TEMA 32

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

ÍNDICE

1. INTRODUCCIÓN
2. TIPOS ESTRUCTURADOS DE DATOS
    2.1. Arrays
    2.2. Cadenas de caracteres
    2.3. Estructuras
    2.4. Guiones
3. PUNTEROS
4. ESTRUCTURAS DINÁMICAS DE DATOS
    4.1. Listas lineales
    4.2. Árboles
5. FUNCIONES
    5.1. Punteros a funciones
    5.2. Funciones predefinidas en C
6. RUTINAS EN LENGUAJE ENSAMBLADOR
7. GRÁFICOS
8. BIBLIOGRAFÍA

1. INTRODUCCIÓN

Los tipos complejos de datos son los que se crean a partir de otros. Tienen gran importancia en C. Entre ellos podemos destacar las tablas o arrays, las cadenas de caracteres, las estructuras y los guiones.

Uno de los elementos más difícil de comprender y manejar en C es el puntero. Sin embargo, su uso nos permitirá diseñar estructuras dinámicas cuya dimensión no es fija y cuyo uso es imprescindible en ciertas aplicaciones.

2. TIPOS ESTRUCTURADOS DE DATOS

2.1. Arrays o Tablas

La *declaración* de un array especifica el nombre del array, el número de elementos del mismo y el tipo de éstos.

Arrays unidimensionales. La declaración de un array de una dimensión, se hace de la forma:

tipo nombre [tamaño];
tipo nombre [];

tipo — indica el tipo de los elementos del array. Puede ser cualquier tipo excepto "void".

nombre — es un identificador que nombra al array.

tamaño — es una constante que especifica el número de elementos del array. El tamaño puede omitirse cuando se inicializa el array, cuando se declara como un parámetro formal en una función o cuando se hace referencia a un array declarado en otra parte del programa.

Ejemplo:

```
int lista [100];
```

Arrays multidimensionales. La declaración de un array de varias dimensiones se hace de la forma:

tipo nombre [expr-cte] [expr-cte] ...;
tipo nombre [] [expr-cte] ...;

La primera "expr-cte" puede omitirse cuando se inicializa el array, cuando se declara como un parámetro formal en una función o cuando hace referencia a un array declarado en otra parte del programa.

El lenguaje C no chequea los límites de una array. Es responsabilidad del programador el realizar este tipo de operaciones.

Para dimensionar un array se pueden emplear constantes o expresiones a base de constantes de cualquier tipo entero.

Para *acceder a un elemento de un array*, se hace mediante el nombre del array seguido de uno o más subíndices, dependiendo de las dimensiones del mismo, cada uno de ellos encerrado entre corchetes. Un subíndice puede ser una constante, una variable o una expresión cualquiera.

Ejemplo:

```
int tabla [20] [80];
tabla [2] [3] = {{0, 1, 2}, {5, 4, 7}};
```

2.2. Cadenas de caracteres

Una cadena de caracteres es un array unidimensional, en el cual todos sus elementos son de tipo "char":

char cadena [longitud];

Un array de caracteres puede ser inicializado asignándole un literal. Por *ejemplo*:

```
char cadena [] = “abcd”;
```

Este ejemplo inicializa el array de caracteres "cadena" con cinco elementos (cadena[0] a cadena[4]). El quinto elemento, es el carácter nulo (\0), con el cual C finaliza todas las cadenas de caracteres.

Si se especifica el tamaño del array de caracteres y la cadena asignada es más larga que el tamaño especificado, se obtiene un error en el momento de la compilación. Por ejemplo:

```
char cadena [3] = "abcd";
```

Operador size of (tamaño de). Este operador da como resultado el tamaño en bytes de su operando. Cuando dicho operando es un array, el resultado es el tamaño total del array.

Función gets(). Leer una cadena de caracteres. La función "gets" lee una línea de la entrada estándar, "stdin", y la almacena en la variable especificada. Esta variable es un puntero a la cadena de caracteres leída.

Función puts(). Escribir una cadena de caracteres. La función "puts" escribe una cadena de caracteres en la salida estándar "stdout", y reemplaza el carácter nulo de terminación de la cadena (\0) por el carácter nueva línea (\n).

