ESCUELA DE PREPARACIÓN DE OPOSITORES
**E. P. O.**
C/. La Merced, 8 – Bajo A Telf.: 968 24 85 54
30001 MURCIA

# INF32

Lenguaje C: manipulación de estructuras de datos dinámicas y estáticas. Entrada y salida de datos. Gestión de punteros. Punteros a funciones.

# SAI35

Lenguaje C: Manipulación de estructuras de datos dinámicas y estáticas. Entrada y salida de datos. Gestión de punteros. Punteros a funciones. Gráficos en C.

## Esquema.

# 1 Introducción.

Entender y saber utilizar los punteros es fundamental para programar en C. Por tres razones: mediante los punteros se consigue el paso de parámetros por referencia, es decir, que una función pueda modificar sus argumentos de llamada; se utilizan para manejar la asignación dinámica de memoria; y el uso de punteros puede mejorar el funcionamiento de ciertas operaciones.

Además, los punteros permiten pasar funciones como argumentos de una función. También están muy relacionados con los arrays y proporcionan una vía alternativa de acceso a los elementos individuales del array. Pero aunque la utilización de punteros produce programas más eficientes, a veces son peligrosos, y es que utilizar punteros de forma incorrecta es muy fácil, y el uso incorrecto puede causar fallos muy difíciles de localizar.

Existen estructuras mediante las cuales podemos solucionar determinados problemas en los que no sabemos la cantidad de memoria que vamos a necesitar durante la ejecución del programa. Por este motivo necesitamos disponer de métodos que nos permitan acceder a las direcciones de memoria que se necesiten durante la ejecución del programa, con la posibilidad de liberarlas cuando no las necesitemos, ofreciéndonos también la posibilidad de modificar su tamaño y estructura en tiempo de ejecución. Estas estructuras reciben el nombre de dinámicas.

Además, en este tema vamos a estudiar la entrada/salida de datos en C, que si bien no forma parte del lenguaje como tal, existe una biblioteca estándar formada por un conjunto de funciones diseñadas para proporcionar un sistema estándar de entrada/salida a los programas C, siendo parte del C estándar ANSI. Concluiremos nuestro estudio repasando las principales librerías gráficas.

# 2 Gestión de punteros.

Un puntero es una variable que contiene la dirección de otra variable. Los punteros se utilizan con abundancia en C, debido en parte a que a veces son la única manera de expresar un cálculo y en parte porque con ellos se obtiene un código más compacto y eficiente.

## *2.1 Punteros y direcciones.*

Puesto que un puntero contiene la dirección de un objeto, se puede acceder al objeto indirectamente a través de él. Supongamos que `x` es una variable, de tipo `int`, y que `px` es un puntero. El operador unitario `&` devuelve la dirección de un objeto, por lo que la proposición `px=&x;` asigna la dirección de `x` a la variable `px`; ahora se dice que `px` "apunta a" `x`. El operador `&` sólo se puede aplicar a variables y a elementos de un array; construcciones como `&(x+1)` y `&3` son ilegales. También es ilegal obtener la dirección de una variable `register`.

El operador unario * toma su operando como una dirección y accede a esa dirección para obtener su contenido. Si y también es de tipo `int`, `y=*px;` asigna a y el contenido de cualquier parte a donde apunte `px`. La secuencia `px=&x; y=*px;` asigna a y el mismo valor que `y=x;`.

También es necesario declarar las variables que intervienen en el ejemplo

```
int x, y;
int *px;
```

La declaración del puntero `px` se entiende como un mnemotécnico; se quiere indicar que la combinación `*px` es del tipo `int`, es decir, si `px` aparece en el contexto `*px`, ello equivale a una variable de tipo `int`. La sintaxis de la declaración de una variable imita a la sintaxis de las expresiones en las que puede aparecer la variable. Este razonamiento es útil en todos los casos de declaraciones complicadas. Por ejemplo, `double atof(),*dp;` indica que `atof()` y `*dp` toman valores del tipo `double` si aparece en una expresión. Se debe observar que en la declaración de un puntero se restringe el tipo de los objetos a los que puede apuntar.

Los punteros pueden aparecer en expresiones. Por ejemplo, si `px` apunta al entero `x`, entonces `px` puede aparecer en cualquier contexto en que pudiera hacerlo `x`. `y=*px+1` asigna a `y` una unidad más que `x`; `printf("%d\n",*px)` imprime el valor actual de `x`; y `d=sqrt((double)*px)` devuelve en `d` la raíz cuadrada de `x`, que se convierte al tipo `double` antes de calcularse.

En expresiones como `y=*px+1` los operadores unitarios * y & tienen mayor precedencia que los operadores aritméticos, por lo que al evaluar la expresión se toma el valor del objeto al que apunta `px`, se le suma 1, y el resultado se asigna a `y`.

También pueden aparecer referencias a punteros en la parte izquierda de una asignación. Si `px` apunta a `x`, entonces `*px=0` pone x a cero, y `*px+=1` incrementa el valor de x al igual que `(*px)++`. En este último ejemplo se necesitan los paréntesis; sin ellos la expresión incrementaría `px` y no al objeto al que apunta, ya que los operadores unitarios como * y ++ se evalúan de derecha a izquierda.

Y por último, puesto que los punteros son variables, se pueden manipular igual que cualquier otra variable. Si `px` es otro puntero a enteros, `py=px` copia el contenido de `px` en `py`, con lo que se consigue que `py` apunte al mismo objeto que `px`.

### *2.2 Aritmética de direcciones.*

Si `p` es un puntero, `p++` incrementa `p` para que apunte al siguiente elemento de los que apunta `p`, y `p+=i` incrementa `p` para que apunte al i-ésimo objeto más allá del que apunta `p`. Estas construcciones y otras similares son las formas más simples y comunes de la aritmética de direcciones.

El lenguaje C es consistente y formal en su manejo de la aritmética de direcciones; la integración de punteros, arrays y aritmética de direcciones es una de sus principales virtudes.

En general, un puntero se puede inicializar como cualquier otra variable, aunque normalmente los únicos valores significativos son `NULL` o una expresión en que aparezcan direcciones de objetos del tipo apropiado definidas previamente.

El lenguaje C garantiza que un puntero que apunte a datos válidos nunca tendrá valor cero. `NULL` se puede emplear para señalar una condición anormal. Se escribe `NULL`

en lugar de cero, sin embargo, para indicar más claramente que es un valor especial para punteros. En general, no tiene sentido asignar enteros a punteros; el cero es un caso especial.

Los punteros se pueden comparar en ciertas circunstancias. Si `p` y `q` apuntan a miembros de un mismo array, se pueden utilizar las relaciones como <, >=, etc. La expresión `p<q` tiene el valor de cierto, por ejemplo, si `p` apunta a un elemento anterior al que apunta `q`. También se pueden emplear las relaciones `==` y `!=`. Todo puntero se puede comparar por igualdad o desigualdad con el valor `NULL`.

