

SISTEMAS EMPOTRADOS

Apunte 1- GUÍA RÁPIDA PARA PROGRAMACIÓN EN LENGUAJE C

- Profesor: Carlos Diego Moreno Moreno

Dpto. de Ingeniería Electrónica y de Computadores.
Área de Arquitectura y Tecnología de Computadores.
Escuela Politécnica Superior. Universidad de Córdoba.

Objetivos.

- El presente documento tiene como objetivo hacer un resumen sobre el lenguaje C, considerando su sintaxis, estructura y uso de sus instrucciones para utilizar en sistemas empotrados.



Índice.

- 1.1.– Introducción.
- 1.2.– Estructura de un programa en C.
 - 1.2.1.– Instrucciones para el compilador.
 - 1.2.2.– Definición de estructuras.
 - 1.2.3.– Definición de variables.
 - 1.2.4.– Funciones.
 - 1.2.5.– Programa principal.
- 1.3.– Tipos.
 - 1.3.1.– Tipos.
 - 1.3.2.– Estructuras.
 - 1.3.3.– Uniones.
 - 1.3.4.– Enumeraciones.
 - 1.3.5.– Modificadores de almacenamiento.

1.4.- Operadores.

- 1.4.1.- Operadores aritméticos.
- 1.4.2.- Operadores de relación y lógicos.
- 1.4.3.- Operadores de incremento y decremento.
- 1.4.4.- Operadores para manejo de *bits*.
- 1.4.5.- Operadores de asignación y expresiones.
- 1.4.6.- Precedencia y asociatividad de operadores.
- 1.4.7.- Conversión de tipos.

1.5.- Control de flujo.

1.6.- Funciones.

1.7.- Punteros (o apunadores).

1.8.- El preprocesador de C.

1.9.- Otros aspectos del lenguaje C.

- 1.9.1.- *Typedef*.
- 1.9.2.- Funciones recursivas.
- 1.9.3.- Gestión dinámica de la memoria.

1.10.- Las librerías del lenguaje C.



Índice.

- 1.1.– Introducción.
- 1.2.– Estructura de un programa en C.
- 1.3.– Tipos.
- 1.4.– Operadores.
- 1.5.– Control de flujo.
- 1.6.– Funciones.
- 1.7.– Punteros (o apuntadores).
- 1.8.– El preprocessador de C.
- 1.9.– Otros aspectos del lenguaje C.
- 1.10.– Las librerías del lenguaje C.

1.1.– Introducción.

- El lenguaje C, es un lenguaje de programación creado en 1972 por Ken Thompson y Dennis M. Ritchie en los Laboratorios Bell como evolución del anterior lenguaje B, a su vez basado en BCPL. Al igual que B, es un lenguaje orientado a la implementación de Sistemas Operativos, concretamente Unix.
- C es apreciado por la eficiencia del código que produce y es el lenguaje de programación más popular para crear *software* de sistemas, aunque también se utiliza para crear aplicaciones y para la programación de microcontroladores de todo tipo.
- Se trata de un lenguaje débilmente tipificado de medio nivel pero con muchas características de bajo nivel.

1.1.– Introducción.

- ❖ Dispone de las estructuras típicas de los lenguajes de alto nivel pero, a su vez, dispone de construcciones del lenguaje que permiten un control a muy bajo nivel. Los compiladores suelen ofrecer extensiones al lenguaje que posibilitan mezclar código en ensamblador con código C o acceder directamente a memoria o dispositivos periféricos.
- ❖ 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. En la práctica, los programadores suelen usar elementos no portables dependientes del compilador o del sistema operativo.

Índice.

- 1.1.– Introducción.
- 1.2.– Estructura de un programa en C.**
- 1.3.– Tipos.
- 1.4.– Operadores.
- 1.5.– Control de flujo.
- 1.6.– Funciones.
- 1.7.– Punteros (o apunadores).
- 1.8.– El preprocessador de C.
- 1.9.– Otros aspectos del lenguaje C.
- 1.10.– Las librerías del lenguaje C.

1.2.– Estructura de un programa en C.

💡 Un programa en C, consta de varias secciones en donde se determinarán qué variables y funciones tendrá el programa, así como la tarea que tendrá que realizar. Su estructura está determinada por las partes siguientes:

- 1.2.1.– Instrucciones para el compilador.
- 1.2.2.– Definición de estructuras.
- 1.2.3.– Definición de variables.
- 1.2.4.– Funciones.
- 1.2.5.– Programa principal.

Índice.

- 1.1.– Introducción.
- 1.2.– Estructura de un programa en C.
 - 1.2.1.– Instrucciones para el compilador.**
- 1.3.– Tipos.
- 1.4.– Operadores.
- 1.5.– Control de flujo.
- 1.6.– Funciones.
- 1.7.– Punteros (o apunadores).
- 1.8.– El preprocessador de C.
- 1.9.– Otros aspectos del lenguaje C.
- 1.10.– Las librerías del lenguaje C.

1.2.1.– Instrucciones para el compilador.

- Las líneas que se escriben en esta sección indican al compilador la realización de diferentes tareas como: la definición de nombres, tomas de decisión durante la compilación, macros e inclusión de archivos entre otras.
- Todas las líneas del preprocesador comienzan con el carácter '#'. Por ejemplo:

```
#include <stdio.h>

#define PI 3.1416 /* Donde PI es el nombre de la etiqueta y su valor es 3.1416 */
```

- Donde todos los caracteres que se encuentren entre "/*" y "*/", son tomados como comentarios y no tienen efecto para la compilación.

Índice.

- 1.1.– Introducción.
- 1.2.– Estructura de un programa en C.
 - 1.2.2.– Definición de estructuras.**
- 1.3.– Tipos.
- 1.4.– Operadores.
- 1.5.– Control de flujo.
- 1.6.– Funciones.
- 1.7.– Punteros (o apunadores).
- 1.8.– El preprocessador de C.
- 1.9.– Otros aspectos del lenguaje C.
- 1.10.– Las librerías del lenguaje C.

1.2.2.– Definición de estructuras.

- En este área se definen todas las estructuras que requiera el programa. Donde una estructura es la colección de variables para la representación del problema que se está tratando. Por ejemplo:

```
struct esfera
{
    float radio;
    float volumen;
};
```

Índice.

- 1.1.– Introducción.
- 1.2.– Estructura de un programa en C.
 - 1.2.3.– Definición de variables.**
- 1.3.– Tipos.
- 1.4.– Operadores.
- 1.5.– Control de flujo.
- 1.6.– Funciones.
- 1.7.– Punteros (o apunadores).
- 1.8.– El preprocessador de C.
- 1.9.– Otros aspectos del lenguaje C.
- 1.10.– Las librerías del lenguaje C.

1.2.3.– Definición de variables.

- ❖ Una vez creados los tipos y estructuras, se procede a la definición de las variables, que son etiquetas con las cuales se tiene acceso a posiciones de memoria para el almacenamiento de información durante la ejecución del programa.
- ❖ Un ejemplo de definición de variable es:

```
struct esfera A;
```

Índice.

- 1.1.– Introducción.
- 1.2.– Estructura de un programa en C.
 - 1.2.4.– Funciones.**
 - 1.3.– Tipos.
 - 1.4.– Operadores.
 - 1.5.– Control de flujo.
 - 1.6.– Funciones.
 - 1.7.– Punteros (o apunadores).
 - 1.8.– El preprocessador de C.
 - 1.9.– Otros aspectos del lenguaje C.
 - 1.10.– Las librerías del lenguaje C.

1.2.4.– Funciones.

- Las funciones son conjuntos de instrucciones encargadas de la realización de tareas específicas. Su existencia permite la representación del problema en función del modelo de programación estructurada.
- Una función que calcula el volumen de una esfera se presenta de la siguiente manera:

```
void volum(void)  
{  
    A.volumen= (4.0 / 3.0) * PI * A.radio * A.radio * A.radio;  
}
```

Índice.

- 1.1.– Introducción.
- 1.2.– Estructura de un programa en C.
 - 1.2.5.– Programa principal.**
- 1.3.– Tipos.
- 1.4.– Operadores.
- 1.5.– Control de flujo.
- 1.6.– Funciones.
- 1.7.– Punteros (o apunadores).
- 1.8.– El preprocessador de C.
- 1.9.– Otros aspectos del lenguaje C.
- 1.10.– Las librerías del lenguaje C.

1.2.5.– Programa principal.

- El programa principal contiene las instrucciones y llamadas a funciones que se ejecutan en primera instancia. Siempre se utiliza el nombre de "main". El programa principal para el cálculo del volumen de una esfera puede ser el siguiente:

```
void main(void)
{
    printf("\n\t Deme el radio de la esfera: ");
    scanf("%f", &A.radio);
    volum();
    printf("\n\t El volumen de la esfera con radio %f, es: %f\n", A.radio, A.volumen);
}
```

- Observación: Es muy recomendable usar tabuladores para la claridad del programa.

Índice.

- 1.1.– Introducción.
- 1.2.– Estructura de un programa en C.
- 1.3.– Tipos.**
 - 1.3.1.– Tipos.**
- 1.4.– Operadores.
- 1.5.– Control de flujo.
- 1.6.– Funciones.
- 1.7.– Punteros (o apunadores).
- 1.8.– El preprocessador de C.
- 1.9.– Otros aspectos del lenguaje C.
- 1.10.– Las librerías del lenguaje C.

1.3.1.– Tipos.

Las variables usadas en lenguaje C, pueden ser de varios tipos. Éstos, definirán el tamaño en *bytes* de cada variable. Los tipos existentes son los siguientes:

void	Sin valor
char	Carácter
short	Entero corto
int	Entero
long	Entero largo
float	Flotante
double	Flotante de doble precisión
signed	Usa signo
unsigned	Sin signo

1.3.1.– Tipos.

- El tipo `void` permite declarar funciones que no devuelven valores y también para la declaración de punteros (o apuntadores) genéricos, es decir capaces de apuntar a cualquier tipo de variable.
- La declaración de una variable emplea la sintaxis siguiente:

```
Tipo nombre_variable, nombre_variable, nombre_variable = constante;
```

- En la declaración de variables se pueden definir una o más variables del mismo tipo en el mismo renglón, por ejemplo:

```
int numero, altura, profundidad;
```

- Todas las líneas terminan con punto y coma (;).

1.3.1.– Tipos.

❖ También se les puede asignar un valor inicial en el momento de la definición. Por ejemplo:

```
char letra_inicial = 'J';  
float elevacion = 2.43, x = 4.0;
```

❖ Es importante mencionar que en el nombre de la variable se pueden emplear letras minúsculas, mayúsculas y el carácter '_' (línea de subrayado).

❖ También se pueden definir *arrays* unidimensionales y multidimensionales.

```
char nombre[30];  
unsigned char meses[2][12] =  
{  
    {31,28,30,31,30,31,30,31,31,30,31,31},  
    {31,28,30,31,30,31,30,31,31,30,31}  
};
```

1.3.1.– Tipos.

💡 Las cadenas de caracteres se representan usando doble comilla, por ejemplo: "Es un día soleado", mientras que para un solo carácter se emplea la comilla, por ejemplo: 'a', 'b', 'z'. Las cadenas de caracteres en C, se terminan con el carácter especial '\0'. Otros caracteres especiales (también se les denomina secuencias de escape) son:

- 💡 '\a' carácter de alarma
- 💡 '\b' retroceso
- 💡 '\f' avance de hoja
- 💡 '\n' nueva línea
- 💡 '\r' retorno de carro
- 💡 '\t' tabulador horizontal
- 💡 '\v' tabulador vertical

- 💡 '\\' diagonal invertida
- 💡 '\?' interrogación
- 💡 '\'' apóstrofo
- 💡 '\"' comillas
- 💡 '\ooo' número octal
- 💡 '\xhh' número hexadecimal
- 💡 '\0' carácter nulo

Índice.

- 1.1.– Introducción.
- 1.2.– Estructura de un programa en C.
- 1.3.– Tipos.
 - 1.3.2.– Estructuras.**
- 1.4.– Operadores.
- 1.5.– Control de flujo.
- 1.6.– Funciones.
- 1.7.– Punteros (o apunadores).
- 1.8.– El preprocesador de C.
- 1.9.– Otros aspectos del lenguaje C.
- 1.10.– Las librerías del lenguaje C.

1.3.2.– Estructuras.

- ❖ Una estructura es una colección de variables que se agrupan bajo una sola referencia.
- ❖ La sintaxis general es la siguiente:

```
struct nombre_estructura {  
    elemento 1;  
    elemento 2;  
    .  
    .  
    .  
} variable_struct;
```

- ❖ Donde la palabra reservada es "struct", el nombre del conjunto de variables se pone en "nombre_estructura", se abre una llave y enseguida se ponen las variables (cada elemento, usa la declaración de variables mostradas en la sección anterior); al terminar de definir todos los elementos, se cierra la estructura con una llave.

1.3.2.- Estructuras.

- ❖ Hasta aquí, sólo se ha definido la estructura, sin embargo, no hay ninguna instancia (creación física) de la misma. Por tanto, se hace necesario definir una variable cuyo tipo sea la estructura predefinida.
- ❖ Usemos como ejemplo una estructura ya vista:

```
struct esfera
{
    float radio;
    float volumen;
};
struct esfera A;
```

- ❖ Para acceder a un campo de la estructura, se usa el nombre de la variable, seguida por un punto y después por el nombre del campo que se va a usar.

A.radio = 4.0;

A.volumen = 4332.8998

Índice.

- 1.1.– Introducción.
- 1.2.– Estructura de un programa en C.
- 1.3.– Tipos.
 - 1.3.3.– Uniones.**
- 1.4.– Operadores.
- 1.5.– Control de flujo.
- 1.6.– Funciones.
- 1.7.– Punteros (o apunadores).
- 1.8.– El preprocessador de C.
- 1.9.– Otros aspectos del lenguaje C.
- 1.10.– Las librerías del lenguaje C.

1.3.3.– Uniones.

- Existe la posibilidad de usar de manera más eficiente la memoria de la computadora, esto se puede lograr empleando la definición de unión. La sintaxis es similar a la de la estructura.

```
union nombre_de_la_union {  
    elemento 1;  
    elemento 2;  
    ...  
} variable_union;
```

- La diferencia estriba, en que, en este caso las variables se solapan. Por ejemplo:

```
union libros {  
    char inicial;  
    int num;  
} lib;
```

- En este caso la unión emplea 2 *bytes* para la variable "*num*", y uno de esos *bytes* se emplea para la variable "*inicial*", por tanto, no hay 3 *bytes* en la definición, sino sólo dos.