Funciones para manipular cadenas de caracteres. Las declaraciones de las funciones, para manipular cadenas de caracteres, que a continuación se describen, están en el fichero a incluir **string.h**. La sentencia:

```
# include <string.h>
```

Se requiere solamente para declaraciones de función.

Algunas *funciones para manipular cadenas de caracteres* son:

1. **strcat** (cadena1, cadena2). Esta función añade la cadena2 a la cadena1, termina la cadena resultante con el carácter nulo y devuelve un puntero a cadena1.

2. **strchr** (cadena, c). Esta función devuelve un puntero a la primera ocurrencia de c en cadena o un valor NULL si el carácter no es encontrado. El carácter c puede ser un carácter nulo (‘\0’).

3. **strcmp** (cadena1, cadena2). Esta función compara la cadena1 con la cadena2 y devuelve un valor:

   < 0 si la cadena1 es menor que la cadena2
   = 0 si la cadena1 es igual a la cadena2 y
   > 0 si la cadena1 es mayor que la cadena2.

4. **strcpy** (cadena1, cadena2). Esta función copia la cadena2, incluyendo el carácter de terminación nulo, en la cadena1 y devuelve un puntero a cadena1.

5. **strlen** (cadena). Esta función devuelve la longitud en bytes de cadena, no incluyendo el carácter de terminación nulo

6. **strncat** (cadena1, cadena2, n). Esta función añade los primeros n caracteres de cadena2 a la cadena1, termina la cadena resultante con el carácter nulo y devuelve un puntero a cadena1. Si n es mayor que la longitud de cadena2, se utiliza como valor de n la longitud de cadena2.

7. **strcmp** (cadena1, cadena2, n). Esta función compara los primeros n caracteres de cadena1 y cadena2 distinguiendo mayúsculas y minúsculas, y devuelve un valor:

   < 0 si la cadena1 es menor que la cadena2
   = 0 si la cadena1 es igual a la cadena2 y
   > 0 si la cadena1 es mayor que la cadena2.

   Si n es mayor que la longitud de la cadena1, se toma como valor la longitud de la cadena1.

8. **strncpy** (cadena1, cadena2, n). Esta función copia n caracteres de la cadena2, en la cadena1 (sobrescribiendo los caracteres de cadenas) y devuelve un puntero a cadena1. Si n es menor que la longitud de cadena2, no se añade automáticamente un carácter nulo a la cadena resultante. Si n es mayor que la longitud de cadena2, la cadena1 es rellenada con caracteres nulo (‘\0’) hasta la longitud n.

9. **strstr** (cadena1, cadena2). Esta función devuelve un puntero a la primera ocurrencia de cadena2 en cadena1 o un valor NULL si la cadena2 no se encuentra en la cadena1.

   ```
   char *strstr(char *cadena1, char *cadena2);
   ```

10. **strlwr** (cadena). Convierte las letras mayúsculas de cadena, en minúsculas. El resultado es la propia cadena en minúsculas.

11. **strupr** (cadena). Convierte las letras minúsculas de cadena, en mayúsculas. El resultado es la propia cadena en mayúsculas.

2.3. Estructuras

Una estructura es una agrupación de datos bajo un nombre común. Es un nuevo tipo de datos que puede ser manipulado de la misma forma que los tipos predefinidos como "float, int, char," entre otros. Una estructura se puede definir como una colección de datos de diferentes tipos, lógicamente relacionados. En C una estructura sólo puede contener declaraciones de variables. En otros compiladores, este tipo de construcciones son conocidas como "registros". Crear una estructura es definir un nuevo tipo de datos, denominado *tipo estructura* y declarar una variable de este tipo. En la definición del tipo estructura, se especifican los elementos que la componen así como sus tipos. Cada elemento de la estructura recibe el nombre de *miembro* (campo del registro). La sintaxis es la siguiente:

```
struct tipo_estructura
{
declaraciones de los miembros
};
```

"tipo_estructura" es un identificador que nombra el nuevo tipo definido.

Después de definir un tipo estructura, podemos declarar una variable de ese tipo, de la forma:

```
struct tipo_estructura [variable[, variable]...];
```

Para referirse a un determinado miembro de la estructura, se utiliza la notación:

```
variable.miembro
```

Ejemplo:

```
struct datos    {
                char nombre [42];
                char domicilio [60];
                };
```

Arrays de estructuras. Cuando los elementos de un array son de tipo estructura, el array recibe el nombre de array de estructuras o *array de registros*. Esta es una construcción muy útil y potente.