### *2.3 Punteros como argumentos de funciones.*

Los punteros como argumentos suelen emplearse en el caso de funciones que devuelven más de un valor simple dado que C pasa los argumentos de las funciones por valor. La técnica consiste en usar como argumentos punteros a las variables que se desean modificar, de forma que lo que realmente es pasado por valor es la dirección de memoria de la variable, pudiendo de esta forma modificar el contenido de dichas direcciones.

### *2.4 Punteros a funciones.*

En C una función no es una variable, pero es posible definir un puntero a una función, que puede ser manipulado, pasado a funciones, colocado en arrays, etc. Ilustraremos esto mediante un procedimiento de clasificación u ordenamiento, de modo que si se aporta el argumento opcional -n ordene las líneas de entrada numéricamente en lugar de hacerlo lexicográficamente.

Un ordenamiento a menudo consta de tres partes (una comparación que determina el orden de cualquier par de elementos, un intercambio que invierte el orden y un algoritmo de ordenamiento que realiza comparaciones e intercambia hasta que los objetos estén ordenados). El algoritmo de ordenamiento es independiente de las operaciones de comparación e intercambio, por lo que pasándole diferentes funciones para comparar e intercambiar, podremos efectuar la clasificación con distintos criterios. Esto es lo que haremos en nuestro programa.

La comparación lexicográfica de dos líneas se hace usando `strcmp` y el intercambio mediante `swap`. También necesitaremos una rutina `numcmp` que compare dos líneas con base en un valor numérico y devuelva el mismo tipo de indicación que `strcmp`. Estas tres funciones se declaran en `main`, pasándole a `sort` un puntero a ellas. A su vez, `sort` llama las funciones a través de los punteros. Hemos eliminado el tratamiento de errores en los parámetros, para así concentrarnos en las cuestiones principales.

```
#define LINES 100 /* número máximo de líneas a ordenar */

int main(int argc,char *argv[])
{
   char *lineptr[LINES]; /* punteros a las líneas de entrada */
   int nlines; /* número de líneas leídas */
   int strcmp(), numcmp(); /* funciones de comparación */
   int swap(); /* función de intercambio */
   int numeric=0; /* 1 si se ordena números */

   if(argc>1 && argv[1][0]=='-' && argv[1][1]=='n')
      numeric=1;
   if((nlines=readlines(lineptr,LINES))>=0)
   {
      if(numeric)
         sort(lineptr,nlines,numcmp,swap);
      else
         sort(lineptr,nlines,strcmp,swap);
      writelines(lineptr,nlines);
   }
   else
      printf("Entrada demasiado larga para ordenar\n");
}
```

`strcmp`, `numcmp` y `swap` son direcciones de funciones; como se sabe que son funciones es innecesario el operador &, del mismo modo que no se necesita antes de un nombre de array. El compilador deduce que se ha de pasar la dirección de la función.

```
int sort(char *v[],int n,int (*comp)(),int (*exch)())
{
   int gap,i,j;

   for(gap=n/2;gap>0;gap/=2)
      for(i=gap;i<n;i++)
         for(j=i-gap;j>=0;j-=gap)
         {
            if((*comp)(v[j],v[j+gap])<=0)
               break;
            (*exch)(&v[j],&v[j+gap]);
         }
}
```

Las declaraciones se deben examinar con atención: `int (*comp)()` indica que `comp` es un puntero a una función que devuelve un entero. Es necesaria la primera pareja de paréntesis. Sin ellos `int *comp()` diría que `comp` es una función que devuelve un puntero a un entero, lo cual es una cosa muy diferente.

# 3 Manipulación de estructuras de datos.

## *3.1 Arrays.*

### 3.1.1 Arrays unidimensionales.

En C existe una estrecha relación entre punteros y arrays, suficientemente estrecha como para que se los trate simultáneamente. Cualquier operación que se pueda realizar mediante la indexación de un array se puede realizar también con punteros.

La declaración

```
int a[10]
```

define un array de tamaño 10, es decir un bloque de 10 objetos consecutivos denominados `a[0]`, `a[1]`, ... , `a[9]`. La notación `a[i]` significa el elemento del array que se encuentra a `i` posiciones del comienzo. Si `pa` es un puntero a un entero, declarado como `int *pa` entonces la asignación `pa=&a[0]` hace que `pa` apunte al elemento cero de `a`; es decir, `pa` contiene la dirección de `a[0]`. Ahora la asignación `x=*pa` copiará el contenido de `a[0]` en `x`.

Si `pa` apunta a un elemento particular de un array `a`, entonces por definición `pa+1` apunta al siguiente elemento, y en general `pa-i` apunta a `i` elementos antes de `pa`, y `pa+i` apunta `i` elementos después. Si `pa` apunta a `a[0]`, entonces `*(pa+1)` se refiere al contenido de `a[1]`, `pa+i` es la dirección de `a[i]` y `*pa+i` es el contenido de `a[i]`.

La definición de "sumar 1 a un puntero", y, por extensión, toda la aritmética de punteros establece que el incremento se adecua al tamaño en memoria del objeto apuntado. En `pa+i`, `i` se multiplica por el tamaño de los objetos a los que apunta `pa` antes de ser sumado a `pa`.

La correspondencia entre indexación y aritmética de punteros es muy estrecha. El compilador convierte toda referencia a un array en un puntero al comienzo del array. El efecto es que el nombre de un array es una expresión de tipo puntero. Puesto que el nombre de un array es sinónimo de la posición del elemento cero, la asignación `pa=&a[0]` se puede escribir como `pa=a`.

Una referencia a `a[i]` se puede escribir también como `*(a+i)`. Al evaluar `a[i]`, C lo convierte en `*(a+i)` inmediatamente; las dos formas son completamente equivalentes. Al aplicar el operador `&` a las dos partes de esta equivalencia se deduce que `&a[i]` y `a+i` también son idénticas: `a+i` es la dirección del i-ésimo elemento de `a`. Por otra parte, si `pa` es un puntero, en las expresiones puede aparecer con un subíndice: `pa[i]` es idéntico a `*(pa+i)`. En suma, cualquier expresión en que aparezca un array y un subíndice se puede escribir como un puntero y un desplazamiento y viceversa, incluso en la misma proposición.

Sin embargo, hay una diferencia entre el nombre de un array y un puntero. Un puntero es una variable, por lo que operaciones como `pa=a` y `pa++` son correctas. Pero el nombre de un array es una constante, no una variable; de ahí que las construcciones como `a=pa` o `a++` o `p=&a` sean ilegales.

Cuando se pasa el nombre de un array a una función, se pasa la dirección del comienzo del array. En la función, este argumento es una variable, igual que cualquier otra, por lo que el nombre de un array como argumento es un puntero, o sea una variable que contiene una dirección.

También es posible pasar parte de un array a una función, pasando un puntero al comienzo del subarray. Por ejemplo, si `a` es un array `f(&a[2])` y `f(a+2)` pasan a la función `f` la dirección del elemento `a[2]`, ya que `&a[2]` y `a+2` son expresiones de tipo puntero que referencian el tercer elemento de `a`.

### 3.1.2 Arrays multidimensionales.

El lenguaje C dispone de arrays rectangulares multidimensionales, aunque en la práctica son mucho menos usados que los arrays de punteros. En C, un array bidimensional es realmente un array unidimensional por definición, cada uno de cuyos

elementos es un array. Por este motivo, los subíndices se escriben como `array[i][j]` en lugar de `array[i,j]` como en la mayor parte de los lenguajes. Aparte de esto, un array bidimensional puede ser tratado prácticamente de la misma forma que en otros lenguajes. Los elementos se almacenan por filas, es decir, el subíndice situado más a la derecha varía más rápidamente cuando se accede a los elementos por orden de almacenamiento.

La inicialización de un array se efectúa mediante una lista de valores situados entre llaves. Cada fila de un array bidimensional se inicializa mediante una correspondiente sublista.

En caso de que se deba transferir un array a una función, la declaración del argumento en la función deberá incluir el número de columnas. El número de filas es irrelevante pues lo que se trasmite realmente es, como antes, un puntero.

Dadas las declaraciones

```
int a [10][10];
int *b[10];
```

la utilización de `a` y `b` puede ser parecida, desde el momento en que `a[5][5]` y `b[5][5]` son, ambas, referencias válidas a un entero. Pero `a` es un array verdadero: existen 100 celdas de memoria asignadas y se efectúa el cálculo de subíndices rectangulares convencional para localizar un elemento dado. Sin embargo, a `b` la declaración sólo le asigna 10 punteros, cada uno de los cuales debe hacerse que apunte a un array de enteros. Suponiendo que cada uno apunta a un array de diez elementos, ocupará 100 celdas de memoria más las diez celdas de los punteros. Por tanto, el array de punteros utiliza un poco más de espacio, pudiendo necesitar una inicialización explícita. Pero tiene dos ventajas: el acceso a un elemento se efectúa mediante una indirección a través de un puntero, en lugar de hacerlo mediante una multiplicación y una suma, y las filas del array pueden ser de diferentes longitudes.

### *3.2 Estructuras.*

Una estructura es un conjunto de una o más variables, posiblemente de tipos diferentes, agrupadas bajo un mismo nombre para hacer más eficiente el manejo. Las estructuras ayudan a organizar datos complicados, particularmente en programas grandes, ya que en muchas situaciones permiten tratar como unidad un conjunto de variables relacionadas, en lugar de tratarlas como entidades independientes.

#### 3.2.1 Conceptos básicos.

Una fecha tiene varios componentes, día, mes, año, día del año y nombre del mes. Estas cinco variables se pueden agrupar en una estructura como la siguiente:

```
struct date
{
   int day;
   int month;
   int year;
   int yearday;
   char mon_name[4];
};
```

La palabra clave `struct` introduce la declaración de una estructura, que no es más que una lista de declaraciones encerradas entre llaves. Opcionalmente puede seguir

un nombre a la palabra clave `struct`; se lo denomina nombre de la estructura y se puede emplear en declaraciones posteriores como una abreviatura de la estructura.

Los elementos o variables citados en una estructura se denominan miembros. Un miembro de una estructura, el nombre de una estructura y una variable ordinaria pueden tener el mismo nombre; siempre se pueden distinguir a través del contexto. Por supuesto una regla de buen estilo especifica que sólo se debe usar el mismo nombre para objetos estrictamente relacionados entre sí.

La llave de cierre que termina la lista de miembros puede ir seguida de una lista de variables, como si se tratara de un tipo básico. Si la declaración de una estructura no va seguida de la lista de variables, no se reserva memoria alguna; en este caso se está describiendo una plantilla de la estructura. Si la estructura tiene muchos nombres, dicho nombre se puede utilizar posteriormente en nuevas declaraciones de variables.

Se puede inicializar una estructura externa o estática añadiendo a su definición la lista de inicializadores de los componentes:

```
struct date d={4,7,1776,186,"Jul"};
```

Una estructura automática también se puede inicializar por asignación o llamando a una función que devuelve una estructura del tipo adecuado.

Para referenciar un miembro de una estructura en una expresión, se emplea una construcción de la forma *nombre_de_la_estructura.miembro*. El operador miembro de estructura "." conecta el nombre de la estructura y el del miembro.

Las estructuras se pueden anidar; un registro de una nómina puede ser algo así como

```
struct person
{
   char name[NAMESIZE];
   char address[ADRSIZE];
   long zipcode;
   long ss_number;
   double salary;
   struct date birthdate;
   struct date hiredate;
};
```

La estructura `person` contiene dos fechas. Si declaramos `emp` como `struct person emp;`, entonces `emp.birthdate.month` se refiere al mes del nacimiento. El operador de miembro de estructura es asociativo de izquierda a derecha.

### 3.2.2 Estructuras y funciones.

Las únicas operaciones legales sobre una estructura son copiarla o asignarla como unidad, tomar su dirección con `&`, y tener acceso a sus miembros. La copia y la asignación incluyen pasarlas como argumentos a funciones y devolver valores de funciones. Las estructuras no se pueden comparar.

Si una estructura grande va a ser pasada a una función, normalmente es más eficiente pasar un puntero que copiar la estructura completa. Por ello, se usan con tanta frecuencia que se ha proporcionado una notación alternativa para su manejo. Si `p` es un puntero a una estructura, entonces `p->miembro_de_estructura` es equivalente a `(*p).miembro_de_estructura` y referencia un miembro particular.

Los operadores de estructuras -> y . junto con los paréntesis de las listas de argumentos y los corchetes de las indexaciones tienen la máxima precedencia entre todos los operadores.

### 3.2.3 Arrays de estructuras.

La inicialización de una estructura es análoga a la de otros objetos (a la definición sigue la lista de inicializadores delimitada por llaves). Los inicializadores se agrupan por miembros de una estructura. Se puede encerrar entre llaves los inicializadores de cada fila (estructura), pero estas llaves no son necesarias si los inicializadores son variables simples o cadenas de caracteres y además si están todos presentes. La dimensión del array la calcula el compilador si están presentes los inicializadores y [] se deja vacío.

### 3.2.4 Punteros a estructuras.

Si p es un puntero a una estructura, toda operación aritmética que se realice con él tendrá en cuenta el tamaño de la estructura, por lo que p++ incrementa p en la cantidad adecuada para que apunte al siguiente elemento del array de estructuras. No se debe suponer que el tamaño de una estructura es la suma de los tamaños de los miembros, ya que los requisitos de alineación de los diferentes objetos pueden originar "huecos" en la estructura.

### 3.2.5 Uniones.

Una unión es una variable que puede contener, en distintos momentos, objetos de tipos y tamaños distintos. Las uniones son una herramienta con la que se manipulan diferentes tipos de datos en una única zona de memoria, sin introducir en el programa información dependiente de la máquina.

