LENGUAJE C para MICROCONTROLADORES

El lenguaje de programación C fue desarrollado e implementado bajo el sistema operativo UNIX.
Una de las mejores características de este lenguaje es que no está atado bajo ningún hardware o sistema en particular.
El lenguaje C es ubicado como un lenguaje de programación de nivel medio, que combina los elementos de un lenguaje de alto nivel, con la eficiencia de un lenguaje ensamblador.

Ventajas

- C es un lenguaje con características de bajo nivel.
- Es un lenguaje estructurado.
- C es un lenguaje que te permite crear cualquier tarea que tengas en mente.
- C te da un control completo de nivel de programación.
- Portable.
- El lenguaje C permite la modularización (módulo de visualización, de comunicación, etc.) y los cambios en un módulo no afectan al resto.

Estandarización ANSI C

La primera estandarización del lenguaje C fue en ANSI, con el estándar X3.159-1989. El lenguaje que define este estándar fue conocido vulgarmente como ANSI C. Posteriormente, en 1990, fue ratificado como estándar ISO (ISO/IEC 9899:1990). La adopción de este estándar es muy amplia por lo que, si los programas creados lo siguen, el código es portátil entre plataformas y/o arquitecturas.

Partes de un programa en C para sistemas embebidos (Microcontroladores)

Un programa escrito en C está conformado por pequeños módulos de código llamados funciones.
El módulo principal o función principal está conformado por la función que denominamos:

```
void main(void){

}
```

Las instrucciones que conforman ésta se encierran entre **{ }**.
Sobre la parte superior de la función **`void main(void)`** se pueden declarar los siguientes objetos que pueden ser manipulados por el programa.

1. Directivas para Incluir archivos, **#include**
2. Directivas para incluir macros y/o constantes, **#define**
3. Declaración de Prototipos de las funciones.
4. Declaración de ***Variables Globales***
5. Declaración de tipos de datos.

Dentro de la función **void main(void)** entre **{** y el **for(;;)** encuentran las declaraciones de las ***Variables Locales*** y las instrucciones que definen los procesos a realizarse por única vez luego de efectuarse un **Power On** o un **Reset** en el Microcontrolador.
Dentro del **for(;;)** se encuentra el lazo principal, es decir el lazo de programa que se va a repetir por siempre.
Fuera de la función **void main(void)** y después de la llave **}** que cierra el main aparecen las declaración de las funciones.

```
/* 1. Directivas para Incluir archivos, #include */
#include <hidef.h>
#include "derivative.h"

/* 2. Directivas para incluir macros y/o constantes,  #define */
#define VERDADERO 1
#define FALSO     0
#define LED PTAD_PTAD0

/* 3. Declaración de Prototipos de las funciones. */
void LEDOn (void);

/* 4. Declaración de Variables Globales */
unsigned char Resultado=0;

/* 5. Declaración de tipos de datos. */
typedef union {
  unsigned int  Palabra;
  unsigned char PalabraH;
  unsigned char PalabraL;
} midato;

/* Función Principal */
void main(void) {
/* Declaración de Variables Locales */
  unsigned char Operador=0;

/* Instrucciones que definen los procesos únicos luego de PowerOn */
  SOPT1_COPT=0;
  PTADD=0xFF;

  for(;;)
  {
    /* Lazo Principal */
  }
}

/* Declaración de las funciones */
void LEDOn (void)
{
/* Declaración de Variables Locales */

/* Instrucciones que definen los procesos */
}
```

CONSTANTES

Son valores que el compilador reemplazará en la primera pasada y que no sufrirán ninguna modificación durante la ejecución. Pueden ser de cualquier tipo de dato, y también pueden tener operaciones para resolver. Siempre estarán en la memoria de programa (FLASH) y según como se las defina pueden:

- estar ubicadas en el transcurso del programa, utilizando la directiva **#define**
- estar en una ubicación fija en la memoria, utilizando el modificador **const**

*Constantes enumeradas (**enum**):*

Permiten asociar valores constantes a un nombre en forma similar al #define, con la ventaja de estar dentro de un mismo grupo y poder tomar valores consecutivos.
(Si no se asigna ningún valor, el primer elemento tomará el valor 0)

```
enum <Nombre> {
  Cero,
  Uno,
  Dos,
  Tres
}

enum <Nombre> {
  Primera=1,
  Segunda,
  Tercera,
  Quinta=5
}
```

Esta declaración sería equivalente a hacer:

```
#define Cero    0
#define Uno     1
#define Dos     2
#define Tres    3

#define Primera 1
#define Segunda 2
#define Tercera 3
#define Quinta  5
```

Constantes especiales tipo carácter

Código C	***Significado***	***Hexadecimal***	***Decimal***
'\b'	Espacio atrás <BS>	0x08	8
'\r'	Enter <CR>	0x0D	13
'\n'	Retorno de carro <LF>	0x0A	10
'\0'	Nulo <NULL>	0x00	0
'\f'	Salto de página <FF>	0x0C	12
'\t'	Tabulación horizontal <TAB>	0x09	9
'\"'	Comilla doble	0x22	34
'\''	Comilla simple	0x27	39
'\v'	Tabulación vertical <VT>	0x0B	11