2.4. Uniones

Una unión es una variable que puede contener miembros de diferentes tipos, en una misma zona de memoria.

La declaración de una unión tiene la misma forma que la declaración de una estructura, excepto que en lugar de la palabra reservada "struct" se pone la palabra reservada "unión". Todo lo expuesto para las estructuras, es aplicable a las uniones, excepto la forma de almacenamiento de sus miembros.

union tipo_union
{
declaraciones de los miembros
};

tipo_union es un identificador que nombra el nuevo tipo definido.

Después de definir un tipo unión, podemos declarar una o más variables de ese tipo de la forma:

union tipo_union [variable[, variable] ...];

Para referirse a un determinado miembro de la unión, se utiliza la notación:

variable.miembro

Para almacenar los miembros de una unión, se requiere una zona de memoria igual a la que ocupa el miembro más largo de la unión. Todos los miembros son almacenados en el mismo espacio de memoria y comienzan en la misma dirección. El valor almacenado es sobrescrito cada vez que se asigna un valor al mismo miembro o a un miembro diferente.

Las uniones ofrecen una buena solución cuando se presentan datos alternativos en una misma estructura. Para ello podemos elegir dos estructuras distintas o una que englobe a ambas (union).

Ejemplo:

```
union   {
        struct  {
                int a_servicio;
                long n_nomina;
                }
        struct  {
                char f_contrato [10];
                int t_contrato;
                }
        }
```

3. PUNTEROS

Un puntero es una dirección de memoria, es decir, la posición donde se guarda un dato en la memoria del ordenador. Puede verse como una variable que contiene la dirección de memoria de un dato o de otra variable que contiene al dato. Quiere esto decir, que el puntero apunta al espacio físico donde está el dato o la variable. Un puntero puede apuntar a un objeto de cualquier tipo, incluyendo estructuras, funciones etc. Los punteros se pueden utilizar para crear y manipular estructuras de datos, para asignar memoria dinámicamente y para proveer el paso de argumentos por referencia en las llamadas a funciones.

Un puntero se declara igual que una variable pero anteponiendo el operador de indirección (*) al identificador del puntero, el cual significa "puntero a". Un puntero siempre apunta a un objeto de un tipo particular. Un puntero no inicializado tiene un valor desconocido.

*tipo * var-puntero;*

tipo	especifica el tipo del objeto apuntado; puede ser cualquier tipo, incluyendo tipos agregados.
var-puntero	nombre de la variable puntero.

(Recordemos que el operador ***** devuelve el dato contenido en una dirección de memoria y que el operador **&** devuelve la dirección de memoria donde está guardado un dato)

Ejemplo:

```
int *pint;        /* pint es un puntero a un entero */
char *pnom;       /* pnom es un puntero a una cadena de caracteres */
double *p, *q;    /* p y q son punteros a reales */
```

El *espacio de memoria* requerido para un puntero, es el número de bytes necesarios para especificar una dirección máquina. En la familia de micros 8086, una dirección "near" (dirección con respecto a la dirección base del segmento) necesita 16 bits y una dirección "far" (dirección segmentada) necesita 32 bits.

Punteros y arrays. En C existe, entre punteros y arrays, una relación tal que, cualquier operación que se pueda realizar mediante la indexación de array, se puede realizar también con punteros.

Para clarificar lo expuesto, observar el siguiente programa, realizado primeramente con arrays y a continuación con punteros:

```
/* Escribir los valores de un array. Versión con arrays */

#include <stdio.h>
main()
{
static int lista[] = {24, 30, 15, 45, 34};
int ind;

for (ind = 0; ind<5; ind++)
printf("%d", lista[ind]);
}
```

En este ejemplo, la notación utilizada para acceder a los elementos del array estático, es la expresión: lista[ind].

A continuación se expone la versión con punteros:

```
/* Escribir los valores de un array. Versión con punteros */

#include <stdio.h>
main()
{
static int lista[] = {24, 30, 15, 45, 34};
int ind;

for (ind = 0; ind<5; ind++)
printf("%d", *(lista+ind));
}
```

Esta versión es idéntica a la anterior, excepto que la expresión para acceder a los elementos del array es: *(lista+ind).

