# Preparador Informática

www.preparadorinformatica.com

## TEMA 32. INFORMÁTICA

## LENGUAJE C: MANIPULACIÓN DE ESTRUCTURAS DE DATOS DINÁMICAS Y ESTÁTICAS. ENTRADA Y SALIDA DE DATOS. GESTIÓN DE PUNTEROS. PUNTEROS A FUNCIONES

## TEMA 35. SAI

## LENGUAJE C: MANIPULACIÓN DE ESTRUCTURAS DE DATOS DINÁMICAS Y ESTÁTICAS. ENTRADA Y SALIDA DE DATOS. GESTIÓN DE PUNTEROS. PUNTEROS A FUNCIONES. GRÁFICOS EN C

# TEMA 32 INF: LENGUAJE C: MANIPULACIÓN DE ESTRUCTURAS DE DATOS DINÁMICAS Y ESTÁTICAS. ENTRADA Y SALIDA DE DATOS. GESTIÓN DE PUNTUROS. PUNTEROS A FUNCIONES.

# TEMA 35 SAI: LENGUAJE C: MANIPULACIÓN DE ESTRUCTURAS DE DATOS DINÁMICAS Y ESTÁTICAS. ENTRADA Y SALIDA DE DATOS. GESTIÓN DE PUNTUROS. PUNTEROS A FUNCIONES. GRÁFICOS EN C.

# 1. INTRODUCCION

A lo largo de la historia de la informática, han ido apareciendo diferentes paradigmas de programación y, por tanto, diferentes lenguajes de programación. En primer lugar, apareció la programación secuencial, que consistía en secuencias de sentencias que se ejecutaban una tras otra. El lenguaje ensamblador o el lenguaje COBOL son lenguajes secuenciales. Entonces no existía el concepto de función, que apareció más adelante en los lenguajes procedimentales, como el BASIC o C. La evolución no terminó aquí, y continuó hasta el paradigma más extendido en la actualidad: la programación orientada a objetos. Java, C++ o C# son ejemplos de lenguajes basados en este paradigma.

En el presente tema se estudian algunas de las características del lenguaje C. Este lenguaje es uno de los más importantes en la historia de la informática por su trascendencia en la misma. Sigue siendo uno de los lenguajes más importantes debido a que sigue utilizándose en multitud de aplicaciones y además es la base de otros lenguajes posteriores.

# 2. MANIPULACIÓN DE ESTRUCTURAS DE DATOS ESTÁTICAS Y DINÁMICAS

## 2.1. TIPOS DE DATOS AVANZADOS. ESTRUCTURAS DE DATOS ESTÁTICAS.



- **Enumeraciones:** Es un tipo de dato entero con valores constantes definidos por el usuario o, dicho de otro modo, un tipo enumerado es una lista de valores enteros constantes. Por ejemplo: enum boolean {FALSE, TRUE}. Salvo que se especifique otra cosa, comienzan en 0, así que FALSE será 0 y TRUE 1.
- **Arrays:** Los arrays son bloques de elementos del mismo tipo. El tipo base de un array puede ser un tipo fundamental o 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. Los arrays pueden tener una o varias dimensiones. El array más común en C es la cadena, que simplemente es un array de caracteres terminado por uno nulo.

- **Estructuras:** Una estructura es un agregado de tipos fundamentales o derivados que se compone de varios campos. Cada elemento de la estructura puede ser de un tipo diferente, cosa que no es posible en los arrays:

```
struct nombre_estructura{
      tipo1 campo1;
      ...
      tipoN campoN,
};
```

- **Uniones:** Se definen de forma parecida a las estructuras y se emplean para almacenar en un mismo espacio de memoria variables de distintos tipos. El tamaño de la unión es igual al tamaño del mayor de sus campos:

```
union nombre_union{
      tipo1 campo1;
      ...
      tipoN campoN,
};
```