DECLARACIÓN DE VARIABLES

Una variable es un espacio reservado en la memoria para contener valores que pueden cambiar durante la ejecución de un programa.
Sobre la parte superior de la función ***void main (void)***, dentro de ella ó dentro de cualquier función se deben definir variables, ejemplo:

tipo_de_dato **Nombre**; ó *modificador* tipo_de_dato **Nombre**;

- unsigned char a, b, x;
- int y;
- unsigned int c;
- unsigned char c[10];

En una declaración de variables le estamos diciendo al compilador que debe reservar espacio en memoria, que a cada espacio en memoria le asigne un nombre y un número determinado de bytes, también se le dice que tipos de datos puede almacenar. No hay que olvidar, que la información en el interior de la memoria es siempre binaria; el modo en que se interpreta esta información es siempre arbitrario, es decir, el mismo valor puede usarse para codificar una letra, un número, etc.
El compilador deduce esa información del ***tipo de dato*** que se escribe con cada declaración.
Las variables que se declaran dentro del ***void main (void)*** se dice que son ***variables locales***, las que se declaran fuera se dicen ***globales***, más adelante detallaremos las diferencias entre estos tipos de variables.

Tipos de datos
El lenguaje tiene cinco tipos de datos básicos y cuatro tipos de modificadores. La Tabla muestra la definición de los tipos de datos básicos y el tipo de modificador.

Tipo	***Significado***	***Palabra Clave***	***Reserva***
Carácter	Caracteres	*char*	1 Byte
Entero	Números enteros con signo	*int*	2 Bytes
Largo	Números enteros con signo	*long*	4 Bytes
Flotante	Números Reales con signo	*float*	4 Bytes
Doble	Números Reales con signo	*double*	8 Bytes

Modificadores

Palabra Clave	***Significado***
unsigned	Caracteriza a la variable como positiva.
Volatile	Le indica al compilador que el valor puede cambiar por acción externa.
Static	Localiza a la variable en una posición fija de la RAM, y solo puede ser reconocida y manipulado por el modulo que la definió.
Extern	Se usa para indicarle a un archivo que la variable está declarada en otro archivo.
Const	Indica que la variable declarada no puede ser modificada (Suele almacenarse en la memoria FLASH)
Far	Asigna la variable fuera de la página cero (necesitan direccionamiento extendido)
Near	Asigna la variable en la página cero (menor tiempo de ejecución)

DEFINICIONES DE TIPO – Palabra clave **`typedef`**

Es la manera en que el programador puede definir sus propios tipos de datos (basados en los ya existentes) En ocasiones puede ser útil definir nombres para tipos de datos, 'alias' que nos hagan más fácil declarar variables y parámetros, o que faciliten la portabilidad de nuestros programas.

```
typedef unsigned char midato;

midato Variable=0;
```

PUNTEROS **`*`**

Es un tipo especial de variable que contiene una dirección de memoria; esa dirección puede contener cualquier tipo de dato. Es decir que un puntero es una posición en la memoria que contiene la dirección donde está almacenado un dato y por lo tanto su declaración debe estar acorde con el tipo de dato que tiene que apuntar.

tipo_de_dato ***NombreP**; ó *modificador* tipo_de_dato ***NombreP**;

OPERADORES

Los operadores son los elementos que posibilitan ciertos cálculos cuando son aplicados a Variables o a otros objetos en una expresión.

OPERADORES ARITMÉTICOS	
=	Asignación
-	Resta o substracción
+	Suma o adición
*	Multiplicación o producto
/	Cociente entero de la división
%	Residuo entero de la división
--	Decrementa en 1
++	Incrementa en 1
OPERADORES RELACIONALES (TRUE = 1 - FALSE = 0)	
<	Menor que
>	Mayor que
<=	Menor o igual que
>=	Mayor o igual que
==	Igual que
!=	Distinto que
OPERADORES LÓGICOS BOOLEANOS (TRUE = 1 - FALSE = 0)	
||	O (si se cumple alguna=1 – OR)
&&	Y (si se cumplen todas=1 – AND)
!	NO (si no se cumple=1 - NOT)
OPERADORES ORIENTADOS A BITS	
|	OR
&	AND
~	NOT
^	XOR
>> n	Desplaza n veces a la derecha
<< n	Desplaza n veces a la izquierda
OPERADORES DE PUNTEROS	
&	Dirección de la variable (de referencia)
*	Contenido de la dirección (de indirección)

Operadores de Punteros

El operador **&** nos devuelve la dirección de memoria del operando.