Esto deja constancia de que "lista" es la dirección de comienzo del array. Si a esta dirección le sumamos 1, o dicho de otra manera si "ind" vale 1, nos situamos en el siguiente entero de la lista; esto es *(lista+1) y lista[1] representan el mismo valor. Un incremento de uno sobre una dirección equivale a avanzar no un byte, sino un número de bytes igual al tamaño de los objetos que estamos tratando.

Según esto, hay dos formas de referirnos al contenido de un elemento de un array:

*(array+indice) o array[indice]

Puesto que *(lista+0) es igual que lista[0], la asignación: p = &lista[0] es la misma que la asignación p = lista, donde p es una variable de tipo puntero. Esto indica que la dirección de comienzo de un array es la misma que la del primer elemento. Por otra parte, después de haber efectuado la asignación p = lista, las siguientes expresiones dan lugar a idénticos resultados:

p[ind], *(p+ind), lista[ind], *(lista+ind)

Sin embargo, hay una diferencia entre el nombre de un array y un puntero. El nombre de un array es una constante y un puntero es una variable. Esto quiere decir que las operaciones:

lista = p o lista++	no son correctas, y las operaciones
p = lista o p++	sí son correctas.

Arrays de punteros. *Punteros a punteros*. Se puede definir un array, para que sus elementos contengan en lugar de un dato, una dirección o puntero.

Ejemplo:

```
int *a[10], v; a[0] =
&v; printf("%d",
*a[0]);
```

Este ejemplo define un array de 10 elementos que son punteros a datos de tipo "int" y una variable entera "v". A continuación asigna al elemento a[0], la dirección de "v" y escribe su contenido.

Cuando cada elemento de un array es un puntero a otro array, estamos en el caso de una doble indirección o "punteros a punteros". La característica de poder referenciar un puntero con otro puntero, le da al lenguaje C una gran potencia y flexibilidad para crear estructuras complejas.

Para *especificar que una variable es un puntero a un puntero*, se procede de la forma siguiente:

*tipo **variable;*

Ejemplo:

```
int **pp;          /* pp es un puntero a un puntero */
```

Para escribir un programa utilizando la notación de punteros en lugar de la notación array, nos planteamos únicamente la cuestión de cómo escribir la expresión “tabla[f][c]”, utilizando la notación de punteros. Pues bien, pensemos que un array de dos dimensiones es un array de una dimensión, donde cada elemento es a su vez un array de una dimensión.

Si elegimos una fila, por ejemplo "tabla[1]", o en notación puntero "tabla+1", interpretamos esta expresión como un puntero a un array de 5 elementos; esto quiere decir que "tabla+1" es un puntero a un puntero, o que el contenido de "tabla+1", *(tabla+l), es la dirección del primer elemento de esa fila, "tabla[1][0]", o en notación puntero "*(tabla+1)+0". Las direcciones "tabla+1" y "*(tabla+1) coinciden, pero ambas expresiones tienen diferente significado. Por *ejemplo*:

tabla+1+2 se refiere a la fila tabla[3] y
*(tabla+1) +2 se refiere al elemento tabla[1][2]
((tabla+1) +2) es el contenido del elemento tabla[1][2].

4. ESTRUCTURAS DINÁMICAS DE DATOS

La propiedad característica de las estructuras dinámicas es la facultad que tienen para variar su tamaño y hay muchos problemas que requieren de este tipo de estructuras. Esta propiedad las distingue claramente de las estructuras estáticas fundamentales (arrays y estructuras). Por tanto, no es posible asignar una cantidad fija de memoria para una estructura dinámica, y como consecuencia un compilador no puede asociar direcciones explícitas con las componentes de tales estructuras. La técnica que se utiliza más frecuentemente para resolver este problema consiste en realizar una *asignación dinámica* de memoria; es decir, asignación de memoria para las componentes individuales, al tiempo que son creadas durante la ejecución del programa, en vez de hacer la asignación durante la compilación del mismo.

Asignación dinámica de memoria. Cuando se asigna memoria dinámicamente para un objeto de un tipo cualquiera, se devuelve un puntero a la zona de memoria asignada. Para realizar esta operación disponemos en C de la función **malloc()**.

```
#include <stdlib.h> o <malloc.h>
p = malloc(t);
```

Esta función asigna un bloque de memoria de *t* bytes y devuelve un puntero que referencia el espacio asignado. Si hay insuficiente espacio de memoria o si *t* es 0, la función retorna a un puntero nulo (NULL).

