INGENIERIA DE SISTEMAS Y AUTOMATICA

PRÁCTICAS DE CONTROL POR COMPUTADOR.

MANUAL DE PROGRAMACIÓN EN C.

INTRODUCCION AL LENGUAJE DE PROGRAMACION C

Indice:

.1 - Introducción al C.

.2 - Tipos de datos.

.3 - Expresiones y operadores.

.4 - Control del flujo del programa.

.5 - Tipos de datos avanzados.

.6 - Funciones.

.1 - INTRODUCCION AL C.

C es un lenguaje de programación estructurado de propósito general . Sus instrucciones constan de términos que se parecen a expresiones algebraicas , además de ciertas palabras clave inglesas como **if** , **else** , **for** , **do** y **while** . En este sentido , C recuerda a otros lenguajes de programación estructurados de alto nivel . C tiene también algunas características adicionales que permiten su uso a un nivel más bajo , cubriendo así el vacío entre el lenguaje máquina y los lenguajes de alto nivel más convencionales . Esta flexibilidad permite el uso de C en la programación de sistemas , así como en la programación de aplicaciones.

C se caracteriza por hacer posible la redacción de programas fuente muy concisos , debido en parte al gran número de operadores que incluye el lenguaje . Tiene un conjunto de instrucciones relativamente pequeño , aunque las implementaciones actuales incluyen numerosas funciones de biblioteca que mejoras las instrucciones básicas. Es más , el lenguaje permite a los usuarios escribir funciones de biblioteca adicionales para su

uso propio. De esta forma , las características y capacidades del lenguaje se pueden ampliar fácilmente por el usuario.

Hay compiladores de C disponibles para computadoras de todos los tamaños y los intérpretes de C se están haciendo cada vez más comunes . Los compiladores son frecuentemente compactos , y generan programas objeto que son pequeños y muy eficientes en comparación con los programas generados a partir de otros lenguajes de alto nivel . Los intérpretes son menos eficientes , aunque son de uso más cómodo en el desarrollo de nuevos programas . Muchos programadores comienzan utilizando un intérprete , y una vez que han depurado el programa utilizan el compilador.

Otras característica importante de C es que los programas son muy portables , más que los escritos en otros lenguajes de alto nivel . La razón de esto es que C deja en manos de las funciones de biblioteca la mayoría de las características dependientes de la computadora . Toda versión de C se acompaña de su propio conjunto de funciones de biblioteca , que están escritas para las características particulares de la computadora en la que se instale . Estas funciones de biblioteca están relativamente normalizadas y se accede a cada función de biblioteca de igual forma en todas las versiones de C . De esta forma , la mayoría de los programas en C se pueden compilar y ejecutar en muchas computadoras diferentes con muy pocas o ninguna modificación .

Historia del C .

C fue desarrollado originalmente en los años setenta por Dennis Ritchie en Bell Telephone Laboraoties , Inc. (ahora AT&T Bell Laboratories) . Es el resultado de dos lenguajes anteriores , el BCPL y el B , que se desarrollaron también en los laboratorios Bell . C estuvo confinado al uso en los laboratorios Bell hasta 1978 , cuando Brian Kernighan y Ritchie publicaron una descripción definitiva del lenguaje. La definición de Kernighan y Ritchie se denomina frecuentemente “K&R C”.

Tras la publicación de la definición K&R , los profesionales de las computadoras , impresionados por la muchas características deseables del C , comenzaron a promover el uso del lenguaje . Por la mitad de los ochenta , la popularidad del C se había extendido

por todas partes . Se habían escrito numerosos compiladores e intérpretes de C para computadoras de todos los tamaños y se habían desarrollado muchas aplicaciones comerciales . Es más , muchas aplicaciones que se habían escrito originalmente en otros lenguajes se reescribieron en C para tomar partido de su eficiencia y porbabilidad.

La mayoría de las implementaciones comerciales de C difieren en algo de la definición original de Kernighan y Ritchie . Esto ha creado alguna pequeñas incompatibilidades entre las diferentes implementaciones del lenguaje , disminuyendo la portabilidad que éste intentaba proporcionar . Consecuentemente , el Instituto Nacional Americano de Estándares (ANSI) comenzó a trabajar en una definición normalizada del lenguaje C (comite ANSI X3J11) .

Estructura de un programa en C .

Todo programa en C costa de una o más funciones , una de las cuales se llama main. El programa siempre comenzará por la ejecución de la función main . Las definiciones de las funciones adicionales pueden preceder o seguir a main . Cada función debe contener al menos de :

- Una cabecera de la función , que consta del nombre de la función ,seguido de una lista opcional de argumentos encerrados con paréntesis.

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

- Una sentencia compuesta , que contiene el resto de la función.

Los argumentos son símbolos que representan información que se le pasa a la función desde otra parte del programa.

Cada sentencia compuesta se encierra con un par de llaves , { y }. Las llaves pueden contener combinaciones de sentencias elementales (llamadas sentencias de expresión) y otras sentencias compuestas . Así las sentencias compuestas pueden estar animadas , una dentro de otra . Cada sentencia de expresión debe acabar en punto y coma (;).

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

Estos componentes del programa se discutirán en los siguientes apartados..

Uso del C en el S.C.S.M.P.I.

Mediante el editor gráfico S.C.S.M.P.I. (Software de Control , Supervisión y Monitorización de Procesos Industriales) podemos obtener distintas pantallas gráficas sin preocuparnos del código necesario para la realización de ello, pero para la realización completa de sistemas de control es necesario completar el programa que nos proporciona el editor gráfico mediante la parte de control o tratamiento de datos para lo cual se hace por medio del lenguaje de programación C.

Para poder obtener un total aprovechamiento del software S.C.S.M.P.I., aunque no se es necesario ser un experimentado programador en C , (ya que el programa tiene como misión el posibilitar la programación gráfica a aquellas personas que no son experimentados programadores , o simplemente facilitar la programación a todo tipo de usuarios) , es necesario tener unos ciertos conocimientos tanto acerca de la metodología de programación como su implementación en el lenguaje de programación C .

A continuación se tratan los aspectos más importantes y de mayor interés del lenguaje de programación C , para que sirva de guía para aquellos usuarios del software . Entre los aspectos tratados no se encuentra el uso de las funciones estándar .

Para una mayor profundización en el lenguaje de programación C , o acerca de las funciones de las librerías clásicas se recomienda consultar bibliografía especifica sobre el lenguaje C , cuyos títulos podemos encontrar en la bibliografía utilizada para la realización del proyecto.

.2 - TIPOS DE DATOS.

Los tipos de datos disponibles en C son :

- Número entero : Identificado como **int** , que tiene distintas variantes dependiendo de su longitud y del signo.

  - Dependiendo de su longitud :
  corto : **short** ó **short int**
  normal : **int**
  largo : **long** ó **long int**

  - Dependiendo del signo puede ser :
  sin signo : **unsigned**
  con signo.

Los signos pueden combinarse con cualquiera de los tipos expuestos anteriormente . Se dopría escribir **unsigned short** , **unsigned int** ... Si no se indica nada respecto al signo , se entiende que tiene signo.

- Número real : Todos números reales ó en coma flotante tienen signo . Dependiendo de la longitud se identificarán como :

  float
  double
  long double

- Caracteres : Los caracteres son todos del mismo tamaño , así que se pueden agrupar en función del signo .

Normal : Se identifican como **char** .

Con signo : Se identifican como **signed char** .

Sin signo : Se identifican como **unsigned char** .

A continuación se muestra una tabla donde se indica cada tipo de datos , con su precisión (número de dígitos que ocupan los datos) , y su rango.

Tipo de dato	Precisión	Mínimo	Máximo
unsigned char	8 bits	0	255
char	8 bits	-128	127
enum	16 bits	-32768	32767
int	16 bits	-32768	32767
unsigned int	16 bits	0	65535
short int	16 bits	-32768	32767
unsigned long	32 bits	0	4294967295
long	32 bits	-2147483648	2147483647
float	32 bits	3.4*10exp(-38)	3.4*10exp(-38)
double	64 bits	1.7*10exp(-308)	1.7*10exp(-308)
long double	80 bits	3.4*10exp(-4932)	3.4*10exp(-4932)

Variables y constantes .

Un punto a tener en cuenta es la diferencia entre las expresiones constantes y las variables :

- Una expresión constante es un valor que no puede variar a lo largo del programa.

```
#define VELOCIDAD_LUZ 300000
```

- Una variable es un espacio de memoria identificado por un nombre , donde se van a almacenar valores de un tipo determinado y que pueden variar a lo largo del programa.

```
int alto ;
alto = 5;
alto = 64;
```

Constantes simbólicas.

Una constante simbólica es un nombre que sustituye a una secuencia de caracteres (donde estos caracteres pueden ser una constante numérica , de carácter , de cadena...) . Las constantes simbólicas se definen al comienzo del programa.

Sintaxis:

```
#define nombre valor
```

, donde nombre es un nombre simbólico y valor una cadena de caracteres . No acaba en punto y coma (;) ya que no es una secuencia . Es costumbre escribir las constantes en mayúsculas .

```
#define INTERES 0.25
#define CIERTO 1
#define FALSO 0
```

Declaración de variables.

Para declarar una variable de un tipo determinado , se debe indicar en primer lugar el tipo de dato del que se desea crear la variable , seguido del nombre que se quiere dar a la variable , y con el que luego nos referiremos a ella .

Sintaxis :

```
modo_almacenamiento tipo_almacenamiento nombre_variable ;
```

Por ejemplo para definir una variable de tipo float :

```
float variable ; /* Definición de una variable */
```

Notar que todas sentencias de C terminan con el carácter punto y coma (**;**) . También hay que indicar que cuando queramos introducir un comentario , este debe encerrarse entre " /* " y " */ ".

Los nombres en C deben cumplir las siguientes reglas :

- El nombre debe comenzar por un carácter alfabético.
- El carácter '_' se considera alfabético.
- Las mayúsculas y minúsculas se consideran caracteres distintos.
- Se admiten caracteres numéricos y alfanuméricos .
- Los nombres pueden tener cualquier longitud.
- No pueden usarse como nombres las palabras claves del lenguaje C.

Mediante el operador de asignación " = " , podemos asignar un valor a una variable . Si empleamos este operador en la declaración de la variable , su valor inicial será el indicado por medio del operador.

```
int a;
int b = 5;        /* Inicializa la variable b con el valor 5 */
a = 5;            /* Asigna un valor a la variable a */
```

Ambito de validez .

El campo que hemos denominado modo_de_almacenamiento hace referencia a lo que se va a ver a continuación..

Las variables tienen validez en el ámbito en el que son declaradas . Si una variable es declarada fuera del contexto de cualquier función , su existencia es conocida por cualquier función existente en el fichero en el que han sido declaradas . Su valor puede ser utilizado en cualquier función y éste puede ser alterado por cualquiera de ellas . Estas son las variables globales . La variables locales son las declaradas en el interior de una función , solo tienen validez en la ejecución de la función en la que son declaradas.

En C hay cuatro modos de almacenamiento , estos se anteponen al tipo de declaración de las variables respectivas . Son :

extern , **static** , **auto** y **register**.

El prefijo **extern** , se utiliza en proyectos de programación en los cuales , el código del programa está repartido en varios ficheros. Si en un fichero , declaramos una variable global y queremos que tenga validez en todo el programa , la simple declaración en todos los ficheros en los cuales se haga uso de ella , provocaría un error en compilación , por duplicidad en la declaración . Anteponiendo **extern** a las demás declaraciones , informamos al compilador de que la declaración de la variable está contenida en otro fichero.

Una vez que una función cesa su código y cede el control a la función desde la que fue llamada , todas las variables locales utilizadas son destruidas , a no ser que sean declaradas con la opción **static** . Con esta opción , las variables permanecen entre una y otra llamada y su valor , conservado , puede ser utilizado una y otra vez por la función de llamada.

Si **static** se aplica a una variable global , estamos limitando , el ámbito de validez de ésta , al fichero en la que ha sido declarada . Esto nos permite , en proyectos de gran envergadura , ocultar detalles no necesarios , al resto de los componentes del equipo , y compartimentar e independizar nuestra sección , de la desarrollada por los demás miembros .

Si a la declaración de una variable , se le antepone el prefijo **register** , la variable es almacenada , si ello es posible , en uno de los registro de la CPU . Su utilidad es manifiesta , la velocidad de acceso a su valor es sensiblemente superior a la requerida para el correspondiente acceso a una variable en memoria . Se suele declarar **register** a las variables que controlan bucles .

En el caso de no indicar nada se supone que es de tipo **auto**. Que es la situación normal.

Sintaxis de las expresiones constantes .

Para distinguir entre los diferentes valores constantes , es necesario regirnos por unas series de convenciones :

- Para indicar un entero se escribe el número sin nada más , si el número es sin signo se debe poner el sufijo **u** ó **U** y si es largo añadiremos el sufijo **l** ó **L**:

```
1221341
-4525
15124uL
```

- Cuando un número es real se pone el carácter **F** , cuando tiene decimales no es necesario ponerlo ya que se sobreentiende . Si el número tiene exponente se escribe la letra **e** ó **E**.

```
64946F
2545.12
15E-5
487e4
```

- Un número en hexadecimal va precedido de los caracteres **0x** .

```
Ox2A1D
```

- Un número octal va precedido del carácter **0** .

0138

- Para que un valor se entienda como carácter debe ir entre comillas simple , por ejemplo el carácter ‘1’ equivale al ASCII 49.

Hay caracteres especiales que no pueden escribirse directamente y que requieren una representación especial , la cual podemos ver en las siguiente tabla :

Carácter	Nombre	Sintaxis
‘	Comilla simple	‘\’‘
“	Comilla doble	‘\”’
\	Barra invertida	‘\\’

Otra manera de indicar un carácter concreto es escribir la barra invertida seguida del número del carácter en el juego de caracteres utilizados (en nuestro caso ASCII) . Así por ejemplo ‘\123’ equivale a ‘{‘ . Esto es muy útil para caracteres de control o especiales , aunque existen convenciones especiales para representar determinados caracteres de control , como se ve en la siguiente tabla :

Nombre	Sintaxis
Salto de línea	‘\n’
Tabulador horizontal	‘\t’
Retroceso	‘\b’
Retardo de carro	‘\r’
Salto de página	‘\f’
Alerta (sonido)	‘\a’
Tabulador vertical	‘\v’

.3 - EXPRESIONES Y OPERADORES .

Aquí se pretende estudiar la forma de operar con los datos anteriormente vistos . Las expresiones son operaciones con uno ó más valores que se relacionan mediante operadores.

Operador de asignación.

Se representa con el carácter de igualdad (=).

Sintaxis :

```
variable = expresión ;
```

, donde variable es el nombre de la variable donde almacenamos el valor como resultado de la expresión.

```
valor_salida = 10;
```

Si se realiza una asignación entre distintos tipos de datos , el compilador convierte el valor de la expresión al tipo de dato de la variable (por truncamiento).

Operadores aritméticos .

Los operadores aritméticos son los expuestos en la siguiente tabla :

Operador	Propósito
+	Adición
-	Sustracción
*	Multiplicación
/	División (por truncamiento)
%	Resto de la división entera

Si se desea , se puede convertir el valor resultante de una expresión a un tipo de datos diferentes . Para hacer esto , la expresión debe ir precedida por el nombre del tipo de datos deseado , encerrado entre paréntesis :

(tipo_dato) expresión ;

Supongamos que a es una variable entera , y f es una variable de coma flotante . La expresión (a+f)%4 es inválida , porque el operador (a+f) es de coma flotante en vez de entero. Sin embargo la expresión ((int)(a+f))%4 es una expresión correcta.

Operadores de manipulación de bits.

Estos operadores tratan los datos bit a bit . Podemos agruparlos en 2 grupos : lógicos y de desplazamiento.

Lógicos :

Realizan las operaciones lógicas bit a bit . Los operadores lógicos se muestran en las siguiente tabla :

Operación	Operador
Y	&
O	|
O Exclusivo	^
Negación	~

Todas estos operadores son binarios (entre 2 miembros) , salvo la negación que es un operador unitario que cambia el valor de cada uno de los bit.

Veamos unos ejemplos suponiendo que la variable es de un tipo cuyo tamaño son 4 bits.

10 | 3 → 1010 | 0011 = 1011 ⇒ 11

~5 → ~0101 =10 10 ⇒ 10

Desplazamiento :

Desplazan los bits del primer operando , en la dirección señalada , tantos bits como indique el segundo operando , rellenando con 0 los huecos libres . Los operandos de desplazamiento son : >> (desplazamiento a la derecha) , << (desplazamiento a la izquierda):

valor >> n : desplaza n bits a la derecha el valor de la variable valor .

valor << n : desplaza n bits a la izquierda el valor de la variable valor .

Para ver esto con mucho más detalle vamos a ver un ejemplo y su representación gráfica :

```
unsigned char var;
var = 12 ;   /* 12 = 00001100b */
var << 3;   /* Desplazamos 3 bits a la izquierda */
```

, antes del desplazamiento tenemos :

0	0	0	0	1	1	0	0

, después del desplazamiento tenemos :

0	1	1	0	0	0	0	0

Operadores de relación.

Los operadores de relación los podemos agrupar en 2 : los operadores de comparación y los operadores lógicos.

Operadores de comparación :

Los operadores de comparación se muestran en la siguiente tabla :

Operador	Significado
<	Menor que
<=	Menor o igual que
>	Mayor que
>=	Mayor o igual que
==	Igual que
!=	No igual que

Los operandos se usan para crear expresiones lógicas , cuyo resultado puede tener 2 valores posibles : 1 (cierto) ó 0 (falso).

5 >= 9 ⇒ 0

3 != 4 ⇒ 1

Operadores lógicos :

Los operadores lógicos son los que se muestran a continuación en la siguiente tabla :

Operador	Significado
&&	Y
||	O
!	Negación lógica

- Una operación Y lógica será cierta si ambos operandos son ciertos.
- Una operación O lógica será cierta si alguno de los operandos es cierta.
- El operador negación niega el valor de la expresión lógica.

Operadores compuestos de asignación .

Operadores que realizan de una sola operación una asignación y una operación binaria . Estos operadores son :

***=** **/=** **%=** **+=** **-=** **>>=** **<<=** **&=** **^=** **|=**

El modo de actuación de estos operandos es el siguiente:

exp1 op= exp2 ⇒ exp1 = exp1 op exp2

Por ejemplo exp1 += exp2 es lo mismo que : exp1 = exp1+exp2.

Operador condicional.

La expresión condicional es :

exp1 **?** exp2 **:** exp3

Cuando se evalúa esta expresión condicional , exp1 se evalúa la primera . Si exp1 es cierta (valor no nulo) , entonces se evalúa exp2 y éste es el valor de la expresión condicional . Sin embargo , si exp1 es falsa (valor igual a 0) entonces se evalúa la exp3 y éste es el valor de la expresión condicional.

Ejemplo :

```
mínimo = (f<g)?f:g ;
```

Se evalúa la expresión f<g si ésta es cierta se asigna a mínimo el valor f , de lo contrario se asigna a mínimo el valor de g.

Operadores de incremento y decremento.

La operación de incremento (+=) ó decremento (-=) un valor en una sola unidad puede hacerse con los operandos (++) y (--).

x += 1 , realiza lo mismo que x++

Hay que diferenciar entre poner el operando antes ó después de la variable. Si se pone delante , primero realiza la operación y luego utiliza el valor que contiene , si se pone detrás , primero utiliza su valor y luego realiza la operación. Para ver esto mejor veamos un ejemplo :

n=5 → x=n++ → x=5 y n=6

n=5 → x=++n → x=6 y n=6

Procedencia de los operadores.

A continuación se expone una tabla con las precedencias de todos los operadores . Siempre se evalúan en primer lugar las operaciones con mayor precedencia. En el caso de operaciones de igual precedencia se evalúa según la columna correspondiente a la asociatividad , que indica si empieza a evaluarse de derecha a izquierda ó al revés . En la tabla que a continuación se muestra , también se expone el número de operados que intervienen con cada operando.

Operadores	Asociatividad	Número operandos
() [] -> .	I→D	Unario
! ~ ++ --	D→I	Unario
- * & (tipo)	D→I	Unario
sizeof		Unario
* / %	I→D	Binario
+ -	I→D	Binario
<< >>	I→D	Binario
< <= >= >	I→D	Binario
== !=	I→D	Binario
&	I→D	Binario
^	I→D	Binario
|	I→D	Binario
&&	I→D	Binario
||	I→D	Binario
?:	D→I	Triario
= += -= *= /= %=	D→I	Binario
&= ^= |= <<= >>=	I→D	Binario

.4 - CONTROL DEL FLUJO DEL PROGRAMA .

En un programa no siempre se ejecutan las mismas instrucciones y en el mismo orden. A veces es necesario que dependiendo de señales externas (señales de usuario , del sistema ...) o de los datos que estamos procesando se tomen unas u otras decisiones . Para todo ello disponemos de distintas instrucciones o sentencias que a continuación se van a pasar a exponer.

Proposiciones y bloques.

Una proposición es una expresión seguida de punto y coma (;) . Una proposición compuesta o bloque es un conjunto de declaraciones y proposiciones agrupadas entre llaves ({ }). Después de la llave de cierre no ponemos punto y coma.

Ejemplo:

```
{  /* Bloque */
        int x;  /* Declaraciones */
        x=20;  /* Proposiciones */
        x+=10;
}
```

Sentencia while.

Sintaxis :

```
while (expresión )
                proposición ;
```

La sentencia se repetirá mientras que el valor de la expresión no sea 0 . Si es booleana la expresión , la sentencia se realizará mientras sea cierta .

Ejemplo: Función que visualiza del 0 al 9.

```
#include <stdio.h>
main ()
{
        int digito = 0;
```

```
        while ( digito <= 9 )
        {
                printf ("%d\n" , digito);
                ++ digito ;
        }
}
```

Sentencia do-while.

Sintaxis :

```
do
{
        proposiciones ;
} while (expresión);
```

En el caso anterior la comprobación se realizaba al principio , en este caso la comprobación se realiza al final , de manera que se asegura de que la sentencia se evalúe al menos una vez.

Ejemplo : Función que visualiza del 0 al 9.

```
#include <stdio.h>
main ()
{
        int digito = 0;
        do
        {
                printf ("%d\n" , digito);
                ++ digito ;
```

```
        } while ( digito <= 9 );
}
```

Sentenencia for.

Sintaxis:

for (inicialización **;** expresión **;** progresión **)**
proposición ;

Mientras expresión sea verdadera , se estará ejecutando la proposición . Inicialización es una expresión o conjunto de expresiones , para inicializar las variables de control que intervienen en expresión , y progresión es una expresión o conjunto de expresiones que indica cómo evolucionan las variables de control .

Ejemplo :

```
int a;
...
for (a =0 ; a<10 ; a++)
        printf ("i=%d\n",a);
...
```

No es necesario que incluyamos ni la inicialización , ni la expresión ni la progresión , pero si deben estar los separadores ; . Además se debe de cumplir :

- La ausencia de la inicialización y la progresión no pasa nada , si se inicializa o modifica los valores de la variable en otro lugar.

- Si se omite la expresión , se supondrá que tiene valor 1 por tanto el bucle se ejecutará continuamente , para salir habrá que hacer uso de algún **break** ó **return** .

Ejemplo:

```
int digito = 0;
...
for ( ; digito <= 9 ; )
        printf ("%d \n" ,digito++);
...
```

Sentencia if-else.

Sintaxis :

```
if (expresión)
        proposición1;
else
        proposicón2;
```

Si expresión es verdadera (distinta de 0) se ejecutará la proposición1 ; en caso contrario , se ejecutará la proposición2.

Ejemplo:

```
int x,y,z;
...
/* Cálculo del máximo de 2 números */
if (x>y)
```

```
        z = x;
else
        z = y;
```

Tanto proposición1 como proposición2 pueden ser bloques de código.

Sentencia else-if.

Sintaxis :

```
if (expresión1)
        proposición1;
else if (expresió2)
        proposición2;
...
else if (expresiónN-1)
        proposiciónN-1;
else
        proposiciónN;
```

La siguiente tabla nuestra cuál es la proposición que se ejecuta en función de los valores booleanos de la expresión.

expresión 1	expresión 2	...	expesión N-1	Proposición que se ejecuta.
verdad	X	X	X	proposición 1
falso	verdad	X	X	proposición 2
...	...	...	...	...
falso	falso	falso	verdad	proposición N-1
falso	falso	falso	falso	proposición N

Sentencia switch.

La proposición switch es una decisión múltiple que prueba si una expresión coincide con alguno de entre un número de valores constantes enteros y traslada el control adecuadamente.

Sintaxis:

```
switch (expesión)
{
        case exp_const1 : proposiciones1;
        case exp_const2 : proposiciones2;
        case exp_constN : proposicionesN;
        default : proposiciones ;
}
```

Ejemplo:

```
int mes , dia_mes;
...
printf (“Mes”);
scanf (“%d” ,&mes);
switch (mes)
{
        case 1:
        case 3:
        case 5:
        case 7:
        case 8:
        case 10:
        case 12:
```

```
                dia_mes = 31;
                break;
        case 4:
        case 6:
        case 9:
        case 11:
                dia_mes = 30;
                break;
        case 2:
                dia_mes = 28;
                break;
        default  :
                printf ("%d , mes erróneo.\n", mes );
}
```

Sentencia break.

Se usa para terminar la ejecución de bucles o salir de un **switch** .

Sintaxis:

```
break ;
```

En el caso de un switch se debe incluir un break en cada uno de los casos , excepto en el último ya que después de la ejecución del último saldrá directamente del switch .

Ejemplo :

```
switch (color)
{
        case 'r':
                printf ("rojo");
                break;
```

```
        case 'b':
                printf ("blanco");
                break;
        default :
                printf ("ERROR");
}
```

Si incluimos un **break** en un bucle **while** , **do-while** o **for** entonces se transfiere el control fuera del bucle en el momento en el que se encuentra la sentencia **break** . Esto proporciona una forma conveniente de terminar un bucle cuando se detecta una error o alguna condición irregular .

Sentencia continue.

Se usa para saltar el resto de la pasada actual a través de un bucle . El bucle no termina cuando se encuentra una sentencia **continue** , sencillamente no se ejecutan las sentencias que se encuentran a continuación en él y salta directamente a la siguiente pasada a través del bucle.

Sintaxis:

continue ;

Se puede usar con las sentencias **while** , **do-while** o **for** .

Ejemplo:

```
do
{
        scanf ("%f",&x);
        if (x<0)
        {
```

```
                printf ("ERROR-valor negativo de x");
                continue;
        }
        /* Procesamiento de valores no negativos de x */
        ...
} while (x<=100);
```

En el ejemplo cuando se introduzca un valor negativo , este no llegará a procesarse pues cuando llegue a **continue** , mandará el control al principio del bucle para realizar la siguiente iteración.

Sentencia goto.

Se utiliza para alterar la secuencia de ejecución normal de un programa , transfiriendo el control a otra parte del programa.

Sintaxis:

```
goto etiqueta;
...
etiqueta : sentencia ;
```

Cada sentencia etiquetada dentro de un programa (mejor dicho dentro de una función) debe tener una única etiqueta , es decir 2 sentencias no pueden tener la misma etiqueta.