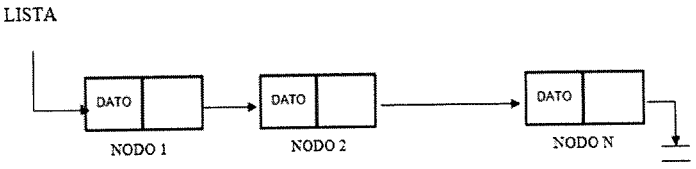
- **Tipos definidos por el usuario:** Es una declaración que permite definir tipos de datos que ya tenemos con nombres diferentes y también con nuevos tipos de datos. La sintaxis es la siguiente: `typedef tipo nuevo_tipo`
- **Campos de bits:** Un campo de bit es un método predefinido por C para poder acceder a un bit de un byte. Se basa en la estructura, pues un campo de bit no es más que un tipo especial de estructura al que le añadimos el número de bits que se desea reservar para cada elemento. El formato es:

```
struct nombre_campo_bit
{
      tipo nombre1 : longitud;
      ...
      tipo nombreN : longitud;
}variables_campo_bit;
```



## 2.2. ESTRUCTURAS DE DATOS DINÁMICAS.

### 2.2.1. LISTAS ENLAZADAS




En C, para crear una lista debemos definir la clase de elementos que van a formar parte de la misma. Un tipo de dato genérico podría ser:

```
struct nodo
{
   int dato;
   struct nodo *sig;
};
```

En las listas simples enlazadas existe un nodo especial: el primero. Diremos que nuestra lista es un puntero a ese primer nodo. Eso es porque mediante ese único puntero podemos acceder a toda la lista, por eso, es muy importante que nuestro programa nunca pierda el valor del puntero al primer elemento. Cuando el puntero que usamos para acceder a la lista vale NULL, diremos que la lista está vacía. Con las listas se pueden realizar las siguientes operaciones básicas:

a) **Crear lista:** Si queremos crear una lista debemos inicializar el puntero de la lista a NULL.
b) **Añadir o insertar elementos:** Para añadir un nuevo elemento debemos crear el nodo haciendo la reserva dinámica de memoria para el nodo normalmente usando la función "malloc". Ahora si la lista está vacía apuntaremos directamente la lista a ese elemento, y si ya tiene elementos habrá que buscar el último elemento de la lista y apuntar su puntero "siguiente" al nuevo registro.
c) **Buscar o localizar elementos:** Muy a menudo necesitaremos recorrer una lista. Para esto dependemos de si la lista está ordenada o no. Si la lista no está ordenada debemos realizar una búsqueda secuencial y en el caso de que sí esté ordenada se busca hasta que lo hayamos encontrado el elemento o el valor del nodo en el que estamos sea mayor del elemento a buscar.
d) **Borrar elementos:** Lo primero será localizar el nodo a eliminar, si es que existe. Pero sin perder el puntero al nodo anterior. Partiremos del nodo primero, y del valor NULL para anterior. Y avanzaremos hasta que se termine la lista o hasta que encontremos el valor a borrar. Una vez encontrado modificamos el puntero "siguiente" y liberamos la memoria del nodo borrado.

Además de la lista enlazada sencilla existen otros tipos de listas:

- Listas circulares: el último nodo enlaza con el primero.
- Listas doblemente enlazadas: cada nodo tiene un doble enlace para apuntar al nodo siguiente y anterior.
- Multilistas: cada uno de los nodos de la lista puede apuntar a otras listas.

### 2.2.2. LISTAS DOBLEMENTE ENLAZADAS

Una lista doblemente enlazada es una lista lineal en la que cada nodo tiene dos enlaces, uno al nodo siguiente, y otro al anterior. El nodo típico es el mismo que

para construir las listas que hemos visto, salvo que tienen otro puntero al nodo anterior. En C:

```
struct nodo {
      int valor;
      struct nodo *siguiente;
      struct nodo *anterior;
};

typedef struct nodo tipoNodo;
typedef tipoNodo *pNodo;
typedef tipoNodo *Lista;
```

El movimiento a través de listas doblemente enlazadas es más sencillo y las operaciones de búsqueda, inserción y borrado, también tienen más ventajas. Además, tenemos el mismo repertorio de operaciones sobre este tipo de listas y se realizan de forma similar salvo que debemos tener en cuenta la utilización y las posibles modificaciones del puntero "anterior".

### 2.2.3. PILAS

Una pila se puede implementar tanto por medio de estructuras de datos estáticas como dinámicas. En C los tipos que definiremos normalmente para manejar pilas serán casi los mismos que para manejar listas, tan sólo cambiaremos algunos nombres:



```
struct nodo
{
      int valor;
      struct nodo *siguiente;
};

typedef struct nodo tipoNodo;
typedef tipoNodo *pNodo;
typedef tipoNodo *Pila;
```

Una pila es una lista abierta, así que sigue siendo muy importante que nuestro programa nunca pierda el valor del puntero al primer elemento, igual que pasa con las listas abiertas. Teniendo en cuenta que las inserciones y borrados en una pila se hacen siempre en un extremo, lo que consideramos como el primer elemento de la lista es en realidad el último elemento de la pila. Las **operaciones** con pilas son muy simples, no hay casos especiales, salvo que la pila esté vacía.

**Push:** Añadir un nuevo elemento a la cima de la pila. Para ello creamos un nodo con el valor que colocaremos en la pila, hacemos que nodo→siguiente apunte a Pila y finalmente hacemos que Pila apunte a nodo.

**Pop, leer y eliminar un elemento:** Eliminar el elemento situado en la cima de la pila. Sólo existe un caso posible, ya que sólo podemos leer desde un extremo de la pila. Partiremos de una pila con uno o más nodos, y usaremos un puntero auxiliar. Hacemos que nodo auxiliar apunte al primer elemento de la pila, es decir a Pila. Asignamos a Pila la dirección del segundo nodo de la pila: Pila->siguiente. Guardamos el contenido del nodo para devolverlo como retorno de la función (recordemos que la operación pop equivale a leer y borrar). Liberamos la memoria asignada al primer nodo, el que queremos eliminar. (Si la pila sólo tiene un nodo, el proceso sigue siendo válido, ya que el valor de Pila->siguiente es NULL, y después de eliminar el último nodo la pila quedará vacía, y el valor de Pila será NULL.)

### 2.2.4. COLAS

Para la implementación en C de una cola los tipos que definiremos normalmente serán casi los mismos que para manejar listas y pilas:

```
struct nodo
{
      int valor;
      struct nodo *siguiente;
};
typedef struct nodo tipoNodo;
typedef tipoNodo *pNodo;
typedef tipoNodo *Cola;
```



Una cola es una lista abierta así que sigue siendo muy importante que nuestro programa nunca pierda el valor del puntero al primer elemento, igual que pasa con las listas abiertas. Además, debido al funcionamiento de las colas, también deberemos mantener un puntero para el último elemento de la cola, que será el punto donde insertemos nuevos nodos. Las **operaciones** con colas son muy sencillas, prácticamente no hay casos especiales, salvo que la cola esté vacía.

**Enqueue (encolar):** Añadir elemento en una cola. Hacemos que nodo->siguiente apunte a NULL. Si ultimo no es NULL, hacemos que ultimo->siguiente apunte a nodo. Y actualizamos último, haciendo que apunte a nodo. (Si primero

es NULL, significa que la cola estaba vacía, así que haremos que primero apunte también a nodo.)

**Dequeue (desencolar):** Leer y eliminar un elemento en una cola. Hacemos que nodo apunte al primer elemento de la pila, es decir a primero. Asignamos a primero la dirección del segundo nodo de la pila: primero->siguiente. Guardamos el contenido del nodo para devolverlo como retorno, recuerda que la operación de lectura en colas implica también borrar. Liberamos la memoria asignada al primer nodo, el que queremos eliminar. Si primero es NULL, hacemos que ultimo también apunte a NULL, ya que la lectura ha dejado la cola vacía.

### 2.2.5. ÁRBOLES

Un árbol es una estructura de datos jerárquica formada por una serie de nodos conectados por una serie de aristas que verifican que:

- Hay un único nodo raíz.
- Cada nodo, excepto la raíz, tiene un único nodo padre.
- Hay un único camino desde la raíz hasta cada nodo.

**Implementación en C de árboles:** Es importante conservar siempre el nodo raíz ya que es el nodo a partir del cual se desarrolla el árbol, si perdemos este nodo, perderemos el acceso al árbol. El nodo típico de un árbol difiere de los nodos que hemos visto hasta ahora, aunque sólo en el número de nodos.

```
#define ORDEN 5 //por ejemplo 5
struct nodo
{
   int valor;
   struct nodo *rama[ORDEN];
};
typedef struct nodo tipoNodo;
typedef tipoNodo *pNodo;
typedef tipoNodo *Arbol;
```

**Operaciones básicas con árboles:** De nuevo tenemos casi el mismo repertorio de operaciones de las que disponíamos con las listas: Añadir elementos, buscar o localizar elementos, borrar elementos, moverse a través del árbol y recorrer el árbol completo. Analicemos los recorridos en profundidad.

**Recorridos:** Los recorridos de un árbol se suelen implementar mediante recursividad. En todos los casos se sigue a partir de cada nodo todas las ramas

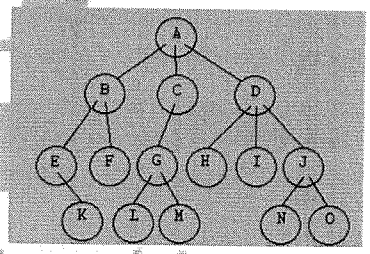
una por una. Supongamos que tenemos un árbol de orden tres, y queremos recorrerlo por completo. Partiremos del nodo raíz: `RecorrerArbol(raiz);`

La función RecorrerArbol, aplicando recursividad, será tan sencilla como invocar de nuevo a la función RecorrerArbol para cada una de las ramas:

```
void RecorrerArbol(Arbol a) {
   if(a == NULL)
      return;
   RecorrerArbol(a->rama[0]);
   RecorrerArbol(a->rama[1]);
   RecorrerArbol(a->rama[2]);
}
```

Para recorrer un árbol los tres tipos de recorridos más habituales son: Preorden: el valor del nodo se procesa antes de recorrer las ramas; Inorden: el valor del nodo se procesa después de recorrer la primera rama y antes de recorrer la última; Postorden: El valor del nodo se procesa después de recorrer todas las ramas. Veamos por ejemplo el recorrido inorden (es el recorrido que más sentido tiene en el caso de árboles binarios) usando el siguiente árbol de ejemplo:

```
void InOrden(Arbol a) {
   if(a == NULL)
      return;
   RecorrerArbol(a->rama[0]);
   Procesar(dato);
   RecorrerArbol(a->rama[1]);
   RecorrerArbol(a->rama[2]);
}
```






Suponiendo que *"Procesar(dato)"* fuera simplemente mostrarlo en pantalla obtendremos: K E B F A L G M C H D I N J O

Una vez vistos conceptos generales sobre árboles vamos a centrarnos en los árboles binarios de búsqueda que son los más habitualmente utilizados.

**Árboles binarios de búsqueda (ABB):** Se trata de árboles de orden 2 en los que se cumple que, para cada nodo, el valor de la clave de la raíz del subárbol izquierdo es menor que el valor de la clave del nodo y que el valor de la clave raíz del subárbol derecho es mayor que el valor de la clave del nodo. El repertorio de operaciones que se pueden realizar sobre un ABB es parecido al que realizábamos sobre otras estructuras de datos:

**Buscar un elemento de un ABB:** Partiendo siempre del nodo raíz, el modo de buscar un elemento se define de forma recursiva. Si el árbol está vacío, terminamos la búsqueda: el elemento no está. Si el valor del nodo raíz es igual

que el del elemento que buscamos, terminamos la búsqueda con éxito. Si el valor del nodo raíz es mayor que el elemento que buscamos, continuaremos la búsqueda en el árbol izquierdo. Si el valor del nodo raíz es menor que el elemento que buscamos, continuaremos la búsqueda en el árbol derecho.

El valor de retorno de una función de búsqueda en un ABB puede ser un puntero al nodo encontrado, o NULL, si no se ha encontrado.

**Insertar un elemento:** Para insertar un elemento nos basamos en el algoritmo de búsqueda. Si el elemento está en el árbol no lo insertaremos. Si no lo está, lo insertaremos a continuación del último nodo visitado. Necesitamos un puntero auxiliar para conservar una referencia al padre del nodo raíz actual. Vamos moviendo por el árbol yendo a izquierda si el elemento es menor que el nodo y a derecha si el elemento a insertar es mayor que el nodo visitado hasta que lleguemos al último nodo visitado posible y hagamos la inserción en el lado correspondiente dependiendo del valor del nodo. Este modo de actuar asegura que el árbol sigue siendo ABB.

**Borrar un elemento:** Para borrar un elemento también nos basamos en el algoritmo de búsqueda. Si el elemento no está en el árbol no lo podremos borrar. Si Se trata de un nodo hoja lo borraremos directamente y si se trata de un nodo rama, en ese caso no podemos eliminarlo, puesto que perderíamos todos los elementos del árbol de que el nodo actual es padre. En su lugar buscamos el nodo más a la izquierda del subárbol derecho, o el más a la derecha del subárbol izquierdo e intercambiamos sus valores. A continuación, se elimina el nodo hoja.



### 2.2.6. GRAFOS

Un grafo es un conjunto de nodos conectados donde cada nodo se denomina vértice y la conexión entre dos de ellos se denomina arista o arco. Los grafos se utilizan para representar aplicaciones de la vida cotidiana, como redes o rutas de transporte así como en muchas aplicaciones de la informática también.

Se pueden distinguir dos tipos de grafos, **no dirigidos** y **dirigidos**:

- **Grafo no dirigido**: las aristas del grafo son bidireccionales.
- **Grafo dirigido**: las aristas del grafo son unidireccionales.

Cuando el grafo tiene un número fijo de nodos se puede implementar como una **matriz de adyacencia**. Sin embargo, lo más habitual es que el número de nodos del grafo pueda variar por lo que se implementa el grafo mediante **listas de adyacencia** (estructura dinámica). Para representarlos mediante listas de adyancencia tendremos que cada nodo debe contener información del nodo en sí y una lista que contendrá los enlaces a todos los nodos relacionados con él, con el peso de cada uno de los enlaces.

```
struct nodo{
      char[20] etiqueta; //podría ser de cualquier tipo
      int nodo; //número de nodo
      struct nodo *sig;
      struct arco *enlaces;
}
struct arco{
      int peso;
      struct nodo *origen;
      struct nodo *destino;
      struct arco *sig;
}
typedef struct nodo *grafo;
```

## 3. PUNTEROS. GESTIÓN DE PUNTEROS. PUNTEROS A FUNCIONES

Los punteros son una de las poderosas herramientas que ofrece el lenguaje C a los programadores, sin embargo, son también una de las más peligrosas porque son una fuente frecuente de errores en los programas de C. Un puntero es una variable que contiene una dirección de memoria. Normalmente esa dirección es una posición de memoria de otra variable, por lo cual se suele decir que el puntero "apunta" a la otra variable. La sintaxis de la declaración de una variable puntero es: `tipo *nombre;` El tipo base de la declaración sirve para conocer el tipo de datos al que pertenece la variable a la cual apunta la variable de tipo puntero. Esto es fundamental para poder leer el valor que almacena la zona de memoria apuntada por la variable puntero y para poder realizar ciertas operaciones aritméticas sobre los mismos. Ejemplo:

`int *p; (*p es un entero, por tanto, p es un puntero a entero).`

C proporciona dos operadores relacionados con las direcciones de memoria:

- Operador indirección: A partir de una variable tipo puntero nos proporciona el dato apuntado.
- & Operador dirección: A partir de una variable nos da la dirección de memoria donde se almacena dicha variable

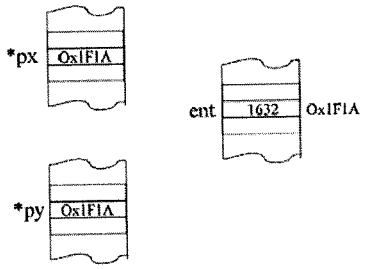
Ejemplo: la variable c contendrá el valor 15, pues *a devuelve el valor de la dirección que sigue (a la que "apunta" la variable puntero), y con anterioridad hemos hecho que a contenga la dirección de memoria de la variable b usando para ello el operador &.

```
int *a,b,c;
b=15;
a=&b;
c=*a;
```

## 3.1. OPERACIONES SOBRE PUNTEROS

Con las variables de tipo puntero es posible realizar algunas operaciones:

a) Asignación de punteros: Es posible asignar el valor de una variable de tipo puntero a otra variable de tipo puntero. Por ejemplo:




```
int *px,*py,ent;
px=&ent;
py=px;
```

Entonces py contiene el valor de px, y por ello, py también "apunta" a la variable ent.

b) Aritmética de punteros: Sólo dos operaciones aritméticas se pueden usar con punteros, la suma y la resta (e incremento y decremento). Estos operadores incrementan o decrementan la posición de memoria a la que "apunta" la variable puntero. El incremento o decremento se realiza de acuerdo al tipo base de la variable de tipo puntero, de ahí la importancia del tipo del que se declara la variable puntero. La posición a la que apunta a se incrementa o decrementa en num*sizeof(tipo).

c) Comparaciones de punteros. Sobre las variables de tipo puntero es posible realizar operaciones de comparación entre ellas.

## 3.2. PUNTEROS Y ARRAYS

Existe una estrecha relación entre los punteros y los arrays. Consideremos el siguiente fragmento de código:

```
char cadena[100],*p;
p=cadena;
```

Este fragmento de código pone en la variable puntero p la dirección del primer elemento del array cadena. Entonces, es posible acceder al valor de la décima

posición del array mediante cadena [9] y *(p+9) (recuérdese que los índices de los arrays empiezan en 0). Esta estrecha relación entre los arrays y los punteros queda más evidente si se tiene en cuenta que el nombre del array sin índice es la dirección de comienzo del array, y, si además, se tiene en cuenta que un puntero puede indexarse como un array unidimensional, por lo cual, en el ejemplo anterior, podríamos referenciar ese elemento como p[9].

Es posible también, como con cualquier otro tipo de datos, definir un array de variables puntero. Por ejemplo, la declaración para un array de punteros int de tamaño 10 es: `int *x[10];`

### 3.3. INDIRECCIÓN MÚLTIPLE Ó PUNTEROS A PUNTEROS

Dado, además, que un puntero es también una variable, es posible definir un puntero a un puntero. Supongamos que tenemos lo siguiente:

```
int a,*b,**c;
b=&a;
c=&b;
```

Y entonces, **c tiene el valor de la variable a, pues c es un puntero a una variable que ya es de tipo puntero.

