datos, estamos indicando al lenguaje cómo se organizan los datos en memoria cuando utilicemos dichas estructuras. Además, el lenguaje c provee un

mecanismo para definir “alias de tipos” utilizando la palabra reservada `typedef`. Un alias de tipo no es otra cosa
que un sinónimo para referirse a un tipo de dato ya existente o predefinido por el programador.

Sintaxis:

Sin typedef	Con typedef

`#include <stdio.h>`
`struct nombreEstructura`
`{`
`  [miembros];`
`};`
 	
`int main()`
`{`
`  struct nombreEstructura nombreVariable;`
`.`
`.`
`.`
	`#include <stdio.h>`
	`struct nombreEstructura`
	`{`
	`  [miembros];`
	`};`
	`typedef struct nombreEstructura nombreTipoDato;`
	
	`int main()`
	`{`
	`  nombreTipoDato nombreVariable;`
	`.`
	`.`
	`.`

Ejemplo: completo

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

struct sPunto
{
   double x;
   double y;
   double z;
};
typedef struct sPunto tPunto;
imprimirPunto(tPunto p);

int main()
{
   tPunto p1, p2;
   p1.x = 2.1;
   p1.y = 3;
   p1.z = 0;
   scanf("%lf", &p2.x);
   scanf("%lf", &p2.y);
   scanf("%lf", &p2.z);
   p1.x = p1.x + p2.x;
   p1.y = p1.y + p2.y;
   p1.z = p1.z + p2.z;
   imprimirPunto(p1);
   return 0;
}
imprimirPunto(tPunto p)
{
    printf("\n P1(x,y,z) = (%.2lf, %.2lf, %.2lf)\n",p.x, p.y, p.z);
}
```

Nota:Se observa que para acceder a cada componente del struct se utiliza el operador ***.*** *(punto.)*

Ejemplo: definición de estructuras complejas

```
struct sFecha
{
   int dia, mes, anio;
};
struct sEmpleado
{
   int legajo;
   char nombre[40];
   char apellido[50];
   struct sFecha fechaNac;
   struct sFecha fechaIng;
};
```

Tipos de datos predefinidos (Algunos)

TABLA CON ALGUNOS TIPOS DE DATOS PREDEFINIDOS EN C			
ENTEROS: números completos y sus negativos			
Palabra reservada:	**Ejemplo**	**Tamaño (byte)**	**Rango de valores**
ENTEROS: números enteros – algunas variantes			
int	*-850*	*4*	*$-2*10^{31}+1$ a $2*10^{31}-1$*
short	*-10*	*2*	*$-2*10^{15}+1$ a $2*10^{15}-1$*
unsigned int	*45689*	*4*	*0 a $2*10^{32}-1$*
long long	*5689*	*8*	*$-2*10^{63}+1$ a $2*10^{63}-1$*
REALES: números con decimales o punto flotante			
float	85.3	4	$3.4x10^{-38}$ a $3.4x10^{38}$
double	20.3	8	$1.7x10^{-308}$ a $1.7x10^{308}$
long double	58.0	10	$3.4x10^{-4932}$ a $1.1x10^{4932}$
CARÁCTER: letras, digitos, símbolos, signos de puntuación.			
char	'O'	1	-127 ... 127 o 0 ... 255 (si es unsigned)
Algunos tamaño (de reales o enteros) y por consecuencia el rango de valores podrán variar en función del modelo de datos que la plataforma utilice. Para obtener información sobre el tamaño de una variable utilizar la función sizeof (). De todos modos la información consignada es la más usual para computadoras de uso comercial (32 o 64 bit – SO windows o Linux) – [año 2015]			

Formatos

Código	Formato
%d	*Entero decimal con signo.*

`%i`	*Entero decimal con signo.*
`%c`	*Carácter sencillo*
`%s`	*Cadena (char*. Los caracteres se imprimen hasta encontrar el '\0')*
`%f`	*Float: Real. Notación decimal [-] mmm.ddd*
`%ld`	*Entero largo*
`%u`	*Entero decimal sin signo.*
`%lf`	*Double: Real Doble precisión [-] mmm.ddd*
`%h`	*Entero corto*
`%o`	*Octal*
`%x`	*Hexadecimal*
`%e`	*Notación Científica*
`%p`	*Puntero*
`%%`	*Imprime Porcentaje*

Caracteres de escape

Carácter de Escape	Descripción
`\n`	Nueva línea
`\r`	Retorno de carro. (es la tecla enter)
`\t`	Tabulador horizontal. Mueve el cursor al próximo tabulador
`\v`	Tabulador vertical.
`\a`	Hace sonar la alarma del sistema
`\\`	Para imprimir un carácter de diagonal invertida
`\?`	Para imprimir el carácter del signo de interrogación
`\”`	Para imprimir una doble comilla

Operadores aritméticos

Operador	Propósito
`+`	Suma
`-`	Resta
`*`	Multiplicación
`/`	División
`%`	Resto de la división entera

Operadores binarios de relación o comparación

Operador	Significado
<	Menor que
<=	Menor o igual que
>	Mayor que
>=	Mayor o igual que
==	Igual que (Para realizar comparaciones)
!=	No es igual a

Operadores unarios

Operador	Propósito
-	Menos Unario: Es el signo menos que va delante de una variable, constante o expresión.
++	Operador Incremento: Hace que la variable, constante o expresión se aumente en uno.(Ej: `c++` es equivalente a `c=c+1`)
--	Operador Decremento: Hace que su variable, constante o expresión disminuye en uno.(Ej: `c--` es equivalente a `c=c-1`)

Operadores de asignación

Operador	Explicación
=	expresión 1=expresión 2. Equivale a transferir el valor de la expresión 2 a la expresión 1
+=	expresión 1=expresión 2. Equivale a: expresión 1=expresión 1 + expresión 2
-=	Ej.: `i-=1`. equivale `a: i=i-1`
*=	Ej.: `J*=2`. Equivale `a: j=j*2`
/=	Ej.: `K/=m`, equivale `a: k=k/m`
%=	Ej.: `p%n`. Equivale a: `p=p%n`

Código ASCII

Dec	Char	Dec	Chr	Dec	Chr	Dec	Chr
0	NUL (null)	32	Space	64	@	96	`
1	SOH (start of heading)	33	!	65	A	97	a
2	STX (start of text)	34	"	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL (bell)	39	'	71	G	103	g
8	BS (backspace)	40	(	72	H	104	h
9	TAB (horizontal tab)	41	)	73	I	105	i
10	LF (NL line feed, new line)	42	*	74	J	106	j
11	VT (vertical tab)	43	+	75	K	107	k
12	FF (NP form feed, new page)	44	,	76	L	108	l
13	CR (carriage return)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (device control 1)	49	1	81	Q	113	q
18	DC2 (device control 2)	50	2	82	R	114	r
19	DC3 (device control 3)	51	3	83	S	115	s
20	DC4 (device control 4)	52	4	84	T	116	t
21	NAK (negative acknowledge)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of trans. block)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[	123	{
28	FS (file separator)	60	<	92	\	124	|
29	GS (group separator)	61	=	93	]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL