PROGRAMACIÓN EN *C*

4.1.- Introducción

C, al igual que cualquier lenguaje humano natural o de programación informática, se compone de un conjunto de elementos y de un conjunto de reglas que permiten la comprensión del lenguaje combinándolos correctamente. Este conjunto de elementos y reglas de un lenguaje se denomina ***sintaxis del lenguaje***. Puede decirse, por lo tanto, que C dispone de su propia sintaxis.

Los elementos básicos a los que se ha hecho referencia en el párrafo anterior forman el ***léxico del lenguaje***. A partir del léxico y teniendo en cuenta las reglas sintácticas, pueden construirse elementos mas complejos del lenguaje a los que se llama ***programas***

Los elementos básicos que constituyen los programas C son los siguientes:

- **Palabras clave o reservadas**

Son un conjunto de palabras predefinidas, que se ***escriben siempre en minúsculas*** y que tienen un significado especial en el lenguaje C. Son pues las que constituyen realmente el lenguaje.

El estándar ANSI C define las siguientes 32 palabras clave:

auto	**double**	**int**	**struct**
break	**else**	**long**	**switch**
case	**enum**	**register**	**typedef**
char	**extern**	**return**	**union**
const	**Flota**	**short**	**unsigned**
continue	**for**	**signed**	**void**
default	**goto**	**sizeof**	**volatile**
do	**if**	**static**	**while**

- **Separadores**

Son usados para separar los diferentes elementos que forman el lenguaje. Se consideran separadores, en C, los ***espacios en blanco,*** los ***tabuladores,*** los ***retornos de carro***, los ***comentarios***, el ***punto y coma*** (***;***), las ***llaves*** (***{ }***), etc.

- **Operadores**

Representan las operaciones de tipo aritmético, lógico, relacional, de asignación, etc. Que se utilizan en el desarrollo de un programa.

- **Identificadores**

Son los nombres de los objetos (***variables, constantes, etc.***) definidos por el programador. Se componen de letras y números, si bien el primer carácter debe ser forzosamente una letra.

4.1.1.- Forma general de un programa en C

Todo programa en C consta de uno o más módulos llamados ***funciones.*** Una de las funciones ha de llamarse forzosamente ***main***. El programa siempre comenzará por la ejecución de la función ***main***, la cual puede acceder a las demás funciones. Las definiciones de las funciones adicionales se deben realizar aparte, bien precediendo o bien siguiendo a la función ***main.***

Cada función debe contener:

1. Una *cabecera* de la función, que consta del nombre de la función, seguido de una lista opcional de *argumentos* encerrados entre paréntesis

2. Una lista de *declaración* de argumentos, si se incluyen éstos en la cabecera

3. Una *instrucción compuesta* que contiene el resto de la función

Los argumentos son símbolos que representan información que se le pasa a la función desde otra parte del programa (a los argumentos también se les suele llamar *parámetros*).

Cada instrucción compuesta se encierra entre un par de llaves (*{ }*). Las llaves pueden contener combinaciones de instrucciones elementales (denominadas *instrucciones de expresión*) y otras instrucciones compuestas. Así, las instrucciones compuestas pueden estar anidadas, una dentro de otra. Cada instrucción de expresión debe acabar en un punto y coma.

Los comentarios pueden aparecer en cualquier parte del programa, mientras estén situados entre los delimitadores ***/*** y *******/*. Los comentarios son útiles para identificar los elementos principales de un programa o para explicar la lógica subyaciente de éstos.

Así, un programa C sencillo sería:

/* Programa para calcular el área de un círculo */	/* TITULO (COMENTARIO) */
#include <stdio.h>	/*ACCESO A ARCHIVO DE BIBLIOTECA */
Main()	/* CABECERA DE FUNCIÓN */
{	
float radio, area;	/* DECLARACIÓN DE VARIABLES */
printf ("Radio = ? ");	/* INSTRUCCIÓN DE SALIDA */
Scanf("%f", &radio);	/* INSTRUCCIÓN DE ENTRADA */

area = 3.14159 * radio * radio	/* INSTRUCCIÓN DE ASIGNACIÓN */
printf ("Area = %f ", area);	/* INSTRUCCIÓN DE SALIDA */
}	

Es interesante subrayar las siguientes características del programa indicado anteriormente:

1. El programa está escrito en minúsculas ya que se han utilizada palabras reservadas. Los comentarios también se suelen escribir en minúsculas salvo en el caso en el que se quiera hacer un especial énfasis en su contenido.

2. La primera línea es un comentario que describe el propósito del programa

3. La segunda línea contiene la referencia a un fichero especial (stdio.h) que contiene información que se debe incluir en el programa cuando se compila. La inclusión requerida de esta información será manejada automáticamente por el compilador.

4. La tercera linea es la cabecera de la función ***main***. Los paréntesis vacíos que siguen al nombre indican que esta función no incluye argumentos.

5. Las cinco líneas restantes está sangradas y encerradas entre un par de llaves. Estas cinco líneas integran la función compuesta dentro de ***main***.

6. La primera línea sangrada es una ***declaración de variables***. Establece los nombres simbólicos de ***radio y area*** como variables en coma flotante.

7. Las otras cuatro líneas sangradas son ***instrucciones de expresión***

8. La cuarta línea sangrada es un tipo especial de instrucción de expresión llamada ***instrucción de asignación***

9. Toda instrucción de expresión dentro de una instrucción compuesta acaba en un punto y coma. Esto es necesario en toda instrucción de expresión.

4.2.- Características deseables de un programa

Son características importantes para considerar un programa como bien escrito, no solamente en lenguajes C sino en cualquier lenguaje de programación:

1.- ***Integridad***. Referida a la corrección de los cálculos. Toda posible ampliación de un programa tendrá sentido si los cálculos se realizan de forma correcta.

2.- ***Claridad***. Considerada como la facilidad de lectura de un programa en conjunto, con particular énfasis en la lógica subyacente. Si el programa está escrito de forma clara, será posible para otro programador seguir la lógica del programa sin mucho esfuerzo. Uno

de los objetivos al diseñar C fue el desarrollo de programas claros y de fácil lectura, a través de un enfoque de la programación ordenado y disciplinado.

3.- ***Sencillez***. La claridad y corrección de un programa suelen verse favorecidas por hacer las cosas de forma tan sencilla como sea posible, consistente con los objetivos del programa en su conjunto. De hecho, suele ser deseable sacrificar cierta cantidad de eficiencia computacional con vistas a no complicar la estructura del programa.

4.- ***Eficiencia***. Relacionada don la velocidad de ejecución del programa y utilización eficiente de la memoria. Es un objetivo importante a conseguir aunque no debe hacerse a expensas de la claridad o la sencillez.

5.- ***Modularidad***. Relacionada con el hecho de obtener programas lo mas generalizados posibles dentro de un margen razonable.

4.3.- Elementos del Lenguaje.

4.3.1.- Caracteres

El conjunto de caracteres del lenguaje C son:

- **Letras: A...Z, a...z.**
- **Dígitos: 0...9.**
- **Carácter de subrayado (guión bajo): _**

Estos caracteres son utilizados para la creación de constantes, identificadores y palabras clave. El compilador de C distingue las mayúsculas y minúsculas.

Son caracteres con un particular interés:

- **Espaciadores o separadores:** Espacio blanco
- **Tabulador horizontal** HT.
- **Tabulador vertical** VT.
- **Avance de página.** FF
- **Nueva línea** LF.
- **Indicador de fin de fichero:** CTRL+Z (MS-DOS), CTRL+D (UNIX).
- **Especiales:** , . ; : ? ' " () [] {} < ! || /\ + # % & ^ * - = >
- **Secuencias de escape:** Es un conjunto de caracteres no imprimibles. Estos caracteres se denominan secuencias de escape y están formados por el carácter "\" seguido de una letra o de una combinación de dígitos.

Secuencia	Acción
\n	Nueva línea
\t	Tabulación horizontal
\v	Tabulación vertical
\b	Backspace (retorno)

\r	Retorno de carro
\f	Alimentación de página (solo impresoras)
\a	Bell (alerta, pitido)
\'	Comilla simple
\"	Comilla doble
\\	Backslash (barra invertida)
\ddd	Carácter ASCII. Representación octal
\xdd	Carácter ASCII. Representación hexadecimal.

4.3.2.- Identificadores

Los identificadores son nombres creados para designar constantes, variables, tipos, funciones, etc., que forman parte de un programa.

Un identificador consta de uno o más caracteres (letras, dígitos y caracteres de subrayado); el primero debe ser una letra o el carácter _. El número de caracteres no debe superar 31.

En C/C++ hay diferencia entre mayúsculas y minúsculas. Entre todos los identificadores distinguimos un conjunto de ellos, predefinidos, con un significado especial para el compilador (palabras clave).

4.3.3.- Palabras Clave.

En C estándar las palabras clave son las siguientes:

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Además, cada compilador añade al conjunto de palabras estándar, las suyas de propia creación. Para el compilador de Borland son añadidas:

asm	-asm	cdecl	-cdecl
-cs	-es	-export	-far
-fastcall	-huge	huge	interrupt
-interrupt	loadds	-near	pascal
-pascal	-saveregs	-seg	-ss

4.3.4. Comentarios

Un comentario en un programa o fichero fuente es una línea escrita en código fuente sin efecto para el compilador. En lenguaje C podemos escribir comentarios de una línea y un bloque de comentario.

```
Línea:      /*     comentario     */
Bloque:     /*     comentario
                   comentario
                ....
                   cometario      */
```

4.4.- Tipos de Datos.

a) Tipos fundamentales o tipos base.

Se denominan tipos fundamentales o tipos base en un determinado lenguaje de programación, a aquellos tipos de datos que pueden ser utilizados en las distintas declaraciones de los elementos que intervienen en el programa sin necesidad de definirlos. Es decir, son aquellos tipos reconocidos directamente por el compilador.

En C los tipos fundamentales se reparten en 4 grupos:

- Enteros, con signo y sin signo.
- Reales en como flotante, en simple y doble precisión.
- Carácter.
- **Vacío.**

La norma ANSI prevé, en teoría, seis tipos de datos **enteros** obtenidos a partir de las palabras clave ***int***, ***short***, ***unsigned*** y ***signed,*** que reflejan características tales como tamaño de memoria ocupada por datos de un tipo determinado (tamaño), forma de representación binaria del dato sobre el espacio definido (magnitud-signo, complemento a dos, etc.).

La norma ANSI también prevé la existencia de tres tipos de **coma flotante**, obtenidos con las palabras clave ***float, double*** y ***long***, dos tipos de **carácter** con ***char, signed*** y ***unsigned*** y un tipo vacío **void**.

Para el C/C++ de Borland los tipos base o fundamentales se esquematizan en el cuadro siguiente:

Declaración de tipo	Tamaño en bytes	Rango	Límites (min-máx)	Fichero cabecera
signed char **char**	1	conjunto de caracteres ASCII		
unsigned char char	1	Conjunto de caracteres ASCII		
short short int **int** signed short signed short int signed int signed	2	-32768 a +32767	INT_MIN INT_MAX	limits.h
unsigned short unsigned short int **unsigned int** unsigned	2	0-65535	UINT_MIN UINT_MAX	limits.h
long long int signed long signed long int	4	-2 147 483 648 a 2 147 483 648	LONG_MIN LONG_MAX	limits.h
unsigned long unsigned long int	4	0-4 294 967 295	ULONG_MIN ULONG_MAX	limits.h
float	4	1.17 E-38 a 3.40 E+38 precisión 6 díg. 1.19 E-7	FLT_MIN FLT_MAX FLT_EPSILON	float.h
double	8	2.22 E-308 a 1.79 E+308 precisión 15 díg. 2.22 E-16	DBL_MIN DBL_MAX DBL_EPSILON	float.h
long double	10	3.6 E-4392 a 1.18 E+4392 precisión 19 díg. 1.08 E-19	LDBL_MIN LDBL_MAX LDBL_EPSILON	float.h
void	-	-	-	

En C existen cinco tipos básicos atómicos: **carácter** (char), **entero** (int), **coma flotante en simple precisión** (float), **coma flotante en doble precisión** (double) y **sin valor** (void) y cuatro modificadores que alteran el valor del tipo base (***signed, unsigned, long y short***); a partir de los tipos básicos y los modificadores se constituyen los tipos fundamentales que aparecen en el cuadro anterior.

b) Tipos enumerados.

La declaración de un tipo enumerado es simplemente una lista de valores que pueden ser asumidos por una variable del tipo definido. Los valores del tipo enumerado se representan mediante identificadores que serán las constantes del nuevo tipo.

Ejemplo:

```
enum dia_semana
{
 lunes,
 martes,
 miércoles,
 ...
 sábado,
 domingo
};
```

Con esta declaración el tipo dia_semana es ya conicido, podremos por lo tanto, definir variables del tipo anterior.

```
enum dia_semana ayer, hoy;
```

Estas variables pueden tomar cualquiera de los valores incluidos en la declaración de tipo.

Sintaxis:

```
enum nombre_de_tipo_enumerado
{
 lista de nombre de constantes enteras,
...
};
enum nombre _de_tipo_enumerado lista de variables...
```

Los valores asociados a los identificadores de constantes son 0 para la primera constante, 1 para la segunda, etc., aunque con una simple asignación, cuando se declara la constante, sirve para modificar estos enteros asociados.

En el ejemplo lunes vale 0, marte1, miércoles 2, etc.

c) typedef.

La palabra clave typedef permite declarar nuevos nombres de tipos de datos para tipos fundamentales o previamente definidos. Estos nuevos nombres pueden ser utilizados para sucesivas declaraciones.

Sintaxis:

typedef nombre_de_tipo nuevo_nombre_de_tipo

Ejemplo:

```
typedef int ENTERO;
 ...
ENTERO n;
```

d) Tipos estándar.

Algunos de los programas de las librerías de C utilizan datos cuyos tipos están definidos en ficheros cabecera (ficheros.h).

