TEMA 4

ELEMENTOS DE UN PROGRAMA REDACTADO EN C

1. **Introducción.**
2. **Tipos de datos en C.**
3. **Palabras clave.**
   - 3.1 Identificadores.
   - 3.2 Palabras reservadas.
   - 3.3 Separadores.
   - 3.4 Formato libre.
4. **Comentarios.**
5. **Operadores y Operaciones.**
6. **Expresiones y su evaluación.**
   - 6.1 Expresiones.
   - 6.2 Prioridad y orden de evaluación.
   - 6.3 Comportamiento de operadores aritméticos en caso de excepción
7. **Conversión de tipos.**
8. **Ámbito de las variables.**
9. **Sintaxis de las instrucciones en C.**
   - 9.1 Instrucciones y bloques.
   - 9.2 Las operaciones de E/S en C.
   - 9.3 Sentencias de control
   - 9.4 Sentencias del preprocesador
10. **Uso de las funciones en C.**
11. **Paso de parámetros por valor y por referencia.**
12. **Otras funciones predefinidas en C.**

1. INTRODUCCIÓN

C al igual que los lenguajes naturales (castellano, francés, etc.) o que otros lenguajes de programación (Pascal, Fortran, Cobol, etc.) se compone de un conjunto de elementos y de un conjunto de reglas que nos permiten combinarlos correctamente así como definir elementos nuevos. A este conjunto de reglas se le denomina *sintaxis de C*.

Por otro lado, C al igual que el resto de lenguajes de programación, se compone de una serie de elementos básicos o léxico, a partir de los cuales y teniendo en cuenta las reglas sintácticas, podremos construir elementos más complejos del lenguaje. Los elementos básicos con los que se construyen los programas de C son los siguientes:

a) Palabras clave o reservadas:

Son un conjunto de palabras predefinidas, que se escriben siempre en **minúsculas** y que tienen un significado especial para el lenguaje C. Son las que constituyen realmente el lenguaje. Entre las palabras reservadas que utiliza C, están : case, char, int, float, etc.

b) 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.

c) Operadores:

Representan operaciones de tipo aritmético, lógico, relacional, asignación, etc.

d) Identificadores:

Son nombres de objetos (variables, constantes, etc.) definidos por el programador.

Los identificadores se construyen con letras y dígitos, si bien el primer carácter debe ser una letra.

2. TIPOS DE DATOS EN C

Las variables y constantes son los objetos o datos básicos que se manipulan en un programa. Cada variable o constante queda definida en el programa mediante un identificador y por un valor que puede variar durante la ejecución del programa, en el caso de las variables, o permanecer fijo, en el caso de las constantes. Las variables y constantes que se pueden definir en C se agrupan en conjuntos organizados, denominados **tipos de datos.** Cada tipo de datos queda definido por el conjunto o rango de valores que pueden tomar sus variables y constantes, así como por las operaciones que podemos realizar con dichos valores. Además el número de valores que forman parte de un tipo **quedará determinado por el número de bits** que utilice el ordenador para representarlos internamente.

En C los tipos de datos se clasifican en:

a) *Fundamentales o básicos*. Que son los siguientes:

i) **char**

Sus valores son cualquier carácter del juego ASCII. Tienen una longitud de 8 bits o sea 1 byte. Cualquier carácter del código ASCII puede representarse con 8 bits (2^8= 256 caracteres).

Ej.: char LETRA;

ii) **int**

Cada elemento de este tipo de datos es un entero. Normalmente tiene la longitud de 16 bits o sea 2 bytes. Aunque dicha longitud puede depender del sistema operativo del ordenador en el que se ejecute. (2 ó 4 bytes)

Ej.: int EDAD;

iii) **float**

Cada elemento es un número real en punto flotante de precisión simple. Su longitud es de 32 bits o sea 4 bytes.

Ej.: float LONGITUD, ALTURA;

iv) **double**

Cada elemento es un número real en punto flotante de doble precisión . Su longitud es de 64 bits.

Ej.: double DISTANCIA;

Sobre estos cuatro tipos de datos se pueden aplicar una serie de modificadores que nos permiten variar su rango de valores y/o su longitud:

short, se aplica al tipo entero (int), cuando queremos un rango de valores enteros menor que el que nos proporciona. (**short int** X;, que es equivalente a poner **short** X;) (2 bytes).

long, se aplica a los enteros (int) y reales de doble precisión (double), para aumentar el rango de valores de éstos últimos.

(**long int** Y; que es equivalente a poner **long** Y; , **long double** Z;).

El tipo long int utiliza 4 ó 8 bytes, el tipo long double 10 bytes

unsigned, se aplica a los tipos int, char, long y short para quedarnos sólo con los valores positivos asociados a cada uno de sus rangos.

(**unsigned int** ó **unsigned**
unsigned char desde 0 hasta 255 en código ASCII
unsigned short
unsigned long)

v) **puntero**

Se trata normalmente de **variables** que contienen la dirección de memoria de otra variable.

Ej.: int *DIRECCION;

vi) **void**

Usado para la declaración de funciones que no retornan ningún valor o para declarar punteros a un tipo no específico. Si void aparece entre paréntesis a continuación del nombre de una función, no es interpretado como un tipo. En este caso indica que la función no acepta argumentos.

Ej.: void main (); int respuesta (void);

Según lo anteriormente dicho los tipos de datos quedarían con los rangos siguientes:

TIPO	**LONGITUD**	**RANGO**
unsigned char	8 bits	0 a 255
char	8 bits	-128 a 127
unsigned int	16 bits (32 bits)	0 a 65.535 (4.294.967.295)
short int	16 bits	-32.768 a 32.767
int	16 bits (32 bits)	-32.768 a 32.767 (2.147.483.648)
unsigned long	32 bits	0 a 4.294.967.295
long	32 bits	-2.147.483.648 a 2.147.483.648
float	32 bits	3,4 x e-38 a 3,4 x e+38
double	64 bits	1,7 x e-308 a 1,7 x 1e+308
long double	80 bits	3,4 x e-4932 a 3,4 x e+4932

b) Tipos compuestos. Son aquellos que se construyen a partir de los anteriores. Las variables y constantes que pertenecen a estos nuevos tipos de datos(vectores, tablas, registros, etc.), se denominan, **estructuras de datos**¸ y son estudiadas en temas posteriores , por lo que no van a ser tratadas en éste.

3. PALABRAS CLAVE O RESERVADAS

3.1. LOS IDENTIFICADORES.

Los identificadores se emplean para nombrar y hacer referencia a los diferentes elementos de un programa (variables, constantes, nombres de funciones, etc.) utilizadas por el propio programa.

Los identificadores se eligen combinando las letras mayúsculas y minúsculas, las cifras y el carácter de subrayado. Estas combinaciones están sujetas a las siguientes reglas:

- El primer carácter no puede ser una cifra.
- El identificador elegido no puede coincidir con una palabra reservada.
- El identificador no puede contener espacios en blanco, ni letras acentuadas, ni letras especiales (ñ, Ñ), ni otros símbolos ortográficos (! (?) . ; ,)

Ejemplos de identificadores válidos son:

DIA, renta_2001, importe, fecha_nac, etc

Ejemplos de identificadores no válidos son:

1_planta (empieza con una cifra), edad minima (contiene un espacio en blanco), edad_mínima (contiene una letra acentuada), año_95 (contiene la letra ñ).

En C las letras mayúsculas y minúsculas son diferentes y, por ello, los identificadores: nombre, Nombre y NOMBRE, se refieren a variables diferentes.

La norma ANSI establece que los identificadores pueden tener cualquier longitud, pero que a efectos de identificación solo se consideran como significativos o los 32 primeros caracteres en los nombres internos.

Las palabras reservadas no se pueden utilizar como identificadores.

3.2 LAS PALABRAS CLAVE O RESERVADAS.

Las palabras clave o reservadas son las utilizadas por el lenguaje C para fines muy definidos. La lista de palabras clave es diferente en cada compilador. La norma ANSI considera la siguiente lista

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

Otros compiladores además de estas establecen otras palabras clave:

asm	delete	if	return	try
auto	do	inline	short	typedef
break	double	int	signed	union
case	else	long	sizeof	unsigned
catch	enum	new	static	virtual
char	extern	operator	struct	void
class	float	private	switch	volatile
const	for	protected	template	while
continue	friend	public	this	
default	goto	register	throw	

3.3 LOS SEPARADORES.

Dos identificadores consecutivos deben separase por un signo especial (, ; + * / = ( ) { } etc.) cuando las reglas del lenguaje de programación (la sintaxis) lo exijan, y cuando las reglas no impongan un símbolo concreto, deben separarse obligatoriamente o por un espacio en blanco o por un carácter de fin de línea.

Por ejemplo, se debe escribir obligatoriamente
char c,d; (dejando espacio en blanco entre char y c**)**
se puede escribir (así queda más legible)
char c, d; (dejando espacio en blanco entre las variables)
y no se puede escribir
charc,d; (sin separar con espacio en blanco char y c)

3.4. EL FORMATO LIBRE.

C no impone restricciones a la hora de componer un programa. Por ejemplo una instrucción puede escribirse en varias líneas y una línea puede contener varias instrucciones. Por ello y para facilitar la lectura de los programas, se han elaborado un conjunto de convenios cuyo cumplimiento no es obligatorio, pero cuyo uso está muy extendido.

Por ejemplo, se podría escribir:

```
#include<stdio.h>
main(){int X,Y; printf("Introduce dos numeros\n\n");
scanf("%d%d",&X,&Y); printf("La suma de%dy%des%d\n\n",X,Y,X+Y);}
```

pero es mucho más legible escrito así, identado y colocando cada sentencia en líneas distintas:

```
#include<stdio.h>
main()
 {
        int X,Y;
        printf("introduce dos numeros\n\n");
        scanf("%d%d",&X,&Y);
        printf("La suma de%dy%des%d\n\n",X,Y,X+Y);
 }
```

4. LOS COMENTARIOS

Los comentarios permiten documentar los programas y así facilitar tanto su comprensión como el trabajo de una posterior modificación.

Dado que los comentarios no se convierten en código objeto, pues son ignorados por el compilador, su uso no incrementa los tiempos de compilación y ejecución y en consecuencia como reporta beneficios se aconseja su uso.

Un comentario en C comienza con los caracteres " **/*** " y termina con los caracteres "***/**". La norma ANSI no permite el uso de comentarios anidados, de forma que algo de este estilo

```
/* comienzo del comentario1 /* comentario2*/ fin comentario1*/
```

daría error, pues para el compilador de C que respete escrupulosamente la norma ANSI el comentario empieza al descubrirse el símbolo **/*** y termina con la primera aparición de */. Así pues, en nuestro caso *fin comentario1* es interpretado por el compilador como instrucciones y por tanto errores de sintaxis.

Para un comentario que ocupe una línea también es posible usar // al comienzo de la línea.

Ejemplo:

```
/* Este es un comentario
de dos líneas*/
// Este es un comentario de una línea.
```

5. LOS OPERADORES Y OPERACIONES

Las operaciones que se pueden realizar con los datos (variables o constantes) de un determinado tipo vienen determinadas por los operadores que usa el lenguaje. C utiliza los siguientes tipos de operadores:

a) *Operador de asignación*
La forma más general de asignación es: NombreVariable = expresión

Ej.: x = 9;

También se puede utilizar el operador de asignación precedido de otro operador, de la siguiente forma: NombreVariable operador = expresion , lo que equivale a escribir: NombreVariable = (NombreVariable)operador(expresion).

Ej.: x += 5; /*es equivalente a poner x = x+5 */

El operador que puede preceder al de asignación es uno de los siguientes:
+, -, *, /, %, >>, << (en binario AND, XOR, OR serían:) &, ^, |
Cuando en una asignación se utilizan variables de distinto tipo hay que hacer una conversión de tipos, de tal forma que el valor de la expresión del lado derecho se convierte al tipo de datos del lado izquierdo.
El hecho de que la asignación sea una expresión, permite realizar asignaciones múltiples y usarla dentro de otras expresiones. Ej: X = Y = 3 . En este caso a las dos variables se les asigna el valor 3, ya que no se puede interpretar Y = 3 como expresión lógica al disponer del operador = = para ello. Esta última situación sería X = Y = = 3.

Ejemplo:

```
/* Uso de los operadores de asignación */
#include <stdio.h>
main() /* Realiza varias operaciones */
{
        int a=1,b=2,c=3,r;
        a+=5;
        printf("a + 5 = %d\n",a);
        c-=1;
        printf("c - 1 = %d\n",c);
        b*=3;
        printf("b * 3 = %d",b);
}
```

b) O*peradores aritméticos*
En C hay dos tipos de operadores aritméticos, los que actúan sobre dos operandos o **binarios** y los que lo hacen sobre uno solo o **unitarios**.
Los operadores binarios actúan sobre dos operandos del mismo tipo para dar un resultado del mismo tipo que los operandos; pero como C no es un lenguaje fuertemente tipado, el compilador también sabe darles un significado cuando se trata de operandos de distinto tipo .

Los operadores aritméticos binarios son:

- + suma (a+b).
- - resta (a-b).
- ***** producto (a*b).
- / división (a/b), cuando los operandos son enteros el operador devuelve un entero despreciando la parte decimal, si la hubiera (división entera), y cuando son números representados en coma flotante, devuelve el cociente en coma flotante.
- **%** módulo (a%b) que sólo actúa sobre enteros y da el resto de la división entera del primer operando (a) entre el segundo (b). Por ejemplo, 29%6 vale 5. La norma ANSI establece que el operador % sólo tenga significado cuando los operandos sean enteros y positivos.

Los operadores aritméticos unitarios actúan sobre un solo operando. Son operadores unitarios:

- opuesto (cambio de signo, enteros o reales): - (-x ó -a+v)
- complemento a 1 (enteros): ~ *(alt + 126) (inversa en binario)*

En C no hay un operador específico para la potenciación. Para ello, se puede usar el operador producto (x^3 = x*x*x) o la función *pow*. La función pow pertenece a la librería estándar de C y se encuentra en el fichero **math.h**.
Su formato sería: **double pow (double x, double y)** y devuelve el valor x^y.

Otros operadores aritméticos que utiliza C son:

- - - (decremento), + + (incremento).

Los dos últimos son operadores propios de C y se pueden utilizar de las dos formas siguientes:

a) precediendo a una variable (incremento)

```
x=5;
printf("%d", ++x); // Imprimiría 6, pues primero incrementa x en una
                   //unidad y luego la imprime
y = --x;           // es equivalente a : x = x-1;  y = x
```

b) sucediendo a una variable (decremento)

```
x = 5;
printf("%d", x++); /* Imprimiría 5 y luego incrementa x en una unidad , por
   lo que  al finalizar la ejecución de printf  x=6. (pero se visualiza 5) */
y = x--;           // es equivalente a : y = x; x = x-1
```

Ejemplo:

```
/* Uso de los operadores aritméticos */
#include <stdio.h>
main() /* Realiza varias operaciones */
{
        int a=1,b=2,c=3,r;
        r=a+b;
        printf("%d + %d = %d\n",a,b,r);
        r=c-a;
        printf("%d - %d = %d\n",c,a,r);
        b++;
        printf("b + 1 = %d",b);
}
```

c) *Operadores relacionales y lógicos:*

Los operadores relacionales son :

- > mayor
- >= mayor o igual
- < menor
- <= menor o igual
- = = igual
- != distinto

Los operadores lógicos son:

- ! not
- && and
- || or

Los operadores lógicos para el manejo de bits son:

- & and
- | or
- ^ xor
- << desplazamiento a la izquierda (desplaza todos los bits a la izda, un numero de veces indicado => Desplazar una vez a la izda equivale a multiplicar por 2)
- >> desplazamiento a la derecha (desplaza todos los bits a la dcha, un numero de veces indicado => Desplazar una vez a la dcha equivale a dividir por 2)

En C **no existe el tipo booleano** (formado por los valores true , false), para representarlo se utilizan los enteros, asignándoles 0 a falso y cualquier otro valor a verdadero.

Ejemplo:

```
/* Uso de los op. lógicos AND,OR,NOT. */

#include <stdio.h>
main()                    /* Compara un número introducido */
{
  int N;
  printf("Introduce un número: ");
  scanf("%d",&N);
  if(!(N >=0))
        printf("El número es negativo");
  else if((N <=100)&&( N >=25))
        printf("El número está  entre 25 y 100");
     else if((N <25)||( N >100))
              printf("El número no está  entre 25 y 100");
}
```

Ejemplo:

```
/* Uso de los operadores relacionales.  */
#include <stdio.h>
main() /* Compara dos números entre ellos */
{
  int a,b;
  printf ("Introduce el valor de A: ");
  scanf ("%d",&a);
  printf ("Introduce el valor de B: ");
  scanf("%d",&b);
  if(a>b)
    printf ("A es mayor que B");
  else if(a<b)
    printf ("B es mayor que A");
  else
    printf ("A y B son iguales");
}
```

d) *Operadores especiales:*

i) **sizeof**

Permite conocer el tamaño en bytes que ocupan los valores asociados a cualquier tipo de datos válido en C.

ii) **new**, **delete**

Permiten reservar y liberar memoria dinámica en C respectivamente.

iii) **->**, **.** (operador punto)

Permiten acceder a los campos de una variable de tipo struct declarada como dinámica en el primer caso y como estática en el segundo.

iv) **&** (operador de dirección)

Es un operador que devuelve la dirección de memoria de su operando.

v) ***** (operador de indirección)

Es un operador que devuelve el contenido de la variable localizada en la dirección de memoria que le sigue

Ej.: `x = 5; y =18; y=*(&x);  /*y=5 */`

vi) **,** (operador coma)

Permite encadenar varias expresiones

`x={y=3,y+1} // es equivalente a poner: y=3; x=y+1=3+1=4`

vii) Operador casting: Fuerza a que una expresión sea de un determinado tipo

`x= (int) (y/z) //siendo: y, z, variables de tipo float`

6. EXPRESIONES Y SU EVALUACIÓN

6.1 EXPRESIONES

Una **expresión** en C, es cualquier combinación válida de operadores, constantes, variables o funciones.

Como ya sabemos de pseudocódigo, según los datos que intervienen y el tipo de operadores empleados, existen varios tipos de expresiones:

Aritméticas: X = 56 + 77
Relacionales: A >= B
Lógicas: (X =Y) and (X < 4)
Concatenación: CADENA1 + CADENA2
Etc.

Para evaluar una expresión el compilador tiene en cuenta las reglas de prioridad existente entre los distintos operadores así como convertir el tipo de cada uno de los operandos al tipo mayor.

Por otro lado C dispone de las **secuencias de escape** que se usan para acciones como: nueva línea, tabular y para representar caracteres no imprimibles. Una secuencia de escape está formada por el carácter \ seguido de una letra o de una combinación de dígitos. C tiene predefinidas las siguientes:

\n	nueva línea (salta a la siguiente linea)
\t	tabulador horizontal
\v	tabulador vertical (para impresora)
\b	backspace (retroceso)
\r	retorno carro (se posiciona en la primera columna)
\f	alimentación de página (sólo para impresora)
\a	bell (pitido)
\’	comilla simple
\”	comilla doble
\\	backslash (barra invertida)
\ddd	carácter ASCII, representación octal
\xdd	carácter ASCII, representación hexadecimal

Ejemplo:

```
/* Uso de las secuencias de escape */
#include <stdio.h>
main() /* Escribe diversas sec. de escape */
{
  printf("Me llamo \"Nemo\" el grande");
  printf("\nDirección: C\\ Mayor 25");
  printf("\nHa salido la letra \'L\'");
  printf("\nRetroceso\b");
  printf("\176");                    /* corresponde en binario a 126 (~) */
  printf("\n\tEsto ha sido todo");
}
```

6.2 PRIORIDAD Y ORDEN DE EVALUACIÓN DE LOS OPERADORES

La siguiente tabla muestra la prioridad y asociatividad (sentido de la evaluación) de los operadores. Los operadores situados en la misma línea tienen la misma prioridad y las filas están colocadas en orden de prioridad decreciente

OPERADORES	**ASOCIATIVIDAD**
( ) [ ] -> .	De Izquierda a Derecha
! ~ ++ - - - * & sizeof	**(De Derecha a Izquierda)**
*** / %**	De Izquierda a Derecha
+ -	De Izquierda a Derecha
<< >>	De Izquierda a Derecha
< <= >= >	De Izquierda a Derecha
&	De Izquierda a Derecha
^	De Izquierda a Derecha
|	De Izquierda a Derecha
== !=	De Izquierda a Derecha
&&	De Izquierda a Derecha
? :	**De Derecha a Izquierda**
||	De Izquierda a Derecha
= += - = *= %= / =	**De Derecha a Izquierda**
,	De Izquierda a Derecha

Ejemplo:

```
/* Jerarquía de los operadores */
#include <stdio.h>
main() /* Realiza una operaci¢n */
{
        int a=6,b=5,c=4,d=2,e=1,x,y,z,r;
        x=a*b;
        printf("%d * %d = %d\n",a,b,x);
        y=c/d;
        printf("%d / %d = %d\n",c,d,y);
        z=x+y;
        printf("%d + %d = %d\n",x,y,z);
        r=z-e;
        printf("%d = %d",r,a*b+c/d-e);
}
```

6.3 COMPORTAMIENTO DE OPERADORES ARITMÉTICOS EN CASO DE EXCEPCIÓN

La norma ANSI no establece qué comportamiento debe adoptar el compilador cuando se intenta una división por cero o cuando el resultado de un cálculo es superior o inferior a la capacidad del tipo empleado. Algunos de los comportamientos usuales son:

********** **CÁLCULO CON NÚMEROS ENTEROS**

- **Desbordamiento de la capacidad**

Operando con enteros no se realizan pruebas sobre el posible desbordamiento de la capacidad de representación del tipo utilizado.

El resultado de desbordar la capacidad es: o la interpretación del uno en el bit de mas peso como indicador de un número negativo, o la pérdida sistemática de los bits más significativos del resultado de la operación. Veamos algunos ejemplos:

Ejemplo:

```
/* ESTE PROGRAMA MUESTRA QUE LOS BITS  DE DESBORDAMIENTO SE DESPRECIAN
*/
            #include<stdio.h>
            main()
            {
              short n=32000, p=32000, q;        // el tipo short  usa 2 bytes => maximo 32767
              q=n+p;                            // se desborda   64000
              printf("%d",q);
            }
```

cuya ejecución devuelve -1536

Ejemplo:

```
 /* ESTE PROGRAMA MUESTRA QUE LOS BITS MÁS SIGNIFICATIVOS SE DESPRECIAN
*/
              #include<stdio.h>
              void main(void)
              {
                unsigned short n=64000, p=64000, q;
              // el tipo unsigned short => 65535 como máximo
                q=n+p;                    // se desborda:   128000
                printf("%u",q);
              /* %u es el código necesario para representar adecuadamente los enteros sin signo*/
              }
```

cuya ejecución devuelve 62464 (ahora no interviene el signo)

- **División por cero.**

Comprobemos el siguiente programa:

Ejemplo:

```
                        #include<stdio.h>
                        void main(void)
                          {
                           int n, p, q;
                           n=35;
                           p=0;
                           q=n/p;
                           printf("%d \n", q);
```

```
}
```

Dará como resultado "Divide error".

**************** CÁLCULOS CON NÚMEROS EN COMA FLOTANTE.**

- **Desbordamiento de la capacidad.**

El desbordamiento se produce porque el tipo de los operandos no es suficiente para recoger el valor del resultado. El siguiente programa muestra esta situación

Ejemplo:

```
#include<stdio.h>
main()
 {
  float x, y;
  x=1e200;
  y=x*x;
  printf("El cuadrado de %e es: %e \n %f \n", x, y, y);
 }
```

cuya ejecución devuelve:

```
El cuadrado de +INF es: +INF
+INF
```

No se debe confundir el desbordamiento de la capacidad de representación al evaluar una expresión con la asignación a una variable de un valor no representable en su tipo.

- **División por cero**

Se indica en tiempo de ejecución mediante el mensaje tipo *Error de sistema al dividir por cero o de desbordamiento.*

- **Resultados indefinidos.**

Cuando el valor de una variable era mayor que el máximo o menor que el mínimo del tipo al que pertenece la variable, el compilador generalmente lo indicaba mediante +INF y -INF.

Si un operando toma uno de estos valores como resultado de la operación se puede obtener una de estas tres salidas

- Un nuevo valor del mismo tipo, esto es, +INF o -INF.
- Un desbordamiento de la capacidad.
- Un error de rango (*Floating point: Square Root of Negative Number*).

El siguiente programa muestra estas situaciones

Ejemplo:

```
#include<stdio.h>
main()
  {
    float x, x1, y;
    x = 1e40;                          /*Se le da un valor grande para que tome el valor +INF*/
      printf ("x: %f  \n", x);           /*Los códigos %f y %e son los que permiten
                                         representar números en coma flotante*/
    y = x + x;
    printf ("y = INF + INF = %e \n",y);        /*Se visualiza +INF*/
    y = x * x;
```

```
    printf ("y = INF * INF = %e \n", y);        /*Se visualiza +INF*/
    x1 = 1e8;
    y = x / x1;
    printf ("y = INF / x1 = %e \n",y);          /*Se visualiza +INF*/
    x1=-1e40;                                   /*A x1 se le da el valor -INF*/
    printf (“x1 = %f\n”, x1);
    y = x / x1;
    printf (" y = INF/(-INF)=%e \n", y);  /*Ahora se visualiza Floating point error: Stack fault
    */
  }
```

- **Subdesbordamiento de la capacidad**

Esta situación se produce cuando como resultado de una operación se obtiene un valor inferior a la capacidad del tipo, por ejemplo

Ejemplo:

```
float x = 1e-30, y;
y = x * x;
printf ("%e",y);
```

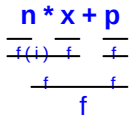
7. CONVERSIÓN DE TIPOS

Una expresión mixta es aquella en la que intervienen dos operandos de distinto tipo. Por ejemplo, con las declaraciones int n,p; float x; la expresión n * x + p es una expresión mixta.

La expresión es evaluada por el compilador siguiendo las reglas de evaluación; es decir, en primer lugar se evalúa n*x. Por tratarse de operandos de tipo diferente el compilador debe introducir las instrucciones necesarias para convertir el valor de n (tipo int) en un valor de tipo float. El compilador no convierte int en float, sino que proporciona las instrucciones para que se realice la conversión, y estas instrucciones se ejecutan a la vez que las del programa.

El resultado es de tipo float ya que el producto en realidad se realiza sobre dos datos de tipo float.

La suma de este resultado con p, es la suma de un dato de tipo float con otro de tipo int. Por tanto También es necesaria la conversión previa de int a float. El resultado final es de tipo float ya que la operación se realiza sobre operandos float.



Las conversiones de tipo de int a float se denominan CONVERSIÓN DE AJUSTE DE TIPO. Las conversiones de ajuste de tipo solo pueden hacerse en el sentido en que se permita salvar el valor inicial, es decir, que respete la integridad de los datos.

Las situaciones que requieren una conversión de tipo son:

i) Operaciones entre variables de distinto tipo: conversión al tipo más grande, “**promoción**”.

ii) Asignación de un resultado a una variable de otro tipo: puede ser una promoción, o una “**pérdida de rango**”. Como vimos en el apartado anterior, esta última situación no garantiza que el resultado final sea correcto.

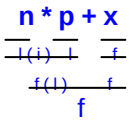
En cualquier operación en que aparezcan tipos diferentes, se elevan las categorías de los tipos de menor rango para igualarlos a la del mayor:

int → **long** → **float** → **double** → **long double**

Por tanto se puede convertir directamente long en double o int en long double, pero nunca se podrá convertir float en long, es decir en otro que se encuentre a su izquierda.

Dada una expresión mixta, la elección de la conversión se hace considerando uno a uno los operandos afectados y nunca se elige considerando la expresión global. Por ejemplo, con las declaraciones int n; long p; float x; la expresión n*p+x se evalúa como sigue

1. Conversión de n en tipo long.
2. Multiplicación por p.
3. Conversión del resultado de la multiplicación a tipo float.
4. Suma con x.
5. El resultado es de tipo float.



Los operadores numéricos no están definidos para los datos de tipo short y char, y en consecuencia las conversiones de ajuste de tipo no son suficientes para regular todos los casos que se puedan presentar.

char → **int** **short** → **int**

Para los casos en los que intervienen datos de tipo short o char, la norma ANSI establece que todo valor de uno de estos dos tipos presente en una expresión aritmética se convierte automáticamente en un dato de tipo int con independencia del tipo de los demás operandos. Estas conversiones automáticas se denominan promociones numéricas o conversiones sistemáticas. Por ejemplo, con las declaraciones short p1, p2, p3; float x; la expresión p1 * p2 + p3 * x, se evalúa como sigue:

1. Se promociona p1 a int.
2. Se convierte p2 en dato de tipo int.
3. Se promociona p3 a int.
4. Multiplicación de p1 por p2 (el resultado es de tipo int)
5. Conversión de ajuste de tipo de int a float en p3
6. Multiplicación de p3 por x
7. Conversión de ajuste de tipo int a float
8. Suma. El resultado es tipo float.

No es recomendable mezclar en una misma expresión enteros con y sin signo, pues las conversiones que resultan suelen carecer de sentido.

La conversión sistemática de char a int tiene sentido cuando un valor de tipo char se considera no como un carácter determinado sino como el código de ese carácter, esto es, como un conjunto de 8 bits. La ultima interpretación puede hacer corresponder a ese valor el del entero que representa la combinación binaria. Por ejemplo, en ASCII el carácter L se representa mediante la combinación binaria 01001100 que equivale al número entero 76.

Esta asociación de 76 por L no es la misma en todos los ordenadores ni en los distintos sistemas operativos.

La conversión de char a int permite emplear variables de 1 octeto para representar números enteros y Así ahorrar memoria.

Dada la declaración char c1, c2; , las expresiones c1+1 y c1-c2 se evalúan como siguen

a) c1+1

1. Se promociona c1 a int
2. Se realiza la suma de los dos enteros
3. El resultado entero contiene el código de c1 más uno.

b) c1-c2

1. Se promociona c1 a int
2. Se promociona c2 a int
3. Se realiza la resta.

Los enteros en C pueden tener signo o no, y dado que los datos de tipo char pueden considerarse como enteros, conviene conocer las conversiones pertinentes. Así, unsigned char promociona a int tomando valores comprendidos entre 0 y 255, mientras que signed char lo hace tomando valores comprendidos entre -127 y 128.

Por ultimo, la instrucción c1=c1+1 donde c1 es una variable de tipo char, evalúa la expresión c1+1 resultando un número entero que se convierte en un valor del tipo de c1 (char), es decir, se realiza una conversión de char a int. Todo opera como si la instrucción incrementase el valor de c1 en 1.

Ejemplo:

```
#include<stdio.h>
main()
  {
    char c1, c2;
    c1='m';
    c2=23;
    printf("c1-c2 = %d equivale al caracter c1-c2 = %c \n", c1-c2, c1-c2);
    c1=c1+1;
    printf("c1 = %d equivale al caracter c1= %c \n", c1, c1);
  }
```

Por otro lado el programador puede realizar una conversión de forma explícita mediante el **casting**, indicando entre paréntesis el tipo al que se desea convertir la variable o la expresión.

Ejemplo**:**

```
int  i,j;
float res;
…….
res = (float)  i / j;              /* division real con enteros */
printf (“ res vale %f”, res);
```

8. ÁMBITO DE LAS VARIABLES

Cualquier variable que vaya a ser utilizada en un programa en C debe ser declarada e inicializada previamente, como ya hemos visto en algunos de los ejemplos anteriores, de una de las dos formas siguientes :

tipo nombre_variable; ó tipo nombre_variable = valor ;

/* En ambos casos nos declaramos una variable nombrada con el identificador *nombre_variable* que pertenece a un *tipo* de datos e inicializada en el segundo caso con el dato *valor* */

Ejs.: int EDAD = 27; char RESPUESTA = 's';
int X = 1, Y, Z;

Las **variables** en C, se pueden declarar :

a) Al principio de un bloque (entendido éste como una sentencia compuesta y delimitada por los simbolos { }) , es decir, antes de cualquier sentencia del bloque. Estas variables, que se denominan **locales,** sólo pueden ser utilizadas por sentencias o instrucciones del bloque en el que están definidas.

Además sólo existen durante la ejecución del código asociado al bloque en el que son declaradas, destruyéndose al finalizar la misma. Las variables locales se inicializan como el resto de variables.

El lugar más común en el que se declara una variable local es en el cuerpo de una función.

Ejemplo:

```
/* Uso de las variables */
#include <stdio.h>
main() /* Suma dos valores */
{
  int num1=4, num2, num3=6;
  printf ("El valor de num1 es %d",num1);
  printf ("\nEl valor de num3 es %d",num3);
  num2 = num1+num3;
  printf ("\nnum1 + num3 = %d",num2);
}
```

b) Fuera de las funciones, por ejemplo antes de la definición de la función principal main(). En este caso se denominan variables **globales** y son conocidas por todos los bloques del programa (incluidos los cuerpos de las distintas funciones), pudiéndose acceder a ellas desde cualquiera de ellos.

Estas variables existen durante toda la ejecución del programa, por lo que están ocupando siempre memoria , por ello y porque las variables globales hacen que el código de una función sea poco transportable, se deben utilizar el menor número de ellas.

Ej.:

```
……
int X;
void main()
{
        int Y;
        ……..
}
```

X es global y la variable Y es local a la función principal

Ejemplo:

```
/* Declaración de variables */
#include <stdio.h>
int a;
main() /* Muestra dos valores */
{
  int b=4;
  printf("b es local y vale %d",b);
  a=5;
  printf("\na es global y vale %d",a);
}
```

En cuanto a la declaración e inicialización de **constantes**, se pueden realizar de dos formas:

a) En el entorno del programa mediante la directiva *define*, del siguiente modo:

#define Nombre_Constante valor // El simbolo # debe escribirse en la 1ª columna
// de la línea.

Ej.: #define CLAVE 3121

b) En los mismos lugares que una variable mediante la palabra reservada *const*

const tipo NombreConstante = valor;

Ej.: const int CLAVE = 3121;

La declaración o inicialización de constante o variable termina con un punto y coma.

Ejemplo:

```
/* Uso de las constantes */
#include <stdio.h>
#define pi 3.1416
#define escribe printf
main() /* Calcula el perímetro */
{
  int r;
  escribe("Introduce el radio: ");
  scanf("%d",&r);
  escribe("El perímetro es: %f",2*pi*r);
}
```

9. SINTAXIS DE LAS INSTRUCCIONES

9.1 INSTRUCCIONES Y BLOQUES

En C, cualquier expresión válida acabada en ; es una *instrucción o sentencia* simple.

Ej.: **Z = X + Y;**

Un conjunto de declaraciones y sentencias simples se pueden agrupar en una sentencia compuesta o *bloque* para que puedan ser tratadas como una sentencia única. Las sentencias compuestas empiezan con una { (llave abierta) y terminan con una } (llave cerrada). Generalmente las sentencias compuestas se utilizan dentro de las sentencias de control del programa para indicar el conjunto de operaciones a realizar cuando se cumpla una condición ó el conjunto de operaciones que se deben repetir mientras o hasta que se cumpla una condición ó para definir el *cuerpo de una función.*

Ej**.;**
```
{ X = 3;
  Y = 5;
  Z = X + Y + 7;}
```

9.2 OPERACIONES DE ENTRADA Y SALIDA DE DATOS EN C

En C al igual que en la mayoría de los lenguajes de programación los datos de entrada que van a ser almacenados y manipulados por el programa, se proporcionan por teclado, entrada estándar (**stdin**), o mediante ficheros de datos almacenados en sistemas de almacenamiento secundarios como por ejemplo el disco duro, y los resultados que se obtienen se visualizan en la pantalla, salida estándar (**stdout**), o son almacenados en un fichero de datos.

Mientras que otros lenguajes de programación utilizan instrucciones para realizar las E/S, C lo hace utilizando funciones de librerías predefinidas para ello. Por lo que para poder utilizarlas tendremos que incluir en nuestro programa el fichero cabecera en el que están definidas: **stdio.h**.

Por ello todo fichero que use estas funciones deberá acceder a dicha librería incluyendo la línea:

#include <stdio.h>

En el caso de E/S por teclado y pantalla, las dos funciones de E/S con formato, más importantes son:

i) *printf*: que visualiza en pantalla datos cuyo formato de salida queda definido en la propia función. Su prototipo es:

int printf (char *CadenaDeControl, lista_de_argumentos)

Devuelve la cadena que se va a visualizar o un valor en el fichero de salida estándar (stdout). El valor entero corresponde al nº de caracteres escritos y es negativo en caso de error.

CadenaDeControl es un literal, es decir, una cadena de caracteres entre comillas dobles, que está formada por una combinación de caracteres y especificadores de formato, que se corresponden con cada uno de los argumentos. Si existen más argumentos que especificadores, los argumentos en exceso se ignoran. Estos especificadores de formato se construyen de la siguiente forma:

%[flags][AnchoCampo].[Precision]Type[SecuenciaEscape]

Flags Para justificar la salida. Si el AnchoCampo reservado es mayor que el dato, éste se justifica a la derecha, a no ser que usemos un – en cuyo caso se justifica a la izquierda, quedando en blanco el resto de caracteres.

AnchoCampo Nº total de caracteres que queremos reservar.

Precision Para nº decimales que queremos visualizar.

Type especifica el tipo de dato que se va a visualizar:

%d ó %i para los enteros.
%f para los float (reales en coma flotante, formato decimal)
%e float en notación científica (reales en forma exponencial)
%g usa %e o %f, la que sea más corta de las representaciones
%ld para los long int
%lf para los double
%Lf para los long double
%c para los char (un único caracter)
%s para las cadenas
%u para los enteros sin signo
%p para los punteros
%o para los datos representados en octal
%x para los datos representados en hexadecimal
%% imprime un signo %
Otros: %le, %lg, para los double en los e y g; %hu para short unsigned; %lu long unsigned...
Type es obligatorio que aparezca al igual que el %

SecuenciaEscape permite añadir caracteres: salto de línea \n , tabuladores \t ,etc.

La lista_de_argumentos está formada por las variables y constantes que se quieren visualizar. Deberá haber el mismo número de especificadores de formato como argumentos tenga la lista. En el caso de las cadeneas, este formato, por ejemplo %5.7s visualiza la cadena con al menos 5 caracteres de largo y no más de siete. Si la cadena es más larga, el ordenador truncará los decimales desde el final.

Ejemplos:

```
printf("%-5.2f",123.234)          |123.23|
printf("%5.2f",3.234)             |  3.23|
printf("%10s","hola")             |      hola|
printf("%-10s","hola")            |hola      |
printf("%5.7s","123456789")       |1234567|
```

Ejemplo:

```
printf  ("%d \n %c \n",5,  'c');
visualizará: 5
             c
```

Ejemplo:

```
/* Uso de la sentencia printf() 1. */
#include <stdio.h>
main() /* Saca por pantalla una suma */
{
        int a=20,b=10;
        printf("El valor de a es %d\n",a);
        printf("El valor de b es %d\n",b);
        printf("Por tanto %d+%d=%d",a,b,a+b);
}
```

Ejemplo:

```
/* Uso de la sentencia printf() 2. */
#include <stdio.h>
main() /* Modificadores 1 */
{
        char cad[]="El valor de";
        int a=-15;
        unsigned int b=3;
        float c=932.5;
        printf("%s a es %d\n",cad,a);
        printf("%s b es %u\n",cad,b);
        printf("%s c es %e o %f",cad,c,c);
}
```

Ejemplo:

```
/* Uso de la sentencia printf() 3. */
#include <stdio.h>
main() /* Modificadores 2 */
{
        char cad[ ]="El valor de";
        int a=25986;
        long int b=1976524;
        float c=9.57645;
        printf("%s a es %9d\n",cad,a);
        printf("%s b es %ld\n",cad,b);
        printf("%s c es %.3f",cad,c);
        getch();
}
```

Se visualizaría:

El valor de a es 25986
El valor de b es 1976574
El valor de c es 9.576

ii) *scanf*: que se utiliza para recoger datos del teclado y almacenarlos en una variable. Su prototipo es el siguiente:

int scanf (char *CadenaDeControl, lista_de_argumentos)

CadenaDeControl está formado por especificadores de formato, construidos como con printf.

lista_de_argumentos está formada por las **direcciones** de las variables en las que se van a almacenar los datos que se recojan desde el teclado

Ejemplo:

```
int x;
printf ("Introduce un entero:");
scanf ("%d",&x);
```

Ejemplo:

```
/* Uso de la sentencia scanf. */
#include <stdio.h>
main() /* Solicita dos datos */
{
        char nombre[10];
        int edad;
        printf("Introduce tu nombre: ");
        scanf("%s",nombre);
        printf("Introduce tu edad: ");
        scanf("%d",&edad);
}
```

Solución de problemas con el buffer intermedio de teclado

Cuando usamos alguna función de E/S de flujo, se produce un almacenamiento de información intermedio (buffered). Como la entrada y salida de datos en disco es una operación lenta, este intercambio de información entre la entrada estándar (stdin) y la salida estándar (stdout), normalmente el teclado y la pantalla, aligera la ejecución de la información.

La información que ha de salir por pantalla se ubica en el buffer, y cuando éste está lleno, es cuando se envían los datos para ser procesados. Sin embargo muchos lenguajes de alto nivel tienen un problema con el almacenamiento intermedio, y es que, no todas las aplicaciones pueden realizarse si no reciben los datos intermedios de forma secuencial y no por bloques.

Por ejemplo, si nuestro programa ejecuta varias sentencias de salida “printf”, por ejemplo, con datos pequeños tipo char (8 bits), que no llenan en todo momento el buffer intermedio, éste no se vacía y el programa pierde la información. Simplemente no continúa o finaliza con pérdida de datos.

Usualmente, la solución consiste en llamar a una función adecuada, para vaciar el buffer en el momento que sea necesario. El método y la función varían según el lenguaje de programación. C, resuelve el problema vaciando automáticamente el buffer, siempre que el programa termina. Pero esto no es suficiente, en determinadas circunstancias, al usar varias veces las instrucciones de E/S, puede ocurrir que el programa, aparentemente correcto, no lee adecuadamente los datos de E/S. Éste queda paralizado, esperando un dato que ya hemos introducido..... Entonces nos vemos obligados a forzar la salida de datos del buffer en el momento que nos interese. Para ello usamos la función **fflush**.

Observemos el siguiente ejemplo:

```
#include <stdio.h>
#include <conio.h>
void main()
{  char OPCION='S';
   int a,b;
   printf ("Introduce un numero  ");
   scanf ("%d",&b);
   while (OPCION=='S')
   {       for (a=1;a<10;a++)
           printf ("\n%d x %d = %d",a,b,a*b);
   printf ("\nDeseas operar otra vez (S/N):");
   // fflush(stdin);
   scanf ("%c",&OPCION);
   }
   getch();
}
```

Según habilitemos o no la línea con fflush, el programa funciona o no.

9.3 SENTENCIAS DE CONTROL

C utiliza las mismas sentencias de control que cualquier lenguaje de programación estructurado. Estas son las siguientes:

a) **Sentencias Alternativas o de Seleción**. En C se utilizan tres tipos:

i) Sentencias if-else

Se utilizan para elegir qué sentencia simple o compuesta, será la siguiente en ejecutarse. La elección se realizará tras evaluar una expresión. Su sintaxis es:

```
if (expresión)
        Sentencia1;
[ else
        Sentencia2;]
```

En esta sentencia se evalúa expresión, si es cierta se ejecutará a continuación Sentencia1, y si es falsa no se realiza ninguna acción. En el caso de que aparezca la palabra reservada else y la expresión sea falsa, se ejecutará Sentencia2. (Sentencia1 y 2 pueden ser un bloque de instrucciones, en cuyo caso irán entre llaves).

Ejemplo:

```
/* Uso de la sentencia condicional if. */
#include <stdio.h>
main() /* Simula una clave numérica de acceso, con un único intento */
{
  unsigned long USUARIO, CLAVE=18276;
  printf("Introduce tu clave: ");
  scanf("%d",&USUARIO);
  if (USUARIO == CLAVE)
        {
        printf("Acceso permitido");
        printf("Enhorabuena");
        }
else
        printf("Acceso denegado");
}
```

En C existe el **operador condicional** ?: , equivalente al if-else, que es el único operador con tres operandos en C, (ternario), y que tiene la siguiente forma general:

expresion_1 ? expresion_2 : expresion_3;

Se evalúa expresion_1. Si el resultado de dicha evaluación es ***true*** (#0), se ejecuta expresion_2; si el resultado es ***false*** (=0), se ejecuta expresion_3.

Ejemplo: (x > y ? x = x + 1 : y = y + 1)

ii) Sentencia if-else compuesta

```
if (expresión)
        Sentencia1;
else if (expresion2)
        Sentencia2;
.............
[else  (expresionf)
        Sentenciaf;]
```

En esta sentencia se evalúa expresión, si es cierta se ejecutará a continuación Sentencia1, y si es falsa no se realiza ninguna acción. En el caso de que aparezcan las palabras reservadas else if y expresión2 sea cierta se ejecutará Sentencia2. Pueden aparecer varios else-if y un else al final de todos ellos para ejecutar una acción por defecto, en caso de no ser cierto ninguno de ellos.

Ejemplo:

```
/* Uso de la sentencia condicional else-if. */
#include <stdio.h>
main()        /* Escribe bebé,,niño, joven o adulto */
{  int edad;
   printf("¿Cuántos años tienes?: ");
   scanf("%d",&edad);
   if (edad<1)
        printf("No es una edad correcta.");
   else if (edad<3)
        printf("Eres un bebé,");
   else if (edad<13)
        printf("Eres un niño");
   else if (edad<18)
        printf("Eres un joven");
   else
        printf("Eres un adulto");
}
```

iii) Sentencia switch

Elige la siguiente instrucción a ejecutar tras evaluar si una expresión es igual a uno entre varios valores. Su formato o sintaxis es:

```
Switch (expresion)
 {
        case  ValorConstante1 :
                Sentencia1;
                break;
        case  ValorConstante2 :
                Sentencia2;
                break;
                ...
        [default: sentenciaN;]
 }
```

Se comprueba si el valor de expresión es igual a alguna de las constantes en cuyo caso ejecutará la sentencia correspondiente y se pasarán a ejecutar las siguientes instrucciones al switch La sentencia N asociada a la palabra reservada default que es opcional, se ejecutará si no se cumple que el valor de expresión sea igual a alguno de los valores constantes especificados. Si no existe la cláusula default y el valor de expresión no coincide con ninguna constante no se ejecutará ninguna de las sentencias.