El operador ***** considera a su operando como una dirección y devuelve su contenido

```
direccion = &Variable;
dato = *direccion;
*direccion = dato;
```

Ejemplo:

```
unsigned int A,B;   // Supongamos que A se almacena en la pos. 0x80
unsigned int *DIR;

DIR=&A;   // DIR almacena la dirección de A, entonces A=0x80;
B=*DIR;   // DIR almacena la dirección de A, entonces B=A;

*DIR=103; // A la dirección apuntada por DIR le asigna 103, A=103;
```

FUNCIONES

Su objetivo es proveer un mecanismo que permita invocar una secuencia de código, que pueda ser llamada varias veces dentro del programa. Esto no sólo facilita el entendimiento del código, sino que genera código reutilizable.
El ***PROTOTIPO*** de una función es la línea de código que la declara indicado: que devuelve la función al programa (después de ser invocada) - su nombre – sus argumentos (datos que necesita que la transfiera el programa).

```
<tipo_de_dato_que_retorna> nombre (<tipo_de_dato_que_recibe>){

}
```

Si la función no retorna valores y/o no necesita argumento, se puede definir:

```
void nombre (<tipo_de_dato_que_recibe>){

}
```

ó,

```
<tipo_de_dato_que_retorna> nombre (void){

}
```

Si el prototipo se declara sin retorno y sin argumentos explícitos, el compilador asume que tanto el retorno como el argumento son del tipo ***int*** (esta situación suele generar un warning)
Ejemplos de definición de funciones:

```
int nombre (int dato1, int dato2){

}
```

```
void nombre (char dato){

}
```

Tener en cuenta que una función puede retornar uno y solo un valor. Y por supuesto este prototipo debe encontrarse antes del llamado a la función.

Cuando la función retorna un valor, lo hace a través de la palabra clave **`return`** seguido por el valor.

```
int Suma (int dato1, int dato2){
  int Resultado=0;
  Resultado=dato1+dato2;

  return (Resultado);
}
```

Dentro de los () de la palabra calve **`return`** podemos incluir operaciones directamente:

```
int Suma (int dato1, int dato2){

  return (dato1+dato2);
}
```

Cuando en un programa llamamos a una función, debemos entregarle los valores con los que va a realizar las operaciones, le pasamos los argumentos a la función.
Estos argumentos pueden pasarse:

- Por valor, este método copia el valor del argumento en la posición esperada por la función

```
void main (void){
  int Resultado;
  Resultado = Suma (24,3); // dato1=24 – dato2=3
                           // Retorna 27 en Resultado
}
```

- Por Referencia, este método copia el valor del argumento en la posición esperada por la función

```
void Suma (int *dato1, int *dato2, int *res){
  *res=*dato1+*dato2;
}

void main (void){
  int d1=24, d2=3, res=0;
  Suma (&d1,&d2,&res);
}
```

Archivos de Cabecera (Header .h)

Ahora que sabemos lo que es una función, podemos expandir un poco más el concepto de librerías.

Todo proyecto deberá estar conformado por varios módulos; cada uno de estos módulos tiene asociado un archivo con extensión ***.c*** y otro con extensión ***.h***.

El archivo con extensión ***.h*** contiene las funciones, las variables y los macros globales que serán utilizados por otros módulos:

El archivo con extensión ***.c*** que necesite referenciar las declaraciones de otro módulo, puede incluirlas usando la directiva `#include`.

SENTENCIAS DE CONTROL y CICLOS REPETITIVOS

Se llama flujo de control de un programa al orden en que se ejecutan las instrucciones que lo conforman. El flujo de control de los programas, es lineal, esto significa que el computador ejecuta una a una las instrucciones que se le indican, sin alterar el orden en que se escriben. Pero en muchos procesos que es necesario programar, es importante indicarle al computador que ejecute un conjunto determinado de instrucciones, cuando se cumpla una determinada condición y que ejecute otras cuando la condición no se cumpla, para esos casos el lenguaje C nos da la instrucción if else.
Una expresión lógica en C es una sentencia que al ser evaluada, el computador da un valor 0 si es falsa y un valor distinto de cero si es verdadera.

Operadores de Relación
Los siguientes operadores los utilizaremos para construir expresiones lógicas y establecen relaciones que pueden ser falsas o verdaderas.

Operador	***Significado***
>=	Mayor o igual que
<=	Menor o igual que
==	Igual que
!=	Diferente que
>	Mayor que
<	Menor que

Ejemplos:

```
A+B>=C*2
```

La expresión anterior compara el valor de la suma de A y B con el doble del valor de C, si la suma es mayor o igual que el doble de C entonces el resultado de la expresión es distinto de 0, en otro caso es 0.

La expresión: **`X=='S'`** compara el valor almacenado en la variable X con la letra S, el resultado da diferente de cero si en la variable X hay una letra distinta a la S.

La expresión: **`z=a+b>=c*2;`** compara el valor de la suma de a y b con el doble de c, si es menor almacena en z el valor 0 si es mayor o igual almacena un valor distinto de cero.

Operadores lógicos

C se tiene los siguientes operadores lógicos para formar expresiones lógicas más complejas.

Operador	***Significado***
&&	y
||	o
!	No

Si tenemos la expresión: **`(5>3) && (Y<X)`**, al evaluarla el resultado:

- Da 0 si no se cumple alguna de las dos condiciones.
- Da 1 si y solo si se cumplen ambas condiciones.