Ejemplo:

clock_t	definido en time.h:	typedef long coch_t;
size_t	definido en stdio.h:	typedef unsigned int size_t;
FILE	definido en stdio.h:	typedef strusct_iobuf FILE;

etc..

La utilización de tipos estándar de datos, favorece la portabilidad del código y la independencia de este código respecto de la máquina. Me olvido de los tipos particulares definidos en un compilador particular; utilizo su nombre estándar sabiendo que la definición del tipo en el fichero.h se encarga del resto.

4.5.- Constantes y Variables.

a) Constantes.

Una constante es un dato que no cambia durante la ejecución del programa; existen constantes numéricas, de carácter y de cadenas de caracteres.

Las constantes numéricas pueden ser enteras y en coma flotante; las enteras pueden expresarse en octal, hexadecimal y decimal.

Octal	**0**ddd
Hexdecimal	**0x**ddd
int	ddd
long	ddddd**L**
ungigned int	ddddd**U**
unsigned long	dddd**UL**

Las constantes numéricas en coma flotante pueden ser float, double o long double:

float	ddd.dd**F**	o	dd.dde dd**F**
double	ddd.ddd		
long double	ddd.dd**L**		

Las constantes carácter van encerradas entre comillas simples 'carácter' y las constantes de cadenas de caracteres van encerradas entre comillas dobles "cadena".

b) Variables.

Una variable es una posición de memoria con nombre, empleada para guardar un dato, que puede ser modificado durante la ejecución del programa. Todas las variables en C tienen que ser declaradas antes de poder ser utilizadas.

Sintaxis: tipo nombre_de_variable

4.6.- Operadores

En C los operadores pueden clasificarse en:

a) Aritméticos.

+ - * / % Los operandos tienen que ser numéricos, reales o enteros.

b) Lógicos.

&& (AND)
|| (OR)
! (NOT)

Los operadores son "boleanos" (cierto o falso), sabiendo que cierto es algo distinto de cero y falso es algo igual a cero. El resultado es un entero "cierto" distinto de cero y "falso" igual a cero. Los operadores son punteros, enteros o reales.

c) Relacionales.

< > <= >= == !=

Los operandos son enteros, reales o punteros. Una expresión cierta vale 1 y falsa vale 0.

d) Unitarios.

Tienen un sólo operando:

- Cambia de signo al operando (complemento a dos).
~ Complemento a uno (Carácter ASCII 126).

e) Operador ternario.

Se utiliza en expresiones condicionales.

Expresión1 ? Expresión2 : Expresión3

Donde Expresión1 tiene que ser entero, real o puntero; si el resultado de la evaluación de Expresión1 es cierta se realiza la Expresión2, en caso contrario se realiza la Expresión3.

Ejemplo:

mayor (a>b) ? a : b;

mayor=a si a>b en caso contrario mayor=b.

f) Operador coma (,).

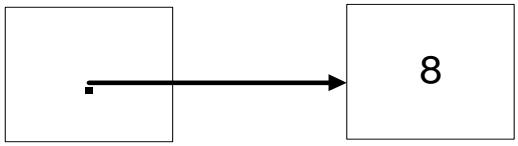
Un par de expresiones separadas por comas son evaluadas de izquierda a derecha.

g) Operadores para punteros.

Dirección **&** Este operador proporciona la dirección del operando.

Indirección ***** Permite el acceso a un dato contenido en un variable, de forma indirecta, a través de un puntero.

Ejemplo:



Ejemplo:

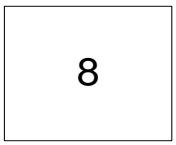
Si p es un puntero que puede guardar la dirección de una variable entera.

p=&A (si A = 8)

y si AUX es una variable entera

AUX=*p.

AUX guardará el valor almacenado en A



AUX

h) Operador sizeof (tamaño de).

Este operador da como resultado el tamaño en bytes de su operando. El resultado es de tipo size_t (unsigned int).

Sintaxis: **sizeof(expresión)**

Donde expresión es un identificador o un nombre de un tipo.

i) Lógicos de manejo de bits

Operador	Descripción
&	AND a nivel de bits.
|	OR a nivel de bits.
^	XOR a nivel de bits.
<<	Desplazamiento a la izquierda n bits.
>>	Desplazamiento a la derecha n bits.

Los operandos tienen que ser enteros: char, int, long, enum.

j) Operadores se asignación.

Operador	Descripción	Ejemplo C	Pseudocódigo
++	Incremento.	a ++	a ←a+1
- -	Decremento.	a - -	a ←a-1
=	Asignación simple.	a =1	a ← 1
* =	Multiplicación más asignación.	a *=5	a ←a*5
/ =	División más asignación.	a /=2	a←a/2
% =	Módulo más asignación	a %2	a ← a%2
+ =	Suma más asignación.	a +=2	a←a+2
- =	Resta más asignación	a -=3	a←a-3
<< =	Desplazamiento a la izquierda más asig.	a<<=1	a ←a<<1
>> =	Desplazamiento a la derecha más asig.	a>>=1	a ← a>>1
& =	Operación AND sobre bits más asig.	a & b	a ← a & b
| =	Operación OR sobre bits más asig.	a |= b	a ← a | b
^ =	Operación XOR sobre bits más asig.	a ^= b	a ← a ^ B

k) Los operadores . y ->.

Sirven para referenciar elementos individuales componentes de las estructuras y uniones (tipos estructurados de datos en C).

Cuando se trabaja con la estructura (estructura o unión) directamente, se utiliza el punto (.) para hacer referencia a cualquier elemento componente.

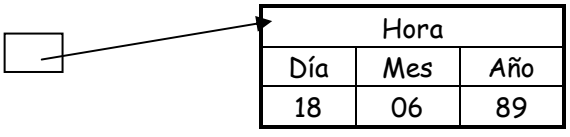
Si se trabaja con un puntero a la estructura, utilizaremos la flecha (->) para hacer referencia a los elementos individuales.

Ejemplo:

Hora		
Día	Mes	Año
18	06	89

Hora.Dia=18
Hora.Mes=06
Hora.Año=89

Si **phora** es un puntero a la estructura Hora: **phora = &Hora**



```
phora->Dia=18
phora->Mes=06
phora->Año=89
```

l) Conversión de tipos.

Automática:

Cuando los operandos en una expresión son de diferentes tipos, en cada operación binaria se produce una conversión automática a un tipo de datos común, de acuerdo con las siguientes reglas:

1. Cualquier operando *float* es convertido a *double* ; esto quiere decir que en C la aritmética en coma flotante se realiza en doble precisión.

2. En una operación determinada, si un operando es *long double*, el otro operando es convertido a *long double*.

3. Si un operando es de tipo *double,* el otro operando es convertido a *double*.

4. Cualquier operando de tipo *char o short* es convertido a *int.*

5. Cualquier operando de tipo *unsigned char* o *unsigned short* es convertido a *unsigned int*.

6. Si un operando es de tipo *unsigned long* el otro operando es convertido a *unsigned long*.

7. Si un operando es de tipo *long e*l otro operando es convertido a *long*.

8. Si un operando es de tipo *unsigned int* el otro operando es convertido a *unsigned int.*

Es decir, los operandos que intervienen en una determinada expresión, son convertidos de forma automática, al tipo de operando de operando de precisión más alta.