Índice.

- 1.1.– Introducción.
- 1.2.– Estructura de un programa en C.
- 1.3.– Tipos.
 - 1.3.4.– Enumeraciones.**
- 1.4.– Operadores.
- 1.5.– Control de flujo.
- 1.6.– Funciones.
- 1.7.– Punteros (o apunadores).
- 1.8.– El preprocessador de C.
- 1.9.– Otros aspectos del lenguaje C.
- 1.10.– Las librerías del lenguaje C.

1.3.4.– Enumeraciones.

- La enumeración es una lista de valores posibles que puede tomar una variable. Por ejemplo:

```
enum ciudades Córdoba, Sevilla, Madrid, Barcelona;
```

```
enum ciudades ciud;
```

- Donde "ciud" sólo puede tomar cualquiera de los nombres dados en la declaración previa.

Índice.

- 1.1.– Introducción.
- 1.2.– Estructura de un programa en C.
- 1.3.– Tipos.
 - 1.3.5.– Modificadores de almacenamiento.**
- 1.4.– Operadores.
- 1.5.– Control de flujo.
- 1.6.– Funciones.
- 1.7.– Punteros (o apunadores).
- 1.8.– El preprocessador de C.
- 1.9.– Otros aspectos del lenguaje C.
- 1.10.– Las librerías del lenguaje C.

1.3.5.– Modificadores de almacenamiento.

- 💡 Con los modificadores de almacenamiento se altera la forma en que se crea el almacenamiento. Los modificadores son: `extern`, `auto`, `register`, `const`, `volatile` y `static`.
- 🌿 Modificador “`extern`”: Se emplea cuando un programa se encuentra dividido en varios archivos. Como las variables sólo se definen una vez, se hace necesario hacer referencia de ellas en los archivos que así lo requieran. La sintaxis es:

```
extern tipo la_variable; /* Se debe indicar también el tipo de la variable o función */
```

- 🌿 Modificador “`auto`”: Todas las variables son “`auto`” por omisión, y son objetos locales a un bloque y al salir de éste son descartadas.

1.3.5.– Modificadores de almacenamiento.

- Modificador “**register**”: Los objetos declarados como “register” son automáticos y sus valores se almacenan en registros del microprocesador.
- Modificador “**const**”: Las variables de tipo “const” no pueden cambiarse por el programa durante la ejecución. Una variable “const” recibirá su valor desde una inicialización explícita o por medio de alguna dependencia del *hardware*.
- Modificador “**volatile**”: Las variables modificadas por “volatile” indican que éstas pueden ser modificadas de forma no explícita por el programa. Por ejemplo: se podría guardar la hora exacta en una variable y que la asignación la hiciera la rutina de atención a la interrupción del reloj.
- Modificador “**static**”: Se usa para mantener una variable local en existencia durante la ejecución de un programa, de manera que no se tiene que crear y destruir cada vez que se encuentra a la rutina donde está declarada.

Índice.

- 1.1.– Introducción.
- 1.2.– Estructura de un programa en C.
- 1.3.– Tipos.
- 1.4.– Operadores.**
- 1.5.– Control de flujo.
- 1.6.– Funciones.
- 1.7.– Punteros (o apunadores).
- 1.8.– El preprocessador de C.
- 1.9.– Otros aspectos del lenguaje C.
- 1.10.– Las librerías del lenguaje C.

1.4.– Operadores.

Los tipos de operadores que existen son:

- Aritméticos.
- De relación y lógicos.
- De incremento y decremento.
- Para manejo de *bits*.
- De asignación y expresiones.
- Expresiones condicionales.

Índice.

- 1.1.– Introducción.
- 1.2.– Estructura de un programa en C.
- 1.3.– Tipos.
- 1.4.– Operadores.
 - 1.4.1.– Operadores aritméticos.**
- 1.5.– Control de flujo.
- 1.6.– Funciones.
- 1.7.– Punteros (o apunadores).
- 1.8.– El preprocessador de C.
- 1.9.– Otros aspectos del lenguaje C.
- 1.10.– Las librerías del lenguaje C.

1.4.1.– Operadores aritméticos.

💡 Los operadores aritméticos binarios son:

- + **Suma de dos operandos.**
- **Resta entre dos operandos.**
- *
- / **División entre dos operandos.**
- % **Módulo entre dos operandos.** Da el resto de la división resultante entre el operando1 (numerador) y el operando2 (denominador).

💡 La sintaxis es: operando1 operador operando2

💡 La división entera trunca cualquier parte fraccionaria.



Índice.

- 1.1.– Introducción.
- 1.2.– Estructura de un programa en C.
- 1.3.– Tipos.
- 1.4.– Operadores.
 - 1.4.2.– Operadores de relación y lógicos.**
- 1.5.– Control de flujo.
- 1.6.– Funciones.
- 1.7.– Punteros (o apunadores).
- 1.8.– El preprocessador de C.
- 1.9.– Otros aspectos del lenguaje C.
- 1.10.– Las librerías del lenguaje C.

1.4.2.– Operadores de relación y lógicos.

💡 Los operadores de relación son:

>	Mayor que
>=	Mayor o igual
<	Menor que
<=	Menor o igual
==	Idéntico
!=	Diferente a

💡 Los operadores lógicos son:

&&	Y (AND)
	O (OR)

💡 El resultado del uso de estos operadores puede ser falso o verdadero. Se considera como falso el valor de cero, y como verdadero, todo aquel valor diferente de cero.

💡 La sintaxis es: operando1 operador operando2

Índice.

- 1.1.– Introducción.
- 1.2.– Estructura de un programa en C.
- 1.3.– Tipos.
- 1.4.– Operadores.
 - 1.4.3.– Operadores de incremento y decremento.**
- 1.5.– Control de flujo.
- 1.6.– Funciones.
- 1.7.– Punteros (o apunadores).
- 1.8.– El preprocessador de C.
- 1.9.– Otros aspectos del lenguaje C.
- 1.10.– Las librerías del lenguaje C.

1.4.3.– Operadores de incremento y decremento.

💡 Los operadores de incremento y decremento, añaden o quitan una unidad al operando que están afectando, respectivamente. De tal manera que:

`++i;` es equivalente a: `i=i+1;` y después se emplea el valor de `i`

`i++;` es equivalente a: se usa el valor de `i`, a continuación se incrementa

`--i;` significa: `i=i-1;` y después se usa el valor de `i`

`i--;` significa: se usa el valor de `i`, a continuación se decrementa en uno

💡 Como nota importante se debe poner énfasis en la precedencia de cada expresión.



Índice.

- 1.1.– Introducción.
- 1.2.– Estructura de un programa en C.
- 1.3.– Tipos.
- 1.4.– Operadores.
 - 1.4.4.– Operadores para manejo de *bits*.**
- 1.5.– Control de flujo.
- 1.6.– Funciones.
- 1.7.– Punteros (o apunadores).
- 1.8.– El preprocessador de C.
- 1.9.– Otros aspectos del lenguaje C.
- 1.10.– Las librerías del lenguaje C.

