Sistemas y Aplicaciones Informáticas

Tema 35. Lenguaje C: Manipulación de Estructuras de Datos Dinámicas y Estáticas. Entrada y Salida de Datos. Gestión de Punteros. Punteros a Funciones. Gráficos en C.

1. Ámbito de docencia.

- Sistemas informáticos monousuario y multiusuario (ASI 1).
- Sistemas informáticos multiusuario y en red (DAI 1).
- Sistemas operativos en entornos monousuario y multiusuario (ESI 1).

2. Lenguaje C: Manipulación de estructuras de datos.

2.1. Estructuras de datos estáticas.

2.1.1. Arrays mono y multidimensionales. Declaración e inicialización.

- Un array es una colección de variables del mismo tipo que se referencian por un nombre común. El compilador reserva espacio para un array y le asigna posiciones de memoria contiguas, cuya dirección más baja corresponde al primer elemento y la dirección más alta al último. Se puede acceder a cada elemento de un array mediante un índice.
- En C no se comprueba los límites del array, por tanto es responsabilidad del programador asegurar que no se accede a otras posiciones de memoria contiguas a las del array. El tamaño en memoria de un array es número de elementos * tamaño del tipo de datos en bytes.
- Se denomina un array unidimensional a un array en el que se puede acceder a sus elementos a través de un único índice. Se declara *tipo nombre_array[tamaño];* donde *tipo* define el tipo de datos que se puede almacenar en el array y *tamaño* es el número de elementos del array. En C el índice empieza en cero y acaba en *tamaño-1*. Para acceder a una posición del array se utiliza el índice y los corchetes de la siguiente manera: *nombre_array[índice];*
- Se denomina un array multidimensional a un array en el que para identificar un elemento del mismo se necesitan dos o más índices. Se declara de la siguiente manera:

  tipo nombre_array[índice1][índice2]…[índice_n];
- Un array bidimensional se corresponde con el concepto de matriz, y utiliza dos índices para acceder a sus elementos de la siguiente manera: *nombre_array[índice_filas][índice_columnas];*
- La manera en que se ocupa la memoria es por filas, es decir, primero el subarray correspondiente a la fila 1, después el subarray correspondiente a la fila 2 y así sucesivamente. Su tamaño en memoria es número de filas * número de columnas * tamaño del tipo de datos en bytes.
- Los arrays se pueden inicializar recorriéndolos y colocando un valor en cada elemento del mismo en tiempo de ejecución, o bien al declararlos se pueden rellenar de la siguiente manera:

  tipo nombre_array[índice1] …[índice_n] = {dato0, dato1,…,dato_n };

2.1.2. Cadenas. Arrays de cadenas. Declaración e inicialización.

- Una cadena es un array unidimensional de caracteres que acaba en un carácter nulo `'\0'`. Hay que tener en cuenta que C incluirá el carácter nulo, por tanto la longitud de la cadena que se declare deberá ser de un carácter más que el número de caracteres a almacenar.
- Un array de cadenas es equivalente a una matriz de tipo `char`, en la cual cada fila se considera una cadena y el número de columnas es la longitud máxima de la cadena. El acceso se puede realizar elemento a elemento como si fuera un array bidimensional, o bien especificando sólo el índice izquierdo correpondiente a la fila. Se declara como un array bidimensional de caracteres:

```
char texto[30][80];
```

- La inicialización de cadenas se realiza utilizando comillas dobles, y el compilador se encarga de introducir el carácter nulo al final. Cuando la inicialización de cadenas resulta tedioso porque hay que contar los caracteres que lo componen, C permite inicializar arrays indeterminados:

```
char texto[] = "Error de ejecución";
```

- Cuando se usan arrays indeterminados, el compilador asignará el espacio suficiente para guardar los datos de la inicialización. En el caso de arrays multidimensionales, se deben especificar todas las dimensiones excepto la dimensión más a la izquierda en la declaración (la primera).

