ACADEMIA CARTAGENA99
C/ Cartagena 99, Bº C , 28002 Madrid
91 51 51 321
academia@cartagena99.com.es

APUNTES PARA EL CURSO DE C AVANZADO E INTRODUCCIÓN A C++

1 INTRODUCCIÓN - 3 -

2 PUNTEROS - 3 -

2.1 OPERACIONES BÁSICAS - 4 -
2.1.1 DECLARACIÓN - 4 -
2.1.2 ASIGNACIÓN - 5 -
2.1.3 DESREFERENCIA DE UN PUNTERO (OPERADOR '*'). - 6 -
2.1.4 PUNTEROS A PUNTEROS - 7 -
2.2 EJEMPLOS DE USO - 7 -
2.2.1 PARÁMETROS POR REFERENCIA A FUNCIONES - 8 -
2.3 PRECAUCIONES CON LOS PUNTEROS - 10 -
2.3.1 PUNTEROS NO INICIALIZADOS - 10 -
2.3.2 PUNTEROS A VARIABLES LOCALES FUERA DE ÁMBITO - 10 -
2.4 ARITMÉTICA DE PUNTEROS - 11 -
2.5 PUNTEROS Y VECTORES - 11 -
2.5.1 ARRAYS - 12 -
2.5.2 PASO DE VECTORES COMO PARÁMETROS A FUNCIONES - 13 -
2.6 PUNTEROS Y ESTRUCTURAS - 14 -
2.6.1 EL OPERADOR -> - 15 -
2.7 MEMORIA DINÁMICA: MALLOC Y FREE - 15 -
2.7.1 PUNTEROS VOID* - 15 -
2.7.2 EL OPERADOR SIZEOF - 16 -
2.7.3 FUNCIÓN FREE - 16 -
2.7.4 EJEMPLOS DE USO DE MALLOC, FREE Y SIZEOF - 16 -
2.7.5 PRECAUCIONES CON LA MEMORIA DINÁMICA - 18 -
2.8 PUNTEROS A FUNCIONES - 18 -

3 ESTRUCTURAS DINÁMICAS DE DATOS - 21 -

3.1 LISTAS ABIERTAS (LISTAS ENLAZADAS) - 21 -
3.1.1 INSERCIÓN - 22 -
3.1.2 BORRADO - 22 -
3.1.3 RECORRIDO - 22 -
3.1.4 INCONVENIENTES DEL USO DE LAS LISTAS ENLAZADAS - 22 -
3.2 PILAS - 23 -
3.3 COLAS - 24 -
3.4 LISTAS CIRCULARES - 24 -
3.5 LISTAS DOBLEMENTE ENLAZADAS - 24 -
3.6 ÁRBOLES - 25 -
3.7 ÁRBOLES BINARIOS - 25 -

3.7.1 RECORRIDO DE ÁRBOLES BINARIOS - 26 -
3.7.2 ÁRBOLES BINARIOS DE BÚSQUEDA (ABB) - 26 -
3.7.3 BORRADO EN ÁRBOLES BINARIOS DE BÚSQUEDA - 27 -
3.8 ÁRBOLES AVL - 28 -

4 MANEJO DE FICHEROS EN C - 30 -

4.1 FUNCIONES C PARA FICHEROS - 30 -
4.1.1 FUNCIONES Y TIPOS C - 30 -
4.1.2 FUNCIONES C PARA FICHEROS DE ACCESO ALEATORIO - 32 -
4.2 ARCHIVOS SECUENCIALES - 33 -
4.2.1 ORDENAR ARCHIVOS SECUENCIALES - 34 -
4.3 ARCHIVOS DE ACCESO ALEATORIO - 35 -
4.3.1 ORDENAR ARCHIVOS DE ACCESO ALEATORIO - 36 -
4.4 FICHEROS DE ÍNDICES - 37 -
4.4.1 BÚSQUEDA BINARIA - 38 -