Ejemplo: Transferencia del control fuera de un bucle si hay un error.

```
/* bucle principal */
scanf ("%f",&x);
while (x<=100)
{
        if (x<0) goto error;
```

```
            ...
            scanf ("%f",&x);
}

/* rutina de tratamiento del error */
error :  {
                printf ("ERROR-valor negativo de x");
                ...
            }
```

Aunque su uso no es recomendado . Hay situaciones en las que esta permitido y es bueno usarlo . Por ejemplo en alguna situación que estamos dentro de muchos bucles anidados y se quiere salir fuera de varios de ellos , podemos usar **break** varias veces pero sería más útil usar **goto**.

.5 - TIPOS DE DATOS AVANZADOS.

Llamamos datos avanzados o derivados a aquellos constituidos a partir de los tipos da datos fundamentales o de otros tipos derivados. Vamos a ver los siguientes tipos de datos : enumeraciones , punteros , arrays, cadenas de caracteres , estructuras , uniones y campos de bits.

Enumeraciones.

Una enumeración es un tipo de dato entero con valores constantes definidos por el usuario . Para verlo más claramente veamos un ejemplo , definiendo una enumeración de tipos de reguladores :

```
enum tipo_reguladores { RP , RPI , RPD , RPID };
```

Cada nombre se almacena como un entero , y toma el valor inmediatamente superior al nombre anterior . Si no se indica nada el primer nombre asume el valor 0.

De esta manera para referirnos al valor RP de una variable de tipo_regulador podemos hacerlo de 2 maneras :

- Indicando el nombre : variable = RP;

- Indicando el entero al que hace referencia : variable = 0;

Arrays .

Los arrays son bloques de elementos del mismo tipo. El tipo base de un array puede ser un tipo fundamental o un tipo derivado . Los elementos individuales del array van a ser accesibles mediante una secuencia de índices . Los índices , para acceder al array , deben ser variables o constantes de tipo entero . Se define la dimensión de un array como el total de índices que necesitamos para acceder a un elemento particular del array.

La definición formal de un array N-dimensional es la siguiente :

```
tipo_array nombre_array [rango1][rango2]...[rangoN] ;
```

Por tanto para definir una variable como array hay que indicar después del tipo y del nombre , el número de elementos a almacenar , encerrado entre corchetes.

Ejemplos:

```
int vector [10] ; /* Reserva espacio de memoria para 10 enteros */
int matriz_entera[10][10];  /* Matriz  de 10 filas por 10 columnas */
```

Los valores del array pueden inicializarse :

```
int vector [10] = { 1 , 3 , 3 , 4 , 5 , 5 , 0 };
```

, que dará lugar a que en la memoria se almacenen los siguientes valores :

1 , 3 , 3 , 4 , 5 , 5 , 0 , ? , ? , ?

, como se observa anteriormente , sino se inicializan todos los elementos del array , aquellos a los que no les asignamos ningún valor tomarán un valor que no podemos predecir.

Para acceder a un valor almacenado en un arrya hacemos uso del operador **[]** , indicando en él , el número de elemento solicitado (este número va del 0 al n-1 , siendo n el número total de elementos que contiene el array).

```
b = valor [4] ; /* Asigna a b el valor de la componente 4+1 = 5 del array */
```

En el caso de que inicialicemos el array en la definición puede omitirse el número de elementos , ya que se sobreentiende :

```
int valores [] = { 1 , 0 , 5 };  /* Crea un array de 3 elementos */
```

Los array unidimensionales de tipo char se conoces como cadenas de caracteres , y los arrays de cadenas de caracteres (matrices de caracteres) se conocen como tablas.

Cadena de caracteres.

Una cadena de caracteres no es más que un grupo de caracteres. Una cadena de caracteres es un array de caracteres que finaliza con el carácter terminador '\0' (carácter 0 del juego de caracteres ASCII) . De manera que para almacenar una cadena será necesario declarar la variable con tantos espacios como caracteres mas 1 , el cual es necesario para albergar al carácter terminador . Así por ejemplo :

```
char cadena [5] = "Hola" ;
```

,que se almacenaría en memoria de la siguiente forma :

H	o	l	a	\0

También puede omitirse el número de elementos indicado entre paréntesis , lo que dará lugar a la mismo que antes.

Puntero .

Un puntero es una variable que contiene la dirección de memoria de otra variable . Por tanto permite acceder a esa variable indirectamente. Un puntero únicamente apunta a una dirección de memoria , no crea una variable en su lugar. De modo que cuando se define un puntero , se asigna un espacio de memoria donde almacenar un valor y se puede operar con ese valor , pero además se puede acceder al contenido de la dirección de memoria a la que este apunta.

Declaración de un puntero:

La declaración de un puntero tiene la siguiente sintaxis :

```
tipo_base *puntero;
```

Se escribe el tipo de dato al que apunta , seguido de un asterisco (*) y del nombre del puntero.

Ejemplo:

```
char *pCaracter;
```

La variable pCaracter es un puntero a un carácter , cuando se hace esto no se reserva memoria para contener un carácter , sino que se reserva memoria para contener una dirección.

OPERADORES :

- Operador **&** : Permite obtener la dirección de memoria de una variable.

Ejemplo:

```
char c='a';
char *pCaracter = &c; /* pCaracter tiene almacenada la dirección de c */
```

- Operador * : Permite acceder a los datos almacenados en las dirección de memoria almacenada en el puntero al que se le aplica el operador.