Si tenemos la expresión: **`(5>3) || (Y<X)`**, al evaluarla el resultado:

- Da 0 si no se cumple ninguna de las dos condiciones.
- Da 1 si se cumple alguna de las dos condiciones.

Por último el operador **`!`** se lo puede usar para generar una operación del tipo negada.
Si no se cumple que **`(5>3) && (Y<X)`** el resultado es verdadero, ejemplo:
!**`(5>3) && (Y<X)`**, al evaluarla el resultado:

- Da 1 si no se cumple alguna de las dos condiciones.
- Da 0 si y solo si se cumplen ambas condiciones.

Sentencia if…else

La instrucción **`if`** evalúa la expresión lógica, si ésta es verdadera, ejecuta la instrucción definida en <acción A>, si es falsa se ejecuta la instrucción inmediata al **`else`** definida en <acción B>, después de eso ejecuta la instrucción definida en <acción C> y las que le siguen.
Caben dos aclaraciones:
Primero: dentro de cada acción puede haber más de una instrucción, segundo
Segundo: Puede existir **`if`** sin **`else`**

```
if (expression_logica){
  <acción A>;
} else {
  <acción B>;
}
  <acción C>;
/**************************/
if (expression_logica){
  <acción A>;
}
  <acción C>;
```

Sentencia while

La instrucción **while** se utiliza para que se ejecute de manera repetida durante un número finito de veces un conjunto de instrucciones.
La instrucción **while** repite la ejecución de las instrucciones encerradas en **{ }** mientras la condición es verdadera. Sintaxis:

```
while (condición){
  <instrucciones>;
}
```

Las llaves encerrando las instrucciones le indican al compilador cuales son las instrucciones que se deben ejecutar mientras la condición se cumple.
Dentro de las instrucciones a repetir debe haber al menos una instrucción que haga que la condición sea falsa, de lo contrario no saldrá del ciclo.
A la condición también la puede hacer falsa un módulo de Hardware (Pin, interrupción, etc).

Sentencia do..while

La instrucción **do..while** es muy parecida a la vista anteriormente, con la diferencia que esta primero ejecuta las instrucción y luego evalúa la condición con la instrucción **while**.

```
do{
  <instrucciones>;
} while (condición);
```

Sentencia for

La instrucción **for** se utiliza para que se ejecute de manera repetida durante un número finito de veces un conjunto de instrucciones. Como vemos es muy similar al **while** con la diferencia que en esta conocemos de antemano la cantidad de veces que se va a repetir el lazo.
La sintaxis de la forma general de la sentencia **for** es:

```
for (<inicialización>;<condición>;<incremento/decremento>){
  <instrucciones>;
}
```

El bucle for permite muchas variaciones, pero existen tres partes principales:
<inicialización> Normalmente es una sentencia de asignación que se utiliza para la inicializar una variable de control del bucle.
<condición> Es una expresión lógica que determina cuando finaliza el bucle.
<incremento/decremento> Define como cambia la variable de control cada vez que se repite el bucle.
Las tres secciones se separan por ; (punto y coma) el bucle se repite hasta que la condición sea falsa.

```
for (i=0;i<55;i++){
  <instrucciones>;  //bucle que se repite 55 veces
}
```

Como un caso particular, tenemos el **for** sin una **<condición>,** como la condición no existe el el lazo no terminara nunca; es conocido como bucle infinito o for infinito.
Esta estructura es la que debe utilizarse como lazo principal dentro de la función **main**, para que el programa se ejecute indefinidamente.
Recordar: Asegurarse de no utilizar instrucciones que fuercen al programa a salir de este tipo de **for**.

```
for (;;){            //bucle infinito
  <instruccio 1>;
  <instruccio 2>;
  <instruccio 3>;
  <instruccio 4>;
}
```

Operador coma
La coma en C tiene como función encadenar varias expresiones. Esencialmente, la coma produce una secuencia de operaciones.
Se puede pensar en el operador coma, como "haz esto y esto y esto".
Más adelante veremos un ejemplo.

Instrucción break
Se puede usar para forzar la terminación inmediata de un bucle, saltando la evaluación de la condición normal del ciclo.
Cuando se encuentra la instrucción **break;** dentro de un bucle finaliza inmediatamente, y la ejecución continúa con la instrucción que le siguen al ciclo/bucle.

```
for (i=0;i<10;i++){
  if(TABLA1[i]>128) break;
  TABLA2[i]=TABLA1[i];
}
<instrucción 1>;
<instrucción 2>;
```

La sentencia **for** inicializa a las variable **i**, y se ejecuta mientras **i<10.**
El **for** repite la comparación y la guarda en **TABLA2[]** mientras esta sea menor a 128. Cuando sea mayor a 128 finaliza forzosamente el **for** y continua ejecutando las instrucciones a continuación de este.

Instrucción continue

La sentencia **continue;** fuerza una nueva iteración del bucle y salta cualquier código que exista entre medios.

```
for (j=0,i=0;i<10;i++){
  if(TABLA1[i]<128) continue;
  TABLA2[j]=TABLA1[i];
  j++;
}
```

La sentencia **for** inicializa a las variables **j** e **i**, y se ejecuta mientras **i<10.**
El **for** anterior repite la comparación mientras esta sea menor a 128. Cuando sea mayor a 128 copia al valor en una nueva tabla.