5 C++ CONCEPTOS BÁSICOS. PROGRAMACIÓN ORIENTADA A OBJETOS - 40 -

5.1 PROGRAMACIÓN ORIENTADA A OBJETOS (POO) - 40 -
5.2 APORTACIONES DE C++ - 42 -
5.2.1 ARGUMENTOS POR OMISIÓN EN FUNCIONES - 43 -
5.2.2 FUNCIONES SOBRECARGADAS - 43 -
5.2.3 AMBIGÜEDADES - 44 -
5.2.4 REFERENCIAS - 44 -
5.2.5 OPERADORES *NEW* Y *DELETE* - 48 -
5.2.6 OTROS - 48 -
5.3 TRABAJANDO CON CLASES - 49 -
5.3.1 CLASES - 49 -
5.3.2 DEFINICIÓN DE CLASES - 49 -
5.3.3 EL PUNTERO IMPLÍCITO THIS - 50 -
5.3.4 FUNCIONES MIEMBRO Y OBJETOS CONSTANTES - 51 -
5.3.5 CONSTRUCTORES - 51 -
5.3.6 DESTRUCTORES - 54 -
5.3.7 FUNCIONES AMIGAS DE UNA CLASE - 55 -
5.3.8 OPERADORES SOBRECARGADOS - 56 -
5.3.9 HERENCIA. CLASES DERIVADAS - 58 -
5.3.10 PUNTEROS A OBJETOS DE UNA CLASE DERIVADA - 60 -
5.4 FUNCIONES VIRTUALES - 60 -
5.5 POLIMORFISMO - 61 -

1 INTRODUCCIÓN

Estos apuntes son los que se entrega a los alumnos como material básico de apoyo en el curso. No obstante, el curso es de contenido principalmente práctico, y la colección de ejercicios completa, que es la esencia del curso en sí, no aparece en el presente documento.

Este curso está pensado para ingenieros que deseen repasar los aspectos más delicados de C, así como introducirse en la programación orientada a objetos en C++. Resulta **ideal para todas aquellas empresas que se dediquen al desarrollo de aplicaciones en C y C++**, ya que, a menudo, cuando los ingenieros terminan la carrera suelen haber olvidado la mayoría de los conceptos importantes para la programación.

Como puede observarse, el curso empieza directamente con el tema de punteros, dando por hecho que los alumnos conocen todo lo referente a tipos de datos en C, operadores en C, sentencias de control de flujo, arrays, funciones, ámbitos de las variables, definición de constantes…

Si usted quisiera contratar este curso para su empresa podría solicitar la modificación del temario para adaptarlo a las características de su empresa.

Tal y como está estructurado en el presente documento, el curso constaría de unas 40 horas, según la profundidad con que se desee abordar cada uno de los puntos.

Puede utilizarse cualquier compilador C/C++ (DevCpp, Visual C++, Borland Builder, Turbo Cpp, CVI, MinGWStudio, LCC…).

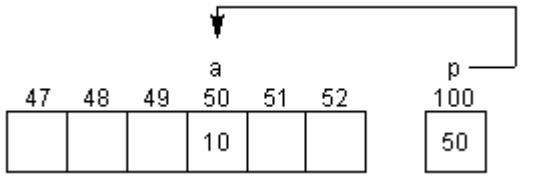
Para cualquier consulta referente a este curso no duden en ponerse en contacto con nosotros en: academia@cartagena99.com.es.

2 PUNTEROS

El concepto de puntero está basado en la idea de una variable cuya misión es contener la dirección de otra variable o la dirección de memoria de un dato. Se dice, entonces, que el puntero apunta a la variable. Según el tipo de variable al que apunte tendremos un puntero a entero, un puntero a float, char, etc.

Para trabajar con punteros es importante conocer el significado de los operadores de dirección-de (símbolo &) e in-dirección (símbolo *), que también se puede leer como lo-apuntado-por. De este modo *&var* puede leerse como “dirección de var”, y **pun* puede leerse como “lo apuntado por pun” o “lo que hay en la dirección contenida en pun”.

En el siguiente gráfico se puede observar lo descrito. La memoria está dividida en bloques numerados. Imaginemos que en la posición de memoria 50 tenemos la variable “a”, cuyo valor es 10. A continuación, si declaramos el puntero “p”, que contiene la dirección de la variable “a”, vemos como el puntero “p” apunta a la dirección que contiene el valor 10.