Para liberar un bloque de memoria asignado por la función "malloc()", utilizaremos la función **free()**.

```
#include <stdlib.h> o <malloc.h>
free(void *p);
```

4.1. Listas lineales

Si deseamos una lista de elementos u objetos de cualquier tipo, originalmente vacía, que durante la ejecución del programa vaya creciendo y decreciendo elemento a elemento, según las necesidades previstas en el programa, entonces tenemos que construir una *lista lineal* en la que cada elemento apunte o direccione el siguiente. Por este motivo, una lista lineal se la denomina también *lista enlazada*.

Para construir una lista lineal primero tendremos que definir la clase de objetos que van a formar parte de la misma. De una forma genérica el tipo definido será de la forma:

```
typedef struct id tipo_objeto;
struct id
{
/* declaración de los miembros de la estructura */
tipo_objeto *siguiente;
};
```

Ejemplo:

```
typedef struct datos elemento;
struct datos
{
int dato;
elemento *siguiente;
}
```

Pilas. Una pila es una lista lineal en la que todas las inserciones y supresiones (y normalmente todos los accesos), se hacen en un extremo de la lista. Un ejemplo de esta estructura es una pila de platos. En ella, el añadir o quitar platos se hace siempre por la parte superior de la pila. Este tipo de listas reciben también el nombre de listas LIFO (Last In First Out - último en entrar, primero en salir).

4.2. Árboles

Un árbol es una estructura no lineal formada por un conjunto de *nodos* y un conjunto de *ramas*. En un árbol existe un nodo especial denominado *raíz*. Un nodo del que sale alguna rama, recibe el nombre de *nodo de bifurcación* o nodo rama y un nodo que no tiene ramas recibe el nombre de *nodo terminal* o nodo hoja.

Árboles binarios. Un árbol binario es un conjunto finito de nodos que consta de un *nodo raíz* que tiene dos subárboles binarios denominados *subárbol izquierdo* y *subárbol derecho*. Evidentemente, la definición dada es una definición recursiva, es decir, cada subárbol es un árbol binario.

Esta definición de árbol binario, sugiere una forma natural de representar árboles binarios en un ordenador: debemos tener dos enlaces (izdo. y dcho.) en cada nodo, y una variable de enlace *raíz* que nos direcciona el árbol. Esto es:

```
typedef struct datos nodo;
struct datos
{
/* declaración de miembros */
nodo *izdo;
nodo *dcho;
};
```

Si el árbol está vacío, "raíz" es igual a NULL; en caso contrario, "raíz" es un puntero que direcciona la raíz del árbol, e "izdo" y "dcho" son punteros que direccionan los subárboles izquierdo y derecho de la raíz, respectivamente.

Hay varios *algoritmos* para el manejo de estructuras en árbol. Una idea que aparece repetidamente en estos algoritmos es la noción de *recorrido de un árbol*. Este es un método para examinar sistemáticamente los nodos de un árbol, de forma que cada nodo sea visitado solamente una vez.

Pueden utilizarse tres formas principales para recorrer un árbol binario: "preorden", "inorden" y "postorden". Cuando se visitan los nodos en "preorden", primero se visita la raíz, después el subárbol izquierdo y por último el subárbol derecho. Cuando se visitan los nodos en "inorden", primero se visita el subárbol izquierdo, después la raíz y por último el subárbol derecho. Cuando se visitan los nodos en "postorden", primero se visita el subárbol izquierdo, después el subárbol derecho y por último la raíz.

Evidentemente, las definiciones dadas son definiciones recursivas, ya que, recorrer un árbol utilizando cualquiera de ellas, implica recorrer sus subárboles empleando la misma definición.

5. FUNCIONES

Una función es una colección independiente de declaraciones y sentencias, generalmente enfocadas a realizar una tarea específica. Todo programa C consta al menos de una función, la función "main()". Además de ésta, puede haber otras funciones cuya finalidad es, fundamentalmente, descomponer el problema general en subproblemas más fáciles de resolver y de mantener. La ejecución de un programa comienza con la función "main()".