Una unión tiene por objeto proporcionar una variable que contenga legítimamente diferentes tipos. Su sintaxis se basa en las estructuras. Por ejemplo,

```
union u_tag
{
   int ival;
   float fval;
   char *pval;
} uval;
```

donde la variable uval tiene que ser lo suficientemente grande como para contener el mayor de los tres tipos, sin importar la computadora que se esté utilizando (el código es independiente de las características del hardware). Cualquiera de estos tipos puede asignarse a uval y usarse en expresiones, en tanto su empleo sea consistente: el tipo utilizado debe ser el más recientemente almacenado. Es responsabilidad del programador recordar cuál es el tipo que hay en la unión.

Sintácticamente se tiene acceso a los miembros de una unión como *nombre_unión.miembro* o *puntero_a_unión->miembro* igual que con las estructuras.

Las uniones pueden aparecer dentro de estructuras y arrays, o viceversa. La notación para acceder a un miembro de una unión dentro de una estructura (o viceversa) es idéntica a la empleada con estructuras anidadas.

Una unión es una estructura donde todos los miembros tienen desplazamiento cero. La estructura es lo suficientemente grande como para contener el mayor de los

miembros, y la alineación es la apropiada para todos los tipos de la unión. Al igual que las estructuras, las únicas operaciones permitidas con las uniones son tener acceso a un miembro y tomar su dirección; las uniones no se pueden asignar, ni ser pasadas a funciones, ni ser devueltas por funciones. Los punteros a uniones pueden usarse de forma idéntica a los punteros a estructuras.

### 3.2.6 Typedef.

C dispone de una declaración denominada `typedef` para la creación de nuevos nombres de tipos de datos. Por ejemplo, la declaración

```
typedef int LENGTH;
```

propone el nombre LENGTH como un sinónimo de `int`. El "tipo" LENGTH se puede emplear en declaraciones, castings, etc., de la misma manera que se haría con el tipo `int`.

Vamos a resaltar que una declaración `typedef` no crea un nuevo tipo en ningún sentido; simplemente añade un nuevo nombre a uno ya existente. Tampoco se amplía la semántica: las variables así declaradas tienen exactamente las mismas propiedades que si se hubieran declarado explícitamente. En realidad, `typedef` se comporta como `#define`, excepto que al ser interpretado por el compilador puede efectuar las sustituciones textuales que exceden las capacidades del preprocesador de C.

Hay dos razones importantes por las que se utilizan declaraciones `typedef`. La primera es para parametrizar un programa contra problemas de portabilidad. Si se utiliza `typedef` con los tipos de datos que pueden ser dependientes de la instalación, sólo se tendrán que cambiar los `typedef` cuando se lleve el programa a otro computador. Una práctica común es emplear `typedef` para las cantidades enteras y luego hacer las elecciones apropiadas de `short`, `int` y `long` en cada computadora. El segundo propósito de `typedef` es facilitar la documentación de un programa.

### 3.2.7 Estructuras dinámicas.

*Estructuras autorreferenciadas.*

Supongamos que deseamos resolver el problema de contar las ocurrencias de todas las palabras de algún texto. Puesto que de entrada no conocemos la lista de palabras, no podemos ordenarlas para emplear un algoritmo de búsqueda binaria. Tampoco podemos realizar una búsqueda lineal de cada palabra que encontremos ya que el programa podría tardar demasiado en ejecutarse. Una solución es mantener en todo momento ordenadas las palabras, colocando en su sitio correspondiente cada palabra que llega. Esto no podemos hacerlo desplazando las palabras en un array lineal, pues se emplearía mucho tiempo. En lugar de esto se va a utilizar una estructura de datos denominada árbol binario.

El árbol contiene un nodo por cada palabra diferente; cada nodo contiene: un puntero a los caracteres de la palabra, un contador del número de ocurrencias, un puntero a su hijo izquierdo (que es un nodo), y un puntero a su hijo derecho (que también es un nodo). Ningún nodo puede tener más de dos hijos, pudiendo tener cero o uno.

Los nodos se relacionan entre sí de la siguiente manera: dado un nodo, el subárbol izquierdo de este nodo contiene palabras que son (lexicográficamente) menores que la palabra del nodo, y el subárbol derecho contiene palabras que son

mayores que la palabra del nodo. Para averiguar si una palabra ya está en el árbol, se comienza examinando la raíz y comparando la nueva palabra con la almacenada en ese nodo. Si coinciden es que la palabra ya está en el árbol. Si la palabra es menor que la del nodo, se continúa la búsqueda por el subárbol izquierdo; si la palabra es mayor, es investigada en el subárbol derecho. Si no existe descendiente en la dirección requerida, ello quiere decir que la palabra no existe en el árbol y que su sitio correcto en el árbol es el del descendiente que no existe. Este proceso de búsqueda es inherentemente recursivo puesto que la búsqueda en cualquier nodo se realiza de igual manera en sus hijos. La inserción y la impresión se realizan de una forma natural mediante rutinas recursivas.

Volviendo a la descripción de un nodo, claramente se describe mediante una estructura con cuatro componentes:

```
struct tnode
{
   char *word; /* puntero a los caracteres */
   int count; /* número de ocurrencias */
   struct tnode *left; /* hijo izquierdo */
   struct tnode *right; /* hijo derecho */
}
```

Esta declaración "recursiva" del nodo puede parecer sospechosa, pero es totalmente correcta. Es ilegal que una estructura contenga una instancia de si misma, pero se declaran `left` y `right` como punteros a un nodo, no como un nodo en sí.

*Gestión dinámica de memoria.*

La asignación dinámica de memoria es la forma en que un programa puede obtener memoria mientras se está ejecutando. A las variables globales se les asigna memoria en tiempo de compilación. Las variables locales usan la pila. Sin embargo, durante la ejecución de un programa no se pueden añadir variables globales o locales. Pero en algunos casos necesitamos hacer programas que usen cantidades de memoria variables. La memoria obtenida mediante las funciones de asignación dinámica de C se obtiene del montón. Aunque el tamaño del montón es desconocido, generalmente contiene una gran cantidad de memoria libre.

La ventaja de la programación dinámica frente a la estática es que el número de elementos de la estructura de datos se ajusta en cada instante a las necesidades del programa. La estructura dinámica de datos aumenta y disminuye según las operaciones que se lleven a cabo, y cuando ya no es necesaria la estructura, se puede liberar toda la memoria que ocupa.

Los punteros y la asignación dinámica de memoria en C, hacen posible los arrays dinámicos y otras construcciones importantes como las listas enlazadas o los árboles.

Un programa en C crea y usa cuatro regiones de memoria lógicas diferentes que sirven para funciones específicas. La primera región es la memoria que contiene realmente el código del programa. La siguiente región es la memoria donde se guardan las variables globales. Las dos regiones restantes son la pila (*stack*) y el montón (*heap*). La pila se usa para muchas cosas durante la ejecución del programa. Mantiene las direcciones de vuelta para las llamadas a funciones, así como las variables locales. El montón es la región de memoria libre que puede usar el programa mediante las funciones de asignación dinámica para estructuras dinámicas como las listas enlazadas y los árboles.