El código correspondiente a este gráfico es el siguiente:

```
int a = 10;
int *p;
p = &a;
```

Como vemos, no tendría sentido asignar directamente la dirección 50 al puntero “p”, ya que no sabemos cuál es la dirección donde está almacenado el valor de “a”. Por lo tanto, utilizamos el operador de dirección & para poder obtener la dirección de la variable “a”.

Además, un puntero puede contener la dirección de memoria de un dato, sin necesidad de tener declarada una variable intermedia. Para ello se hace necesaria la reserva de memoria, no podemos elegir las posiciones que queramos arbitrariamente, ya que podrían estar ocupadas por otros datos.

2.1 Operaciones básicas

2.1.1 Declaración

Un puntero se declara anteponiendo el modificador *, que se suele leer como “puntero a”.

```
int *pun;
```

Se puede leer como “int es lo apuntado por pun” o un “int es lo que hay en la dirección contenida en pun”. Esta es la declaración de un puntero a entero. Es decir, si aparece una variable “a” declarada de la siguiente forma, y asignamos al puntero “pun” la dirección de dicha variable “a”, podemos utilizar *pun en su lugar:

```
int a;
```

Por lo tanto, *pun es un entero que está localizado en la dirección de memoria especificada en pun, así como “a” es un entero localizado en la dirección de memoria que podemos obtener mediante &a.

Un puntero inicializado siempre apunta a un dato de un tipo en particular.
Un puntero no inicializado, no se sabe a dónde ni a qué apunta.

Del mismo modo tenemos:

```
float *pun2// puntero a float
char *pun3// puntero a char
```

El espacio de memoria requerido para almacenar un puntero es el número de bytes necesarios para especificar una dirección de la máquina en la que se esté compilando el código. Hoy día suele ser el valor típico de 4 bytes.

2.1.2 Asignación

A un puntero se le puede asignar la dirección de una variable del tipo adecuado. Esta dirección puede encontrarse bien en otro puntero, obtenerse de la propia variable mediante el operador “dirección de” (&), a través del nombre de un array (lo veremos posteriormente), o a través del valor devuelto por una función.

Algo que no tiene demasiada lógica es asignar directamente una dirección a un puntero, dado que no sabemos si esa dirección de memoria estará ocupada o no. Más adelante se verá el mecanismo para asignar una posición de memoria a un puntero. Como hemos visto en el ejemplo de la introducción, sin necesidad de reservar memoria para un contenido apuntado por un puntero, podemos asignar el valor de una dirección válida, como por ejemplo la dirección de una variable ya declarada, mediante el operador &.

Un ejemplo de asignación es el siguiente:

```
int a = 10, *p, *q;
q = &a;
*p = 20;
```

Al puntero “q” se le asigna la dirección de la variable “a”. Esta asignación es correcta y cada vez que se actúe sobre *q, es como si estuviéramos actuando sobre “a”.

El puntero p no ha sido inicializado y a su contenido se le está asignando el valor 20. Es decir, a una dirección de memoria que no sabemos cuál es, especificada por “p”, está siendo modificada al valor 20. Esto no tiene mucho sentido puesto que esa posición de memoria puede pertenecer incluso a otro proceso. En los sistemas operativos actuales esto daría error si esa posición de memoria no estuviera libre, por lo que no causaríamos daño a otras aplicaciones que estén ejecutándose, pero no es la forma adecuada de asignación.

2.1.3 Desreferencia de un puntero (operador ‘*’).

El operador * también se conoce como el operador de indirección. Si en el ejemplo anterior accedemos a *q, obteniendo los mismos resultados que si accedemos a la variable “a”, se dice que estamos accediendo al “contenido de q”. Así que también se lee como “contenido de”.

Si hemos asignado correctamente un valor a un puntero, como en el ejemplo anterior hicimos con el puntero “q”, al escribir *q será como si escribiéramos “a”.