Sentencia switch

Esta sentencia permite verificar si una variable de tipo **char** o **int** tiene un valor determinado. Tiene la siguiente sintaxis:

```
switch(variable){
  case constante_1:
    <instrucciones_1>;
  break;

  case constant_2:
    <instrucciones_2>;
  break;
   ...
  case constant_n:
    <instrucciones_n>;
  break;

  default:
    <instrucciones_por_defecto>;
 }

  <instrucciones>;
```

La sentencia **switch** compara el valor de la variable con la constante_1, si son iguales ejecuta **<instrucciones_1>** y llega al **break**, y ejecuta luego las **instrucciones** siguientes a la llave }.
Si no son iguales compara el valor de la variable con el valor de la constante_2, sino son iguales compara con el de la constante_3, y así sucesivamente. Si no existe ninguna constante igual al valor de la variable ejecuta el **default**, ***si este existe***, sino continúa con la ejecución de las **instrucciones** después de la llave }. El **break** en cada caso es opcional, sino existe se ejecutan las instrucciones siguientes hasta encontrar un **break**.

VECTORES Y MATRICES

Los vectores y matrices también conocidos como Arreglos/Arrays, son variables en memoria que pueden almacenar más de un dato del mismo tipo. Estas variables tienen una característica y es que nos referimos a los elementos, almacenados en los vectores o matrices, utilizando un nombre y números enteros que indican la posición de la celda en el vector o la matriz.

En C podemos tener vectores que almacenen datos de tipo:

- Carácter
- Enteros
- Reales
- Punteros
- Estructuras

Propiedades:

1. Los datos individuales se llaman ***elementos***.
2. Todos los elementos tienen que pertenecer al ***mismo tipo de dato***.
3. Todos los datos son almacenados en celdas contiguas en la memoria RAM, y el subíndice del primer dato es el ***cero***.
4. El nombre del ***array*** es una constante que representa la ***dirección*** en memoria que ocupa el primer elemento del vector.

***Declaración de un Vector*:**

Para declarar un vector siga la siguiente estructura:

```
<tipo_de_dato> <nombre>[<tamaño>]
```

Donde el **tamaño** es un número entero que indica cuantas posiciones del tipo **<tipo_de_dato>** tiene el vector **nombre**.

Las posiciones en memoria se numeran desde **[0]** hasta **[tamaño-1]**.

Inicialización de un Vector:

El lenguaje permite darle valores a las celdas de un vector de la siguiente manera:

1. Por omisión, cuando son creados. El estándar ANSI de C especifica que siempre que declaremos un vector (arreglo), este se inicializa con ceros.
2. Explícitamente, al declarar el vector suministrando datos iniciales.

```
<tipo_de_dato> <nombre>[<tamaño>]={<lista_de_valores>};
```

Ejemplo:

```
unsigned char Vector[3]={1,2,3};
```

3. Durante la ejecución del programa.
   Mediante la ejecución de un ciclo **while** o **for,** el programa almacena los elementos en el vector. Ejemplo:

```
unsigned char TABLA[10];

for (i=0;i<10;i++){
  TABLA[i]=(i*i)+2;
}
```

***Declaración de una Matriz*:**

El lenguaje C nos permite declarar *Arrays* de forma de poder tratarlos como matrices: Para declarar una *matriz* siga la siguiente estructura:

```
<tipo_de_dato> <nombre>[<números_filas>][<números_columnas>]
```

De esta manera queda reservada en memoria una matriz de:
[<números_filas>] x **[<números_columnas>]** x **<tipo_de_dato>**

Las posiciones en memoria se numeran desde **[0][0]** hasta **>[<filas-1>][<columnas-1>]**

Inicialización de una Matriz:

El lenguaje permite darle valores a las celdas de un vector de la siguiente manera:

1. Por omisión, cuando son creadas. El estándar ANSI de C especifica que siempre que declaremos una *matriz*, esta se inicializa con ceros.
2. Explícitamente, al declarar la matriz suministrando datos iniciales.

```
<tipo_de_dato> <nombre>[<filas>][<columnas>]={
                                        {<valores_fila_0>},
                                        {<valores_fila_1>},
                                              ...
                                      {<valores_fila_n-1>}
                                            };
```

Ejemplo:

```
unsigned char Matriz[3][3]={
                              {1,2,3},
                              {4,5,6},
                              {7,8,9}
                            };
```

3. Durante la ejecución del programa.
   Mediante la ejecución de un ciclo **while** o **for,** el programa almacena los elementos en la matriz. Ejemplo:

```
unsigned char MATRIZ[10][10];

for (i=0;i<10;i++){
  for (j=0;j<10;j++){
    MATRIZ[i][j]=(i+j)*2;
  }
}
```

***Cadena de caracteres*:**

Para almacenar en la memoria una cadena de caracteres, es necesario en el lenguaje C, declarar un Array (vector) de caracteres.