Además hay que tener en cuenta que en una asignación, el tipo resultante de la expresión, es convertido al tipo de la variable sobre la que recae la asignación:

- Los *caracteres* se convierten a *enteros.*
- Los *enteros* se convierten a *caracteres*, desechando los bits de mayor peso.
- Los *reales* se convierten a *enteros* perdiendo la parte fraccionaria.
- Los *double* se convierten a *float* perdiendo precisión.

También tiene lugar una conversión automática de tipos cuando un valor es pasado como argumento en la llamada a una función. Las conversiones se realizan independientemente sobre cada argumento.

"Casting" o conversión explícita de tipos.

En algunas ocasiones es necesario forzar una determinada conversión a un tipo de datos, indicando explícitamente el tipo requerido en la conversión. Se debe seguir las reglas de la conversión automática, ya que todas las conversiones no están permitidas.

La **sintaxis** para la conversión explícita del tipo de una expresión es :

(nombre_de_tipo) expresión

Ejemplo:

Si queremos que A = B * H / 2 sea un valor real, tenemos varias opciones.

```
A = (float)B * H / 2
A = B * (float)H / 2
A = B * H / 2.0
```

m) Precedencia de operadores.

( ),[ ],., ->	**Izquierda a Derecha**
-,!,*,&,++,--,sizeof,"casting"	**Izquierda a Derecha**
***,/,%,+,-**	**Izquierda a Derecha**
<<,>>	**Izquierda a Derecha**
<,>,<=,>0	**Izquierda a Derecha**
==,!=	**Izquierda a Derecha**
&	**Izquierda a Derecha**
^	**Izquierda a Derecha**
|	**Izquierda a Derecha**
&&,||	**Izquierda a Derecha**
?:	**Derecha a Izquierda**
=,*=,/=,%=,+=,-=,<<=,>>=,&=,|=,^=	**Derecha a Izquierda**
,	**Izquierda a Derecha**

4.7.- Expresiones

Una expresión en C es cualquier combinación válida de operadores y operandos (constantes y variables). Según sean los operadores y operandos que forman parte de la expresión podemos clasificar las expresiones:

- Aritméticas.
- Lógicas.
- Relacionales.
- Etc...

Si en una expresión aparecen operandos de diferente tipo serán convertidos a un tipo único (tipo de la expresión).

4.8.- Estructura de un programa en C

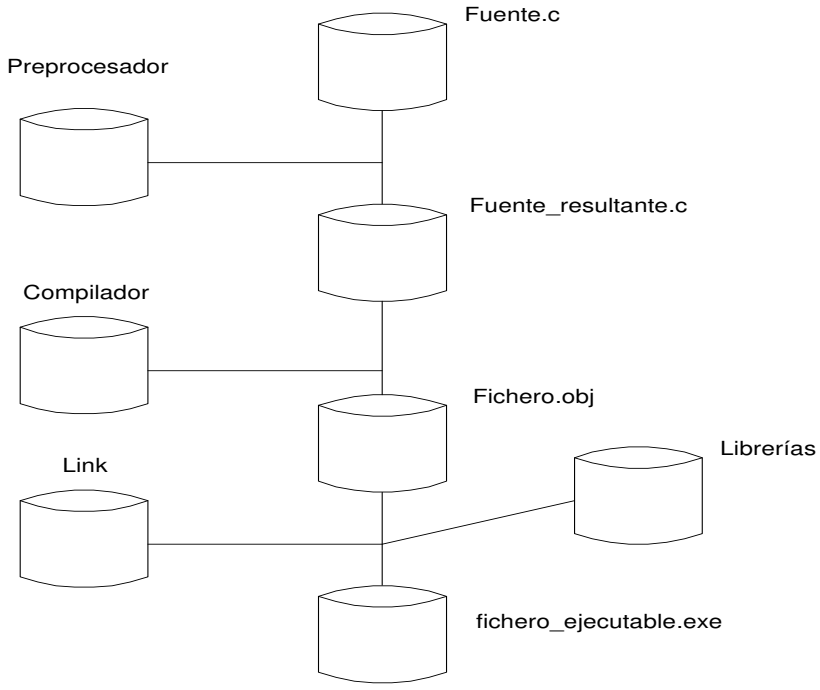
4.8.1.- Directivas

Comienzan con el símbolo # y son ordenes dadas al preprocesador para efectuar determinadas acciones sobre el fichero fuente C.

El preprocesador de C es un procesador de texto que manipula el archivo que contiene el texto del programa fuente, como una primera fase de la compilación.

Cuando se ejecuta la orden para compilar un fichero, el preprocesador procesa el texto fuente antes de que lo haga el compilador, realizando las siguientes acciones:

1. Se elimina caracteres dependientes del sistema.
2. El texto fuente se descompone en elementos reconocibles por el C.
3. Se ejecuta las directivas, se expanden las macros, etc..
4. Se compila el código fuente resultante.



El preprocesador de C/C++ contiene las siguientes directivas:

Directivas de compilación condicional.

Permiten compilar selectivamente partes del código fuente del programa: Compilación condicional

#if **#if expresión_constante**
secuencia de sentencias
#else **#else**
secuencia de sentencias
#endif **#endif**

#elif Se emplea para anidar #if... #elif...("else if")

#ifdef **#ifdef nombre_de_macro**
secuencia de sentencias
#endif

Significa que si se ha definido previamente nombre_de_macro con #define, se compila el bloque de código.

#ifndef **#ifndef nombre_de_macro**
secuencia de sentencias
#endif

Significa que si nombre_de_macro no ha sido definido en una sentencia #define, se compila el bloque de código.

#include **#include <fichero>**

Indica al compilador que tiene que incluir el archivo fuente especificado en el fichero fuente que contiene la directiva. También puede especificarse el nombre del fichero entre comillas para algunas situaciones concretas. Ejemplo: **#include <stdio.h>**

#define **#define nombre_de_macro secuencia_de_caracteres**

Sirve para definir un identificador y asociarlo a una secuencia de caracteres. El preprocesador sustituirá cada aparición del identificador en el programa fuente por esta secuencia. Ej : **#define PI 3.14**

#undef **#undef nombre_de_macro**

Elimina la definición previamente realizada con #define

#line **#line número "nombre_de_archivo"**

Cambia los contenidos de **_LINE_** (guarda el número de linea de la línea que se está compilando) y **_FILE_** (cadena que contiene el nombre del archivo fuente que se está compilando). Se utiliza principalmente en depuración.

#error **#error mensaje_de_error**

Fuerza al compilador a detener la compilación.

#pragma ....

4.8.2.- Enlace (Linkado). Librerías y Archivos de cabecera

El proceso de linkado de un programa tiene dos funciones:

- Enlazar varios fragmentos de código objeto.
- Convertir direcciones simbólicas, escritas en el formato reutilizable del fichero objeto, en direcciones reales.