A continuación se proponen dos ejemplos sencillos para comprobar los resultados que se obtendrían al ejecutar el código:

Ejemplo 1.

```
int a=3,b=5;
int *pun1, *pun2, *pun3;
pun2=pun1;
pun1=&a;
a=*pun1+1;
printf("%d\n",*pun1);
printf("%d\n",*pun2);
pun3=pun1;
printf("%d\n",*pun3);
pun2=&b;
*pun2=*pun3+3;
printf("%d\n",b);
```

Ejemplo 2.

```
int x, y;
int *p1, *p2;
p1= &x;
y=7;
y=(*p1);
x=3;
printf("%d ",*p1);
printf("%d ",y);
p2=&y;
(*p2)=(*p1)*2;
(*p1)=y+3;
printf("%d %d",y,x);
```

2.1.4 Punteros a punteros

Del mismo modo que un puntero puede contener la dirección de un entero, de una char o un float, también puede contener la dirección de otro puntero. Se trata entonces de un puntero a puntero a int, puntero a puntero a char etc.

```
int x, y;
int *p1,*p2 , **pp;
p1 = &x;
p2 = &y;
pp = &p1;
y = 5;
**pp=7;
printf("%d ",x);
printf("%d ",*p1);
printf("%d ",*p2);
y = (**pp)+(*p2);
pp = &p2;
printf("%d",**pp);
p2 = &x;
printf("%d",**pp);
```

2.2 Ejemplos de uso

Una de las utilidades más importantes de los punteros es la de referenciar los datos sin necesidad de copiarlos. Como ventaja obtenemos no tener que copiar los datos a los cuales apunta un puntero. Y como inconveniente, se pueden cometer problemas de ambigüedad de los datos, dado que al modificar los datos apuntados por un puntero, esos mismos datos están siendo apuntados por otro puntero.

En un caso concreto, este tipo de referencia se utiliza mucho en las llamadas a funciones, cuyos argumentos serán punteros a datos.

2.2.1 Parámetros por referencia a funciones

Las funciones en C sólo devuelven un valor, es decir, que si sólo usáramos el valor de retorno, cada llamada a una función solamente podría modificar una variable del ámbito desde el que es llamada. Una forma de solventar esta limitación es usando los punteros. Por medio de los punteros podemos indicarle a la función la dirección donde se encuentran las variables que queremos que modifique. De este modo una función podrá modificar tantas variables del ámbito de la llamada como queramos.

Cuando se llama a una función, el valor de los parámetros reales se pasa a los parámetros formales. Todos los argumentos, excepto los arrays, por defecto se pasan por valor. Por ejemplo:

```
void main()
{
      ...
      nombre_funcion (arg1, arg2);
      ...
}

nombre_funcion (argumento1, argumento2)
{
      ...
}
```

En este caso, “arg1” y “arg2” son los parámetros reales. Los parámetros formales son “argumento1” y “argumento2”. En este caso, lo que está ocurriendo es que se hace una copia del valor del argumento, no se le pasa la dirección a la función. La función “nombre_funcion” no puede alterar las variables que contienen los valores pasados, es decir, “arg1” y “arg2” seguirán teniendo el mismo valor en la función “main”.

Si lo que se desea es alterar los contenidos de los argumentos especificados en la llamada a la función, hay que pasar esos argumentos por referencia. Se le pasa la dirección de cada argumento y no su valor. Por lo tanto, los argumentos formales deben ser punteros, y utilizaremos el operador “&” en la llamada de la función.

A continuación se muestran los ejemplos correspondientes a los dos tipos de paso de parámetros:

Ejemplo 1.

```
#include <stdio.h>

void intercambio (int x, int y);

void main ()
{
      int a = 20, b = 30;
      intercambio (a, b);
      printf("a vale %d y b vale %d\n", a, b);
}

void intercambio (int x, int y)
{
      int z = x;
      x = y;
      y = z;
}
```

Ejemplo 2.

```
#include <stdio.h>

void intercambio (int * x, int * y);

void main ()
{
      int a = 20, b = 30;
      intercambio (&a, &b);
      printf("a vale %d y b vale %d\n", a, b);
}

void intercambio (int *x, int *y)
{
      int z = *x;
      *x = *y;
      *y = z;
}
```

EJERCICIOS:

1. Realizar una función que devuelva por dos argumentos que se le pasan por referencia, las dos soluciones reales de una ecuación de segundo grado. El valor retornado por la función será 1 si pudo calcular las soluciones y 0 si no las pudo calcular (por ser complejas) (*enviar consultas o sugerencias a*: academia@cartagena99.com.es).