### 3.4. PUNTEROS A FUNCIONES



Al igual que cualquier otro tipo de dato, una función ocupa una dirección de memoria, y, por tanto, puede ser apuntada por un puntero. Un puntero a función es una variable que almacena la dirección de una función. Esta función puede ser llamada más tarde, a través del puntero. Este tipo de construcción es útil pues encapsula comportamiento, que puede ser llamado a través de un puntero. Veamos un ejemplo sencillo que crea un puntero a una función de imprimir:

```
#include <stdio.h>
void imprime(){
  printf("Imprimiendo\n");
}
int  main(){
  void (*puntero_funcion)(void)=imprime;
  puntero_funcion (); //Llama a imprime
  return 0;
}
```

Generalmente, los punteros a funciones se usan en la programación de bajo nivel, por ejemplo, para creación de controladores de dispositivos, etc.

# 4. ENTRADA Y SALIDA DE DATOS

Para el correcto funcionamiento de la entrada y salida en C es necesario incluir el archivo <stdio.h>, mediante la directiva del preprocesador #include,

## 4.1. ENTRADA/SALIDA DESDE CONSOLA

La entrada y salida desde consola se refiere a las operaciones que se producen en el teclado y la pantalla del ordenador. Las más utilizadas son:

a) La función *getchar()*, lee un carácter desde el teclado y la función *gets()* lee un string desde el teclado hasta que se pulsa un retorno de carro.

b) La función putchar() escribe un carácter a la pantalla del ordenador y la función puts() escribe un string en pantalla.

e) La función scanf() se usa para leer cualquier tipo de dato predefinido desde el teclado, y convertirlo, de forma automática, al formato interno adecuado. La función se define como: `int scanf(cadena de control,arg1,arg2,...,argN);`

La cadena de control indica el formato de los datos a leer. Dicha cadena de control consta de tres clases de caracteres que son los especificadores de formato, los caracteres de espacio en blanco y los caracteres que no sean espacios en blanco. Los especificadores de formato están precedidos por el signo %, y dicen a la función que tipo de datos van a ser leídos a continuación. Los especificadores de formato validos son por ejemplo "%s" si es una cadena de caracteres ó "%d" para números enteros.



f) La función printf() se usa para escribir cualquier tipo de dato a la pantalla. Su formato es: `int printf(cadena de control,arg1,arg2,...,argN);`

Donde cadena de control hace referencia a una cadena de caracteres que contiene información sobre el formato de salida y arg1,arg2,...,argN son argumentos que representan los datos de salida. La cadena de control está compuesta por grupos de caracteres, con un grupo de caracteres por cada dato de salida. Cada grupo de caracteres debe empezar por un signo de porcentaje (%) funcionando igual que en la función scanf().

## 4.2. ENTRADA/SALIDA POR ARCHIVOS

En C un archivo puede ser desde un archivo de disco a un terminal o una impresora. Se asocia una secuencia con un archivo específico realizando una operación de apertura y una vez que esté abierto, la información puede ser intercambiada entre éste y el programa. Un puntero a un archivo es un puntero a una información que define varias cosas sobre él, incluyendo el nombre, el estado y la posición actual del archivo. Para obtener una variable de tipo puntero a archivo se debe utilizar una sentencia como: `FILE *puntero;`

La función fopen() se encarga de abrir un archivo. Su definición es:

```
FILE *fopen(char *nombre,char *modo);
```

Donde nombre es un string que contiene el nombre del archivo que queremos leer y modo es otro string que contiene el modo de apertura deseado. Por ejemplo si usamos "r" es abrir para lectura o "w" para escritura.

La función fclose() cierra un archivo. Su definición es: `int fclose(FILE *fp);`

Una vez abierto un archivo, y hasta que se proceda a cerrarlo es posible leer, escribir, etc., en el, según se indique en el modo de apertura:

- La función getc() lee caracteres del archivo asociado y la función fgets() funciona de igual forma que la función gets().
- La función putc() escribe un carácter en el archivo asociado y la función fputs() funciona igual que la función puts().
- Las funciones fscanf() y fprintf() funcionan de forma similar a sus equivalentes sobre la consola scanf() y printf(), solo que leen o escriben del archivo asociado a fp.

```
int fscanf(FILE *fp,const char *formato[,dirección,...]);
int fprintf(FILE *fp,const char *formato[,argumento,...]);
```

- La función fread() permite leer un bloque de datos.
- La función fwrite() permite escribir un bloque de datos.