1.4.4.– Operadores para manejo de bits.

&	AND de bits.	Ejemplo : 0x0F & 0xFE => 0x0E
	OR inclusivo de bits.	Ejemplo: 0x1E 0x01 => 0x1F
^	OR exclusivo de bits.	Ejemplo: 0xF0 ^ 0x02 => 0xF2
<<	Desplazamiento a la izquierda.	Ejemplo: 0x02 << 2 => 0x08
>>	Desplazamiento a la derecha.	Ejemplo: 0x04 >> 2 => 0x01
~	Complemento a uno.	Ejemplo: ~0xF0 => 0x0F

Índice.

- 1.1.– Introducción.
- 1.2.– Estructura de un programa en C.
- 1.3.– Tipos.
- 1.4.– Operadores.
 - 1.4.5.– Operadores de asignación y expresiones.**
- 1.5.– Control de flujo.
- 1.6.– Funciones.
- 1.7.– Punteros (o apunadores).
- 1.8.– El preprocessador de C.
- 1.9.– Otros aspectos del lenguaje C.
- 1.10.– Las librerías del lenguaje C.

1.4.5.– Operadores de asignación y expresiones.

- El operador de asignación '=' se usa como se muestra a continuación:

$x = 5 + 4;$

- Y puede interpretarse como: el resultado de sumar 5 + 4 se asigna a la variable x.
- También se puede usar para la suma de dos variables y que el resultado quede en una tercera. Por ejemplo:

$x = y + z;$

- Supongamos,

$x = 4;$

- Cuando se tiene $x = x + 5$, primero se suma a x el 5 que implica que se obtenga el número 9, y después se asigna a x. Por tanto el valor final de x es idéntico a 9 ($x == 9$).

1.4.5.– Operadores de asignación y expresiones.

- Otra forma de expresar lo anterior es:

$x += 5;$

que es equivalente a:

$x = x + 5$

- De esta manera " $+=$ " es un operador de asignación.
- En general se puede decir que:

expresión1 op= expresión2

equivale a

expresión1 = (expresión1) op (expresión2)

Índice.

- 1.1.– Introducción.
- 1.2.– Estructura de un programa en C.
- 1.3.– Tipos.
- 1.4.– Operadores.
 - 1.4.6.– Precedencia y asociatividad de operadores.**
- 1.5.– Control de flujo.
- 1.6.– Funciones.
- 1.7.– Punteros (o apunadores).
- 1.8.– El preprocessador de C.
- 1.9.– Otros aspectos del lenguaje C.
- 1.10.– Las librerías del lenguaje C.

1.4.6.– Precedencia y asociatividad de operadores.

- 💡 La precedencia y asociatividad de los operadores se muestra en la tabla siguiente.
- 💡 Téngase en cuenta que los operadores que están en la misma línea tienen la misma precedencia.
- 💡 Los renglones están en orden de precedencia decreciente, por ejemplo, '*', '/', y '%' todos tienen la misma precedencia, la cual es más alta que la de '+' y '-' binarios.
- 💡 El "operador" () se refiere a la llamada a una función.

1.4.6.– Precedencia y asociatividad de operadores.

Operadores	Asociatividad
() []	Izquierda a derecha
! ~ ++ -- + - * & (tipo) sizeof	Derecha a izquierda
* / %	Izquierda a derecha
+ -	Izquierda a derecha
<< >>	Izquierda a derecha
< <= > >=	Izquierda a derecha
== !=	Izquierda a derecha
&	Izquierda a derecha
^	Izquierda a derecha
	Izquierda a derecha
&&	Izquierda a derecha
	Izquierda a derecha
?:	Derecha a izquierda
= += -= *= /= %= &= ^= = <=>=	Derecha a izquierda
,	Izquierda a derecha

Índice.

- 1.1.– Introducción.
- 1.2.– Estructura de un programa en C.
- 1.3.– Tipos.
- 1.4.– Operadores.
 - 1.4.7.– Conversión de tipos.**
- 1.5.– Control de flujo.
- 1.6.– Funciones.
- 1.7.– Punteros (o apunadores).
- 1.8.– El preprocessador de C.
- 1.9.– Otros aspectos del lenguaje C.
- 1.10.– Las librerías del lenguaje C.

1.4.7.– Conversión de tipos.

- ❖ Cuando un operador tiene operandos de tipos diferentes, éstos se convierten a un tipo común de acuerdo con un reducido número de reglas. Estas reglas son:
 - › Si cualquier operando es `long double`, conviértase el otro a `long double`.
 - › De otra manera, si cualquier operando es `double`, conviértase el otro a `double`.
 - › De otra manera, si cualquier operando es `float`, conviértase el otro a `float`.
 - › De otra manera, conviértase `char` y `short` a `int`.
 - › Despues, si cualquier operando es `long`, conviértase el otro a `long`.
- ❖ También se puede convertir de un tipo a otro usando el operador "cast". Que consiste en anteponer a la variable o función el tipo que se desea. Para ello se usan paréntesis. Por ejemplo:

```
char caracter;  
int entero;  
entero = (int) caracter;
```



Índice.

- 1.1.– Introducción.
- 1.2.– Estructura de un programa en C.
- 1.3.– Tipos.
- 1.4.– Operadores.
- 1.5.– Control de flujo.**
- 1.6.– Funciones.
- 1.7.– Punteros (o apunadores).
- 1.8.– El preprocessador de C.
- 1.9.– Otros aspectos del lenguaje C.
- 1.10.– Las librerías del lenguaje C.

1.5.– Control de flujo.

- ❖ Mediante las proposiciones de control de flujo se indica el orden en que se realizará el proceso.
- ❖ Las expresiones vistas anteriormente, se vuelven proposiciones cuando se terminan con punto y coma ';'.
- ❖ Las llaves '{', '}' se emplean para agrupar declaraciones y proposiciones dentro de una proposición compuesta o bloque.
- ❖ Para el control del programa existen las proposiciones siguientes: **if-else**, **else-if**, **switch**, **while**, **for** y **do-while**.

1.5.– Control de flujo: If – else

- La proposición if-else se usa para expresar decisiones. La sintaxis es:

```
if(expresión)
    proposición1;
else
    proposición2;
```

- Si la expresión es verdadera, se ejecuta la proposición1, de lo contrario se realiza la proposición2.

- No es indispensable el else, es decir, se puede tener una proposición de la forma:

```
if (expresión)
    proposición1;
```

- Se debe hacer notar que la proposición1 o la proposición2, pueden ser proposiciones compuestas o bloques. Por ejemplo:

```
if( x == 1 )
{
    y = r*j;
    printf("El resultado de y es: %u", y);
}
else
{
    printf("Se tiene un valor incorrecto de x");
    printf("Vuelva a repetir los pasos 1 al 8");
}
```

1.5.– Control de flujo: if – else. Expresiones condicionales

💡 Otra forma de escribir la proposición if-else es usando una expresión condicional.

```
expresión1 ? expresión2 : expresión 3
```

💡 que es equivalente a:

```
if (expresión1 )  
    expresión2;  
else  
    expresión3;
```

💡 Si las expresiones 2 y 3 son compuestas, cada proposición debe estar separada por coma ','.