Ejemplo :

```
int *pEntero;
int x,y;
pEntero = &x;
y = *pEntero ;  /*  Ahora y contiene el mismo valor que x */
```

- Operadores aritméticos : Con los punteros se pueden realizar las operaciones de adición (+ , += , ++) , substracción (- , -= , --) y asignación (=).

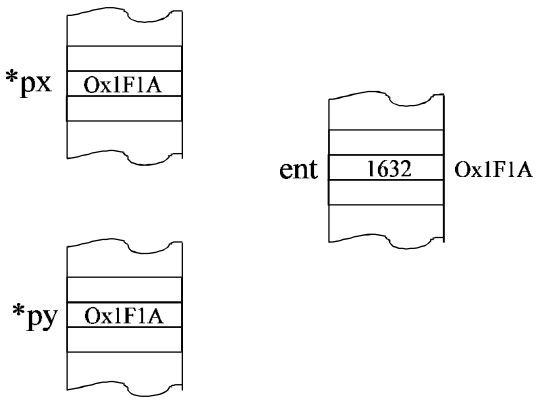
Ejemplo :

```
int *pEntero;
pEntero += 5 ;   /* Incrementa pEntero para que apunte 5 enteros más adelante */
```

Ejemplo : Uso del operador de asignación entre punteros.

```
int ent;
int *py;
int *px = &ent;
py = px ; /* hacemos que px apunte a una variable ent */
```

Gráficamente tendremos :



Estructuras .

Una estructura es un agregado de tipos fundamentales o derivados que se compone de varios campos. A diferencia de los arrays , cada elemento de la estructura puede ser de un tipo diferente.

Definición de una estructura.

La forma de definir una estructura es :

```
struct nombre_estructura
{
        tipo1 campo1;
        tipo2 campo2;
        ...
        tipoN campoN;
};
```

Ejemplo:

```
struct fecha
{
        unsigned short dia;
        unsigned short mes;
        unsigned int año;
};
struct fecha hoy ;
```

Hoy es una variable declarada de tipo struct fecha.

Procesamiento de una estructura :

Para acceder a los campos de una estructura utilizaremos el operador '**.**' . Así para acceder al campo mes de la variable anterior , escribiremos:

```
hoy.mes = 12;
```

Se pueden declarar arrays de estructuras :

Ejemplo :

```
struct
{
        float real , imaginaria ;
} vetor [10] ;
```

La variable vector es un array unidimensional de números complejos . Para acceder a la parte real del elemento 5 , escribiremos :

```
vector [4].real = 10;
```

También se pueden declarar punteros a estructuras . En estos casos , el acceso a los campos de la variable se hace por medio del operador '->'.

Ejemplo :

```
struct alumno
{
        char nombre [61] ;
        float nota;
};
struct alumno alumno , *pa ;
...
pa = &alumno ;
pa -> nota = 10.0 ;
```

Por último , diremos que puesto que el tipo de cada campo de una estructura puede ser un tipo fundamental o derivado , también puede ser otra estructura . Tendremos así declaradas estructuras dentro de estructuras.

Ejemplo :

```
struct fecha
```

```
{
        int dia , mes , año ;
};
struct alumno
{
        char nombre [61];
        struct fecha fecha_nacimiento ;
        float nota ;
};
struct alumno  alumno;
```

Con estas declaraciones podemos hacer asignaciones como :

```
alumno.fecha_nacimiento.mes = 12 ;
```

Tipos definidos por el usuario.

Es una declaración que permite definir tipos de datos , que ya tenemos , con nombres diferentes y también con nuevos tipos de datos.

Sintaxis :

typedef tipo nuevo_tipo;

Ejemplo :

```
typedef int edad ;   /* Ahora edad es un dato de tipo int */
edad varon , hembra ;
```

También es muy útil para definir estructuras , ya que luego no será necesario poner **struct** nombre_escritura donde se haga referencia .

Ejemplo:

```
typedef struct
```

```
{
        int dia , mes , año ;
} fecha;
fecha hoy , mañana ;
```

Uniones .

Las uniones se definen de forma parecida a las estructuras y se emplean para almacenar en un mismo espacio de memoria variables de distintos tipos .

Sintaxis :

```
union nombre_union
{
        tipo1 campo1;
        tipo2 campo2;
        ...
        tipoN campoN;
};
```

El tamaño de la unión no va a ser igual a la suma de cada uno de sus campos como ocurre con las estructuras , sino que es igual al tamaño del mayor de sus campos .

Ejemplo :

```
union numero_mixto
{
        float real;
        int entero;
};
```

El tamaño de esta unión es de 4 bytes (tamaño del campo real) . Con esa declaración vamos a poder hacer asignaciones como :

```
numero.entero = 10;
numero.real = 124;
```

Hay que insistir en que esta unión no ocupa el espacio de sus dos campos , sino el del mayor de ellos . Si después de las asignaciones anteriores imprimimos el valor de numero.entero veremos que no es 10.

Campo de bits.

C brinda la posibilidad de definir variables cuyo tamaño en bits puede no coincidir con un múltiplo de 8 .

Sintaxis :

```
struct nombre_campo
{
        unsigned campo1 : tamaño1 ;
        unsigned campo2 : tamaño2 ;
        ...
        unsigned  campoN  :  tamañoN
;
};
```

Ejemplo :

```
struct byte
{
        unsigned char b0:1;
        unsigned char b1:1;
        unsigned char b2:1;
        unsigned char b3:1;
        unsigned char b4_7:4;
};
```

.6 - FUNCIONES.

Una función es un conjunto de instrucciones , que se ejecutan cuando se le llama , es decir , cuando se referencia a esta función desde otro punto del programa. Para que la función pueda procesar distintos datos o de distinta forma dependiendo del punto del programa desde el que se le llame , se puede pasar a la función la información que debe tratar en forma de parámetros y ésta puede devolver el resultado de este tratamiento como valor de retorno.