Existen tres ficheros que son abiertos de forma automática al comenzar la ejecución del programa, y cerrados, también de forma automática, al terminar la misma. Estos archivos son la entrada standard (stdin), la salida standard (stdout) y la salida standard de error (stderr). Normalmente estos ficheros están asociados a la consola, pero pueden redireccionarse a cualquier otro dispositivo.

## 5. GRÁFICOS EN C

Las funciones que llevan a cabo los gráficos en C permiten generar dibujos y presentaciones de los programas en general. Las funciones de salida dependen del adaptador y del monitor que se esté utilizando. El controlador seleccionado se carga desde el disco durante la inicialización de la biblioteca de gráficos llamado initgraph(), y se necesita incluir la librería #include<graphics.h>. Para activar el modo Gráfico se debe de empezar llamando a la función initgraph():

```
void initgraph(int *controlador, int *modo, char *camino)
```

Se carga en la memoria el controlador de gráficos correspondiente al número determinado por el controlador. El parámetro modo apunta a un entero que especifica el modo de vídeo que van a usar las funciones de gráficos. El parámetro camino especifica el directorio donde se encuentra el controlador. Si no se especifica se busca en el directorio actual. Para desactivar el modo gráfico del programa se usa la función `void closegraph(void)` que implica la devolución al sistema de la memoria que se utilizaba para tener los controladores y las fuentes gráficas en uso.

La escritura en modo gráfico se puede llevar a cabo con las funciones especiales para ello. Las más utilizadas son:

- `void outtext(char *cadena de texto );` Muestra en una pantalla en modo gráfico una cadena de texto en la posición actual (en los modos gráficos no existe un cursor visible, pero la posición actual se conserva como si existiera uno invisible).
- `void outtextxy(int x,int y, char *cad)` Escribe un texto en la posición determinada por las coordenadas x,y en la ventana gráfica.
- `void settextstyle(int fuente, int direccion, int tamaño_car);` Para cambiar el tipo de letra.

Hay muchas otras funciones importantes relacionadas con los gráficos y que podemos encontrar en la librería Graphics, nombraremos algunas de ellas:

- `circle(int x,int y,int angulo_inicial,int angulo_final,int radio); //círculo`
- `ellipse(int x,int y,int angulo_inicial,int angulo_final,int radio_en_x, radio_en_y);//elipse`

- `line(int x,int y,int x2,int y2); //línea`
- `rectangle(int x,int y,int x2,int y2); //rectángulo`
- `bar(int x,int y,int x2,int y2); //barra o rectángulo "relleno"`
- `setcolor(color_elegido); //elegir el color`

## 6. CONCLUSIÓN

En el presente tema hemos estudiado diversas características del lenguaje C. Este lenguaje es de suma importancia por varios motivos principales:

- El lenguaje de programación C sigue siendo uno de los más populares, solo por detrás de Java y a muy corta distancia de éste.
- También sigue siendo uno de los lenguajes más demandados por las empresas.
- Conocer las bases de este lenguaje facilita el aprendizaje y comprensión de otros lenguajes de programación similares, también muy implantados a nivel empresarial.

Principalmente por estos motivos y por muchos otros es tan importante conocer el lenguaje de programación C y dominarlo perfectamente.



## 7. BIBLIOGRAFÍA

- Kernighan, B. y Ritchie, D: **El lenguaje de programación C**. Editorial Pearson
- Ceballos, F.J.: **C/C++ Curso de Programación**. Editorial Ra-Ma
- Adiego, J. y Llanos, D. **Fundamentos de informática y programación en C**. Editorial Paraninfo.