2.1.3. Tipos de datos definidos por el usuario.

2.1.3.1. Estructuras. Arrays de estructuras. Declaración. Anidamiento.

- Una estructura es una colección de variables no necesariamente del mismo tipo que se referencian por un nombre común. Las variables que forman la estructura se denominan elementos de la estructura, que suelen mantener una relación lógica entre ellos. Una vez definida la estructura, se pueden declarar variables del nuevo tipo de dato definido.
- Una estructura se define de la siguiente manera:

```
struct nombre_estructura {
    tipo nombre_elemento1;
    tipo nombre_elemento2;
};
```

donde *nombre_elemento* son variables de cualquiera de los tipos válidos en C. La declaración de una estructura acaba en punto y coma porque se considera una sentencia.

- Para declarar una variable del tipo estructura se puede declarar como una variable cualquiera:

```
struct nombre_estructura nombre_variable;
```

- Para referenciar a los elementos de la estructura se utiliza el operador punto:

```
nombre_estructura.nombre_elemento1
```

- Según el estándar ANSI, es posible asignar una variable estructura a otra. Al ser una estructura un tipo de datos válido en C, se pueden definir arrays de estructuras. Para ello, primero se define la estructura y después se declara una variable array de ese tipo de la siguiente manera:

```
struct nombre_estructura nombre_array[tamaño];
```

donde *tamaño* es la longitud del array y cada elemento del array será a su vez una estructura.

- Los elementos de una estructura pueden ser tipos de datos simples, o puede ser un array mono o multidimensional. También puede ser otra estructura, lo que se denomina anidamiento.

2.1.3.2. Campos de bits. Declaración.

- El lenguaje C permite acceder a bits individuales dentro de un byte. El método que usa C para acceder a los bits se basa en la estructura. Un campo de bit es un elemento de una estructura donde se define su longitud en bits de la siguiente manera:

```
struct nombre_estructura {
    tipo nombre_elemento1:longitud1;
    tipo nombre_elemento2:longitud2;
} lista_variables;
```

- Los campos de bits se pueden declarar como `int`, `unsigned` o `signed`. Por ejemplo:

```
struct info {
   unsigned alarma:1;
   unsigned delta:4;
   unsigned alfa:3;
} estado;
```

- Para asignar valores a un campo de bits, se emplean los mismos métodos que en el resto de estructuras. No es necesario que algunos de los campos de bit tengan nombre si no son necesarios, basta simplemente con declarar su longitud en la estructura. Del mismo modo, si no se van a usar los últimos bits no es necesario declararlos.
- Los campos de bits tienen ciertas restricciones, como que no se puede tomar la dirección de un campo de bit (operador `&`) y no se pueden declarar arrays de campos de bits. En cambio, se pueden mezclar campos de bits con otras variables en la declaración de la estructura.

2.1.3.3. Uniones. Declaración.

- Una unión es una posición de memoria compartida por dos o más variables diferentes, en general de distinto tipo. La definición es similar a la de la estructura:

  union nombre_unión {
  tipo nombre_elemento1;
  tipo nombre_elemento2;
  } lista_variables;

- El compilador guarda espacio para la variable de mayor tamaño declarada. En cualquier momento el dato guardado se puede utilizar como cualquiera de las variables definidas.

2.1.3.4. Redefinición de tipos de datos existentes.

- La palabra clave `typedef` permite crear nuevos nombres para tipos de datos existentes:

  typedef tipo_existente nuevo_nombre;

- Una vez definido el nuevo nombre del tipo, se puede utilizar en la declaración de variables como si fuera un tipo estándar de C. Se puede usar para redefinir cualquiera de los tipos válidos en C, es decir, los tipos estándar y cualquiera de las estructuras definidas por el usuario.

2.2. Estructuras de datos dinámicas.

2.2.1. Pilas. Concepto. Implementaciones.

- Se caracterizan porque las operaciones de inserción y eliminación de elementos se realizan solamente en un extremo de la estructura, que se denomina habitualmente cima.
- Existen dos posibles implementaciones en los lenguajes de alto nivel:
  - *Estática*. Es la mejor opción, por rapidez y sencillez, cuando se conoce el número máximo de datos que deben ser almacenados. De esta manera, la pila se representa mediante un registro con un campo formado por un vector que almacena los elementos de la pila, y un campo cima que almacena el índice del último elemento introducido en la pila:

```
typedef int tPila;
typedef struct pila {
   tPila elem[MAX];
   int cima;
} Pila;
```