Por otro lado, es importante resaltar que si se omiten los break, cuando se entre en un caso que coincida con la expresión, se ejecutarán, no sólo las sentencias del caso coincidente, sino que además se ejecutan todas las siguientes pertenecientes a los case sucesivos, a no ser que aparezca un break en alguno de ellos o se finalice el switch.

Ejemplo:

```
/* Uso de la sentencia condicional switch1. */
#include <stdio.h>
main() /* Escribe el d¡a de la semana */
{
  int dia;
  printf("Introduce el ordinal del día de la semana(1 al 7): ");
  scanf("%d",&dia);
  switch(dia)
  {
     case 1: printf("Lunes"); break;
     case 2: printf("Martes"); break;
     case 3: printf("Mi,rcoles"); break;
     case 4: printf("Jueves"); break;
     case 5: printf("Viernes"); break;
     case 6: printf("S bado"); break;
     case 7: printf("Domingo"); break;
  }
}
```

Ejemplo:

```
/* Uso de la sentencia condicional switch2. */
#include <stdio.h>
main()          /*Indica en qué posible base Bi, Oc, De o He,
                  se encuentra un dígito */
{
  char dig;
  printf("Introduce un dígito: ");
  scanf("%c",&dig);
  switch(dig)
  {
     case ‘0’:
     case ‘1’: printf("Binario\n");
     case ‘2’:
```

```
    case ‘3’:
    case ‘4’:
    case ‘5’:
    case ‘6’:
    case ‘7’: printf (“Octal\n”);
    case ‘8’:
    case ‘9’:
    case ‘A’:
    case ‘B’:
    case ‘C’:
    case ‘D’:
    case ‘E’:
    case ‘F’: printf (“Hexadecimal\n”);
    }
}
```

b) Sentencia break

Esta sentencia finaliza la ejecución de una sentencia “do”, “for”, “switch” o “while”. Cuando se encuentra anidada, break sólo finaliza la ejecución de la sentencia en la que está incluida.

c) Sentencias repetitivas

i) Sentencia while

Permite repetir un conjunto de instrucciones mientras una expresión sea cierta. Como la expresión es lo primero en evaluarse, si dicha expresión es falsa (=0) inicialmente, las instrucciones no se ejecutarán.

La sintaxis de esta instrucción es la siguiente:

```
while (expresion)
   {
        sentencias;
   }
```

Ejemplo:

```
/* Uso de la sentencia while. */
#include <stdio.h>
main()        /* Escribe los números del 1 al 100 */
{
  int N=1;
  while(N<=100)
  {
    printf("%d\n",N);
    N++;
  }
}
```

ii) *Sentencia do while*

Repite la ejecución de un conjunto de sentencias hasta que una expresión sea cierta, dicha expresión se evaluará tras haberse ejecutado al menos una vez el conjunto de sentencias.

La sintaxis de esta instrucción es la siguiente:

```
do
{
   sentencias;
}  while (expresion);
```

Ejemplo:

```
/* Uso de la sentencia do...while. */
#include <stdio.h>
main() /* Muestra un menú si no se pulsa 4 */
{
  char OPCION;
  do{
    printf ("1.- Iniciar\n");
    printf ("2.- Abrir\n");
    printf ("3.- Grabar\n");
    printf ("4.- Salir\n");
    printf ("Escoge una opción (1, 2, 3 ó 4): ");
    OPCION=getchar();
    switch(OPCION)
                {
                  case '1':  printf("Has elegido Opción 1");
                             break;
                  case '2':  printf("Has elegido Opción 2");
                             break;
                   case '3':  printf("Has elegido Opción 3");
                }

     }
while(OPCION!='4');
}
```

iii) *Sentencia for*

Semánticamente es igual al while pero sintácticamente es algo diferente:

```
for (expresion1;expresion2;expresion3)
{
        sentencias;
}
```

Esto es equivalente a:

```
expresion1;
while (expresion2)
        {
                sentencias;
                expresion3;
         }
```

En expresion1 se inicializa la variable de control del for; expresion2 es la condición de permanencia en el bucle; expresion3 modifica en cada iteracion del for la variable de control para hacer que expresión2 sea falsa y poder terminar las repeticiones. Estas reglas no serán tenidas en cuenta si queremos realizar un bucle infinito (no es estructurado).

Ejemplo:

```
/* Uso de la sentencia for. */
#include <stdio.h>
main() /* Escribe la tabla de multiplicar */
{
  int N, i, producto;
  printf ("Introduce un número: ");
  scanf ("%d",&N);
  for (i=0;i<=10;i++)
   {
    producto = N * i;
    printf ("\n%3d X %d = %4d\n", i,N, producto);
   }
}
```

d) Sentencia continue

Esta sentencia, cuando aparece dentro de un bloque controlado por un “do”, “while” o “for”, pasa el control para que se ejecute la siguiente iteración.

Ejemplo:

```
/* Uso de la sentencia CONTINUE. */
#include <stdio.h>
void main() /* Escribe del 1 al 100 menos los múltiplos de 11 */
{
  int NUM =1;
  while(NUM <=100)
  {
    if ((NUM%11)==0)
    {
          NUM ++;
          continue;
     }
  printf("%d\n", NUM);
  NUM ++;
  }
}
```

e) Sentencia goto

Esta sentencia nos permite saltar dentro de una función a otra sentencia marcada con una etiqueta.

En realidad, “goto” , **nunca** es necesario y su uso mal pensado, conduce a programas mal estructurados. No es aconsejable su uso.

Ejemplo:

```
.................
while (...)
        {
          .......
          if (A<B)
             goto lugar;   // elegimos la palabra “lugar” como etiqueta
        }
.................
lugar: printf (“ A es mayor que B”);
.................
```

9.4 SENTENCIAS DEL PREPROCESADOR

El ***preprocesador*** del lenguaje C permite crear ***macros*** (sustitución en el programa de constantes simbólicas o texto, con o sin parámetros), realizar compilaciones condicionales e incluir archivos, todo ello antes de que empiece la compilación propiamente dicha. El preprocesador de C reconoce los siguientes comandos:

#define, #undef
#if, #ifdef, #ifndef, #endif, #else, #elif
#include
#pragma (da instrucciones de implementación al compilador sobre cómo interpretar el código fuente y generar el ejecutable)
#error (instruye al compilador para generar un mensaje de error definido por el usuario)

Los comandos más utilizados son: **#includ**e, **#defin**e.

Comando #include

Cuando en un archivo ***.c*** se encuentra una línea con un **#include** seguido de un nombre de archivo, el preprocesador la sustituye por el contenido de ese archivo.
La sintaxis de este comando es la siguiente:

#include "nombre_del_archivo"
#include <nombre_del_archivo>

La diferencia entre la primera forma (con comillas "...") y la segunda forma (con los símbolos <...>) estriba en el directorio de búsqueda de dichos archivos. En la forma con comillas se busca el archivo en el directorio actual y posteriormente en el directorio estándar de librerías (definido normalmente con una variable de entorno del MS-DOS llamada INCLUDE, en el caso de los compiladores de Microsoft). En la forma que utiliza los símbolos <...> se busca directamente en el directorio estándar de librerías. En la práctica, los archivos del sistema ***(stdio.***h, ***math.***h, etc.) se incluyen con la segunda forma, mientras que los archivos hechos por el propio programador se incluyen con la primera.
Este comando del preprocesador se utiliza normalmente para incluir archivos con los ***prototipos*** (declaraciones) de las funciones de librería, o con módulos de programación y prototipos de las funciones del propio usuario. Estos archivos suelen tener la extensión ***.h,*** aunque puede incluirse cualquier tipo de archivo de texto.

Comando #define

El comando **#define** establece una ***macro*** en el código fuente. Existen dos posibilidades de definición:

#define NOMBRE texto_a_introducir
#define NOMBRE(parámetros) texto_a_introducir_con_parámetros

Antes de comenzar la compilación, el preprocesador analiza el programa y cada vez que encuentra el identificador NOMBRE lo sustituye por el texto que se especifica a continuación en el comando **#define**. Por ejemplo, si se tienen las siguientes líneas en el código fuente:

```
#define e 2.718281828459
...
void main(void) {
double a;
a= (1.+1./e)*(1.-2./e);
...
}
```

al terminar de actuar el preprocesador, se habrá realizado la sustitución de **e** por el valor indicado y el código habrá quedado de la siguiente forma:

```
void main(void) {
double a;
a= (1.+1./2.718281828459)*(1.-2./2.718281828459);
...
}
```

Este mecanismo de sustitución permite definir constantes simbólicas o valores numéricos (tales como **e**, **pi**, etc.) y poder cambiarlas fácilmente, a la vez que el programa se mantiene más legible. De forma análoga se pueden definir ***macros con parámetro***s: Por ejemplo, la siguiente macro calcula el cuadrado de cualquier variable o expresión,

```
#define CUAD(x) ((x)*(x))
void main() {
double a, b;
...
a = 7./CUAD(b+1.);
...
}
```

Después de pasar el preprocesador la línea correspondiente habrá quedado en la forma:

```
a = 7./((b+1.)*(b+1.));
```

Obsérvese que **los paréntesis son necesarios** para que el resultado sea el deseado, y que en el comando **#define** no hay que poner el carácter (;). Otro ejemplo de ***macro*** con dos parámetros puede ser el siguiente:

```
#define C_MAS_D(c,d)   (c + d)    /* no ponemos paréntesis, deberíamos poner ((c) + (d))
void main() {
double a, r, s;
a = C_MAS_D(s,r*s);
...
}
```

con lo que el resultado será:

```
a = (s + r*s);
```

El resultado es correcto por la mayor prioridad del operador (*) respecto al operador (+).
Cuando se define una *macro con argumentos* conviene ser muy cuidadoso para prever todos los posibles resultados que se pueden alcanzar, y garantizar que todos son correctos. En la definición de una *macro* pueden utilizarse *macros* definidas anteriormente. En muchas ocasiones, *las macros son más eficientes que las funcione*s, pues realizan una sustitución directa del código deseado, sin perder tiempo en copiar y pasar los valores de los argumentos.
Ejemplo:

```
#define max(A,B)   ((A)>(B) ? (A) : (B))
.....................
x = max (p+q,r+s);
/* Se expande a
x = ((p+q) > (r+s) ? (p+q) : (r+s)); */
```

No obstante hay que tener cuidado con los efectos laterales. Así pues :

```
x  = max( i++, j++);
```

se expande evaluando dos veces los postincrementos del elemento mayor.
Es recomendable tener presente que el comando **#define**:
– No define variables.
– Sus parámetros no son variables.
– En el preprocesamiento no se realiza una revisión de tipos, ni de sintaxis.
– Sólo se realizan sustituciones de código.
Es por estas razones que los posibles errores señalados por el compilador en una línea de código fuente con **#define** se deben analizar con las substituciones ya realizadas. Por convención entre los programadores, ***los nombres de las macros se escriben con mayúscula***s.
Existen también muchas *macros predefinidas a algunas variables internas del sistem*a. Algunas son:

_ _DATE_ _	Fecha de compilación
_ _FILE_ _	Nombre del archivo
_ _LINE_ _	Número de línea
_ _TIME_ _	Hora de compilación

Ejemplo:

```
printf (“ %s | %s | Número de línea: %d”, _ _FILE_ _,_ _DATE_ _,_ _LINE_ _);
```

Se puede definir una *macro* sin *texto_a_sustituir* para utilizarla como ***señal*** a lo largo del programa. Por ejemplo:

```
#define COMP_HOLA
```

La utilidad de esta línea se observará en el siguiente apartado.