Por ejemplo:

```
(A.radio > 4) ?  
    printf("\n Funciona\n"), printf("A ver??\n")  
: printf("\n Es más pequeño");
```

1.5.– Control de flujo: `else` – `if`

💡 Con esta expresión se complementa la expresión vista en el punto anterior. Su sintaxis es:

```
if( expresión )
    proposición;
else if( expresión )
    proposición;
else if( expresión )
    proposición;
else if( expresión )
    proposición;
else
    proposición;
```

💡 Mediante esta expresión se puede seleccionar una condición muy específica dentro de un programa, es decir, que para llegar a ella se haya tenido la necesidad del cumplimiento de otras condiciones.

1.5.– Control de flujo: `switch`

- La proposición `switch`, permite la decisión múltiple que prueba si una expresión coincide con uno de los valores constantes enteros que se hayan definido previamente. Su sintaxis es:

```
switch( expresión ) {  
    case exp-const: proposiciones  
    break;  
  
    case exp-const: proposiciones  
    break;  
  
    case exp-const:  
    case exp-const: proposiciones  
    break;  
  
    default: proposiciones  
}
```

1.5.– Control de flujo: `switch`

- ❖ Se compara la "expresión" con cada una de las opciones "exp-const", y en el momento de encontrar una constante idéntica se ejecutan las proposiciones correspondientes a ese caso.
- ❖ Al terminar de realizar las proposiciones del caso, se debe usar la palabra reservada "break" para que vaya al final del `switch`.
- ❖ Si ninguno de los casos cumple con la expresión, se puede definir un caso por omisión, que también puede tener proposiciones.
- ❖ En todos los casos pueden ser proposiciones simples o compuestas. En las compuestas se usan llaves para definir el bloque.

1.5.– Control de flujo: while

- La proposición "while" permite la ejecución de una proposición simple o compuesta, mientras la "expresión" sea verdadera. Su sintaxis es:

```
while( expresión )
      proposición
```

- Por ejemplo:

```
while( (c = getchar()) == 's' || c == 'S' )
      printf("\nDesea salir del programa?");
while( (c = getchar()) == 's' || c == 'S' )
{
    printf("\nDesea salir del programa?");
    ++i;
    printf("\nNúmero de veces que se ha negado a salir: %u",i);
}
```

1.5.– Control de flujo: **for**

- La proposición "for" requiere tres expresiones como argumento. Las expresión1 y la expresión3 comúnmente se emplean para asignaciones, mientras que la expresión2 es la condición que se debe cumplir para que el ciclo "for" se siga ejecutando. La sintaxis es:

for(expresión1; expresión2; expresión3)

proposición

- Ejemplo:

```
for(i=0; i <= 500; ++i)  
printf("\nEl número par # %i, es: %i", (2*i), i);
```

1.5.– Control de flujo: do – while

- ❖ La proposición "do-while" es similar a la proposición "while", se ejecuta el ciclo mientras se cumpla la condición dada en "expresión".
- ❖ La diferencia estriba en que en el "do-while" siempre se evalúa al menos una vez su "proposición", mientras que en el "while" si no se cumple la "expresión" no entra al ciclo.

❖ Sintaxis:

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

❖ Ejemplo:

```
i=0;
do
{
    printf("\nEl número par # %i, es: %i", (2*i), i);
    ++i;
}
while( i < 500 );
```

1.5.– Control de flujo: **break** y **continue**

- ❖ Cuando se quiere abandonar un ciclo en forma prematura debido a que ciertas condiciones ya se cumplieron, se puede usar la proposición "break". Ésta sirve para las proposiciones "for", "while" y "do-while".
- ❖ También se tiene otra proposición relacionada, y esta es el "continue"; su función es la de ocasionar la próxima iteración del ciclo.

Ejemplos:

```
for(h=1; h <= 1000; ++h)
{
    if( h%5 == 0 )
        continue;
    printf("Número que no es múltiplo de 5: %i", h);
```

```
while(1)
{
    if( getchar() == '$' )
        break;
    printf("\nNo puedo parar, hasta que presione '$'");
```

Índice.

- 1.1.– Introducción.
- 1.2.– Estructura de un programa en C.
- 1.3.– Tipos.
- 1.4.– Operadores.
- 1.5.– Control de flujo.
- 1.6.– Funciones.**
- 1.7.– Punteros (o apunadores).
- 1.8.– El preprocessador de C.
- 1.9.– Otros aspectos del lenguaje C.
- 1.10.– Las librerías del lenguaje C.

1.6.– Funciones.

- Las funciones son fragmentos de un programa que realizan tareas específicas. Esto permite dar claridad al programa y facilita su mantenimiento. La sintaxis de las funciones es:

```
tipo nombre_función (tipo argumento)
{
    expresión1;
    expresión2;
    expresión3;
    ...
}
```

- El "*tipo*" determina la clase de variable que devolverá la función. Los tipos definidos para las variables en la sección 1.3.1 son aplicables a las funciones.
- El "*nombre_función*" deberá ser único en todo el programa y no se podrán usar ni palabras reservadas del lenguaje, ni funciones ya preestablecidas en las librerías del compilador.

1.6.– Funciones.

- ❖ A una función se le pueden pasar argumentos, con el fin de que con esos valores se realicen cálculos específicos. Los argumentos se separan por comas. Por ejemplo:

```
void integra(float x, float dx, float lim_inf, float lim_sup)
{
    .
    .
}
```

- ❖ El paso de argumentos entre funciones puede ser por valor o por dirección. Finalmente, las funciones pueden devolver un resultado, es decir, que al finalizar la ejecución de la función se produjo un dato que se va a usar posteriormente dentro del programa y por ello es necesario que sea dado como respuesta. Para realizar esto se emplea la palabra reservada "return". Por ejemplo:

1.6.– Funciones.

```
float cuadrado(float x)
{
    float y;
    y = x * x;
    return(y);
}
void main(void)
{
    float xx, yy, rr;
    ...
    rr = sqrt( (cuadrado(xx) + cuadrado(yy)) );
    printf("\nEl valor de la hipotenusa es: %f", rr);
}
```

- 💡 La función "main", es la rutina principal de un programa en C. A partir de ella se llaman al resto de las funciones.
- 💡 Una función puede llamar a otra función. En lenguaje C existe la recursión.



Índice.

- 1.1.– Introducción.
- 1.2.– Estructura de un programa en C.
- 1.3.– Tipos.
- 1.4.– Operadores.
- 1.5.– Control de flujo.
- 1.6.– Funciones.
- 1.7.– Punteros (o apuntadores).**
- 1.8.– El preprocessador de C.
- 1.9.– Otros aspectos del lenguaje C.
- 1.10.– Las librerías del lenguaje C.

1.7.– Punteros (o apuntadores).

- ❖ Una variable puede ser usada por su nombre o por la dirección donde está el contenido de la variable.
- ❖ Para poder utilizar las direcciones de las variables se utilizan los punteros.
- ❖ Sintaxis:

```
tipo *nombre_variable;
```

- ❖ La diferencia entre una declaración de una variable normal y un puntero es el uso del asterisco.

1.7.– Punteros (o apuntadores).