* *Dinámica*. Es la única opción cuando no se conoce el número máximo de datos que deben ser almacenados. La pila se representa como una sucesión de nodos creados en memoria dinámicamente. Cada nodo es un registro con dos campos, uno de ellos almacena uno de los elementos de la pila y el otro es un puntero al siguiente nodo. El puntero del último nodo señala un valor nulo, y existe un puntero externo que señala al primer nodo de la pila:

```
typedef int tPila;
typedef struct pila {
   tPila elem;
   struct pila *sig;
} Pila;
```

2.2.2. Colas. Concepto. Implementaciones.

- Se caracterizan porque las operaciones de inserción y eliminación de elementos se realizan en los extremos opuestos de la estructura. La inserción se produce en el extremo derecho o final de la estructura, mientras que la eliminación se realiza en el extremo izquierdo o inicio.
- Existen dos posibles implementaciones en los lenguajes de alto nivel:
  * *Estática*. La representación más eficiente se realiza mediante una cola circular. De esta manera, la cola se representa mediante un registro con un campo formado por un vector que almacena los elementos de la cola, un campo inicio que almacena el índice anterior al primer elemento de la cola y un campo final que almacena el índice del último elemento:

```
typedef int tCola;
typedef struct cola {
   tCola elem[MAX+1];
   int ini, fin;
} Cola;
```

  * *Dinámica*. Se representa como una sucesión de nodos creados en memoria dinámicamente. Cada nodo es un registro con dos campos, uno de ellos almacena uno de los elementos de la cola y el otro es un puntero al siguiente nodo. El puntero del último nodo señala un valor nulo, y existen dos punteros externos: uno señala al nodo de inicio, y otro al nodo final:

```
typedef int tCola;
typedef struct cola {
   tCola elem;
   struct cola *sig;
} Cola;
```

2.2.3. Listas enlazadas. Concepto. Implementación. Tipos.

- Son estructuras de datos que se caracterizan porque las operaciones de inserción y eliminación de elementos se realizan en cualquier punto según un determinado criterio de ordenación.
- En las listas enlazadas cada elemento de la estructura de datos (estructura lógica) no ocupa la posición de memoria (estructura física) contigua a la del elemento que le precede; por tanto para evitar el movimiento de elementos en inserciones o eliminaciones, la lista se representa como una sucesión de nodos creados en memoria dinámicamente. Cada nodo es un registro con dos campos, uno de ellos almacena uno de los elementos de la lista y el otro es un puntero al siguiente nodo. El puntero del último nodo señala un valor nulo, y existen dos punteros externos: uno señala al nodo de inicio de la lista, y otro señala a un punto de interés:

```
typedef struct {
   int clave;
   int dato;
} tLista;

typedef struct lista {
   tLista elem;
   struct lista *sig;
} Lista;
```

- Otros tipos de listas enlazadas son las siguientes:
  - *Listas circulares*. Son listas enlazadas en las que el puntero del último nodo señala al nodo de inicio de la lista. Presentan la ventaja de que partiendo de cualquier elemento se puede hacer un recorrido completo de la lista.
  - *Listas doblemente enlazadas*. Son listas enlazadas en las que cada nodo apunta a su sucesor y a su predecesor. Presentan la ventaja de que se puede hacer un recorrido de la lista en ambos sentidos, pero implican una mayor reserva de memoria por cada nodo.

2.2.4. Árboles. Concepto. Implementación.

- Un árbol es un conjunto finito de cero o más nodos, tal que existe un nodo especial, llamado nodo raíz, y donde los restantes nodos están separados en $n \geq 0$ conjuntos disjuntos, cada uno de los cuales es a su vez un árbol llamado subárbol del nodo raíz. Esto implica que cada nodo del árbol es raíz de algún subárbol contenido en el árbol principal.
- Un árbol binario se representa como una sucesión de nodos creados en memoria dinámicamente. Cada nodo es un registro con tres campos, uno de ellos almacena uno de los elementos del árbol y los otros dos son punteros que señalan al primer nodo del subárbol izquierdo y al primer nodo del subárbol derecho respectivamente. Los punteros de los nodos sin subárbol izquierdo o sin subárbol izquierdo señalan un valor nulo. En C se expresa de la siguiente manera:

```
typedef struct {
   int clave;
   int dato;
} tArbol;

typedef struct arbol {
   tArbol elem;
   struct arbol *izq, *der;
} Arbol;
```