Comandos #ifdef, #ifndef, #else, #endif, #undef

Uno de los usos más frecuentes de las *macros* es para establecer bloques de compilación opcionales. Por ejemplo:

```
#define COMP_HOLA
void main()
   {
#ifdef COMP_HOLA      // si está definida la macro llamada COMP_HOLA
        printf("hola");
#else                       // si no es así
        printf("adios");
#endif
    }
```

El código que se compilará será **printf("hola")** en caso de estar definida la *macro* **COMP_HOLA**; en caso contrario, se compilará la línea **printf("adios"**). Esta compilación condicional se utiliza con frecuencia para desarrollar *código portable* a varios distintos tipos de computadores; según de qué computador se trate, se compilan unas líneas u otras. Para eliminar una *macro* definida previamente se utiliza el comando ***#undef***:

```
#undef COMP_HOLA
```

De forma similar, el comando **#ifndef** pregunta por la *no-definición* de la *macro* correspondiente.

Ejemplo:

```
#include <stdio.h>
#include <math.h>
//#define DAT_REALES  //si habilitamos esta línea se opera con reales simples
#define DAT_REALES_DOBLE
void main()
{
  float A,B,H;
  double AA,BB,HH;
#ifdef DAT_REALES
#undef DAT_REALES_DOBLE printf ("Introduzca los catetos: ");
  printf ("Introduzca los catetos: ");
  scanf ("%f %f",&A,&B);
  //H=pow((pow(A,2)+pow(B,2)),0.5);
  H=pow((pow(A,2.)+pow(B,2.)),0.5);
  printf ("La hipotenusa vale %f",H);
  printf ("\nHemos operado con reales simples");
#endif
#ifdef DAT_REALES_DOBLE
  printf ("\nIntroduzca los catetos: ");
  scanf ("%lf %lf",&AA,&BB);
  //H=pow((pow(AA,2)+pow(BB,2)),0.5);
  HH=pow((pow(AA,2.)+pow(BB,2.)),0.5);
  printf ("La hipotenusa vale %lf",HH);
  printf ("\nHemos operado con reales dobles");
#endif
}
```

10. USO DE LAS FUNCIONES EN C

Funciones

La programación modular y estructurada se implementan en C, utilizando funciones.
Las funciones son uno de los elementos más importantes de un programa en C. Una función es un bloque o conjunto de instrucciones y datos que resuelven un problema claramente definido.
La única función que debe existir obligatoriamente en un programa es main(), función principal.

La ejecución de un programa acaba cuando lo hace la función principal o cuando desde alguno de sus módulos se ejecuta la instrucción **exit**.
En una función hay que distinguir las siguientes partes:

a) Declaración o prototipo de la función.

La declaración de una función consiste en describirla antes de utilizarla, para que el compilador pueda encontrar su definición y conocer sus parámetros formales así como el tipo de datos que va a retornar la función. Es decir la declaración nos indica cuántos parámetros tiene la función y de que tipo son, así como el tipo del valor retornado.

El *prototipo* o *declaración* de una función se realiza:

i) En el caso de módulos o funciones principales, antes de la función main, en el **entorno** del programa que también contiene la declaración de variables globales del programa.

ii) En el caso de submódulos, dentro del entorno del módulo o función en la que se incluye el submódulo.

El formato de la declaración de una función es:

TipoDatoRetorno NombreFuncion (Tipo [Parametro1],Tipo [Parametro2],...);

Siendo:

TipoDatoRetorno el tipo al que pertenece el valor devuelto por la función. Puede ser cualquier tipo de datos (simple o compuesto) válido en C.
En caso de ser el tipo **void**, la función se comportará como un procedimiento.

Tipo [Parametro1], Tipo [Parametro2],.... son los tipos a los que pertenecen las informaciones adicionales o parámetros actuales con los que se va a operar dentro de la función. Cuando queramos declarar una función sin argumentos, pondremos después del nombre y entre paréntesis el tipo **void**:
TipoDatoRetorno NombreFuncion (void);
No es necesario poner el nombre de los parámetros: Parametro1,...etc, pero es conveniente hacerlo para indicar su funcionalidad.

Ejemplos:

```
int Producto(int A, int B);   // o    int Producto(int , int );
void PedirDatos(void);
void Menu(int OPCION);   // o    void Menu(int);
```

b) Definición de la función:

En la definición es donde se escribe el código de la función, es decir, el conjunto de sentencias que describen la acción o acciones que realiza la función.

El formato de la definición de una función es :

```
TipoDatoRetorno NombreFuncion (Tipo Parametro1,Tipo Parametro2,...)
        {
          Declaracion de variables locales;
          Sentencias ejecutables;
          return [(expresion)];
        }
```

A la primera línea de la definición se le denomina **cabecera** de la función. Los tipos de datos y el orden en el que aparecen deben coincidir con los especificados en la declaración de la función.

Al resto, lo que va entre llaves, se le denomina **cuerpo** de la función, y es donde se declaran las variables locales de la función y donde se escriben las instrucciones que tienen que ejecutarse cuando se llame a la función..

Parametro1 y Parametro2, son los parámetros formales de la función.

El nombre de la función así como el de los tipos de los parámetros formales de la cabecera de la definición se deben corresponder con los del prototipo.

La sentencia **return** hace que finalice la ejecución de la función devolviendo el control a la función que la llamó. Además devuelve a dicha función el valor de expresión que tiene que ser del mismo tipo que TipoDatoRetorno. Expresión, no es de obligado uso y si TipoDatoRetorno es el tipo void, de hecho, en este caso se puede omitir toda la línea. Se trataría de un procedimiento.

Ejemplo:

```
int Producto (int A, int B)
 {
    return (A*B);
 }
```

Las funciones que no estén declaradas y *definidas* (bloque de instrucciones) simultáneamente, antes de la función main(), han de ser *declaradas* en el entorno de la función principal main() (cabecera) y *definidas* después del bloque de intrucciones de dicha función principal (este segundo procedimiento es el que usamos).

La definición de una función no puede contener la definición de otra función. Esto no quiere decir que no pueda contener la declaración de otra función, pues en este caso la segunda será un submódulo de la primera. Cuando una función es llamada por otra, la definición de la segunda debe escribirse a continuación de la primera, como ocurre en la función principal main().

En nuestros programas la mayoría de las funciones van a ser llamadas por la función main(), es decir, el resto de funciones que utilicemos serán submódulos de la función main(), por ello:

Las declararemos en el entorno (cabecera) de la función main().

Su definición se realizará a continuación de la definición de la función main().

En el caso de haber una función que fuese submódulo de otra, seguiremos el mismo razonamiento que para la función main():

Declaramos la función submódulo dentro de la función en la que la vamos a utilizar.

Definimos la función submódulo a continuación de la función en la que la vamos a utilizar.

c) Llamada a una función:
Para poder utilizar una función hay que invocarla. Para ello hay que poner el nombre de la función seguido de los argumentos actuales (si los tiene) que se le van a pasar a la función. Dichos argumentos habrá que escribirlos entre paréntesis y separados por comas en caso de ser varios.
Si el TipoDatoRetorno es void, la llamada a la función será una instrucción más, ya que se trata de un procedimiento:

Ej.: visualizar();

En caso contrario la llamada de la función formará parte de la expresión situada en la parte derecha de una instrucción de asignación:

Ej: total = division (8,4) + 34 - 45;

siendo división una función que calcula la división de dos números.

Ejemplos:

1. Dados el nº de lados de un polígono regular y su longitud (regular => la misma para todos los lados), determinar su perímetro y su área.

```
#include <stdio.h>
#include <math.h>
float PER(int X, float Y);
float ARE(float X, float Y);
void main()
{
  int N;
  float lado,apo,pi=3.1415;
  printf("Introduce el numero de lados y su longitud: ");
  scanf("%d %f",&N,&lado);
  apo=lado/(2*tan(2*pi/(2*N)));
  printf("Perimetro=%10f\n", PER(N,lado));
  //printf("apotema=%10f\n", apo);
  printf("Area=%10f",ARE(PER(N,lado),apo));
}
```

```
float PER(int X, float Y)
{
    return X*Y;
}
```

```
float ARE(float X, float Y)
{
    return (X*Y/2);
}
```

2.- Suma de los N primeros términos de la sucesión de Fibonacci:

```
#include <stdio.h>
int FIBBO(int N);
void main()
{
    int N;
    printf ("Introduce N:  ");
    scanf ("%d",&N);
    printf ("La suma de los %d primeros terminos es: %d",N,FIBBO(N));
}
```

```
int FIBBO(int N)
{
  if (N>3)
    return (2*FIBBO(N-1)-FIBBO(N-3));
  else
     switch (N)
        {
         case 1: return (1);
                 break;
         case 2: return (2);
                 break;
         case 3: return (4);
                 break;
        }

}
```

2.- Cálculo del MCD y el MCM de dos números

```
#include <stdio.h>
int MCD(int A, int B);
void main()
{
  int x,y;
  printf ("Introduce x e y:  ");
  scanf ("%d %d",&x,&y);
  printf ("EL MCD es: %d",MCD(x,y));
  printf ("EL MCM es: %d",x*y/MCD(x,y));
}
```

```
int MCD(int A, int B)
{
  if (A%B)
    return MCD(B,A%B);
  else
    return B;
}
```

Clases de almacenamiento de las variables

La clase de almacenamiento de una variable viene determinada por dos características que son, su ámbito o alcance y su tiempo de almacenamiento. El *ámbito* es la parte del código del programa donde la variable es conocida y por tanto es accesible. Por otro lado el *tiempo de almacenamiento* es el tiempo de vida o duración en memoria de la variable durante la ejecución del programa.

El ámbito de una variable queda determinado por el lugar en el que se defina, mientras que el tiempo de almacenamiento queda determinado por el especificador de clase de almacenamiento que le preceda.

Clasificación de las variables en función del lugar donde están definidas

Teniendo en cuenta el lugar del programa donde se definen las variables, y volviendo a repetir los contenidos explicados en el apartado 8, las podemos clasificar en:

a) Variables globales:

Una variable global está definida fuera de las funciones del programa, y normalmente al comienzo del mismo.

Su ámbito es desde su lugar de definición hasta el final del fichero en el que está definida o hasta el final de los ficheros donde está declarada, en el caso de programas con módulos externos.

Su tiempo de almacenamiento está limitado al tiempo que dura la ejecución del programa, por tanto se crea cuando se define y se destruye al terminar la ejecución del mismo.

No es aconsejable el uso de variables globales ya que se pierde eficiencia en la ejecución de los programas y se pierde portabilidad del código.

b)Variables locales:

Una variable local está definida al comienzo del bloque de instrucciones (cuerpo) de una función.

Su ámbito es el conjunto de sentencias donde está definida

Su tiempo de almacenamiento está limitado al tiempo que dura la ejecución del bloque de instrucciones donde está definida, por tanto la variable local se crea al entrar en ejecución el bloque y se destruye cuando finaliza el mismo.

Un tipo especial de variables locales son los parámetros formales. Los parámetros formales tienen por tanto el mismo ámbito y tiempo de almacenamiento que una variable local, pero se diferencian de éstas en que no están definidos al principio del cuerpo de una función, sino en el encabezamiento de la misma.

Las variables globales, y módulos principales que hemos utilizado en pseudocódigo, podemos definirlos como locales de la función main(), y por tanto declararlos dentro de su definición, justo antes del cuerpo de dicha función. Estos elementos siguen siendo globales para los módulos usados dentro de la función main().