- Para que quede claro que es un puntero se presenta la siguiente analogía:
 - Imagínese un libro, éste, tiene hojas y cada una de ellas está numerada.
 - Ahora, supongamos que tiene un índice al final del mismo y se desea hallar información relacionada con el ajedrez.
 - Al encontrar la palabra ajedrez en el índice se tiene el número de página; en ese momento se cae en la cuenta de la necesidad de una hoja para escribir esa información; enseguida se toma la hoja y se anota la página.
 - Acto seguido se va a la página marcada con el número encontrado y se lee la información.
 - En este caso el puntero será la hoja con el número escrito en ella.

1.7.– Punteros (o apuntadores).

- ❖ Cuando se declara "nombre_variable" se indica que se tiene necesidad de un lugar donde se pueda escribir y que éste sea de tipo determinado (que en el ejemplo precedente pudiera ser una pizarra, una hoja, la pared, etc.).
- ❖ Sin embargo, el objeto no está disponible aún, por ello se requiere de una instrucción adicional que indique la acción de tomar la hoja y tenerla dispuesta para su uso.
- ❖ Un ejemplo concreto sería:

```
#include <stdlib.h>
char *caracter;
void main(void)
{
    caracter = (char *)malloc(sizeof(char));
    .
    *caracter = 'O';
}
```

1.7.– Punteros (o apuntadores).

- La etiqueta "caracter" es el lugar en el que se puede escribir la dirección donde está la información que requerimos, (char *) indica el tipo de variable que va a ser, es decir puntero a char (en nuestro ejemplo del libro: tipo hoja que tendrá números de página), mientras que "*caracter" apunta directamente al contenido de la variable (la hoja con el número de página).
- Para tomar un espacio de memoria de la computadora que sirva para fines de almacenar direcciones se debe incluir la cabecera o "header" stdlib.h, después se declara la variable y luego se asigna memoria mediante la función "malloc".
- La función malloc asigna memoria y usa como argumento el número de *bytes* que se quieran usar. Por eso se pone sizeof(char), donde la función sizeof, devuelve el tamaño del tipo que se pone como argumento. Finalmente, para que exista consistencia entre tipos, se usa el operador cast (char *) para indicar que la variable contendrá direcciones para encontrar un char.

1.7.– Punteros (o apuntadores).

- Si se ha definido una variable de manera "normal" (que no se haya definido como puntero), también se puede usar su dirección. Para ello se antepone el carácter '&' a la variable usada.
- Por ejemplo:

```
float x, *y;  
void main(void)  
{  
    x = 4.3;  
    y = &x;  
    printf("El valor de y es: %f", *y);  
}
```

- El resultado del `printf` será que $y = 4.3$.

Índice.

- 1.1.– Introducción.
- 1.2.– Estructura de un programa en C.
- 1.3.– Tipos.
- 1.4.– Operadores.
- 1.5.– Control de flujo.
- 1.6.– Funciones.
- 1.7.– Punteros (o apuntadores).
- 1.8.– El preprocessador de C.**
- 1.9.– Otros aspectos del lenguaje C.
- 1.10.– Las librerías del lenguaje C.

1.8.– El preprocessador de C.

Unión de líneas:

- Las líneas que terminan con el carácter diagonal invertida '/' se empalman eliminando la diagonal inversa y el carácter nueva línea.

Definición y expansión de macros:

- Mediante una macro se pueden definir etiquetas que van ser sustituidas en tiempo de compilación dentro del programa. La sintaxis es como sigue:

```
# define identificador secuencia-de-tokens
# define identificador (lista-de-identificadores) secuencia-de-tokens
# undef identificador
```

- Esta última definición hace que el preprocessador olvide la definición del identificador.

1.8.– El preprocessador de C.

leaf Ejemplos:

```
#define TABSIZE 100  
#define ABSDIFF(a,b) ((a)>(b) ? (a)-(b) : (b)-(a))
```

leaf En el caso del preprocessador las expresiones no terminan con punto y coma.

fan Inclusión de archivos:

leaf Mediante estas directivas un archivo completo se incluye en el programa actual:

```
# include <nombre-de-archivo> /* Pone el archivo según la ruta indicada por el  
sistema */  
  
# include "nombre-de-archivo" /* Busca el archivo en el directorio actual, y si no  
lo encuentra busca en la ruta indicada por el sistema */  
  
# include secuencia-de-tokens /* Expande la secuencia de tokens como texto normal*/
```

1.8.– El preprocessador de C.

Compilación condicional:

- Estas directivas permiten tomar acciones al preprocessador en función de condiciones iniciales de ambiente.
- Uno de los usos más frecuentes de las macros es para establecer bloques de compilación opcionales.
- Comandos `#ifdef`, `#ifndef`, `#else`, `#endif`, `#undef`.
- Por ejemplo:

1.8.– El preprocessador de C.

Compilación condicional:

preprocesador-condicional:

```
    línea-if texto partes-elif parte-else #endif
            opt
    línea-if:
            # if expresión-constante
            # ifdef identificador
            # ifndef identificador
    partes-elif:
            línea-elif texto
            partes-elif (opt)
    línea-elif:
    # elif expresión-constante
            parte-else:
    # línea-else texto
    línea-else:
            #else
```

1.8.– El preprocessador de C.

Ejemplo 1:

```
#define NUMERO 2
#define NUMERO == 1
#define VAR_TOKEN 100
#define conver(x) (x + 5)
#else
#define VAR_TOKEN 200
#define conver(x) (x +6)
#endif
```

Ejemplo 2:

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

1.8.– El preprocessador de C.

- En el caso del Ejemplo 2, 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 procesadores; según de qué procesador 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.
- Un uso muy importante de los comandos `#ifdef` y `#ifndef` es para evitar comandos `#include` del mismo fichero repetidos varias veces en un mismo programa.

1.8.– El preprocessador de C.

Control de línea:

- Ocasionalmente el compilador supone que el número de línea de la siguiente línea fuente está dado por la constante entera decimal y que el archivo actual de entrada está nombrado por el identificador, todo esto para propósito de diagnóstico de errores:

```
# line constante "nombre-de-archivo"  
# line constante
```

Generación de errores:

- Ocasionalmente el preprocessador escribe un mensaje de diagnóstico que incluye la "secuencia de tokens":

```
# error secuencia-de-tokens (opt)
```

1.8.– El preprocessador de C.

Pragmas:

- Realiza acciones dependiendo de la implementación.

```
# pragma secuencia-de-tokens (opt)
```

Directiva nula:

- Una línea del preprocessador de la forma:

```
#
```

- No tiene efecto alguno.

1.8.– El preprocessador de C.

💡 Nombres predefinidos:

- 🌿 `__LINE__` Constante decimal que contiene el número de línea actual.
- 🌿 `__FILE__` Cadena literal que contiene el nombre del archivo que se está compilando.
- 🌿 `__DATE__` Cadena literal que contiene la fecha de compilación, en la forma 'mm dd aaaa'.
- 🌿 `__TIME__` Cadena literal que contiene la hora de la compilación, en la forma "hh :mm :ss".
- 🌿 `__STDC__` La constante 1. Este identificador será definido como 1 sólo en implementaciones que conforman el estándar.

Índice.