Vamos a realizar una pequeña incursión en administradores (o gestores) de memoria. En principio parece deseable que sólo exista un administrador de memoria en un programa, aunque maneje diferentes tipos de objetos. Pero si un gestor acepta peticiones de punteros a `char` y a `struct tnode` se plantean dos cuestiones. En primer lugar, ¿cómo satisfacer los requerimientos de casi todas las computadoras que imponen restricciones de alineación a diferentes tipos de objetos? (por ejemplo, los enteros a menudo suelen ubicarse en una dirección par). En segundo lugar, ¿cómo se puede expresar el hecho de que el gestor devuelva diferentes tipos de punteros?

Las restricciones de alineación se suelen satisfacer fácilmente, al coste de cierta pérdida de memoria, si se asegura que el administrador devuelve siempre un puntero que satisface todas las restricciones de alineación.

La declaración del tipo del gestor puede ser un inconveniente para cualquier lenguaje que realice seriamente la comprobación de tipos. En C, el mejor procedimiento es declarar que el gestor devuelve un puntero a `void`, y forzarlo explícitamente al tipo deseado mediante un casting.

El archivo de cabecera `<stdlib.h>` contiene una función denominada `malloc`, cuyo prototipo es:

```
#include <stdlib.h>
void *malloc(size_t size);
```

que se utiliza para asignar memoria dinámica en tiempo de ejecución. `malloc` recibe el número de bytes que necesitan ser asignados, y devuelve un puntero (una dirección) al bloque de bytes asignados si se encuentra memoria disponible. Si no hay memoria disponible para satisfacer la petición, `malloc` devuelve un puntero `NULL`.

C tiene un operador unitario llamado `sizeof` que calcula el tamaño de cualquier objeto en tiempo de compilación. La expresión `sizeof(objeto)` devuelve un entero igual al tamaño del objeto especificado (el tamaño se expresa en unidades no definidas denominadas "bytes", que son del mismo tamaño que un valor `char`). El objeto puede ser una variable, un array o una estructura, o el nombre de un tipo básico como `int` o `double`, o el nombre de un tipo derivado, como por ejemplo una estructura.

Volviendo al ejemplo anterior, el programa principal lee palabras llamando a una función `getword` y las inserta en el árbol mediante `tree`.

```
#define MAXWORD 20

int main() /* cuenta frecuencia de palabras*/
{
   struct tnode *root, *tree();
   char word[MAXWORD];
   int t;

   root=NULL;
   while((t=getword(word,MAXWORD))!=EOF)
      if(t==LETTER)
         root=tree(root,word);
   treeprint(root);
   return 0;
}
```

La función `tree` también es breve. `main` presenta una palabra a comparar comenzando por la raíz del árbol. En cada etapa, esta palabra se compara con la almacenada en el nodo y se prosigue la búsqueda por el subárbol derecho o izquierdo

mediante una llamada recursiva a `tree`. Finalmente la palabra coincide con una que ya está en el árbol (en este caso se incrementa el contador), o se encuentra un puntero nulo, que indica que se debe crear y añadir un nodo al árbol. Si se crea un nodo, `tree` devuelve su puntero para poder instalarlo en el nodo paterno.

```
struct tnode *tree (struct tnode *p, char *w)
{
   struct tnode *talloc();
   char *strsave();
   int cond;

   if (p==NULL)
   {
      if((p=(struct tnode *)malloc(sizeof(struct tnode)))==NULL)
         exit -1; /* No hay suficiente memoria */
      strcpy(p->word,w);
      p->count=1;
      p->left=p->right=NULL;
   }
   else
      if((cond=strcmp(w,p->word))==0)
         p->count++; /* palabra repetida */
      else if(cond<0) /* las menores van al subárbol izquierdo */
         p->left=tree(p->left,w);
      else /* las mayores van al derecho */
         p->right=tree(p->right,w);
   return(p);
}
```

La memoria para crear el nuevo nodo se obtiene llamando a la función `malloc`. Devuelve un puntero a una zona de memoria donde almacenar el nodo. La nueva palabra se copia a una zona vacía mediante `strcpy`, el contador se inicializa, y los dos hijos toman el valor `NULL`. Esta parte de la rutina se ejecuta al alcanzar un extremo del árbol para añadir un nuevo nodo. `treeprint` imprime el árbol en orden simétrico; dado un nodo, se imprime el subárbol izquierdo (todas las palabras menores que la del nodo), el propio nodo, y luego el subárbol derecho (todas las palabras mayores).

```
void treeprint(struct tnode *p)
{
   if(p!=NULL)
   {
      treeprint(p->left);
      printf("%4d %s\n",p->count,p->word);
      treeprint(p->right);
   }
}
```

Por último, nos falta estudiar la forma de liberar la memoria asignada mediante la función `malloc`. La función inversa a `malloc` es `free`

```
#include <stdlib.h>
void free(void *ptr);
```

## 4 Entrada y salida de datos.

Las operaciones de entrada y salida no forman parte del lenguaje C. La biblioteca estándar de entrada/salida es un conjunto de funciones diseñadas para proporcionar un sistema estándar de entrada/salida a los programas C, siendo parte del C estándar ANSI. Se pretende que las funciones presenten una interfaz conveniente para

la programación, y reflejen las operaciones que proporcionan la mayoría de los sistemas operativos modernos.

Las rutinas son lo suficientemente eficientes como para que los usuarios rara vez las eviten por razón de eficiencia, independientemente de cuán importante sea la aplicación. Por último las rutinas están destinadas a ser portables, en el sentido de que deberán existir en un formato compatible con todo sistema que disponga de C, y que los programas que limiten su interacción con el sistema a las facilidades proporcionadas por la biblioteca estándar se puedan trasladar de un sistema a otro sin cambios esenciales.

No vamos a intentar describir la biblioteca completa; nos interesa más mostrar los aspectos esenciales de escribir programas en C que interactúan con el entorno de su sistema operativo.

### *4.1 Acceso a la biblioteca estándar.*

Todo archivo fuente que utilice funciones de la biblioteca estándar deberá contener la línea `#include <stdio.h>` al principio. El archivo `stdio.h` define ciertas macros y variables empleadas por la biblioteca estándar de entrada/salida.

### *4.2 Entrada y salida estándar: `getchar` y `putchar`*

El mecanismo más sencillo de entrada es leer un carácter de la entrada estándar, generalmente la terminal del usuario, con la función:

```
#include <stdio.h>
int getchar(void);
```

Esta función devuelve el siguiente carácter de la entrada cada vez que se la llama, y el valor EOF cuando encuentra el fin de archivo en la entrada de la que esté leyendo. La biblioteca estándar, define la constante simbólica EOF como -1, pero las comparaciones deben realizarse en términos de EOF para hacerlas independientes del valor específico.