Ejemplo:

declaración →

```
int x,y;     /* variables globales al programa*/
int CalcularSuma(int A, int B);
void main(void)
   {
        .................
        /* CUERPO DE LA FUNCION main*/
        .................
   }
```

o de otro modo

```
int CalcularSuma(int A, int B);
void main(void)
{
    int x,y;      /* variables locales a main y globales para sus módulos*/
    .................
    /* CUERPO DE LA FUNCION main*/
    .................
}
```

definición

```
int CalcularSuma(int A, int B)
   {
       return (A+B);
   }
```

Ejemplo:

```
/* Declaración de variables */

#include <stdio.h>
int a;
main() /* Muestra dos valores */
{
  int b=4;
  printf ("b es local y vale %d",b);
  a=5;
  printf ("\na es global y vale %d",a);
}
```

Clasificación de las variables en función del especificador de clase de almacenamiento

En C se puede indicar al compilador como se quieren almacenar las variables que se van a utilizar en el programa. Para ello se utilizará en la declaración de la variable uno de los tres tipos de especificadores de clase de almacenamiento, de la forma siguiente:

especificador tipo nombre_variable;

Siendo especificador uno de los cuatro valores siguientes:

i) extern

Puesto que en C puede suceder que el programa esté divido en varios ficheros que se pueden compilar por separado y después enlazarlos conjuntamente, y como además las variables globales sólo pueden estar definidas una vez en el programa, aunque sean usadas por varias funciones, debe existir alguna forma de comunicar a todos los módulos (ficheros y funciones) cuáles son las variables globales del programa, y poder así utilizarlas en ficheros distintos al que están definidas. Para ello se declaran en un módulo (Archivo1.c) las variables globales y en aquellos módulos en los que se van a utilizar dichas variables se declaran como extern:

```
// Archivo1.c
        int x, y;
        main()
          {
          ....
          }
// Archivo2.c
        extern int x, y;
        Funcion1
          {
        .......
          }
```

Cuando se compile el Archivo2.c, el compilador no dará error (variable global x duplicada), pues con el especificador extern le indicamos que está definida en otro fichero que forma parte del programa. En nuestro ejemplo en el módulo principal (pues contiene a la función main).

ii) *static*

- Se utiliza para hacer que una variable local no se destruya cuando finalice la ejecución del bloque en el que ha sido declarada, (aunque ésta no sea accesible fuera del bloque, conserva su valor si vuelve a ser usada por el mismo). Al ser local sólo la podrían utilizar las sentencias de su bloque.
- En el caso de variables globales, éste especificador hace que sólo sea reconocida en el fichero en el que se ha declarado, por lo que **no** tiene sentido utilizar con ella en otro módulo, el especificador extern para usarla.
- Y por último en el caso de una función, limita su ámbito al fichero donde está definida, haciéndola invisible para los demás elementos (módulos del programa).

iii) *register*

Indica al compilador que almacene la variable en un registro de la CPU (si es posible), para que el acceso a la misma sea más rápido. Se suele aplicar a las variables locales o parámetros formales. En caso de no ser posible, la variable se almacenará en memoria principal ,como el resto de variables.

Las variables globales que pueden ser usadas en otros módulos mediante extern y las variables locales que usen el atributo static, por defecto se inicializan a 0. Mientras que las variables locales no se inicializan por defecto.

Ejemplo:

Creamos un primer archivo principal llamado *variables.c* en nuestro proyecto *variables.ide* con este contenido

```
#include <stdio.h>
#include <conio.h>
int suma;                                   // variable global
void sumadosvecescinco(void);

void main()
   {
        sumadosvecescinco();
        presentasuma();
        sumadosvecescinco();
        presentasuma();
        getch();
   }

void sumadosvecescinco(void)
{
  static int contador;              // variable local estática, se inicializa a 0
  int i;                            // variable local
  for (i=0;i<2;i++)
    {
     int i=5;                       // variable local que esconde el valor de i
     suma += i;                     // el for se repite 2 veces, la i no se pierde
    }
  contador++;
}
```

Por otro lado creamos el archivo (nodo) *presenta.c* en nuestro proyecto *variables.ide:*

```
void presentasuma(void)
{
  extern int suma;                          //variable externa
  printf ("Ahora la suma vale: %d \n", suma);
}
```

La variable **suma** es global, se inicializa a 0 y la usamos dentro del segundo archivo como variable externa (extern). La variable **contador** es local y estática (static), también se inicializa a 0 y aunque no es accesible fuera de la función sumadosvecescinco, su valor no se destruye, sino que cuando se vuelve a usar en dicha función conserva el valor de salida de la vez anterior. La variable **i** del bucle es una variable local normal (no se inicializa automáticamente). La variable **i** declarada en el interior del bucle es local(no se inicializa), no hace que se pierda el valor de la i anterior, no se trata de las mismas variables aunque posean el mismo identificador. La i interior se crea y se destruye en el bloque y no afecta a la i del bucle. Otra cosa hubiese sido, si en lugar de int i=5; apareciese una expresión con i, esto si la afectaría. Por ejemplo i=5; i++; i=3+4; etc.

Cuadro resumen de los tipos de variables:

TIPO	ESPECIFICADOR DE ALMACENAMIENTO	LUGAR EN EL QUE SE DECLARA	ÁMBITO	VIDA
Global	Ninguno	Antes de todas las funciones	Todo el programa	Hasta que termine el programa.
Global	**Extern**	Antes de cualquier función de un fichero	El fichero fuente en la que está definida como variable global (sin el especificador extern), así como todos los ficheros fuentes dónde éste declarada con el especificador extern	Mientras que cualquiera de los archivos dónde esté declarada esté activo.
Global	**Static**	Antes de cualquier función	Sólo en las funciones definidas en el fichero en el que se declara la variable.	Mientras que el fichero se esté ejecutando
Local	Ninguno	Al principio de un bloque, siendo el bloque más normal en el que se declara el cuerpo de una función	El conjunto de instrucciones que componen el bloque	Mientras se estén ejecutando las instrucciones de su bloque.
Local	**Static**	Al principio de un bloque, siendo el bloque más normal en el que se declara el cuerpo de una función	Fuera de su ámbito o función	Mientras se esté ejecutando el programa.
Local	**Register**	Igual que una local normal	Igual que una local normal	Igual que una local normal

Como ya hemos indicado alguna vez, no es conveniente el uso de variables globales y por tanto el uso de *extern*, debido a la pérdida de eficiencia (las variables globales permanecen en memoria durante la ejecución de todo el programa) y la pérdida de portabilidad (los módulos no son directamente reutilizables debido a las referencias cruzadas de las variables).

11. PASO DE PARÁMETROS POR VALOR Y POR REFERENCIA

Por defecto los parámetros formales de una función reciben a los parámetros actuales **por valor** (los argumentos con los que se inicializan dichos parámetros no cambian de valor tras finalizar la función), pero también se pueden pasar **por referencia** (los argumentos cambian de valor al finalizar la ejecución de la función si durante la ejecución de la misma se ha hecho alguna modificación de los parámetros formales con los que se inicializan dichos argumentos).
Para poder utilizar parámetros por referencia hay que utilizar punteros.
En la *declaración* de la función:

```
TipoDatoRetorno NombreFuncion (TipoParametro1 *[Parámetro]);
```

En la *definición* de la función:

```
TipoDatoRetorno NombreFuncion (TipoParametro1 *Parametro);
        /* cabecera de la definición de una función
        con un parámetro por variable o referencia*/
{
        /* Dentro del cuerpo de la función hay que
        utilizar el parámetro formal como un puntero */
}
```

En la *llamada* a la función, se pasará como parámetro real la dirección en memoria de éste y no su contenido:

```
NombreFuncion (&Parámetro)
```

Ejemplo :

```
.....
A= 3;
B= 8;
Ejemplo(A,&B);  /* &B indica que se pasa la dirección en memoria, no el dato*/
.....
void Ejemplo(int N1, int *N2)
        {
            N1=5;
            *N2=7
            printf("%d", N1);
            printf("%d", *N2);
        }
```

Tras la ejecución de Ejemplo(A,&B), el primer parámetro real pasado conservará su valor anterior A=3, en cambio el segundo no, valdrá B=7.

12. OTRAS FUNCIONES PREDEFINIDAS EN C

Las funciones que podemos utilizar en C las podemos clasificar en tres tipos:

a) La función main()
Es la función más importante de un programa en C. Siempre tiene que existir una única función main(). Esta función es a la que llama el sistema operativo cuando ejecutamos el programa. Como toda función puede tener parámetros que se inicializarán con los valores que escribamos desde el sistema operativo a continuación del nombre de la función cuando ejecutemos el programa. Análogamente puede devolver valores de retorno que serán interpretados por el sistema operativo.

b) Funciones de usuario.
Son las funciones que el propio programador define dentro del programa.

c) Funciones de librería
Son un gran conjunto de funciones predefindas por el lenguaje C, entre las que destacan por ejemplo, las funciones comunes de entrada/salida (scanf (..&variable), printf(" ") , etc.)
Los cuerpos o *definiciones* de las funciones de librería están contenidos en unos ficheros llamados librerías y que tienen extensión “lib”. Normalmente una librería contiene la definición de varias funciones relacionadas entre sí.
Para poder utilizar las funciones de una determinada librería en nuestro programa debemos añadir en la cabecera del mismo una directiva del preprocesador de tipo include del siguiente modo:

```
#include <Nombre_Archivo_Cabecera.h>
                    ó
#include "\ruta\Nombre_Archivo_Cabecera.h"
```

Siendo:
Nombre_Archivo_Cabecera, el nombre de un archivo cabecera, que son los ficheros que contienen la *declaración* o *prototipo* de las funciones de librería que queremos utilizar en nuestro programa.
La primera forma se utiliza cuando el archivo especificado está en el directorio en el que C almacena por defecto todos los ficheros de este tipo. La segunda se utitiza en caso contrario, teniendo que indicar a C la ruta completa en la que está dicho fichero.
Los programadores se pueden crear sus propias librerías y ficheros cabecera.

Ejemplo:

Creamos el siguiente programa principal:

```
/* programa modular.c */
        #include <stdio.h>
        #include "lib.h"
        void main ()
        {
         printf ("%d ",suma (3,4));
         printf ("%d ",producto (3,4));
        }
```

como vemos hemos usado dos funciones sin declararlas ni definirlas en el programa: suma y producto. Al igual que ocurre con la librería *stdio.h* que nos permite el uso de printf, hemos creado nuestra propia libería *lib.h* en la que hemos definido las funciones suma y producto. Cargamos nuestra librería al inicio del programa y podemos hacer uso de ella en los programas que necesitemos:

```
/* librería lib.h */
    int suma (int a,int b)
     {
            return a+b ;
     }

    int producto (int c,int d)
     {
            return c*d ;
     }
```

Podemos usar nuestra librería en cualquier proyecto, al igual que hacemos con las librerías estándar: stdio.h, math.h , conio.h, ctype.h, etc.