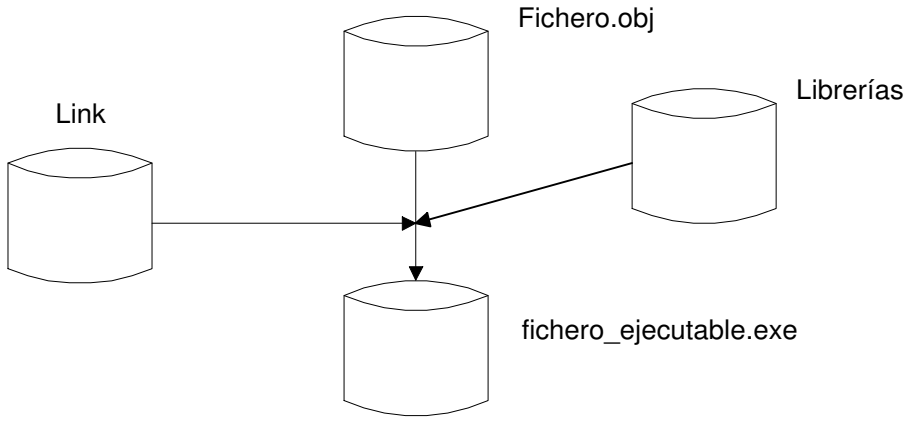
El resultado final de la fase de "linkado" en el desarrollo de un programa es la creación del programa ejecutable correspondiente.

Cuando la programación es modular los fragmentos de código que se enlazan son módulos objetos incluidos en distintas librerías. Este conjunto de librerías puede estructurarse en:

- Librerías estáticas: Librerías estándar del sistema y librerías de usuario.
- Librerías dinámicas.

Librerías estáticas.

Una librería estática incluye el código objeto de un conjunto de módulos. El código objeto de un determinado módulo se incluye o añade al programa durante el enlazado (linkado) del mismo.



El estándar ANSI de C da nombre y describe un conjunto de funciones que deben ser preprocesadas por cualquier compilador de C que se ajuste al estándar ANSI. Este conjunto de funciones se encuentra agrupado en librerías. Cuando un programa llama a una determinada función (módulo) contenido en una librería del sistema, el enlazador busca el código de esa función y lo añade al programa (librerías estándar del sistema).

Además cada usuario puede crear sus propias librerías de funciones. (Librerías de usuario).

Librerías dinámicas.

En el caso de librerías dinámicas el código real del módulo o función reside en un archivo aparte hasta que el programa que utiliza dicha función comienza su ejecución.

Estos módulos o funciones son cargados en tiempo de ejecución y residen en librerías especiales denominadas librerías de enlace dinámico ó DLL (Dinamic Link Library). Se emplean DLL´s cuando se programa para Windows (conjunto de funciones de la Interfaz de Programación de Aplicaciones o lo que es lo mismo las API´s)

Archivos cabecera.

Durante la fase de compilación de un determinado programa el compilador verifica que cada sentencia escrita es correcta. Cuando tiene que verificar que una sentencia de llamada a un determinado módulo o función es correcta necesita determinada información acerca del tipo de valor devuelto o parámetros formales, para esto será necesario incluir algunas sentencias que nos proporcionan esta información.

Cada una de las funciones definidas en una librería de C/C++ tiene asociado un archivo denominado archivo cabecera (con extensión .h). En este archivo cabecera se encuentra una descripción de la función que permitirá al compilador comprobar que cada llamada a una función es correcta durante la fase de compilación (prototipo de la función).

Un fichero cabecera contiene además definiciones de constantes, tipos de datos, etc. utilizados por las distintas funciones. Un fichero cabecera es incluido en un programa C/C++ mediante la directiva *#include <fichero_cab.h>*.

4.9.- Sentencias simples y compuestas

Una sentencia es la unidad ejecutable menor de un programa C/C++.

Una sentencia simple representa una sola acción a realizar en el programa. Una sentencia simple se escribe en una solo línea y termina con punto y coma (;).

Una sentencia compuesta representa la ejecución de un conjunto de sentencias (bloque).

Ejemplo.

Pseudocódigo	Sentencia	Tipo
<leer A>	Scanf("%d",&A);	Simple
A ← 10	A = 10;	Simple

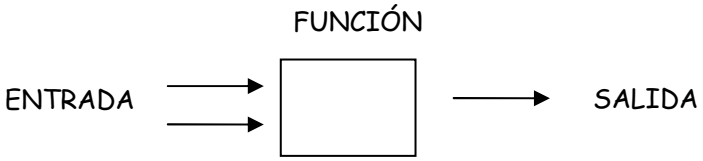
Mientras A > 5
<imprimir A>
A ← A-1
Fin_mientras
	While (A > 5)
	{
	Printf ("%d",A);
	A--;
	}
		Compuesta

4.10.- Funciones

En C/C++ el único módulo permitido es la función. Una función es un colección independiente de declaraciones y sentencias que realizan una determinada tarea.

Todo programa C/C++ consta de al menos una función: main(), pero además pueden existir otras.

Gráficamente podemos representar una función así:



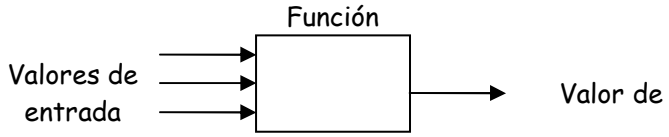
Una función es un módulo que realiza una determinada tarea; el módulo debe recibir una serie de datos (datos de entrada) y devolverá un resultado que será devuelto por la función (dato de salida).

4.11.- Estructura de un programa en C

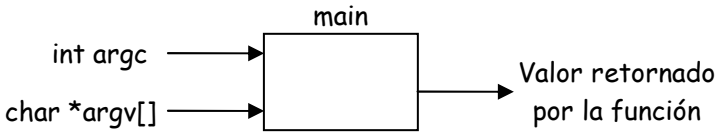
El lenguaje C permite dividir un determinado programa en varios bloques de sentencias, denominados módulos, siguiendo las reglas de cohesión y acoplamiento, etc. expuestas en la teoría de descomposición modular.

En el lenguaje C el único módulo permitido es el módulo FUNCIÓN. Una función es una colección de sentencias que realizan (ejecutan) una tarea específica.

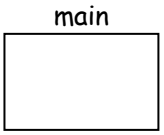
Gráficamente se puede representar un esquema de la función:



Todo programa C debe contener una función principal de nombre "main", desde donde comienza la ejecución.



Por ahora emplearemos una estructura para la función "main" simplificada; es decir, sin valores de entrada y sin valor de salida.



Según esto la estructura de un programa C/C++ es:

- Conjunto de directivas dirigidas al preprocesador
- Conjunto de declaraciones globales de variables y constantes
- Conjunto de declaraciones globales de funciones utilizadas.

Esto es

```
/* Definición de la función principal */

void main (void)
{
 /* Cuerpo de la función principal */
Conjunto de declaraciones locales a "main" de variables y constantes.
 Conjunto de declaraciones locales a "main" de funciones utilizadas.
 Conjunto de sentencias y llamadas a otras funciones.
 ...
 /* fin del cuerpo de la función principal */
}
/* Definición de otra función */
encabezamiento de la función
{
 /* cuerpo */
}
        ...
```

4.12.- Ámbito de las variables, constantes, funciones, etc.