Cuando se almacena una cadena de caracteres, se guarda después del último carácter, el carácter **NULL**; que se especifica '\0'. Por esta razón para declarar arrays de caracteres es necesario que el número de posiciones que se declaren sea de uno más que la cadena más larga que pueda contener. Esto significa que, si se desea almacenar una frase como **Caracter**, necesitamos declarar un Array de caracteres de 9 elementos.

```
unsigned char FRASE[9]= "Caracter";
```

En memoria la variable FRASE almacena el dato así:

Posición	FRASE								
Elemento	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
Valor	'C'	'a'	'r'	'a'	'c'	't'	'e'	'r'	0

Inicialización de Cadena de caracteres:

1. Explícitamente, al declarar el vector suministrando datos iniciales.

```
unsigned char <nombre>[<tamaño>]= "<Cadena_de_caracteres>";

unsigned char <nombre>[<tamaño>]= {lista_de_caracteres};
```

Ejemplos:
De esta manera se añade automáticamente el carácter NULL al final de la cadena.

```
unsigned char PALABRA[7]= "Cadena";
```

De esta manera se determina automáticamente el tamaño de la cadena.

```
unsigned char PALABRA[]= "Cadena";
```

De esta manera hay que que definir el tamaño y asignar el carácter NULL en la última posición:

```
unsigned char PALABRA[7]= {'C','a','d','e','n','a','\0';
```

2. Durante la ejecución del programa.
Mediante la ejecución de un ciclo **while** o **for,** el programa almacena los elementos en el Array. Ejemplo:

```
unsigned char FRASE[<tamaño>];

for (i=0;i<(tamaño-1);i++){
  FRASE[i]=<Caracter>;
}
FRASE[(tamaño-1)]= '\0';
```

NOTA:
En el lenguaje C está prohibido realizar una asignación de la siguiente manera:

```
unsigned char FRASE[10];

FRASE= "Prohibido";
```

VARIABLES DE TIPO PUNTERO

¿Qué es una variable de tipo puntero?

Una variable de tipo puntero es una variable que almacena la dirección en memoria en que inicia una variable.

a
0x80

DATOS	
POSICIÓN	CONTENIDO
0x80	'C'
0x81	'A'
0x82	'R'
0x83	'L'
0x84	'O'
0x85	'S'
0x86	'\0'

En el gráfico anterior la variable **a** es una variable de *tipo puntero*, pues almacena el número 0x80 que es la dirección en memoria donde comienza la ***variable datos***, que es un vector de caracteres con 7 posiciones.

¿Cómo se declara una variable de tipo puntero?

Para declarar una variable de tipo puntero proceda de la siguiente forma:

```
<modificador><tipo_de_dato> *Puntero;
```

Ejemplo:

```
unsigned char *Puntero;
```

La variable Puntero es una variable puntero a variable **unsigned char**, en otras palabras, Puntero puede guardar la dirección en memoria en que empiezan variables de tipo **unsigned char**.

¿Cómo se le da valores a una variable de tipo puntero?

Para almacenar en una variable de tipo puntero la dirección en que está una variable en memoria

debemos utilizar el operador **&** (operador de dirección)

El operador **&** lo utilizamos en los programas, cuando deseamos almacenar en una variable de tipo puntero la dirección en que se inicia una variable en memoria.

```
unsigned char Variable;
unsigned char *Puntero;

Puntero=&Variable;
```

*El operador *, operador de indirecto*

Dicho operador cuando se escribe a la izquierda de una variable de tipo puntero, permite acceder a la variable a la que apunta el puntero y de esta manera poder leer o escribir el valor de la variable apuntada.

Ejemplo:

```
unsigned char Variable_1,Variable_2;
unsigned char *Puntero;

Puntero=&Variable_1;
Variable_2=*Puntero;
*Puntero=2;
```

La variable `Puntero` guarda la dirección de la posición en memoria reservada para la variable `Variable_1`. Se puede decir que `Puntero` apunta a `Variable_1`.

```
Puntero=&Variable_1;
```

La variable `Variable_2` guarda el valor que tiene la variable `Variable_1`.

```
Variable_2=*Puntero;
```

La variable `Variable_1` ahora guarda el valor `0` .

```
*Puntero=2;
```

Variable Puntero para acceder a un Vector

Para entenderlo vamos a usar un ejmplo:

```
unsigned char VECTOR[5]={1,3,4,5,6};
unsigned char Valor;
unsigned char *Puntero;

Puntero=&VECTOR[0];
for (i=0;i<5;i++){
  Valor=*Puntero;
  Puntero++;
}
```

La variable `Puntero` guarda la dirección de la posición en memoria reservada para el elemento `[0]` de la variable `VECTOR`, que como vimos anteriormente es un Array de 5 elementos.

```
Puntero=&Variable_1;
```

Por cada pasada del **`for`** la variable `Valor` guarda el valor que tiene la variable apuntada por `Puntero.` Como vimos en el paso anterior, la variable `Puntero` se inicializo con la dirección de `VECTOR[0]`.