2.3 Precauciones con los punteros

Cuando se trabaja con punteros son frecuentes los errores debidos a la creación de punteros que apuntan a alguna parte inesperada, produciéndose una "violación de memoria". Por lo tanto, debe ponerse la máxima atención para que esto no ocurra, inicializando adecuadamente cada uno de los punteros que se utilicen.

Además, en C es importante liberar la memoria reservada para los punteros previamente, con el fin de que no se produzcan las denominadas "lagunas de memoria".

2.3.1 Punteros no inicializados

Uno de los principales problemas que podemos encontrar es la utilización de un puntero que no ha sido inicializado correctamente. Tenemos dos maneras de asignar a un puntero una posición de memoria correcta para que no se produzcan estos errores:

- Asignación de la dirección de una variable o del nombre de un array.
- Asignación de la dirección de comienzo de una zona de memoria reservada dinámicamente (mediante la función *malloc*).

En el siguiente ejemplo podemos ver el uso de los punteros inicializados de forma incorrecta:

```
int a, *p, *q;

a = 10;
q = &a;     /* Asignamos al puntero q la dirección de la
            variable a */
a += 2;     /* Incrementamos a dos unidades */
printf ("Valor de a = %d\tValor de q = %d\n", a, *q);
*p += 2;    /* Incrementamos el contenido de la direccion
            apuntada por p dos unidades. ERROR */
printf ("Valor de p = %d\n", *p);   /* ERROR */
```

Como podemos ver en el ejemplo, el contenido de la dirección apuntada por p no podemos incrementarlo 2 unidades, dado que no sabemos a qué dirección apunta. Al declarar el puntero *p, la dirección que contiene es desconocida, así que no podemos tampoco imprimir el contenido de la dirección apuntada por este puntero.

2.3.2 Punteros a variables locales fuera de ámbito

Debemos tener cuidado especial al asignar a un puntero la dirección de una variable local. Por ejemplo:

```
int * funcion (void)
{
      int a = 10;
      int *q;

      q = &a;
```

```
        return q;
}

void main()
{
        int *p;
        p = funcion ();
}
```

Esto producirá un error dado que *funcion* devuelve la dirección de una variable a la que desde la función *main* no podemos acceder, porque es una variable local de otra función.

2.4 Aritmética de punteros

A los punteros les podemos sumar o restar una cantidad entera. Esto provoca que se incremente o se decremente la posición de memoria apuntada por él. Por lo tanto, quiere decir que el valor del puntero no cambia tantas unidades como pretendemos cambiar, sino en tantas posiciones de memoria como indicamos. De esta forma, el valor del puntero, si incrementamos 2 unidades, se incrementará 2 posiciones de memoria, lo cual no tiene que corresponder en bytes con 2 unidades.

Ejemplo:

```
int a[100];
int *p, *q;
p = &a[3];  /* p apunta a a[3] */
q = &a[0];  /* q apunta a a[0] */
p++;        /* hace que p apunte al siguiente entero: a[4] */
p--;        /* hace que p apunte al entero anterior: a[3] */
p = p + 3;  /* hace que p avance tres enteros: a[6] */
p = p - 3;  /* hace que p retroceda tres enteros: a[3] */
```

Por lo tanto, incrementar el valor del puntero p hace que pase a apuntar al siguiente entero del array, es decir, si el array “a” tuviera sus elementos almacenados en memoria consecutivamente, el valor del puntero “p” se incrementaría en 2 ó 4 unidades (el tamaño en bytes de un entero).

2.5 Punteros y vectores

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

2.5.1 Arrays

Un array o vector es una estructura homogénea, compuesta por varios elementos, todos del mismo tipo y almacenados consecutivamente en memoria.

Los arrays pueden tener una o más dimensiones. Si son de una dimensión se suelen llamar "vector" o "listas", y si tienen dos dimensiones, "tablas".

Para indicar el elemento de un array al que se quiere acceder, se utilizan índices enteros entre corchetes, a continuación del nombre del array.

El subíndice correspondiente al primer elemento es el cero.

Declaración

```
tipo nombre [tamaño];
int vector[10];
```

El tamaño puede ser omitido si es inicializado en el momento de su definición:

```
int vector[] = {1, 2, 3, 4, 5};
```

El tamaño debe ser un número entero o una constante, pero nunca una variable.
El acceso a los elementos de un array se realiza mediante un número entero (índice), una constante o una variable.