Las variables y funciones globales son conocidas en todos los puntos del programa; es decir, pueden ser utilizadas en cualquier parte.

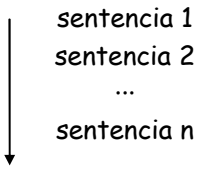
Las variables y funciones locales son conocidas sólo dentro del bloque correspondiente a cada función en la que están definidas.

4.13.- Sentencias

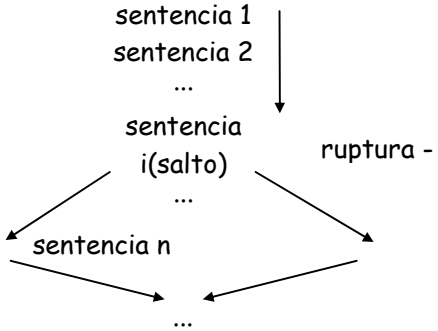
Una sentencia es la unidad ejecutable más pequeña de un programa C.

Un programa está formado por un conjunto de sentencias almacenadas en memoria en el mismo orden en el que van a ser ejecutadas; es decir; secuencialmente.

Un programa puede ser lineal cuando las instrucciones se ejecutan secuancialmente:



Un programa es no lineal cuando existen sentencias de ruptura de secuencia o bifurcación:



4.13.1.- Tipos de sentencias

El lenguaje C dispone de un conjunto determinado de sentencias. Estas sentencias pueden clasificarse en:

Por su extensión

- Simples: Una sola sentencia, terminada en ";".
- Compuestas: Formadas por un conjunto de sentencias encerradas entre llaves {} (bloque).

Por la función que desempeñan

Asignación:

Sintaxis:

variable operador_de_asignación expresión;

Se evaluará la expresión y el resultado es asignado a la variable. El tipo resultante de la expresión debe coincidir con el tipo de la variable.

Ejemplo:	a ← 8	a = 8;
	b ← 5 + 3 * 2 - a	b = 5 + 3*2 – a;

Condicionales:

Permiten tomar una decisión referente a la acción a ejecutar en un programa, basándose en el resultado cierto o falso de una expresión.

Sintaxis:

```
if (expresión)
        sentencia 1;
[else
        sentencia 2;]
```

donde:

expresión: Debe ser una expresión numérica, relacional o lógica.
La evaluación de la expresión da como resultado verdadero (distinto de cero) o falso (cero).

sentencia 1/2: Representa una sentencia simple o compuesta.

Si el resultado de la expresión es "verdadero" se ejecutará lo indicado por la sentencia1; si es "falso", se ejecutará lo indicado por la sentencia2.

Selección múltiple.

Permite ejecutar una de varias acciones, en función del valor de una expresión.

Sintaxis:

```
switch (expresión)
{
 case cte1:  sentencia 1;
                  ...
             break;
 case cte2:  sentencia 2;
                  ...
             break;
 ...
 [default: sentencia default;]
}
```

Donde:

expresión:	expresión entera.
ctei:	valores enteros posibles en la evaluación de la expresión entera (conjunto discreto de valores).
sentencia:	sentencia simple o compuesta.

Evalúa la expresión y compara el resultado de la evaluación con los casos que aparecen en cada "case"; cuando coincide con alguno de las constantes, se ejecutan las sentencias correspondientes al caso hasta encontrar "break".

Si no coincide con ninguna de las constantes especificadas y se encuentra la opción default se ejecutan las sentencias asociadas a esta última opción.

Ejemplo:

Diseñar un fragmento de programa que permita leer un mes (nº correspondiente entre 1 y 12) y almacenar, en la variable días, el número de días correspondiente a cada mes.

```
printf ("Introduzca el mes del año:");
scanf ("%d", &mes);

switch (mes)
{
 case 11:
 case 4:
 case 6:
 case 9:
        dias = 30;
        break;
 case 2:
        dias = 28;
 default:
        dias = 31;
}
printf ("El mes tiene %d días", dias);
```

Repetitivas

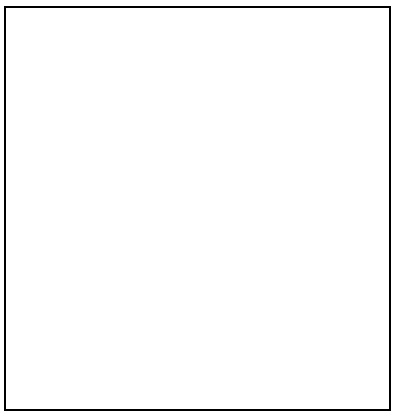
a) Sentencia while.

Permite **repetir** un bloque de sentencias **cero o más veces** dependiendo del valor que se evalúa para una expresión booleana.

Primero se evalúa la condición; si la expresión booleana es "verdadera" se ejecuta el cuerpo del bucle y se vuelve a evaluar la condición. El proceso se repite mientras la condición resulte cierta en cada evaluación y termina cuando la condición evaluada es "falsa".

Sintaxis:

```
while (expresión)
      {
 sentencia;
  . . .
}
```



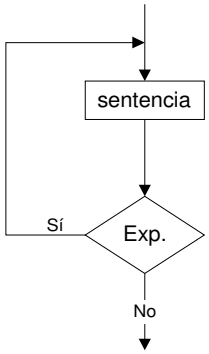
```
mientras (expresión)
     sentencia
          . . .
fin_mientras
```

b) Sentencia do.

Permite **repetir** un bloque de sentencias **una o más veces** dependiendo del valor obtenido al evaluar una expresión booleana (cierto o falso).

Sintaxis:

```
do{
 sentencias;
  . . .
}while (expresión);
```



Diagrama

```
hacer
  Sentencia
    …    …
mientras (expresión)
```

c) Sentencia for.

Permite **repetir** un bloque de sentencias **un número** determinado **de veces** empleando una variable contador.

Sintaxis:

```
for ([var1=ini1, [var2 = ini2]...]; [condición]; incremento)
{
 sentencias;
 ...
}
```

Sentencias de ruptura.

Sentencia break.

Permite finalizar la ejecución de una sentencia do, for, switch o while de forma brusca.

Sintaxis: **break;**

Sentencia continue.

Permite pasar el control a la siguiente iteración. Se emplea en sentencias como while, do, for.

Sintaxis: **continue;**

Sentencia goto (etiquetas).

Permite que la ejecución pase a una parte del código que se identifica a través de una etiqueta.

Sentencia de retorno (return).

Permite expresar el valor retornado por la función.

Sintaxis: **return (valor o variable retornado);**

Sentencias de declaración.

Permiten declarar un objeto del programa, constante, variable, función, etc.