Las funciones van a permitir agrupar bajo un identificador una serie de proposiciones concebidas para realizar alguna tarea específica . La organización de un programa grande en funciones sencillas hará que el programa sea estructurado además de fácil de depurar y mantener .

Una función tiene 3 estados : definición , declaración y llamada.

Definir la función .

Consiste en escribir el código que se ha de ejecutar cuando se llame a esa función.

La definición de una función según el estándar ANSI es :

```
tipo nombre_función (declaración_parámetros)
{
        variables_locales;
        proposiciones;
        return (expresión);
}
```

Kernighany Ritchie , en la primera edición de " El lenguaje de programación C." utilizan la siguiente sintaxis para la definición de funciones :

```
tipo nobre_función (parámetros)
declaración_parámetros;
{
        variables_locales;
        proposiciones;
        return (expresión);
}
```

Ejemplo:

```
factorial (n)
int n;
{
        int i , factorial = 1;
        for (i = 1 ; i<= n ; i++)
                factorial *= i;
        return factorial;
}
```

Una función puede ser de cualquier tipo , excepto tipo función o array . Es decir , una función no puede devolver otra función , ni tampoco un array . Si no se especifica nada , el tipo de una función es **int** .

Las variables locales y los parámetros de la función se reservan en la pila de usuario del programa , por lo que al entrar en la función se reserva espacio para ellos , pero al salir de la función desaparecen . Este tipo de almacenamiento se conoce como automático , en contrastación al almacenamiento estático . Si una variable local va precedida del calificador **static** , su almacenamiento será estático y ocupará la zona de memoria reservada a las variables globales ; además , esa variable existirá durante todo el tiempo de ejecución del programa.

Declarar la función.

Normalmente , necesitamos llamar a una función sin haberla definido todavía , o que se ha definido en un archivo diferente . Cuando el compilador localiza la llamada a una función comprueba que la función existe , que los parámetros que se le pasan son los adecuados y que el uso del valor de retorno es el adecuado para el tipo de dato devuelto . Si la función no está definida previamente , el compilador no puede realizar estas comprobaciones y genera un error . Si no interesa definir la función antes de su llamada , podemos declarar la función , que es una forma de indicar al compilador el formato de la función.

Para declarar una función se debe escribir el tipo del valor de retorno , seguido del nombre de la función y de los tipos de los distintos parámetros encerrados entre paréntesis y separados por comas . La declaración finaliza con un punto y coma (;).

Ejemplo :

```
in sumar ( int , int ) ; /* Función que recibe 2 enteros y devuelve otro */
```

Una regla consiste en definir las funciones en archivos de extensión .C y declararlas en archivos con el mismo nombre y extensión .H , llamados archivos de cabecera . De esta forma , simplemente con incluir estos archivos de cabecera , se consigue que el compilador pueda realizar las comprobaciones necesarias.

Llamar a la función.

Para llamar a una función se indica su nombre y , entre paréntesis , los parámetros que se le quieran pasar . De esta forma , cuando el flujo de programa pase por el punto donde está alguna de las llamadas , se ejecutará la función con los parámetros especificados.

Ejemplo :

```
int a ;
int b;
a = sumar ( b , 3 );
```

Cuando el programa llegue a este punto , ejecutará la función sumar , pasándole como parámetros el valor de b y 7 , guardando el valor que devuelve en la variable a.

Paso de parámetros por valor y por referencia.

Es importante tener en cuenta que los parámetros de una función sólo se pueden modificar a nivel local.

Ejemplo:

```
int f( int x );
{
        x=20;
}

main ()
{
        int x=10;
        f(x);
        printf ("%d\n" , x);  /* Esto va a imprimir un 10 */
}
```

Para modificar un parámetro de forma permanente , hay que trabajar con un puntero al mismo . Así , el programa anterior se podría escribir como sigue.

Ejemplo :

```
int f (int *x)
{
        *x = 20;
}

main ()
{
        int x = 10;
        f ( &x );
        printf ( "%d\n" , x) ; /* Esto va a imprimir el valor 20 */
}
```

El paso de arrays como parámetros de funciones siempre se realiza por referencia . Las estructuras se pueden pasar por valor o por referencia . Si pasamos una estructura por valor , las modificaciones de sus campos serán locales mientras que pasándolas por referencia , las modificaciones serán permanentes.

El hecho de que los parámetros pasados por valor sólo sufran modificaciones a nivel local , se debe a que el parámetro es una copia en la pila de usuario de la variable a la que se refiere . Así , las modificaciones se hacen sobre la copia de la variable que hay en la pila y no sobre la propia variable.

Librerías estándar.

C tiene definidas un conjunto de funciones estándar , implementadas en todos los compiladores de este lenguaje y que el programador puede utilizar . Estas funciones se encuentran en las denominadas librerías estándar. Junto con estas librerías se suministran los archivos de cabecera con las declaraciones de funciones , que debemos incluir en nuestros programas cuando utilicemos alguna de las funciones declaradas en ellos . Las librerías son archivos con extensión .LIB que incluyen el código ya compilado y deben juntarse con nuestro código al realizar el link . Para incluir los ficheros de cabecera debemos usar la directiva **#include** , seguida del nombre del fichero de cabecera que deseemos incluir entre los signos <> o ‘’ según se encuentren el el directorio \INCLUDE o en cualquier otro respectivamente . En algunos ejemplos anteriores ya hemos visto la utilización de algunas funciones de librería como puede ser las funciones printf , scanf ...