- 1.1.– Introducción.
- 1.2.– Estructura de un programa en C.
- 1.3.– Tipos.
- 1.4.– Operadores.
- 1.5.– Control de flujo.
- 1.6.– Funciones.
- 1.7.– Punteros (o apuntadores).
- 1.8.– El preprocessador de C.
- 1.9.– Otros aspectos del lenguaje C.**
 - 1.9.1.– `Typedef`.**
- 1.10.– Las librerías del lenguaje C.

1.9.– Otros aspectos del lenguaje C. `typedef`

- ❖ Esta palabra reservada del lenguaje C sirve para la creación de nuevos nombres de tipos de datos.
- ❖ Mediante esta declaración es posible que el usuario defina una serie de tipos de variables propios, no incorporados en el lenguaje y que se forman a partir de tipos de datos ya existentes.
- ❖ Por ejemplo, la declaración:

```
typedef int ENTERO;
```

- ❖ define un tipo de variable llamado ENTERO que corresponde a int.

1.9.– Otros aspectos del lenguaje C. **typedef**

- Como ejemplo más completo, se pueden declarar mediante **typedef** las siguientes estructuras:

```
#define MAX_NOM 30
#define MAX_ALUMNOS 400
    struct s_alumno { // se define la estructura s_alumno
        char nombre[MAX_NOM];
        short edad;
    };
    typedef struct s_alumno ALUMNO; // ALUMNO es un nuevo tipo de variable
    typedef struct s_alumno *ALUMNOPTR;
    struct clase {
        ALUMNO alumnos[MAX_ALUMNOS];
        char nom_profesor[MAX_NOM];
    };
    typedef struct clase CLASE;
    typedef struct clase *CLASEPTR;
```

1.9.– Otros aspectos del lenguaje C. **Typedef**

- 💡 Con esta definición se crean las cuatro palabras reservadas para tipos, denominadas ALUMNO (una estructura), ALUMNOPTR (un puntero a una estructura), CLASE y CLASEPTR.
- 💡 Ahora podría definirse una función del siguiente modo:

```
int anade_a_clase (ALUMNO un_alumno, CLASEPTR clase)
{
    ALUMNOPTR otro_alumno;
    otro_alumno = (ALUMNOPTR) malloc(sizeof(ALUMNO));
    otro_alumno->edad = 23;
    ...
    clase->alumnos[0]=alumno;
    ...
    return 0;
}
```

1.9.– Otros aspectos del lenguaje C. `typedef`

- ❖ El comando `typedef` ayuda a parametrizar un programa contra problemas de portabilidad.
- ❖ Generalmente se utiliza `typedef` para los tipos de datos que pueden ser dependientes de la implementación.
- ❖ También puede ayudar a documentar el programa (es mucho más claro para el programador el tipo `ALUMNOPTR`, que un tipo declarado como un puntero a una estructura complicada), haciéndolo más legible.

Índice.

- 1.1.– Introducción.
- 1.2.– Estructura de un programa en C.
- 1.3.– Tipos.
- 1.4.– Operadores.
- 1.5.– Control de flujo.
- 1.6.– Funciones.
- 1.7.– Punteros (o apunadores).
- 1.8.– El preprocessador de C.
- 1.9.– Otros aspectos del lenguaje C.
 - 1.9.2.– Funciones recursivas.**
- 1.10.– Las librerías del lenguaje C.

1.9.2.– Funciones recursivas.

- La recursividad es la posibilidad de que una función se llame a sí misma, bien directa o indirectamente.
- Un ejemplo típico es el cálculo del factorial de un número, definido en la forma:

$$N! = N * (N-1)! = N * (N-1) (N-2)! = N * (N-1)*(N-2)*...*2*1$$

- La función factorial, escrita de forma recursiva, sería como sigue:

```
unsigned long factorial(unsigned long numero)
{
    if ( numero == 1 || numero == 0 )
        return 1;
    else
        return numero*factorial(numero-1);
}
```

1.9.2.– Funciones recursivas.

- 💡 Supóngase la llamada a esta función para N=4, es decir factorial(4). Cuando se llame por primera vez a la función, la variable numero valdrá 4, y por tanto devolverá el valor de 4*factorial(3); pero factorial(3) devolverá 3*factorial(2); factorial(2) a su vez es 2*factorial(1) y dado que factorial(1) es igual a 1 (es importante considerar que sin éste u otro caso particular, llamado caso base, la función recursiva no terminaría nunca de llamarse a sí misma), el resultado final será 4*(3*(2*1)).
- 💡 Por lo general la recursividad no ahorra memoria, pues ha de mantenerse una pila con los valores que están siendo procesados.
- 💡 Tampoco es más rápida, sino más bien todo lo contrario, pero el código recursivo es más compacto y a menudo más sencillo de escribir y comprender.

Índice.

- 1.1.– Introducción.
- 1.2.– Estructura de un programa en C.
- 1.3.– Tipos.
- 1.4.– Operadores.
- 1.5.– Control de flujo.
- 1.6.– Funciones.
- 1.7.– Punteros (o apuntadores).
- 1.8.– El preprocessador de C.
- 1.9.– Otros aspectos del lenguaje C.
 - 1.9.3.– Gestión dinámica de la memoria.**
- 1.10.– Las librerías del lenguaje C.

1.9.3.– Gestión dinámica de la memoria.

- Según lo visto hasta ahora, la reserva o asignación de memoria para vectores y matrices se hace de forma automática con la declaración de dichas variables, asignando suficiente memoria para resolver el problema de tamaño máximo, dejando el resto sin usar para problemas más pequeños.
- Así, si en una función encargada de realizar un producto de matrices, éstas se dimensionan para un tamaño máximo (100, 100), con dicha función se podrá calcular cualquier producto de un tamaño igual o inferior, pero aún en el caso de que el producto sea por ejemplo de tamaño (3, 3), la memoria reservada corresponderá al tamaño máximo (100, 100).
- Es muy útil el poder reservar más o menos memoria en tiempo de ejecución, según el tamaño del caso concreto que se vaya a resolver. A esto se llama reserva o gestión dinámica de memoria.

1.9.3.– Gestión dinámica de la memoria.

- Existen en C dos funciones que reservan la cantidad de memoria deseada en tiempo de ejecución. Dichas funciones devuelven (es decir, tienen como valor de retorno) un puntero a la primera posición de la zona de memoria reservada.
- Estas funciones se llaman `malloc()` y `calloc()`, y sus declaraciones, que están en la librería `stdlib.h`, son como sigue:

```
void *malloc(int n_bytes)
```

```
void *calloc(int n_datos, int tamaño_dato)
```

1.9.3.– Gestión dinámica de la memoria.

- ❖ La función `malloc()` busca en la memoria el espacio requerido, lo reserva y devuelve un puntero al primer elemento de la zona reservada.
- ❖ La función `calloc()` necesita dos argumentos: el nº de celdas de memoria deseadas y el tamaño en bytes de cada celda; se devuelve un puntero a la primera celda de memoria.
- ❖ La función `calloc()` tiene una propiedad adicional: inicializa todos los bloques a cero.
- ❖ Existe también una función llamada `free()` que deja libre la memoria reservada por `malloc()` o `calloc()` y que ya no se va a utilizar.

1.9.3.– Gestión dinámica de la memoria.