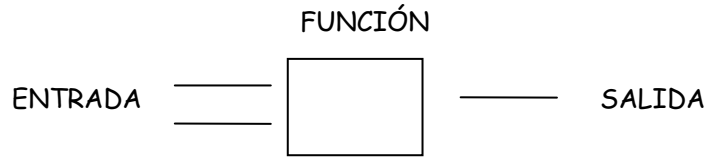
Constante: Tipo_de_la_constante identificador;
Variable: Tipo_de_almacenamiento Tipo_dato identificador;
Función: Tipo_de_valor_devuelto nombre (lista de tipo de los parámetros);

4.14.- Funciones

En C/C++ el único módulo permitido es la función. Una función es una colección independiente de declaraciones y sentencias que realizan una determinada tarea.

Todo programa C/C++ consta de al menos una función: main(), pero además pueden existir otras.

Gráficamente podemos representar una función así:



Una función es un módulo que realiza una determinada tarea; el módulo debe recibir una serie de datos (datos de entrada) y devolverá un resultado que será devuelto por la función (dato de salida).

a) Definición de una función.

La definición consta de una cabecera de la función y de un cuerpo de la función.

La sintaxis de esta definición es:

```
tipo de valor devuelto      nombre_de_la_funcion     (lista de parámetros formales)
{
 declaraciones de variables, constantes locales ...
 declaraciones de funciones locales (utilizadas en esa función)
 sentencias
 ...
 return (variable ó valor devuelto por la función)
}
```

Donde, los **parámetros formales** son variables con las que se trabaja en la función y que reciben los datos de entrada a la función (**parámetros actuales**).

Los parámetros formales son válidos sólo en la función, se crean al entrar en la función, cuando se ejecuta, y se destruyen al salir de la función.

b) Llamada a una función.

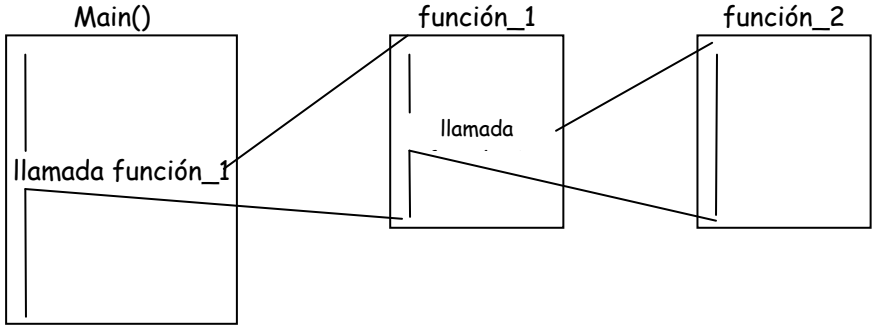
Las sentencias que componen una determinada función son ejecutadas a través de una llamada a la función. La sintaxis de esta llamada es:

```
[variable =]  nombre_de_la_función ( [lista de parámetros actuales ] )
```

Los **parámetros actuales** son los valores proporcionados a la función en la llamada.

La variable que aparece en la llamada es la que recoge el valor devuelto por la función en la sentencia **return**.

Cuando se llama a una función, el control del programa pasa a la función; cuando finaliza la función, el control es devuelto de nuevo al módulo que realizó la llamada, para continuar con la ejecución del mismo a partir de la sentencia que efectuó la llamada.

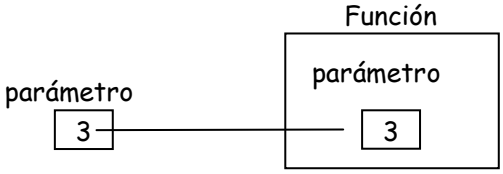


c) Paso de parámetros.

En C/C++ hay principalmente dos modos de pasar parámetros actuales a sus correspondientes parámetros formales, cuando se efectúa la llamada a la función :

1. Por valor.

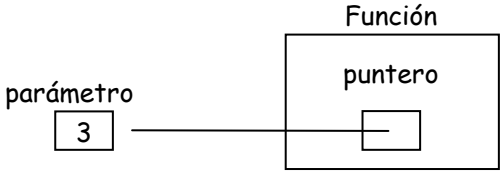
Significa copiar los parámetros actuales en sus correspondientes parámetros formales. No se modifican los parámetros actuales.



Llamada: **Función (parámetro actual)**

2. Por referencia (dirección).

Lo que se transfiere no son los valores sino las direcciones de las variables que contienen esos valores. Los parámetros actuales pueden ser modificados desde la función.



Llamada: **Función (&parámetro actual)**

d) Declaración de una función. Función prototipo.

Una función no puede ser llamada si previamente no está definida o declarada. La declaración de una función se realiza mediante la denominada función prototipo.

Una función prototipo tiene la misma sintaxis que la cabecera de la definición de la función, terminando en punto y coma.

Tipo_de_valor_devuelto nombre_de_la_funcion (lista de parámetros);

El ámbito de los parámetros está restringido a la propia declaración.

4.15.- Ámbito de las variables.

Se denomina ámbito de una variable a la parte de un programa donde dicha variable puede ser referenciada por su nombre.

Una variable puede estar limitada a un bloque (entre { }), a un fichero, a una función, o a una declaración de una función prototipo.

4.15.1.- Variables globales.

Se declaran fuera de todo bloque de un programa (fuera de main() también) y son accesibles desde el punto de su definición o declaración, hasta el final del fichero fuente.

```
#include...
#define.....
variables globales
void main(void)

{

 .......

}

tipo otra_funcion(.......)

{
        ....
}

...
```

4.15.2.- Variables locales.

Una variable local se declara dentro de un bloque delimitado entre { }. El posible acceso a esa variable queda limitado a ese bloque. Una variable local existe y tiene valor desde el punto donde es declarada hasta el final del bloque donde es definida. Una variable local es accesible sólo dentro del bloque al que pertenece.

Cada vez que se realiza una llamada a una función, se pasa el control al bloque de la función para su ejecución y son definidas las variables locales a la función; cuando finaliza la ejecución de la función (sentencia return), las variables dejan de existir.

Cuando se entra en un determinado bloque, delimitado entre llaves, se crean la variables allí definidas; estas variables dejarán de existir al salir del bloque.

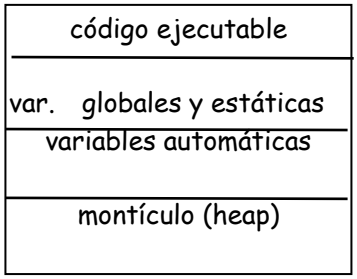
Los parámetros formales de una función, son locales a la función.

Los parámetros que aparecen en la declaración de una función prototipo son locales a la propia declaración de la función.

4.16.- Almacenamiento de variables.

Por defecto todas las variables llevan asociado un tipo de almacenamiento que determina su accesibilidad y existencia (ámbito).

La ejecución de un programa en C, lleva consigo la utilización de un espacio de la memoria del ordenador de la forma siguiente:



El código ejecutable ocupa unas posiciones perfectamente definidas en el momento del "linkado".

A continuación se almacenan las variables globales, ocupando un espacio perfectamente definido en tiempo de compilación, y las variables que designemos explícitamente con almacenamiento permanente (estáticas).