3. Entrada y salida de datos.

3.1. Funciones de E/S por consola. Caracteres especiales.

- El lenguaje C no tiene palabras clave para la entrada y salida de datos, sino que utiliza funciones de la librería `stdio.h`. Las funciones simples de E/S son las siguientes:
  - `int getchar(void)`. Devuelve el carácter leído del teclado y lo convierte a `unsigned int`. En caso de error devuelve `EOF`, constante definida en `stdio.h`.
  - `char *gets(char *cadena)`. Lee una cadena de caracteres introducida por teclado y la sitúa en la dirección apuntada por `cadena`. Recoge todos los caracteres escritos hasta pulsar INTRO e incluye un carácter `'\0'` de fin de cadena al final.

  - `int putchar(int caracter)`. Envía el carácter a la pantalla en la posición actual del cursor. En caso de error o fin de fichero devuelve `EOF`.
  - `int *puts(char *cadena)`. Imprime una cadena de caracteres. En caso de error devuelve `EOF`, en caso contrario devuelve un valor distinto de cero. Para escribir cadenas es una llamada más rápida que `printf()` y ocupa menos espacio.
- Existen funciones que permiten leer y escribir datos en varios formatos, y son las siguientes:
  - `int printf(char *cadena_control, lista_argumentos)`. Devuelve la entrada formateada o bien un valor negativo en caso de error. La cadena de control está formada por especificadores que se escriben *%type* y por los caracteres que se quieran imprimir en pantalla. La lista de argumentos está formada por las variables o constantes que se desean imprimir. Debe haber el mismo número de especificadores de formato que argumentos en la lista. Los especificadores más importantes son:
    - `%c`. Muestra un carácter.
    - `%d`. Muestra un entero con signo.
    - `%u`. Muestra un entero sin signo.
    - `%ent.decf`. Muestra un número real con `ent` dígitos enteros y `dec` decimales.
    - `%s`. Muestra una cadena de caracteres.
    - `%o`. Muestra un numero octal sin signo.
    - `%x`. Muestra un numero hexadecimal sin signo.
  - `int scanf(char *cadena_control, lista_argumentos)`. Recoge datos del teclado y en caso de error devuelve `EOF`. La cadena de control es equivalente a la de `printf()` y está formada por especificadores que se escriben *%type*. La lista de argumentos está formada por las direcciones de las variables que se quieren cargar. Debe haber el mismo número de especificadores de formato que argumentos en la lista.
- El backslash (\) se utiliza para imprimir ciertos caracteres que no se pueden imprimir desde el teclado. Algunos de ellos son espacio atrás (`'\b'`), salto de página (`'\f'`), salto de línea (`'\n'`), tabulación horizontal (`'\t'`) y fin de cadena (`'\0'`).

3.2. Entrada y salida por archivos.

3.2.1. Tipo de dato FILE.

- Cuando se accede a un archivo, debe crearse un puntero a dicho archivo que permita distinguir un archivo de otro. Un puntero a archivo es una variable definida como un puntero al tipo de dato `FILE`, que está definido en la librería `stdio.h`. Se declara como cualquier puntero a cualquier otro tipo de dato. Los datos pueden transferirse en formato binario o en formato texto.
- La definición de la estructura `FILE` depende del compilador, pero en general mantienen un campo con la posición actual de lectura/escritura, un buffer para mejorar las prestaciones de acceso al fichero y algunos campos para uso interno.

3.2.2. Funciones de apertura y cierre de archivos.

- `FILE *fopen(char *nombre, char *modo)`. Abre y crea ficheros en disco. El valor de retorno es un puntero a una estructura FILE. Los parámetros de entrada son:

- *nombre*. Una cadena que contiene un nombre de fichero válido, esto depende del sistema operativo que estemos usando. El nombre puede incluir el camino completo.
- *modo*. Especifica en tipo de fichero que se abrirá o se creará y el tipo de datos que puede contener, de texto o binarios:
  - **r**. Sólo lectura. El fichero debe existir.
  - **w**. Se abre para escritura, se crea un fichero nuevo o se sobrescribe si ya existe.
  - **a**. Añadir para escritura, el cursor se sitúa al final. Si el fichero no existe, se crea.
  - **r+**. Lectura y escritura. El fichero debe existir.
  - **w+**. Lectura y escritura, se crea un fichero nuevo o se sobrescribe si ya existe.
  - **a+**. Añadir para lectura y escritura, el cursor se sitúa al final. Si no existe, se crea.
  - **t**. Tipo texto, si no se especifica "t" ni "b", se asume por defecto que es "t".
  - **b**. Tipo binario.