Para salida, la función:

```
#include <stdio.h>
int putchar(int c);
```

envía el carácter c a la salida estándar, que por definición también es la terminal.

### *4.3 Entrada y salida con formato: `printf` y `scanf`.*

Las rutinas `printf` para salida y `scanf` para entrada permiten la traducción de cantidades numéricas en caracteres y viceversa. También permiten la generación o interpretación de líneas con formato. La función de salida que convierte, da formato e imprime sus argumentos en la salida estándar, bajo el control de la cadena de formato es:

```
#include <stdio.h>
int printf(const char *formato [[,argumento]] ... );
```

La cadena de formato contiene dos tipos de objetos: caracteres ordinarios, que simplemente se copian a la salida estándar, y especificaciones de conversión, cada una de las cuales origina la conversión e impresión del siguiente argumento de `printf`.

Cada especificación de conversión comienza con el carácter `%` y acaba en un carácter de conversión. Entre el carácter `%` y el carácter de conversión puede haber: un

signo menos, que indica ajustar a la izquierda el argumento de este campo; una cadena de dígitos, que indica el tamaño mínimo del campo; un punto, que separa el campo de la siguiente cadena de dígitos; una cadena de dígitos (la precisión), que indica el número máximo de caracteres que se imprimirán de la cadena, o el número de dígitos que se deben imprimir a la derecha del punto decimal de un valor `float` o `double`. Un modificador `l` (ele) de la longitud, que indica que el valor correspondiente es de tipo `long` en lugar de `int`. Los caracteres de conversión y su significado respectivo son:

- d El argumento se convierte a notación decimal.
- o El argumento se convierte a notación octal (sin el cero de relleno).
- x El argumento se convierte a notación hexadecimal sin signo (sin el 0x inicial).
- u El argumento se convierte a notación decimal sin signo.
- c El argumento se toma como un carácter.
- s El argumento es una cadena; se imprimen los caracteres de la cadena hasta encontrar el carácter nulo o alcanzar el número de caracteres indicado en la especificación de precisión.
- e El argumento se toma como un valor `float` o `double` y se convierte a notación decimal con el formato [-]m.nnnnnn E[±]xx donde la longitud de la cadena de enes viene determinada por la precisión (por defecto 6).
- f El argumento se toma como `float` o `double` y se convierte a la notación decimal de la forma [-]mmm.nnnnn donde la longitud de la cadena de enes viene indicada por la precisión (por defecto 6).
- g Emplea el más corto de `%e` o `%f`, los ceros no significativos no se imprimen.

Si el carácter que sigue a `%` no es uno de los anteriores, dicho carácter se imprime, por lo que `%` se imprime con `%%`.

La función equivalente a `printf` pero para la entrada que lee los caracteres, los interpreta de acuerdo con el formato especificado, y almacena los resultados en los restantes argumentos es:

```
#include <stdio.h>
int scanf(const char *formato [[,argumento]] ... );
```

donde cada uno de los argumentos debe ser un puntero que indica donde almacenar el resultado de la conversión.

La cadena de formato contiene usualmente especificaciones de conversión, que se utilizan para dirigir la interpretación de las secuencias de la entrada. La cadena de formato puede contener: blancos, tabuladores o fines de línea ("espacio en blanco" genérico), que no se tienen en cuenta; caracteres ordinarios (no `%`) que se espera que coincidan con el próximo carácter `%`, distinto de blanco del flujo de entrada; especificaciones de conversión, formadas por el carácter `%`, un carácter opcional de suspensión de asignación `*`, un número opcional que indica el tamaño máximo del campo y el carácter de conversión.

Una especificación de conversión dirige la conversión del próximo campo de la entrada. El resultado se almacena normalmente en la variable a la que apunta el argumento correspondiente. Si se indica supresión de la asignación mediante el carácter `*`, no se toma en cuenta el campo de entrada, no se realiza la asignación. Un campo de

entrada se define como una cadena de caracteres delimitada por "espacio en blanco" (blancos, tabuladores y fin de línea); abarca hasta el primer blanco o tantos caracteres como indique el tamaño del campo. Ello quiere decir que `scanf` leerá más de una línea si esto es necesario para cumplir con el tamaño del campo, ya que los terminadores de renglón se consideran como "espacio en blanco".

El carácter de conversión indica la interpretación del campo de entrada; el argumento correspondiente debe ser un puntero según la semántica de las llamadas por valor de C. Son válidos los siguientes caracteres de conversión:

- d Se espera un número decimal en la entrada; el argumento correspondiente debe ser un puntero a un entero.
- o Se espera un entero octal (con o sin cero a la izquierda) en la entrada. El argumento correspondiente debe ser un puntero a un entero.
- x Se espera un entero hexadecimal (con o sin ceros) en la entrada. El argumento correspondiente debe ser un puntero a un entero.
- h Se espera un entero corto (`short`) en la entrada. El argumento correspondiente debe ser un puntero a un entero corto.
- c Se espera un carácter. El argumento correspondiente debe ser un puntero a caracteres; el siguiente carácter de la entrada se almacena en el lugar indicado. Se suprime el salto normal sobre el espacio en blanco. Para leer el siguiente carácter distinto de blanco se utiliza `%1s`.
- s Se espera una cadena de caracteres. El argumento correspondiente debe ser un puntero a un array de caracteres suficientemente grande como para contener la cadena y un carácter /0 de terminación.
- f Se espera un número en punto flotante. El argumento correspondiente debe ser un puntero a `float`. El carácter de conversión `e` es un sinónimo de `f`. El formato de entrada de reales es un signo opcional, una cadena de dígitos que puede contener un campo de exponente opcional con una E o e seguida de un entero que puede ir con signo.

Los caracteres de conversión `d`, `o` y `x` pueden ir precedidos por la letra `l` (ele) para indicar que el argumento correspondiente es un puntero a `long` en lugar de a `int`. De igual manera, los caracteres de conversión `e` o `f` pueden estar precedidos por la `l` (ele) para indicar un puntero a `double` en lugar de a `float` en la lista de argumentos.

### *4.4 Conversión de formatos en la memoria.*

Las funciones `printf` y `scanf` tienen funciones hermanas llamadas `sprintf` y `sscanf` que realizan las mismas conversiones pero que operan en una cadena de caracteres. Su prototipo es:

```
#include <stdio.h>
int sprintf(char *buffer, const char *formato [[,argumento]] ... );
int sscanf(char *buffer, const char *formato [[,argumento]] ... );
```

La función `sprintf` formatea los argumentos de acuerdo con la cadena de formato, pero deja el resultado en `buffer`, y no en la salida estándar. La función `sscanf` realiza las conversiones inversas, explora `buffer` de acuerdo con el formato especificado, y coloca los valores resultantes en los argumentos que deben ser punteros.