2. Ejercicio 1. Crear un array unidimensional y rellenarlo con enteros introducidos a través del teclado. Finalmente mostrar los elementos por pantalla (*enviar consultas o sugerencias a*: academia@cartagena99.com.es).

Ejercicio 2. Crear un array bidimensional para almacenar cadenas de caracteres introducidas a través del teclado (*enviar consultas o sugerencias a*: academia@cartagena99.com.es).

Correspondencia con punteros

La dirección de comienzo de un array viene indicada de dos formas: por el nombre del array y por la dirección del primer elemento del array. De esta forma, podemos asignar esa dirección de comienzo a un puntero y operar con el puntero de la misma forma que si operáramos con el array:

```
int a[5] = {1, 2, 3, 4, 5};
int *p;
int elemento;

p = &a[0];        /* Ambas lineas */
p = a;            /* hacen lo mismo */

/* Formas de acceder a los elementos del array */
```

```
elemento = a[3];        /* elemento vale 4 */
elemento = p[4];        /* elemento vale 5 */
elemento = *(p+2);      /* elemento vale 3 */
```

Además, la notación de acceso de los punteros mediante el operador *, puede ser utilizada para el nombre del array "a":

```
elemento = *(a+3);           /* elemento vale 4 */
```

Una diferencia importante es que el nombre del array es una constante y el puntero es una variable. De tal forma que podemos incrementar la posición del puntero (p++) para avanzar a la siguiente posición y que éste ya no apunte al principio del array, pero no podemos hacer lo mismo con el nombre del array (a++).

2.5.2 Paso de vectores como parámetros a funciones

Una forma cómoda de pasar argumentos a funciones es utilizar el paso por referencia. De esta forma, no se realiza una copia del parámetro al llamar a la función, sino que se indica la dirección donde se encuentra.

Podemos utilizar punteros además de para pasar por referencia argumentos que sean variables, para hacer lo mismo con arrays. Pero además, podemos utilizar la notación de arrays para el paso de los mismos a funciones. Por ejemplo:

```
void funcion (int a[])
/* Otra forma posible seria: void funcion (int *a) */
{
      ...
}
void main()
{
      int a[10];
      funcion (a);
}
```

En este ejemplo se observa que podemos modificar los valores del array "a" en la función, dado que está recibiendo la dirección del comienzo del array.

Ejercicio 1. Realizar un programa que rellene en la función *main* un vector con las edades de unos alumnos. Implementar también una función que discrimine las edades y ponga a cero las edades de los alumnos menores de edad (*enviar consultas o sugerencias a*: academia@cartagena99.com.es).

Ejercicio 2. Realizar un programa que recoja los nombres de un número NUM_PERSONAS de personas y los imprima por pantalla. Para ello, realizar dos funciones: una para leer las cadenas de caracteres que introduce el usuario por teclado y otra para imprimir por pantalla (*enviar consultas o sugerencias a*: academia@cartagena99.com.es).

2.6 Punteros y estructuras

La finalidad de una estructura es agrupar una o más variables, generalmente de diferentes tipos, bajo un mismo nombre para hacer más fácil su manejo. Por ejemplo, una estructura típica podría ser la ficha que almacena datos relativos a una persona: nombre, apellidos, edad, teléfono... A las variables que agrupa dentro de una estructura, se les denomina "miembros".

La sintaxis para crear una estructura es la siguiente:

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

A continuación tendríamos que definir una estructura de este tipo declarado para poder utilizarla. En el siguiente ejemplo se puede ver una estructura que represente la ficha de una persona:

```
struct tficha
/* declaración de una estructura del tipo tficha */
{
      char nombre[20];
      char direccion[20];
      long telefono;
};
/* definición de las estructuras var1 y var2 */
struct tficha var1, var2;
```

En C++ no es necesario especificar la palabra "struct" en la definición de las variables, pero en ANSI C sí. Lo más cómodo para evitar esto es crear un tipo utilizando *typedef*, como se muestra en el siguiente ejemplo:

```
typedef struct ficha
/* declaración de una estructura del tipo tficha */
{
      char nombre[20];
      char direccion[20];
      long telefono;
}tficha;
/* definición de las estructuras var1 y var2 */
tficha var1, var2;
```