Como `Puntero` guarda el valor de la dirección del elemento `[0]`, cada vez que ejecutamos la instrucción `Puntero++` el valor de `Puntero` se incrementa en 1, haciendo que esta apunte al elemento siguiente de Array `VECTOR`.

```
for (i=0;i<5;i++){
  Valor=*Puntero;
  Puntero++;
}
```

La variable `Variable_1` ahora guarda el valor `0` .

```
*Puntero=2;
```

Variable Puntero para acceder a una Cadena de Caracteres

Como vimos, una cadena de caracteres no es más que un Array/Vector donde se almacenan los caracteres, pero con la particularidad que la posición última almacena el valor NULL.
En el siguiente ejemplo, usare un puntero que apunte a una cadena de caracteres para determinar el largo de la cadena almacenada.

```
unsigned char PALABRA[]= "PUNTERO";
unsigned char largo=0;
unsigned char *Puntero;

Puntero=PALABRA;      // Puntero=&PALABRA[0]
while (*Puntero!='\0'){
  largo++;
  Puntero++;
}
```

Veamos en detalle el ejemplo anterior.
La variable `Puntero` guarda la dirección de la posición en memoria donde comienza la cadena de caracteres `PALABRA` . Como vemos en el ejemplo para pasarle la dirección de la cadena no usamos el operador `&` esto se debe a que `PALABRA` almacena la dirección donde comienza la cadena de caracteres, es por esto que es análogo a hacer `Puntero=&PALABRA[0]`

```
Puntero=PALABRA;      // Puntero=&PALABRA[0]
```

Luego el bucle **`while`** , recorre la cadena de caracteres buscando al carácter NULL

```
while (*Puntero!='\0'){
```

Inicialización de un Puntero a Caracteres

Una manera más versátil de manejar una cadena de caracteres en memoria es por medio de la declaración de la forma siguiente:

```
unsigned char *Puntero= "PUNTERO";
```

Este tipo de variable se dice que es un puntero a cadena de caracteres.
Se crea un espacio en memoria de 8 bytes y guarda allí la frase **`PUNTERO`**, determina en que dirección queda almacenado el primer carácter de la frase, luego crea la variable
`*Puntero` y guarda en esa variable la dirección donde quedo almacenado el primer carácter.

Uso de punteros en los parámetros de un función

Cuando en una función se utiliza un puntero como parámetro, al momento de llamar a la función se copia la dirección de la variable que se utiliza en la llamada a la función en el

parámetro que es un puntero. Dentro de la función se usa la dirección que almacena el parámetro, para acceder al dato que se encuentra en dicha dirección. Esto significa que los cambios hechos a los parámetros afectan a las variables usadas en la llamada de la función.
Ejemplo:

```
unsigned char Suma (unsigned char *Op1, unsigned char *Op2);//Prototipo

void main (void){
  unsigned char Operador_1=4;
  unsigned char Operador_2=8;
  unsigned char Resultado=0;

  Resultado=Suma(&Operador_1,&Operador_2);
}

unsigned char Suma (unsigned char *Op1, unsigned char *Op2){
  return (*Op1+*Op2);
}
```

También se puede usar con cadenas de caracteres, usaremos el mismo ejemplo donde contamos la cantidad de caracteres que tenía una cadena.

```
unsigned char Largo (unsigned char *Caracteres);//Prototipo

void main (void){
  unsigned char PALABRA[]= "PUNTERO";
  unsigned char Cantidad=0;

  Cantidad=Largo(PALABRA); //es igual que Cantidad=Largo(&PALABRA[0]);
}

unsigned char Largo (unsigned char *Caracteres){
  unsigned char largo=0;

  while (*Caracteres!='\0') largo++;

  return (largo);
}
```

Trabajando con punteros vimos que teniamos la posibilidad de hacer una asignacion del siguiente tipo:

```
unsigned char *Puntero= "PUNTERO";
```

Teniendo en cuenta esto, podes realizar el ejemplo anterior de la siguiente manera:

```
unsigned char Largo (unsigned char *Caracteres);//Prototipo

void main (void){
  unsigned char Cantidad=0;

  Cantidad=Largo("PUNTERO");
}

unsigned char Largo (unsigned char *Caracteres){
  unsigned char largo=0;

  while (*Caracteres!='\0') largo++;

  return (largo);
}
```

ESTRUCTURAS DE DATOS

En C una estructura es una colección de variables que se referencian bajo el mismo nombre. Una estructura proporciona un medio conveniente para mantener junta información relacionada.
Al crear o definir una estructura se forma una plantilla que puede usar el programador para definir una variable de tipo estructura. Las variables que conforman la estructura son llamadas **campos** de la estructura.