### *4.5 Acceso a archivos.*

Las reglas son sencillas. Antes de leer o escribir en un archivo hay que *abrirlo* mediante la función `fopen` de la biblioteca estándar. `fopen` acepta como argumento un nombre externo de archivo, realiza operaciones y negociaciones con el sistema operativo (cuyos detalles no nos preocupan) y devuelve un nombre interno que deberá emplearse en subsecuentes lecturas o escrituras del archivo.

El nombre interno es un puntero a una estructura que mantiene información sobre el archivo tal como la dirección del buffer, el último carácter leído del buffer, el hecho de que el archivo se va a leer o escribir, etc. Los usuarios no necesitan conocer estos detalles ya que dicha estructura es parte de las definiciones contenidas en `stdio.h`, y se llama `FILE`. Sólo es necesario declarar el puntero al archivo. `FILE` es el nombre de un tipo, no un nombre de una estructura.

Eso es justo lo que ocurre al comenzar la ejecución de un programa, se abren tres archivos automáticamente y se establecen tres punteros para ellos: `stdin`, `stdout` y `stderr` que corresponden a la entrada estándar, la salida estándar y la salida estándar de errores respectivamente. Estos punteros se pueden utilizar en cualquier punto donde aparezca un objeto del tipo `FILE *`, ya que son constantes, no variables, por lo que no se les puede asignar ningún valor.

#### 4.5.1 Apertura del fichero.

Para abrir un fichero la función utilizada es `fopen`, y su prototipo es:

```
#include <stdio.h>
FILE *fopen(const char *nombre_fich, const char *tipo_apertura);
```

Esta función devuelve un puntero a un fichero que se asigna a una variable de tipo `FILE *`. Si existe algún tipo de error al realizar la operación, por ejemplo porque se desee abrir para lectura y éste no exista, devuelve el valor `NULL`.

Los parámetros que utiliza son:

- *nombre_fich*: que es el nombre que tiene el fichero para el sistema operativo.
- *tipo_apertura*: que es una cadena de caracteres que indica el tipo del fichero, texto o binario, y el uso que se va a hacer de él (lectura, escritura, añadir datos, etc.).

Los tipos de apertura existentes se combinan para conseguir abrir el fichero en el modo adecuado.

#### 4.5.2 Lectura/escritura de caracteres y de líneas.

Lo siguiente que necesitamos es una forma de leer o escribir en el archivo una vez abierto. La operación de salida más sencilla sobre un fichero es la escritura en ficheros de texto. Las funciones básicas de escritura de texto son:

```
#include <stdio.h>
int fputc(int c, FILE *stream);
```

que escribe el carácter `c` en el fichero y devuelve el carácter escrito `c` si la operación fue correcta o `EOF` en caso contrario.

```
#include <stdio.h>
int fputs(const char *s, FILE *stream);
```

que escribe la cadena de caracteres `s` (sin incluir el carácter de fin de cadena \0) en el fichero, y devuelve el último carácter escrito si la operación fue correcta o `EOF` en caso contrario.

```
#include <stdio.h>
int fprintf(FILE *stream, const char *formato [[,argumento]] ... );
```

que es similar a la función `printf`. Escribe la cadena de caracteres especificada por el formato en el fichero, sustituyendo en la misma los valores de los argumentos.

Las operaciones de lectura correspondientes a las funciones anteriores son:

```
#include <stdio.h>
int fgetc(FILE *stream);
```

que devuelve el carácter leído del fichero convertido en entero si la operación fue correcta o `EOF` en caso contrario.

```
#include <stdio.h>
char *fgets(char *s, int num_caracteres, FILE *stream);
```

que lee `num_caracteres` del fichero o hasta que encuentra un salto de línea, y los almacena en `s`, añadiendo el carácter de fin de cadena \0. Si no hay error devuelve un puntero a la cadena `s` y si hay error o se ha llegado al fin del fichero devuelve `NULL`.

```
#include <stdio.h>
int fscanf(FILE *stream, const char *formato [[,argumento]] ... );
```

que es similar a la función `scanf`. Lee del fichero los valores de los argumentos colocando los mismos en las direcciones y adaptando la lectura al formato especificado.

Por último, debemos mencionar que existe una función cuyo objetivo es devolver el último carácter leído `c` al fichero, de forma que esté disponible en la próxima ocasión que se realice una lectura. Esta función es:

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

y devuelve `c` en caso de éxito o `EOF` en caso contrario. Los caracteres puestos en el fichero deben ser devueltos en orden inverso, garantizándose sólo que se pueda devolver al fichero un carácter.

### 4.5.3 Cierre del fichero.

Cuando el fichero se deja de utilizar, para liberar la memoria principal que ocupa es necesario cerrarlo. La función para cerrar un fichero es `fclose`, siendo su prototipo:

```
#include <stdio.h>
int fclose(FILE *stream);
```

### 4.5.4 Escritura y lectura por bloques.

Cuando se utilizan dispositivos que no permiten la utilización de las funciones de texto, o cuando se desea utilizar ficheros binarios, las operaciones de acceso al fichero que se deben utilizar son las de escritura y lectura de bloques de datos que graban o leen una serie de bytes en el fichero en formato binario. Esto no quita que también se puedan utilizar estas funciones para escribir o leer texto. Además, las funciones por bloques son más eficientes que las de texto. La función para escribir datos en formato binario en un fichero es:

```
#include <stdio.h>
size_t fwrite(const void *datos, size_t tamano, size_t num, FILE
*stream);
```

que escribe en el fichero tantos datos como indica num, cada uno de los cuales será del tamaño tamano. Los datos escritos los toma de la variable datos. La función devuelve el número de datos escritos, de forma que si el valor devuelto es menor que num indica que se ha producido un error.

La función para leer datos en formato binario de un fichero es:

```
#include <stdio.h>
size_t fread(void *datos, size_t tamano, size_t num_datos, FILE
*stream);
```

que lee del fichero tantos datos como indica num_datos, cada uno de los cuales será del tamaño tamano y los almacena en datos. La función devuelve el número de datos leídos, de forma que si el valor devuelto es menor que num_datos indica o que se ha llegado al final del fichero o que se ha producido un error.

### 4.5.5 Posicionamiento.

Para poder consultar y/o actualizar los datos contenidos en un fichero es necesario disponer de algún mecanismo que proporcione la situación del dato dentro del fichero, así como la disponibilidad de desplazarse en torno a él. En los ficheros existe un puntero de acceso que apunta al siguiente dato que será accedido para leerlo o escribir en él. Los canales permiten un acceso directo, es posible leer datos en cualquier posición. Los canales, como puertos, terminales, etc. que no permiten este tipo de acceso no pueden utilizar las funciones de posicionamiento. La función para colocar el puntero de acceso a un fichero es:

```
#include <stdio.h>
int fseek(FILE *stream, long int posicion, int origen);
```

que coloca el puntero del fichero a tantos bytes del origen como indica posicion contando a partir del origen indicado. Los orígenes posibles son: SEEK_SET (principio del fichero), SEEK_CUR (posición actual), y SEEK_END (final del fichero).