A continuación, se almacenarán aquellas variables que son creadas cada vez que comienza la ejecución de un bloque (función). Son colocadas en la denominada pila del sistema que crece y decrece según las necesidades del programa.

Por último, encontramos el espacio reservado para la asignación de memoria durante la ejecución del programa; es decir, para asignación dinámica. La memoria correspondiente a esta zona se gestiona en un montículo (heap).

Según esto, las variables globales se almacenarán inmediatamente detrás del código, y las locales en la zona de pila. Sin embargo existen unos calificadores que nos permiten modificar este tipo de almacenamiento por defecto:

- **auto**: Almacenamiento en la pila del sistema. Es el almacenamiento por defecto en las variables locales.
- **register** : El almacenamiento se realiza sobre un registro especial del procesador.
- **static** : Las variables son almacenadas en la misma zona que las variables globales.
- **extern** : El almacenamiento es externo.

La **declaración** de una variable será entonces:

[Tipo_de_almacenamiento] Tipo_de_dato identificador;

4.17.- Funciones de la librería del lenguaje C/C++. Funciones de entrada y salida.

Las operaciones de entrada y salida no forman parte del conjunto de sentencias de C, sino que pertenecen a un conjunto de funciones de las librerías de C/C++, cuyos prototipos se encuentran en los ficheros stdio.h (para las funciones estándar), conio.h (para las funciones no estándar), graphics.h (para las funciones gráficas),...

Las funciones de E/S pueden resumirse en el siguiente esquema:

Funciones estándar de E/S.

- **Por consola ó conversacionales.**
  - Con formato :
    printf( )
    scanf( )
  - carácter a carácter :
    getchar( )
    putchar( )
  - cadenas de caracteres :
    puts( )
    gets( )
- **Sobre ficheros.**
  - carácter a carácter :
    fputc( )
    fgetc( )
  - palabra a palabra :
    putw( )
    getw( )
  - cadena a cadena :
    fputs( )
    fgets( )
  - Con formato :
    fprintf( )
    fscanf( )
  - Registro a registro, o en bloque :
    fwrite( )
    fread( )

Funciones de entrada y salida de bajo nivel.

- **Sobre ficheros.**

write( ) son funciones del núcleo.
read( )

Funciones de entrada y salida no estándar.

getch( )
getche( )
putch( )

kbhit( )
cgets( )
cputs( )
cprintf( )
cscanf( )

Otras funciones.

exit( )
clrscr( )
system( )
...

Funciones de entrada y salida estándar.

Su característica fundamental es que la E/S se realiza a través de un buffer o memoria intermedia.

En el entorno C todos los dispositivos tradicionales (teclado y pantalla), terminales, unidades de disco, puertos e impresora, son tratados como archivos. Sea cual sea la naturaleza del dispositivo externo, es tratado por el sistema convirtiéndolo en un dispositivo lógico denominado **flujo, corriente o stream.**

Cuando un programa comienza su ejecución, se abren automáticamente cinco streams, que se corresponden con dispositivos:

stdin dispositivo de entrada estándar (teclado).
stdout dispositivo de salida estándar (pantalla)
stderror dispositivo de salida de errores estándar (pantalla)
stdaux dispositivo auxiliar estándar (puerto serie)
stdprn dispositivo de impresión estándar (impresora paralelo)

Estos dos últimos dependen de la configuración de la máquina y pueden no estar presentes.

printf( )

Escribe datos por consola, a través de stdout. El prototipo está en stdio.h.

```
int printf ( const char *cadena, argumentos) ;
```

este formato puede ser una cadena constante entre "" ó puntero a una cadena constante de caracteres.

Este formato contiene caracteres que se imprimen directamente y especificadores de formato para cada tipo de datos:

%c	carácter
%d	entero
%i	entero
%e ó %E	notación científica de números reales
%f	coma flotante
%g ó %G	Utiliza el mas corto entre %e ó %f
%o	Octal sin signo
%s	Cadenas de caracteres
%u	Enteros sin signo
%x ó %X	Hexadecimal sin signo
%p	Punteros
%n	Puntero a entero al que se asigna el nº de caracteres escritos
%%	Imprime %

El formato para cada tipo de dato contiene además de los caracteres anteriores.

% [flags] [longitud] [.precisión] [F/N/h/l/L] tipo

Flags: - justifica a la izquierda.
+ imprime el signo + .
b imprime blanco para numeros > 0.
forma alterna ... ?

longitud: constante entera positiva.
Un mínimo número de caracteres visualizados.
0 un mínimo número de caracteres completados con 0.
* para pasar las características anteriores como argumentos ("% * . *f", 104, 123.3)

precisión: longitud de caracteres decimales.
F puntero Far
N puntero Near
h short
l long int ld, li
L long double

El entero retornado por la función representa el número de caracteres visualizados, en caso de error se compara este valor retornado con los caracteres que se tenían que imprimir.

scanf().

Función para entrada de datos por consola, el prototipo está incluido en stdio.h.

```
int scanf( const char *cadena, argumentos...);
```

Es una función que trabaja a través del buffer. La función toma los datos de un área de memoria interna (stdin); existe una falta de sincronización entre lo que se escribe por teclado y lo que lee scanf ().

En la cadena que expresa el formato el * indica que el valor leído no es tenido en cuenta y no será asignado a ninguna variable de la lista de argumentos.

getchar().

Realiza la lectura de caracteres a través del buffer, obtiene el siguiente carácter de stdin. Los caracteres son leídos como unsigned char y se convierten a enteros

protipo: int getchar (void);	**stdio.h**

putchar().

Escribe un carácter sobre la salida estándar stdout.

int putchar (int c);	**stdio.h**

puts() y gets().

Permiten la salida y entrada de una cadena sobre ña salida estándar y la entrada estándar respectivamente.

int puts (conts char *cad);	**stdio.h**

char gets (char *cad);	**stdio.h**

getch() y getche()

No es una función estándar. Lee un caracter sin eco o con eco.

int getch (void);	**sin eco**	**conio.h**
int getche (void);	**con eco**	**conio.h**

putch().

Escribe un carácter. No es una función estándar.

int putch (int c);	**stdio.h**

kbhit ().

Devuelve un valor distinto de cero si se ha pulsado una tecla, en caso contrario devuelve cero.

int kbhit (void); **conio.h**

Se emplea para detectar si se ha pulsado una tecla. Sirve para "vaciar" la memoria intermedia del DOS (20 octetos) que almacena los caracteres pulsados en el teclado hasta que sean utilizados.

Ejemplo.

```
while (kbhit() = 0)
        getch ();

for (;;)
{
 if (kbhit() == 0)
        break;
}
```

La función kbhit genera en realidad una llamada a DOS.

cprintf() y cscanf().

Escribe y lee de la memoria interna del DOS respectivamente.

4.18.- Salida de un programa

exit().

Permite la salida del programa (terminación del programa).

prototipo: void exit (int código_salida);

4.19.- Ejecución de órdenes del Sistema Operativo

system().

Permite ejecutar una orden del sistema operativo.

prototipo: int system (conts char

Ejemplo:

```
system ("cls");
system ("dir");
```