Las funciones en C se pueden agrupar en dos grandes bloques:

- Funciones de librería.
- Funciones de usuario.

Las *funciones de librería* son las que están ya creadas en C (predefinidas). Entre estas se encuentran las de entrada y salida, las de control de memoria, las de gestión de cadenas, ...

Estas funciones se encuentran declaradas y definidas en ficheros de texto con extensión ".h". Para utilizarlas hay que poner en nuestro programa las sentencias "#include" necesarias.

Las funciones de usuario son las declaradas y definidas por el programador en el programa, aunque el programador puede crear ficheros ".h" con estas funciones para después utilizarlas poniendo sentencias "#include" en el programa donde quiera utilizarlas; es decir, puede convertirlas en funciones de librería.

Cuando se llama a una función, el control se pasa a la misma para su ejecución; y cuando finaliza, el control es devuelto de nuevo al módulo que llamó, para continuar con la ejecución del mismo a partir de la sentencia que efectuó la llamada.

Definición de una función. La definición de una función consta de la *cabecera* de la función y del *cuerpo* de la función. La sintaxis correspondiente es:

```
[clase] [tipo] nombre([parámetros-formales])
{
[declaraciones]
sentencias;
}
```

clase — Define el ámbito de la función, es decir, desde donde puede ser llamada. La clase puede ser: "extern" o "static".

Una función "static" es visible solamente en el fichero fuente en el cual está definida; y una función "extern" es visible para todos los ficheros fuente que componen el programa. Por defecto, la clase de una función es "extern".

tipo — Indica el tipo del valor devuelto por la función. Puede ser cualquier tipo fundamental o tipo definido por el usuario. Por defecto, el tipo es "int". Una función no puede retornar un array o función, pero si puede retornar un puntero a un array o a una función.

nombre — Es un identificador que indica el nombre de la función. Si el nombre va precedido por el operador asterisco (*), el valor devuelto por la función es un puntero.

parámetros formales — Componen la lista de argumentos de la función. Esta lista consta de un conjunto de variables con sus tipos, separadas por comas y encerradas entre paréntesis. Los parámetros formales son variables que reciben los valores pasados en la llamada a la función.

Si no se pasan argumentos a la función, la lista de parámetros formales puede ser sustituida por la palabra clave "void".

Cuerpo de la función. El cuerpo de una función está formado por una sentencia compuesta que contiene sentencias que definen lo que hace la función. También puede contener declaraciones de variables utilizadas en dichas sentencias. Estas variables, por defecto, son locales a la función.

Valor retornado por una función. Sentencia **return**. Cada función puede devolver un valor cuyo tipo se indica en la cabecera de función. Este valor es devuelto a la sentencia de llamada a la función, por medio de la sentencia "return", cuya sintaxis es la siguiente:

return [(expresión)];

Si la sentencia "return" no se especifica o se especifica sin contener una expresión, la función no devuelve un valor.

Llamada a una función. La llamada a una función tiene la forma:

[variable =] expresión([parámetros-actuales]);

- variable: especifica la variable donde va a ser almacenado el valor devuelto por la función. Notar que la llamada puede prescindir del valor devuelto por la función.

- expresión: especifica una dirección que referencia a una función. Puede ser, una expresión que es evaluada a una dirección de una función, o simplemente un identificador que corresponde con el nombre de la función llamada. Esto significa que una función puede ser llamada a través de un puntero a una función.

- parámetros-actuales: son una lista de expresiones separadas por comas. Las expresiones son evaluadas y convertidas utilizando las conversiones aritméticas usuales. Los valores resultantes son pasados a la función y asignados a sus correspondientes "parámetros formales". El número de expresiones en la lista, debe ser igual al número de parámetros formales, a no ser que se especifique un número variable de argumentos.

Declaración de una función. La declaración de una función, denominada también "función prototipo", permite conocer el nombre, el tipo del resultado, los tipos de los parámetros formales y opcionalmente sus nombres. No define el cuerpo de la función. Esta información permite al compilador chequear los tipos de los parámetros actuales por cada llamada a la función. Una función no puede ser llamada si previamente no está definida o declarada. A esta regla hay una excepción: que la definición de la función se haga con anterioridad a la llamada.