– `int fclose(FILE *fichero)`. Cerrar un fichero almacena los datos que aún están en el buffer de memoria, y actualiza algunos datos de la cabecera del fichero que mantiene el sistema operativo. Además permite que otros programas puedan abrir el fichero para su uso. Un valor de retorno cero indica que el fichero ha sido correctamente cerrado, si ha habido algún error, el valor de retorno es la constante `EOF`. El parámetro es un puntero a la estructura `FILE` del fichero que queremos cerrar.

3.2.3. Funciones de lectura y escritura de un carácter.

– `int fgetc(FILE *fichero)`. Esta función lee un carácter desde un fichero. El valor de retorno es el carácter leído como un `unsigned char` convertido a `int`. Si no hay ningún carácter disponible, el valor de retorno es `EOF`. El parámetro es un puntero a una estructura `FILE` del fichero del que se hará la lectura.

– `int fputc(int caracter, FILE *fichero)`. Esta función escribe un carácter a un fichero. El valor de retorno es el carácter escrito, si la operación fue completada con éxito, en caso contrario será `EOF`. Los parámetros de entrada son el carácter a escribir, convertido a `int` y un puntero a una estructura `FILE` del fichero en el que se hará la escritura.

3.2.4. Funciones de lectura y escritura de cadenas.

– `char *fgets(char *cadena, int n, FILE *fichero)`. Lee cadenas de caracteres. Leerá hasta n-1 caracteres o hasta que lea un retorno de línea. En este último caso, el carácter de retorno de línea también es leído. El parámetro n nos permite limitar la lectura para evitar desbordar el espacio disponible en la cadena. El valor de retorno es un puntero a la cadena leída, si se leyó con éxito, y es `NULL` si se detecta el final del fichero o si hay un error. Los parámetros son: la cadena a leer, el número de caracteres máximo a leer y un puntero a una estructura `FILE` del fichero del que se leerá.

– `int fputs(const char *cadena, FILE *stream)`. Escribe una cadena en un fichero. No se añade el carácter de retorno de línea ni el carácter nulo final. El valor de retorno es un número no negativo o `EOF` en caso de error. Los parámetros de entrada son la cadena a escribir y un puntero a la estructura `FILE` del fichero donde se realizará la escritura.

3.2.5. Funciones de lectura y escritura con formato.

- `int fprintf(FILE *fichero, const char *formato, ...)`. Funciona igual que `printf` en cuanto a parámetros, pero la salida se dirige a un fichero en lugar de la pantalla.
- `int fscanf(FILE *fichero, const char *formato, ...)`. Funciona igual que `scanf` en cuanto a parámetros, pero la entrada se toma de un fichero en lugar del teclado.

3.2.6. Funciones de lectura y escritura en archivos binarios.

- `size_t fread(void *puntero, size_t tamaño, size_t nregistros, FILE *fichero)`. Lee desde un fichero uno o varios registros de la misma longitud y a partir de una dirección de memoria determinada. El usuario es responsable de asegurarse de que hay espacio suficiente para contener la información leída. El valor de retorno es el número de registros leídos, no el número de bytes. Los parámetros son: un puntero a la zona de memoria donde se almacenarán los datos leídos, el tamaño de cada registro, el número de registros a leer y un puntero a la estructura `FILE` del fichero del que se hará la lectura.
- `size_t fwrite(void *puntero, size_t tamaño, size_t nregistros, FILE *fichero)`. Escribe hacia un fichero uno o varios registros de la misma longitud almacenados a partir de una dirección de memoria determinada. El valor de retorno es el número de registros escritos, no el número de bytes. Los parámetros son: un puntero a la zona de memoria donde se almacenarán los datos leídos, el tamaño de cada registro, el número de registros a leer y un puntero a la estructura `FILE` del fichero del que se hará la lectura.