Los miembros de una estructura pueden ser cualquier tipo de variable, incluso una estructura. Para acceder a éstos, debemos utilizar el operador ".", si la estructura es una variable, o "->" si disponemos de un puntero a la estructura. En el ejemplo anterior, para acceder a los campos nombre, direccion y telefono, tendremos que poner:

```
char * nombre = var1.nombre;
char * direccion = var1.direccion;
long telefono = var1.telefono;
```

Podemos asignar una estructura a otra mediante el operador de asignación:

```
var1 = var2;
```

Ejercicio 1. Realizar un programa que defina una estructura que represente la ficha de una persona y almacene los siguientes datos: nombre, dirección, telefono, fecha de nacimiento. La fecha de nacimiento será una estructura que contenga, a su vez: día, mes y año. Este programa debe tomar los datos por teclado y luego mostrarlos por pantalla (*enviar consultas o sugerencias a*: academia@cartagena99.com.es).

Ejercicio 2. Realizar un programa que ordene un grupo de alumnos según la edad. Los datos de los alumnos serán introducidos por teclado y estarán contenidos en una estructura con los siguientes datos: nombre, direccion, edad. Los resultados serán mostrados por pantalla (*enviar consultas o sugerencias a*: academia@cartagena99.com.es).

2.6.1 El operador ->

Como hemos indicado previamente, cuando disponemos de un puntero a una estructura, la forma de acceder a los miembros de la misma es mediante el operador "->". Por ejemplo:

```
typedef struct ficha
/* declaración de una estructura del tipo tficha */
{
      char nombre[20];
      char direccion[20];
      long telefono;
}tficha;
/* definición de las estructuras var1 y var2 */
      tficha *var1, var2;
```

La forma de acceder a los elemenos de la estructura apuntada por var1, será:

```
char * nombre = var1->nombre;
char * direccion = var1->direccion;
long telefono = var1->telefono;
```

2.7 Memoria dinámica: malloc y free

2.7.1 Punteros void*

Un puntero a cualquier objeto no constante se puede convertir a tipo void *. Por eso este tipo de puntero recibe el nombre de puntero génerico.

En el siguiente ejemplo, se puede observar una conversión de un puntero a entero a un puntero genérico:

```
void * p = NULL;
int a = 10, *q = &a;
p = q;
```

En C se permite la conversión implítica de puntero genérico a un puntero de cualquier tipo; sin embargo, en C++ se requiere la conversión explícita. Este tipo de conversión sería la que aparece en el siguiente ejemplo:

```
void * p = NULL;
int a = 10, *q = &a, *t;
p = q;
t = p;
```

Para la reserva de memoria dinámica mediante la función *malloc*, haremos uso de este tipo de conversiones de punteros void a punteros del tipo que deseemos.

2.7.2 El operador sizeof

El operador *sizeof* nos proporciona el tamaño en bytes del argumento que indiquemos. El argumento es un tipo de datos. Nos será útil porque a la función que utilizaremos más adelante para realizar la reserva de memoria de forma dinámica, le tendremos que indicar como parámetro el tamaño en bytes que queremos reservar.

> Ejercicio. Comprobar el tamaño de los tipos de datos: char, int, float, y de la estructura de datos que representa a la ficha de una persona de los ejemplos del punto 1.6 (*enviar consultas o sugerencias a*: academia@cartagena99.com.es).

2.7.3 Función free

Cuando reservamos memoria de forma dinámica, ésta debe ser liberada para que no se produzcan “lagunas de memoria”. Esto es, aunque en los sistemas operativos actuales modernos esto se soluciona, al acabar una función, las variables locales desaparecen y, al acabar el programa, desaparecen todas las variables locales. Si desaparece un puntero que apuntaba a una zona de memoria reservada previamente por nosotros, esa zona de memoria no queda liberada y, si este hecho se repitiera, acabaríamos por dejar llena la memoria de datos a los que no apuntamos.

Cuando hacemos que un puntero apunte a la dirección de comienzo de un array estático, no es necesario liberar la memoria, y de hecho intentarlo produciría un error.

Necesitaremos incluir el archivo stdlib.h en nuestro proyecto para utilizar tanto las funciones free, malloc y realloc.