Declaración de una Estructura

```
struct NombreEstructura {
   <modificador_1><tipo_de_dato_1> campo_1;
   <modificador_2><tipo_de_dato_2> campo_2;
      ...
   <modificador_n><tipo_de_dato_n> campo_n;
};
```

La declaración anterior se usa en C cuando se quiere definir una estructura, donde `NombreEstructura` es el identificador de la estructura, `struct` es la palabra que indica que se está iniciando la declaración de una estructura. La declaración de una estructura finaliza siempre con ;.
Ejemplo:

```
struct Persona {
   unsigned char NOMBRE[20];
   unsigned char APELLIDO[20];
   unsigned char DNI[8];
};
```

Definición de variables de tipo estructura

```
struct NombreEstructura <Variable1>;
```

Como la estructura es un tipo de dato, podemos usar el operador coma para crear mas de una variable del tipo estructura.
Las variables de este tipo de estructura, van a ocupar en RAM 48 Bytes, es decir la suma de los espacios que ocupa cada uno de los **campos** de la estructura.
Otra manera de declarar variables de un tipo estructura es hacerlo al momento de declarar la estructura, veamos un ejemplo:

```
struct Persona {
   unsigned char NOMBRE[20];
   unsigned char APELLIDO[20];
   unsigned char DNI[8];
}Ezequiel,Ignacio;
```

Como se entran datos a un campo de una variable estructura

Para acceder a un **campo** de una variables de tipo estructura, usamos el operador punto (.)

```
    <NombreVariableEstructura>.<CampoEstructura>=<valor>
```

Lo que vayamos a guardar en el campo debe ser del tipo del campo.

Campo de bits

Un campo de bit es un elemento de una estructura definida en términos de bits.

Antes de ver un ejemplo del uso de struct para crear estructuras de campos de bits consideremos el caso en donde se tiene una variable del tipo char (8 bits) y que para la misma se desea que los bits tengan significados específicos. Estructura de bits:

```
struct NombreEstructura {
   <modificador_1><tipo_de_dato_1> campo_1 :x;
   <modificador_2><tipo_de_dato_2> campo_2 :y;
   <modificador_3><tipo_de_dato_3> campo_3 :z;
};
```

Donde `:x; :y; :z;` son el número de bits requeridos para el campo `campo_1 campo_2 campo_3` respectivamente.

Veamos un ejemplo, digamos que el primer bit servirá para controlar alguna condición; los siguientes cuatro bits, o sea del segundo al quinto bit, controlarán otra condición; el bit 6 tendrá otra función; y el séptimo se empleará para contralar otra condición.

```
struct campo_de_bits {
   unsigned char bit_1      :1;
   unsigned char bits_2_5   :4;
   unsigned char bit_6      :1;
   unsigned char bit_7      :1;
};
```

Corresponde a la siguiente colección de bits

campo_de_bits							
7	6	5	4	3	2	1	0
bit_1	bit_1	bits_2_5					bit_1

Como escribir o leer un bit del campo de bits

En el siguiente ejemplo, vamos a crear una variable `Bit_Var` de tipo **`struct`** `campo_de_bits`.

```
struct campo_de_bits Bit_Var;
```

Para leer o escribir el bit identificado como `bit_1` podemos usar las siguientes instrucciones:

```
Bit_Var.bit_1=1;              //Escribir

Bit_Var.bit_7=Bit_Var.bit_6; //Leer
```

Union

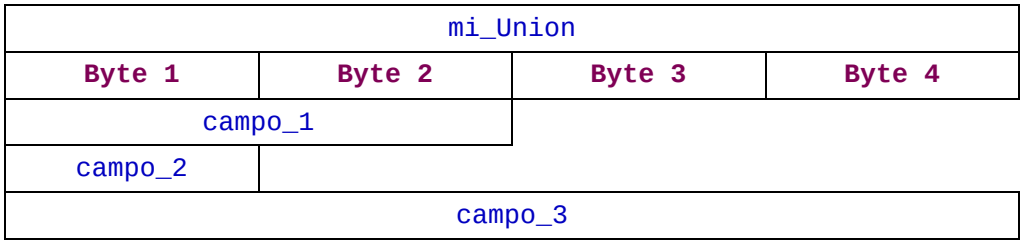
De la misma manera que con la orden **`struct`**, con la orden **`union`** se pueden crear estructuras con nombre y estructuras sin nombre.El mecanismo de acceso a los miembros de una **`union`** es igual al mecanismo de acceso a los miembros de una **`struct`**.
Los miembros de una **`union`** comparten un espacio de almacenamiento común.
En una **`union`**, el compilador reserva el espacio de almacenamiento para la misma de acuerdo con el tipo de la variable de mayor tamaño.

```
union NombreUnion {
   <modificador_1><tipo_de_dato_1> campo_1;
   <modificador_2><tipo_de_dato_2> campo_2;
      ...
   <modificador_n><tipo_de_dato_n> campo_n;
};
```

Ejemplo:

```
union NombreUnion {
   unsigned int  campo_1;
   unsigned char campo_2;
   unsigned long campo_3;
} mi_Union;
```

El siguiente esquema demuestra cómo se reserva la memoria en una **union**:



***Comparación entre* union *y* struct**

```
union NombreUnion {
   unsigned int  campo1;
   unsigned char campo2;
   unsigned long campo3;
} mi_Union;
```

```
struct NombreUnion {
   unsigned int  campo_1;
   unsigned char campo_2;
   unsigned long campo_3;
} mi_Struct;
```