Entre las funciones más importantes asociadas al posicionamiento en ficheros, destacan las siguientes:

```
#include <stdio.h>
void rewind(FILE *stream);
```

que coloca el puntero a principio del fichero.

```
#include <stdio.h>
long int ftell(FILE *stream);
```

que devuelve la posición actual en bytes del puntero del fichero.

### 4.5.6 El manejo de errores.

Después de la ejecución de una llamada al sistema o de ciertas funciones desde un programa en C, el código del error producido se almacena en la variable errno.

- Si hay error, dicha variable tendrá un valor que está definido por unas constantes del sistema que identifican claramente los errores y que dependen

de la función ejecutada (dichos errores se encuentran en el fichero `errno.h`).

- Si no hay error, tendrá un valor desconocido, generalmente, el del último error producido.

La mayoría de las funciones, por tanto, suelen devolver el valor 0 cuando han realizado su operación correctamente o el valor -1 cuando ha existido algún error. Los errores pueden, generalmente, ser mostrados en pantalla utilizando la función `perror`,

```
#include <stdio.h>
void perror(const char *s);
```

que permite mostrar un mensaje sobre el error producido en la salida estándar de errores, ya que recibe una cadena de caracteres y muestra dicha cadena seguida de ":" y un mensaje indicativo del último error producido.

### *4.6 Manejo de errores: `exit`.*

La función `exit` fuerza el fin de la ejecución del programa cuando se la llama.

```
#include <stdlib.h>
void exit(int estado);
```

El argumento de `exit` está disponible para el proceso que llamó a éste, por lo que se puede comprobar si un programa acabó bien o mal. Por convención, el 0 indica que todo ha ido bien, mientras que valores diferentes de 0 se emplean para indicar diferentes situaciones anormales.

### *4.7 Funciones de operación con cadenas.*

La biblioteca estándar dispone de una amplia gama de funciones, de entre las cuales veremos a continuación las más importantes para operaciones con cadenas:

```
#include <string.h>
size_t strlen(const char *cadena);
```

Devuelve la longitud en bytes de la cadena, sin incluir el carácter /0 de fin de cadena.

```
#include <string.h>
char *strcpy(char *cadena1, const char *cadena2);
```

Copia cadena2, incluyendo el carácter /0 de fin de cadena, en la dirección especificada por cadena1, y devuelve cadena1.

```
#include <string.h>
char *strcat(char *cadena1, const char *cadena2);
```

Añade cadena2 a cadena1 y almacena el resultado en la dirección especificada por cadena1, añadiendo al final el carácter /0 de fin de cadena. Devuelve cadena1.

```
#include <string.h>
int strcmp(const char *cadena1, const char *cadena2);
```

Compara lexicográficamente cadena1 y cadena2, devolviendo 0 si ambas cadenas son iguales, un número menor que 0 si cadena1 es menor que cadena2, y un número mayor que 0 si cadena1 es mayor que cadena2.

## 5 Gráficos en C.

Al igual que ocurre con las operaciones de entrada y salida, el lenguaje C no incluye gráficos. Además, y a diferencia de las operaciones de entrada y salida, no existen librerías de funciones estándar a este respecto, por lo que los gráficos en C se facilitan a través de librerías de funciones no estándar con cada producto software concreto.

Existen, no obstante, algunos estándares de librerías de funciones basados en diversos entornos gráficos como por ejemplo OpenGL y DirectX. OpenGL proporciona la especificación de un interfaz de diseño de alto nivel. En contraste, DirectX es de bajo nivel y más difícil de programar, pero obtiene un mejor rendimiento en máquinas sin aceleración hardware. Existen muchas librerías que implementan el interfaz OpenGL, de entre las cuales MESA se distribuye bajo licencia GNU.

## 6 Conclusiones.

Un puntero es una variable que contiene la dirección de otra variable. Con ellos se obtiene un código más compacto y eficiente. El operador unario `&` devuelve la dirección de un objeto. El operador unario `*` toma su operando como una dirección y accede a esa dirección para obtener su contenido. En general, un puntero se puede inicializar como cualquier otra variable, aunque normalmente los únicos valores significativos son `NULL` o una expresión en que aparezcan direcciones de objetos del tipo apropiado definidas previamente.

Los punteros se emplean como argumentos para realizar paso de parámetros por referencia dado que C pasa los argumentos de las funciones por valor. Además, en C es posible definir un puntero a una función, que puede ser manipulado, pasado a funciones, colocado en arrays, etc.

En C existe una estrecha relación entre punteros y arrays. Cualquier operación que se pueda realizar mediante la indexación de un array se puede realizar también con punteros. El compilador convierte toda referencia a un array en un puntero al comienzo del array. El efecto es que el nombre de un array es una expresión de tipo puntero. En C, un array bidimensional es realmente un array unidimensional donde cada uno de cuyos elementos es un array.

Las estructuras son un conjunto variables agrupadas bajo un mismo nombre que permiten tratar como unidad un conjunto de variables relacionadas. Una unión es una variable que puede contener, en distintos momentos, objetos de tipos y tamaños distintos. Además, C dispone de una declaración denominada `typedef` para la creación de nuevos nombres de tipos de datos.

Cuando se trabaja con estructuras dinámicas, el compilador asigna una cantidad fija de memoria para mantener la dirección del componente asignado dinámicamente, en vez de hacer una asignación para el componente en sí. Esto implica que debe haber una clara distinción entre datos y referencias a datos y que consecuentemente se deben emplear tipos de datos cuyos valores sean punteros.

La asignación dinámica de memoria es la forma en que un programa puede obtener memoria mientras se está ejecutando. La memoria obtenida mediante las funciones de asignación dinámica de C se obtiene de la memoria montón. Los punteros y la asignación dinámica de memoria en C, hacen posible los arrays dinámicos y otras construcciones importantes como las listas enlazadas o los árboles.

La función `malloc` se utiliza para asignar memoria dinámica en tiempo de ejecución. `malloc` recibe el número de bytes que necesitan ser asignados, y devuelve un puntero al bloque de bytes asignados si se encuentra memoria disponible. Si no hay memoria disponible para satisfacer la petición devuelve un puntero `NULL`. Para liberar la memoria asignada mediante la función `malloc`, se utiliza la función `free`.

C tiene además un operador unitario llamado `sizeof` que calcula el tamaño de cualquier objeto en tiempo de compilación.

Las operaciones de entrada y salida no forman parte del lenguaje C. La biblioteca estándar de entrada/salida es un conjunto de funciones diseñadas para proporcionar un sistema estándar de entrada/salida a los programas C, siendo parte del C estándar ANSI. Las rutinas son lo suficientemente eficientes como para que los usuarios rara vez las eviten por razón de eficiencia.

En cuanto a gráficos, ni siquiera existen librerías de funciones estándar por lo que se facilitan a través de librerías de funciones no estándar.