Ejemplos:

1. Queremos sumar o restar dos números reales introducidos por teclado, en función de la respuesta S o R (Suma o Resta):

```
#include <stdio.h>  //librería de E S
#include <conio.h> //uso de getch, clrscr
#include <ctype.h> //uso de tolower y toupper
void main()
  {
      float a,b;
      char resp;
      clrscr();
      do{
            printf ("\n\n\t Introduce dos números   ");
            scanf ("%f %f",&a,&b);
            printf ("\n\t Introduce s para sumar o r para restar   ");
            resp=getch();
                  if (tolower(resp)=='s')
                     printf ("\n\n\t El resultado es %f",a+b);
                  else
                    {
                     printf ("\n\t El resultado es %f",a-b);
                           }
            printf ("\n\n\t ¿Desea ejecutar otra vez?   ");
            resp=getch();
          }
    while (tolower(resp)=='s');
    getch();
    clrscr();
 }
```

2. Calcular la media de 100 números enteros introducidos por teclado y visualizarlo:

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
void main(){
      char resp;
      int cont,suma,num;
      float media;
      clrscr();
      do{
            cont=1;
            suma=0;
            while (cont<=100)
               {
                  printf ("\n\n\t Introduce un numero:   ");
                  scanf ("%d",&num);
                  suma=suma+num;
```

```
            cont=cont+1;
          }
        media=suma/100;
        printf ("\n\n\t La media es %f  ",media);
        printf ("\n\n\t ¿Deseas ejecutarlo otra vez? <s/n>  ");
        resp=getch();
        }
    while (tolower(resp)=='s');
    getch();
    clrscr();
}
```

3. Visualizar los múltiplos de 4 comprendidos entre 4 y N, dónde N es un número introducido por teclado:

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
void main()
 {
   char resp;
   int n;
   do{
       do{
          printf ("\n\n\t Introduce un número:   ");
          scanf ("%d",&n);
         }
       while (n<4);
       do{
          if (n%4==0)
            { printf ("\t %d",n);}
          n=n-1;
         }
       while (n>=4);
       printf ("\n\n\t ¿ejecutar de nuevo otra vez? <s/n>   ");
       resp=getch();
     }
   while (tolower(resp)=='s');
   getch();
   clrscr();
 }
```

4. La recompensa de los tres marineros

Un capitán de barco recompensa a tres marineros con más de 200 monedas y menos de 300. Durante la noche, uno de los marineros, se levanta sigiloso y .....;
"....hago 3 montones iguales, me quedo con uno, y esta moneda que sobra la tiro al mar"
Más tarde se levanta otro marinero y actuó igual:
"....hago 3 montones iguales, me quedo con uno, y esta moneda que sobra la tiro al mar"
Al poco rato, se levanta el tercer marinero e hizo igual que sus compañeros:
"....hago 3 montones iguales, me quedo con uno, y esta moneda que sobra la tiro al mar"
A la mañana siguiente, el contable repartió lo que quedaba en tres montones iguales y se quedó con una moneda que sobraba en pago a su trabajo.
¿Cuántas monedas había, y cuántas recibió cada marinero?

```
#include <stdio.h>
#include <conio.h>
void main()
{
  int MON,REP;
  for (MON=200;MON<301;MON++)
    {
     if((MON-1)%3==0)
       {
         REP=(MON-1)*2/3;
        if (((REP-1)%3)==0)
               {
                  REP=(REP-1)*2/3;
                  if (((REP-1)%3)==0)
                         {
                          REP=(REP-1)*2/3;
                          if (((REP-1)%3)==0)
                                 {
                                   REP=(REP-1)/3;
                                   printf ("Habia %d monedas\n",MON);
                                   getch();
                                   clrscr();
                                   printf ("Cada marinero toca a %d monedas",REP);
                                 }
                         }
                  }
           }
    }
}
```

5. “La herencia del rajá” (problema indú del siglo IV)
Un rajá dejó en herencia cierto número de perlas a sus hijas, de este modo:
“...La primera recibirá 1 perla más 1/7 de las trestantes,
.....la segunda recibirá 2 perlas más 1/7 de las restantes,
.....la tercera recibirá 3 perlas más 1/7 de las restantes,
.....y así con todas las hijas.
Al finalizar el reparto, se comprobó con sorpresa que todas las hijas recibieron el mismo número de perlas. ¿Cuántas perlas dejó en herencia? ¿Cuántas hijas tenía el rajá?.

NOTA: la última hija ha de tomar las perlas que deja la penúltima N,de tal forma que 1/7 del resto (o sea 1/7 de 0) sea 0, de otro modo nunca se acabaría el número de hijas. Por tanto sabemos que la última hija la n-ésima recibe N perlas, pero como todas reciben las mismas perlas, deducimos que N hijas por N perlas cada una dan un total de N*N perlas o lo que es lo mismo HIJAS*HIJAS perlas. Por tanto hay que buscar un nº de hijas N que se puedad repartir según las condiones N*N perlas.

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

```
void main()
{
  int J,REPARTO,PERLAS,HIJAS;
  for (HIJAS=1;HIJAS<1000;HIJAS++)
     {
      PERLAS=HIJAS*HIJAS;
      J=1;
      while (PERLAS>0)
         {
          if ((PERLAS-J)%7==0)
            {
             REPARTO=J+(PERLAS-J)/7 ;
             PERLAS=PERLAS-REPARTO;
             if (PERLAS==0)
               {
                printf ("El numero de hijas es %d\n",HIJAS);
                 printf ("Habia %d perlas y tocan a %d\n",HIJAS*REPARTO,REPARTO);
               }
             else if (PERLAS>0)
                J=J+1;
            }
          else
             break;
        }
    }
}
```

6. Elaborar un algoritmo llamado JUGADORES que resuelva el siguiente problema:
Tres jugadores convienen que el que pierda una partida, doblará el dinero que en ese momento tenga cada uno de los otros dos. Después de haber perdido una única partida cada uno de ellos, cada jugador termina con 200 pesetas. ¿Cuánto dinero tenían al principio del juego?
NOTA: En juego siempre hay la misma cantidad 600 ptas.. Cada jugador pede tener en cada partida desde 1 a 598 ptas.. El que pierde siempre tiene que poseer más que los otros, de otro modo se quedaría sin dinero y nunca podría acabar con más de 0 pesetas.

```
#include <stdio.h>
void main()
{
  int i,j,k,j1, j2,j3;
  for(i=1;i<599;i++)
     for(j=1;j<i+1;j++)
        for(k=1;k<i+1;k++)
          {
           j1=i-j-k;
           j2=2*j;
           j3=2*k;
           j2=j2-j1-j3;
           j1=2*j1;
           j3=2*j3;
```

```
            j3=j3-j1-j2;
            j1=2*j1;
            j2=2*j2;
            if ((j1==200)&&(j2==200)&&(j3==200))
                  printf (" Tenian: %-4d %-4d %-4d",i,j,k);
            }
            getch();
}
```

7. **Introducido un nº entero positivo por teclado, se quiere saber el nº de dígitos que posee. Si el nº de dígitos es impar, se desea saber cuál es el dígito central y en caso de ser par, la suma de dichos dígitos:**

```
#include <stdio.h>
#include <math.h>
void main()
{
  unsigned long int N;
  int DIG=1,SUMA,I;
  printf ("Introduce un numero positivo: ");
  scanf ("%ld",&N);
  while ((N/((unsigned long int)pow(10,DIG)))!=0)
    { DIG=DIG+1;}
  printf ("El numero de digitos es %d\n",DIG);
  if (DIG%2==0)
       { SUMA=0;
         for (I=0;I<DIG;I++)
          {
           SUMA=SUMA+(N%((int)pow(10,I+1)))/pow(10,I);
          }
         printf ("La suma es %d\n",SUMA);
       }
  else
         printf ("El central es %d\n",(N/((int)pow(10,DIG/2)))%10);
}
```

Uso de algunas funciones

DELAY, SOUND, NOSOUND

```
void delay (unsigned tiempo)
void sound(unsined frecuencia)
```

El prototipo de delay() se encuentra en *dos.h*, esta función no está definida en el estándar ANSI de C. La función delay() detiene la ejecución del programa durante un tiempo especificado en milisegundos.

Ejemplo1:

```
/* Este programa muestra un mensaje y emite un pitido dos veces. */
#include <stdio.h>
#include <dos.h>
main (void)
{
        printf(" beep beep \n");
        sound (500);
        delay (600);
        nosound();
        delay (300);
        sound(500);
        delay(600);
        nosound();
}
```

En este ejemplo hemos usado dos funciones nuevas sound() y nosound(). La función sound se llama con un argumento de tipo entero (por ejemplo sound(100)) que se convertirá en la frecuencia del sonido a emitir. Al activar sound, el sonido continúa hasta que se ejecute nosound().

Ejemplo2:

```
/* sonido sirena*/
#include <stdio.h>
#include <dos.h>
void siren(int frec);
int S=1;
void main ()
   {
        siren(3000);
   }

void siren(int frec)
    {

         while ((frec>=1000)&&(S==1))
            {
                sound(frec);
                delay(80);
                nosound();
                if (frec<=1000)
                S=0;
                siren(frec-100);
             }
           while ((frec<=3000)&&(S==0))
            {
                sound(frec);
                delay(80);
                nosound();
                if (frec>=3000)
                S=1;
                siren(frec+100);
```

```
        }
    nosound();
}
```

GOTOXY

void gotoxy (int x, int y)

El prototipo de gotoxy() se encuentra en *conio.h*.
La función gotoxy() sitúa el cursor de la pantalla de texto en la posición especificada por x,y. Si alguna, o ambas, coordenadas no son válidas, no ocurre nada.

Ejemplo1:

```
/* Este programa escribe X diagonalmente a lo largo de la pantalla. */
 #include <stdio.h>
#include <conio.h>
main (void)
{
register int i, j;
clrscr();
/* escribir la diagonal de X*/
for (i=1,j=1;j<24;i=i+2,j++)
        {
                gotoxy(i,j);
                cprintf("X");
        }
getch();
clrscr();
}
```

TEXTCOLOR Y TEXTBACKGROUND

void textcolor (int color)
void textbackground(int color)

El prototipo de textcolor() y textbackground se encuentran en *conio.h*.
La función textcolor() establece el color con el que se muestran los caracteres en una pantalla de texto. Los valores válidos para color, se muestran a continuación, junto con sus macros (definidas en conio.h):

Macro	**Equivalente entero**
NEGRO	0
AZUL	1
VERDE	2
CYAN	3
ROJO	4

MAGENTA	5
MARRON	6
GRIS CLARO	7
GRIS OSCURO	8
AZUL CLARO	9
VERDE CLARO	10
CYAN CLARO	11
ROJO CLARO	12
MAGENTA CLARO	13
AMARILLO	14
BLANCO	15
PARPADEAR	128

Esta función no cambia el color de los caracteres que ya estén en la pantalla, solo afecta a los que se escriban después de la ejecución de textcolor().
La función textbackground() sirve para poner un color de fondo de pantalla correspondiente al texto para el que se activa.

Ejemplo1:

```
#include <stdio.h>
#include <conio.h>
void main()
{
  long fg,n,i=5;
  cprintf("Introduce un numero entero:\n");
  scanf ("%d",&n);
  for (fg=0;fg<8;fg++)
  {
        textcolor(fg);
        textbackground(fg+2);
        gotoxy(10,i++);
        cprintf("%10d\n",n);
   }
}
```

Ejemplo2:

```
/*Programa para usar funciones gráficas*/
#include <stdio.h>
#include <conio.h>
#include <math.h>
void main()
{
        int grad,radio=10;
        double radianes;
  //for (radio=2;radio<12;radio=radio+2)
  //      {
        for (grad=0;grad<360;grad++)
```

```
                        {
                                radianes=grad*6.28/360;
                                textcolor(grad+12);
                                gotoxy(40+radio*sin(radianes),12+radio*cos(radianes));
                                cprintf("*");
                        }
   //}
        getch();
}
```

RANDOM, RANDOMIZE

```
int random (int num)
void randomize(void)
```

Los prototipos de random() y randomize() se encuentran en *stdlib.h*. Estas funciones no están definidas en el estándar ANSI de C.

La función random() devuelve un número aleatorio que se encuentra en el rango de 0 a num-1.

La macro randomize() inicializa el generador de números aleatorios a un valor aleatorio. Usa la función time(), por lo que es necesario incluir *time.h* en cualquier programa que use randomize().

Ejemplo1:

```
/*Este programa escribe diez números aleatorios entre 0 y 24:*/

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

void main(void)
{
        int i;
        randomize();
        for(i=0; i<10; i++)
                printf("%-5d",random(25));
}
```

Ejemplo2:

```
/*Programa para generar combinaciones de bonoloto*/
#include <stdio.h>
#include <conio.h>
#include <time.h>
#include <stdlib.h>
void main()
{
        int N, i, j,nuevo, k=5;
        printf ("Introduce el nº de combinaciones: ");
  scanf ("%d",&N);
```

```
  randomize();
  textcolor(2);
  gotoxy(10,k++);
  for (i=1;i<=N;i++)
       {
        for (j=1;j<7;j++)
          {
            nuevo=random(50);
            if (nuevo)
               cprintf ("%5d",nuevo);
            else
                    j--;
          }
        //getch();
        gotoxy(10,k++);
       }
          getch();
}
```

Este programa hay que modificarlo porque se repetirían los números en cada combinación. (usando arrays, no vistos, o declarando 6 variables para controlar su repetición).