3.2.7. Funciones específicas de archivos de acceso aleatorio.

- `int fseek(FILE *fichero, long int desplazamiento, int origen)`. Situa el cursor del fichero para leer o escribir en el lugar deseado. El valor de retorno es cero si la función tuvo éxito, y un valor distinto de cero si hubo algún error. Los parámetros de entrada son: un puntero a una estructura `FILE` del fichero en el que queremos cambiar el cursor de lectura/escritura, el valor del desplazamiento y el punto de origen desde el que se calculará el desplazamiento. El parámetro `origen` puede tener tres posibles valores:
  - `SEEK_SET`. El desplazamiento se cuenta desde el principio del fichero. El primer byte del fichero tiene un desplazamiento cero.
  - `SEEK_CUR`. El desplazamiento se cuenta desde la posición actual del cursor.
  - `SEEK_END`. El desplazamiento se cuenta desde el final del fichero.
- `long int ftell(FILE *fichero)`. Averigua la posición actual del cursor de lectura/escritura de un fichero. El valor de retorno será esa posición, o -1 si hay algún error. El parámetro de entrada es un puntero a una estructura `FILE` del fichero del que queremos leer la posición del cursor de lectura/escritura.

3.2.8. Funciones generales de E/S.

- `int fflush(FILE *fichero)`. Fuerza la salida de los datos acumulados en el buffer de salida del fichero. El valor de retorno es cero si la función se ejecutó con éxito, y `EOF` si hubo algún error. El parámetro de entrada es un puntero a la estructura `FILE` del fichero del que se quiere vaciar el buffer. Si es `NULL` se hará el vaciado de todos los ficheros abiertos.

- `int feof(FILE *fichero)`. Esta función sirve para comprobar si se ha alcanzado el final del fichero. El valor de retorno es distinto de cero sólo si no se ha alcanzado el fin de fichero. El parámetro es un puntero a la estructura `FILE` del fichero que queremos verificar.
- `void rewind(FILE *fichero)`. Sitúa el cursor de lectura/escritura al principio del archivo. El parámetro es un puntero a la estructura `FILE` del fichero que queremos rebobinar.

4. Gestión de punteros.

4.1. Definición. Declaración y operadores. Punteros a estructuras.

- Se puede considerar la memoria de un ordenador como un array de bytes que pueden direccionarse individualmente o por grupos. Un puntero es una variable que es capaz de guardar una dirección de memoria, en la cual se encuentra otra variable. De esta manera, si una variable A contiene la dirección de otra variable B se dice que A apunta a B, o que A es un puntero de B.
- Un puntero se declara de la siguiente manera:

  *tipo *nombre_puntero;*

  donde *tipo* corresponde al tipo de dato de las variables que el puntero puede apuntar, y que puede ser cualquier tipo de datos válido en C. El resultado de una declaración de un puntero es una variable capaz de mantener una dirección del tipo de datos para el que se ha declarado.
- Se puede definir un puntero a una estructura del mismo modo que se hace con otras variables. Los punteros a estructuras se usan en general para pasar por referencia una estructura a una función, para recorrer arrays de estructuras y para crear listas y otras estructuras dinámicas.:

  *struct nombre_estructura *nombre_puntero;*
- Cuando se usa un puntero a una estructura, para acceder a los elementos de la estructura se sustituye el operador punto por el operador `->`, es decir, *nombre_puntero->nombre_elemento1*

4.2. Expresiones con punteros.

- Los punteros, una vez declarados, contienen un valor desconocido. Si se intenta utilizar un puntero sin inicializar el sistema operativo puede llegar a caerse. Para inicializar un puntero se le debe asignar una dirección conocida. Se puede inicializar un puntero a nulo utilizando la constante `NULL` definida en la librería `stdlib.h` para indicar que no apunta a nada.
- Un puntero puede usarse en el lado derecho del operador de asignación para asignar su valor a otro puntero del mismo tipo de dato. Sólo existen dos operaciones aplicables a los punteros: la suma y la resta. Estas operaciones actualizan la dirección que contiene apuntando a la siguiente posición de memoria contigua (suma) o a la anterior posición de memoria contigua (resta) respecto de la posición a la que apunta, en función del tipo de datos del puntero.
- Tanto la suma como la resta se reducen a sumar o restar punteros con enteros. No se pueden sumar dos punteros ni restarlos. Se pueden aplicar los operadores incremento (`++`) y decremento (`--`), puesto que es sumar o restar respectivamente una posición. Además se pueden comparar dos punteros en una expresión relacional.