Paso de parámetros. Hay dos formas de pasar los parámetros actuales a sus correspondientes parámetros formales, cuando se efectúa la llamada a una función:

1. Por valor.
2. Por referencia.

Pasar "parámetros por valor", significa copiar los parámetros actuales en sus correspondientes parámetros formales, operación que se hace automáticamente cuando se llama a una función, con lo cual no se modifican los parámetros actuales.

Pasar "parámetros por referencia", significa que lo transferido no son los valores, sino las direcciones de las variables que contienen esos valores, con lo cual los parámetros actuales pueden verse modificados.

Cuando se llama a una función, los argumentos especificados en la llamada son pasados por valor; excepto los arrays que se pasan por referencia, ya el nombre del array es un puntero a dicho array.

Utilizando la forma de pasar parámetros por valor, pueden ser transferidas constantes, variables y expresiones; y utilizando la forma de pasar parámetros por referencia, solamente se permite transferir las direcciones de variables de cualquier tipo, arrays y funciones.

Para pasar una variable por referencia, se pasa la dirección del parámetro actual a su correspondiente parámetro formal, el cual tiene que ser un puntero. Para ello, se tiene que utilizar el operador "&" antes del nombre de la variable. Para pasar la dirección de un array o de una función, no es necesario este operador antes del nombre del array o del nombre de la función.

5.1. Punteros a funciones

Igual que sucedía con los arrays, el nombre de una función representa la dirección de comienzo de la función; por lo tanto, puede ser utilizado para pasarlo a funciones, colocarlo en arrays, etc.

Para declarar un puntero a una función se procede así:

*tipo (*p_identif)();*

tipo	es el tipo del resultado devuelto por la función.
p_identif	identifica a una variable de tipo puntero. Esta variable recibirá un puntero a una función, dado por el propio nombre de la función.

En el siguiente *ejemplo*, se define un puntero "p" a una función. A continuación, se asigna a "p" la dirección de la función "escribir" y se llama a la función mediante la sentencia: (*p) (5);

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

main()
{
void (*p)(int);     /*p es un puntero a una función */

p = escribir;       /*p = dirección de la función */
(*p) (5);           /* llamada a la función */

void escribir(int a)          /* función escribir */
printf("%d\n", a);
}
```

El nombre de una función representa la "dirección" de comienzo de la misma.

5.2. Funciones predefinidas en C

Hemos estudiado cómo el usuario puede definir sus propias funciones. No obstante C dispone en sus librerías de más de 400 funciones; algunas de ellas ya las hemos visto, como las funciones para entrada/salida, las funciones para manipular cadenas de caracteres etc., y otras las iremos viendo en este apartado.

Funciones matemáticas. Las declaraciones para las funciones matemáticas que a continuación se describen, están en el fichero a incluir "math.h". Quiere esto decir, que cuando se utilice una función matemática en un programa, debe especificarse la directriz:

```
#include <math.h>
```

Los argumentos para esta funciones son de tipo "double" y el resultado devuelto es también de tipo "double".

1. acos(x). Esta función da como resultado el arco, en el rango 0 a , cuyo coseno es x .
2. asin(x). Esta función da como resultado el arco, en el rango - /2 a /2, cuyo seno es x .
3. atan(x). Esta función da como resultado el arco, en el rango - /2 a /2, cuya tangente es x.
4. cos(x). Esta función da como resultado el coseno de x (x en radianes).
5. sin(x). Esta función da como resultado el seno de x (x en radianes).
6. tan(x). Esta función da como resultado la tangente de x (x en radianes).
7. log(x). Esta función da como resultado el logaritmo natural de x.
8. pow(x, y). Esta función da como resultado x elevado a y.
9. sqrt(x). Esta función da como resultado la raíz cuadrada de x.

Otras funciones de interés. Son:

1. rand(). Esta función da como resultado un número pseudoaleatorio entero, entre 0 y 32767. Necesitamos incluir <stdlib.h>

2. srand(arg). Esta función fija el punto de comienzo para generar números pseudoaleatorios. Si no se utiliza, el punto de comienzo siempre es el mismo para cada ejecución, que es el correspondiente a un argumento de valor 1.

3. time(seg). Esta función da como resultado el número de segundos transcurrido desde las 0 horas de 1 de Enero de 1970. Necesitamos incluir <time.h>

6. RUTINAS EN LENGUAJE ENSAMBLADOR

Pensando en C, el lenguaje ensamblador será útil para:

- Escribir funciones en lenguaje ensamblador.
- Incrementar la velocidad en ciertas secciones de código.
- Llamar a rutinas del núcleo del sistema operativo con la instrucción INT. Esto nos permitirá controlar los periféricos distintos de los estándares de entrada y salida.
- Crear rutinas residentes.

Una rutina de ensamblador, escrita en línea con las sentencias C que forman el programa, va precedida por la palabra clave **_asm**. La palabra _asm llama al ensamblador; puede aparecer en cualquier lugar válido para una sentencia C, e irá seguida por una instrucción en ensamblador, o por un grupo de instrucciones en ensamblador encerradas entre llaves.

7. GRÁFICOS

El tratamiento de gráficos que se estudia a continuación es para los compiladores C de BORLAND. Aunque puede haber diferencias con otros compiladores, siempre existirán funciones similares si el compilador incluye una interfaz gráfica. Estas variaciones son debidas a que en el standard C no se contemplan los gráficos.

Antes de poder utilizar cualquier función gráfica hay que cambiar el modo de la pantalla, de modo texto a modo gráfico. Se utiliza la función **initgraph()** para cambiar el modo de vídeo. Tiene el siguiente formato:

*initigraph (int far *drivergraf, int far *modograf, char far *viadriver);*

El argumento *drivergraf* le indica a initigraph() qué controlador (driver) gráfico es necesario cargar para el adaptador gráfico que vamos a utilizar. Si este argumento es igual a 0, se llama a la función detectgraph() automáticamente. Esta función comprueba el hardware de su equipo y selecciona la mayor resolución gráfica que pueda utilizar.

El argumento *modograf* indica el modo gráfico que va a utilizar.

El argumento *viadriver* le indica a initigraph() dónde puede encontrar los ficheros de los controladores gráficos. Si este argumento es NULL, los ficheros de los controladores gráficos deberán estar en el directorio actual.

Cuando se haya acabado de trabajar en modo gráfico, podemos llamar a la función **closegraph()** para volver al modo texto. Esta función libera cualquier memoria que hubiera sido reservada por las funciones gráficas. La función closegraph() sale del modo gráfico y vuelve al modo texto que estaba siendo utilizado cuando se llamó a la función initigraph().

La programación gráfica es sencilla si se utiliza Borland C++. Gracias a la librería BGI (Borland Graphics Interface), Borland ha realizado todo el trabajo difícil que envuelve a la programación gráfica.

Funciones de dibujo y rellenado. Las formas que se pueden dibujar con las funciones de la librería BGI se dividen en dos grandes grupos: *objetos sin rellenar* y *objetos rellenos*. El interior de un objeto sin rellenar es del mismo color que el fondo, sólo se ven los límites exteriores del objeto. El interior de un objeto relleno es de un color que se ha especificado al llamar a la función.

En la librería BGI se pueden encontrar las siguientes funciones de dibujo para objetos sin rellenar:

- line() Dibuja una línea.
- linerel() Dibuja una línea a una distancia relativa a la posición actual.
- lineto() Dibuja una línea de la posición actual a otra que se la especifique.
- rectangle() Dibuja un rectángulo.
- drawpoly() Dibuja un polígono.
- arc() Dibuja un arco.
- circle() Dibuja un círculo.
- ellipse() Dibuja una elipse.

El segundo gran grupo de este tipo de funciones lo componen las funciones que dibujan objetos rellenos. Estas funciones dibujan objetos que se rellenan con un modelo y un color seleccionado. Las siguientes funciones dibujan objetos rellenos:

- bar() Dibuja y rellena una barra bidimensional.
- bar3d() Dibuja y rellena una barra tridimensional.
- fillellipse() Dibuja y rellena una elipse.
- fillpoly() Dibuja y rellena un polígono.
- pieslice() Dibuja y rellena un sector de un diagrama de tarta.
- sector() Dibuja y rellena un sector de un diagrama elíptico de tarta.
- floodfill() Rellena una región limitada.

8. BIBLIOGRAFÍA

García de Sola, J.F.
Lenguaje C y estructura de datos
Mc Graw-Hill, 1992

Salvador Senent
Gestión de Entrada/Salida en C
Anaya Multimedia, 1992