- ❖ Esta función usa como argumento el puntero devuelto por `calloc()` o `malloc()`. La memoria no se libera por defecto, sino que el programador tiene que liberarla explícitamente con la función `free()`. El prototipo de esta función es el siguiente:

```
void free(void *)
```

- ❖ A continuación se presenta un ejemplo de gestión dinámica de memoria para el producto de matriz por vector $\{y\}=[a]\{x\}$. Hay que tener en cuenta que reservando memoria por separado para cada fila de la matriz, no se garantiza que las filas estén contiguas en la memoria.
- ❖ Por otra parte, de esta forma se pueden considerar filas de distinto tamaño. El nombre de la matriz se declara como puntero a vector de punteros, y los nombres de los vectores como punteros. Supóngase que N es una constante simbólica predefinida con el número de filas.

1.9.3.- Gestión dinámica de la memoria.

```
// declaraciones
double **a, *x, *y;
void prod(int , double **, double *, double *);
...
// reserva de memoria para la matriz a
a = calloc(N, sizeof(double *));
for (i=0; i<N; i++)
    a[i]=calloc(N, sizeof(double));
...
// reserva de memoria para los vectores x e y
x = calloc(N, sizeof(double));
y = calloc(N, sizeof(double));
...
prod(N, a, x, y);
...
// definicion de la funcion prod()
void prod(int N, double **mat, double *x, double *y)
{...}
```

1.9.3.- Gestión dinámica de la memoria.

- Con gestión dinámica de memoria es más fácil utilizar matrices definidas como vectores de punteros que matrices auténticas (también éstas podrían utilizarse con memoria dinámica: bastaría reservar memoria para las N filas a la vez y asignar convenientemente los valores al vector de punteros a[i]). El ejemplo anterior quedaría del siguiente modo:

```
// declaraciones
double **a, *x, *y;
void prod(int , double **, double *, double *);
...
// reserva de memoria para el vector de punteros a[]
a = calloc(N, sizeof(double *));
// reserva de memoria para toda la matriz a[][]
a[0] = calloc(N*N, sizeof(double));
// asignación de valor para los elementos del vector de punteros a[]
for (i=1; i<N; i++)
    a[i] = a[i-1]+N;
// el resto del programa sería idéntico
...
```

Índice.

- 1.1.– Introducción.
- 1.2.– Estructura de un programa en C.
- 1.3.– Tipos.
- 1.4.– Operadores.
- 1.5.– Control de flujo.
- 1.6.– Funciones.
- 1.7.– Punteros (o apunadores).
- 1.8.– El preprocessador de C.
- 1.9.– Otros aspectos del lenguaje C.
- 1.10.– Las librerías del lenguaje C.**

1.10.– Las librerías del lenguaje C.

- ❖ A continuación se incluyen en forma de tabla algunas de las funciones de librería más utilizadas en el lenguaje C.
- ❖ **Nota:** La columna tipo se refiere al tipo de la cantidad que devuelve la función. Un asterisco denota puntero, y los argumentos que aparecen en la tabla tienen el significado siguiente:
 - ❖ **c** representa un argumento de tipo carácter.
 - ❖ **d** representa un argumento de doble precisión.
 - ❖ **f** representa un argumento archivo.
 - ❖ **i** representa un argumento entero.
 - ❖ **l** representa un argumento entero largo.
 - ❖ **p** representa un argumento puntero.
 - ❖ **s** representa un argumento cadena.
 - ❖ **u** representa un argumento entero sin signo.

1.10.– Las librerías del lenguaje C.

Función	Tipo	Propósito	Librería
abs (i)	int	Devuelve el valor absoluto de i	stdlib.h
acos (d)	double	Devuelve el arco coseno de d	math.h
asin (d)	double	Devuelve el arco seno de d	math.h
atan (d)	double	Devuelve el arco tangente de d	math.h
atof (s)	double	Convierte la cadena s en un número de doble precisión.	stdlib.h
atoi (s)	long	Convierte la cadena s en un número entero.	stdlib.h
clock ()	long	Devuelve la hora del reloj del ordenador. Para pasar a segundos, dividir por la constante CLOCKS_PER_SEC	time.h
cos (d)	double	Devuelve el coseno de d	math.h
exit (u)	void	Cerrar todos los archivos y buffers, terminando el programa.	stdlib.h
exp (d)	double	Elevar e a la potencia d (e=2.718281...)	math.h

1.10.– Las librerías del lenguaje C.

Función	Tipo	Propósito	Librería
fabs (d)	double	Devuelve el valor absoluto de d	math.h
fclose (f)	int	Cierra el archivo f	stdio.h
feof (f)	int	Determina si se ha encontrado un fin de archivo.	stdio.h
fgetc (f)	int	Leer un carácter del archivo f	stdio.h
fgets (s, i, f)	char *	Leer una cadena s, con i caracteres, del archivo f	stdio.h
floor (d)	double	Devuelve un valor redondeado por defecto al entero más cercano a d	math.h
fmod (d1, d2)	double	Devuelve el resto de d1/d2 (con el mismo signo de d1)	math.h
fopen (s1, s2)	FILE *	Abre un archivo llamado s1, para una operación del tipo s2. Devuelve el puntero al archivo abierto.	stdio.h
fprintf (f...)	int	Escribe datos en el archivo f	stdio.h
fputc (c, f)	int	Escribe un carácter en el archivo f	stdio.h

1.10.– Las librerías del lenguaje C.

Función	Tipo	Propósito	Librería
free (p)	void	Libera un bloque de memoria al que apunta p	malloc.h
fscanf (f...)	int	Lee datos del archivo f	stdio.h
getc (f)	int	Lee un carácter del archivo f	stdio.h
getchar ()	int	Lee un carácter desde el dispositivo de entrada estándar.	stdio.h
log (d)	double	Devuelve el logaritmo natural de d	math.h
malloc (n)	void *	Reserva n bytes de memoria. Devuelve un puntero al principio del espacio reservado.	malloc.h ó stdlib.h
pow (d1 ,d2)	double	Devuelve d1 elevado a la potencia d2	math.h
printf (...)	int	Escribe datos en el dispositivo de salida estándar.	stdio.h
rand (void)	int	Devuelve un valor aleatorio positivo.	stdlib.h
sin (d)	double	Devuelve el seno de d	math.h

1.10.– Las librerías del lenguaje C.

Función	Tipo	Propósito	Librería
sqrt (d)	double	Devuelve la raíz cuadrada de d	math.h
strcmp (s1, s2)	int	Compara dos cadenas lexicográficamente.	string.h
strcasecmp (s1, s2)	int	Compara dos cadenas lexicográficamente, sin considerar mayúsculas o minúsculas.	string.h
strcpy (s1, s2)	char *	Copia la cadena s2 en la cadena s1	string.h
strlen (s)	int	Devuelve el número de caracteres en la cadena s	string.h
system (s)	int	Pasa la orden s al sistema operativo.	stdlib.h
tan (d)	double	Devuelve la tangente de d	math.h
time (p)	long int	Devuelve el número de segundos transcurridos desde un tiempo base designado (1 de enero de 1970).	time.h
toupper (c)	int	Convierte una letra a mayúscula	stdlib.h ó ctype.h

¡Muchas gracias por su atención!

Carlos Diego Moreno Moreno



Área de Arquitectura y Tecnología de Computadores

Departamento de Ingeniería Electrónica y Computadores.

Escuela Politécnica Superior. Universidad de Córdoba