4.3. Punteros y arrays.

- Se puede acceder a cualquier posición de un array usando el nombre del array y utilizando la aritmética de punteros como índice. Es posible crear un puntero a un array de dos maneras:

* Usando el nombre del array como puntero:

```
int dato[3] = {1, 2, 3};
printf ("%d", *(dato+2)); /* Escribiría 3 */
```

* Declarando un puntero del mismo tipo del array e inicializando el puntero a la dirección del array usando la variable del array sin índice o con el índice igual a cero (primera posición).

```
int dato[5];
int *puntero;
puntero = dato; /* Inicialización */
puntero = &dato[0]; /* Otra manera */
```

- Del mismo modo, se puede apuntar a cualquier posición del array con un puntero apuntando a la dirección de esa posición de memoria. También se puede recorrer un array utilizando un puntero al array y utilizando la aritmética de punteros. En este caso, para acceder al contenido de una posición del array se utilizará el puntero y el operador `*`. Un array multidimensional tambíen puede recorrerse con un puntero al array y utilizando la aritmética de punteros
- Si el array se va a recorrer completamente y en orden, la utilización de la aritmética de punteros será el método más rápido,mientras que si se va a acceder al array aleatoriamente es más conveniente usar la indexación, ya que en general será más rápido y más sencillo.
- Una cadena se puede apuntar con un puntero y recorrerla con aritmética de punteros como cualquier otro array. Un caso especial es la inicialización de cadenas de la siguiente manera:

  *char *nombre_cadena = constante_cadena;*
- Cuando se inicializa un puntero a una constante de cadena hay que tener en cuenta que si se modifica el puntero para que apunte a otra dirección, se perderá la constante de cadena.
- Los punteros, como cualquier otro tipo de datos, pueden ser almacenados en un array. Se puede definir una array de punteros de la siguiente manera:

  *tipo *nombre_array[tamaño];*

  donde *tipo* define el tipo de datos a los que apuntan los punteros y *tamaño* es el número de punteros del array. Para acceder a los contenidos de las direcciones apuntadas por los elementos del array es necesario utilizar el operador `*`.

4.4. Punteros a funciones.

- Al igual que ocurre con los arrays, el nombre de una función representa su dirección de comienzo. Para declarar un puntero a una función la sintaxis es la siguiente:

  *tipo (*nombre_función)(tipopar1, tipopar2, …);*

  donde *tipo* es el tipo de datos que devuelve la función y *tipopar* es el tipo de los parámetros usados en la definición de la función. La variable *nombre_función* recibirá un puntero a una función dado por el propio nombre de la función.
- En el siguiente ejemplo se define un puntero a una función. A continuación se asigna al puntero la dirección de una función y se llama a la función a través del puntero:

```
#include <stdio.h>

void escribir(int);
```

```
void main (void) {

      void (*puntero)(int);
      puntero = escribir;
      (*puntero)(5);
}

void escribir(int a) {

      printf("%d\n", a);
}
```

5. Gráficos en C.

- La librería BGI (Borland Graphics Interface) es una librería gráfica que funcionaba en todos los compiladores de Borland de MS-DOS. Para usar esta librería es necesario incluir el fichero de cabecera `graphics.h`.
- Existen dos maneras de trabajar con la pantalla: en modo texto, en el que sólo se pueden visualizar caracteres; y en modo gráfico, en el que se pueden visualizar texto y gráficos. Por defecto se trabaja en modo texto, si se quiere pasar al modo gráfico hay que indicarlo con la función `initgraph(driver, modo, directorio)`, donde `driver` indica la tarjeta que se va a utilizar, `modo` la resolución y `directorio` la localización de la librería BGI.
- Una vez inicializado el modo gráfico, la librería contiene una gran cantidad de funciones para escribir texto, dibujar líneas y círculos, etc. Una vez terminadas las operaciones gráficas, se debe volver al modo texto usando `closegraph()`.