El prototipo de la función free es el siguiente:

```
void free (void * puntero);
```

2.7.4 Ejemplos de uso de malloc, free y sizeof

Para reservar memoria dinámicamente, utilizaremos la función *malloc*. También podemos utilizar la función *realloc* para reasignar un bloque de memoria al que apunta un puntero, por ejemplo para cambiar el tamaño del bloque asignado.

El prototipo de la función malloc es el siguiente:

```
void * malloc (size_t nbytes);
```

Como podemos ver, la función malloc devuelve un puntero de tipo void. Es decir, tendremos que hacer una conversión explícita al tipo de puntero para el cual queremos reservar memoria. Si la reserva de memoria no se ha podido llevar a cabo, el puntero devuelto apuntará a NULL.

En el siguiente ejemplo, se reserva memoria para 100 elementos de tipo entero y se libera al final:

```
void main()
{
      int *p;

      p = (int*)malloc(100*sizeof(int));
      if (p == NULL)
      {
            printf("Fallo al reservar memoria\n");
            return -1;
      }
      free (p);
}
```

Ejercicio 1. Realizar un programa que lea una serie de enteros por teclado, los ordene y los visualice (*enviar consultas o sugerencias a*: academia@cartagena99.com.es). Dicho programa utilizará asignación dinámica de memoria. Para ello utilizar las funciones con el siguiente prototipo:

```
OrdenarEnteros ( int *pent, int numelementos )
VisualizarEnteros ( int *pent, int numelementos )
```

Ejercicio 2. Realizar un programa que reserve memoria de forma dinámica para un puntero de punteros, de tal forma que se puedan almacenar varias cadenas de caracteres. (NOTA: debemos seguir dos pasos: asignar memoria para un array de punteros, cuyos elementos referenciarán cada una de las filas del array; y asignar memoria para cada una de las filas, que contendrán las cadenas de caracteres) (*enviar consultas o sugerencias a*: academia@cartagena99.com.es).

Ejercicio 3. Realizar un programa que lea una lista de N_MAX alumnos y sus correspondientes notas de final de curso. Estas podrán introducirse en formato numérico o alfabético a elegir, para cada alumno; 5 ó aprobado, 7 ó notable, etc. Las estructuras de datos que se manejan están basadas en las siguientes definiciones:

```
typedef union
{
      float fnota;
      char snota[13];
} tunota;
typedef enum
{
      numerico, alfanumerico
```

```
} tiponota;
typedef struct
{
      char nombre[10];
      tiponota tipo;
      tunota nota;
} ficha;
```

Se pide además que se calcule la nota media de la clase.

Para los anteriores cálculos, considerar que los rangos de notas serán: No presentado, Suspenso (0 hasta 5), Aprobado (5 hasta 7), Notable (7 hasta 9), Sobresaliente (9 hasta 10), Matrícula (10).

Asimismo, para el cálculo de la nota media considerar que Suspenso o No Presentado es un 0, Aprobado un 5, Notable un 7, Sobresaliente un 9 y M. De Honor un 10.

Para ello crear una función con el siguiente prototipo:

`void estadistica (ficha alumnos[ ], int numeroalumnos)`

Dicha funcion calculará y presentará por pantalla los datos que se piden. La petición de los nombres de los alumnos y los datos se realizará en el programa principal (*enviar consultas o sugerencias a*: academia@cartagena99.com.es).

Ejercicio 4. Realizar un programa que reserve memoria para un puntero mediante una función *reserva_memoria*. Hacerlo de dos formas:

- La función devuelve el puntero para el que queremos reservar memoria.
- La función no devuelve nada y el puntero se lo pasamos en un argumento (NOTA: se le pasará el puntero por referencia).

(*enviar consultas o sugerencias a*: academia@cartagena99.com.es)

2.7.5 Precauciones con la memoria dinámica

Como ya hemos comentado, hay que tener cuidado con la memoria reservada de forma dinámica en cada función. Cuando una función acaba, las variables y punteros locales desaparecen, de tal forma que debemos liberar la memoria apuntada por los punteros si ésta ha sido reservado mediante las funciones *malloc* o *realloc*.

Además, debemos comprobar que la memoria que pedimos es realmente reservada. Si no, los accesos no serán válidos.

El redimensionamiento de una zona de memoria que tenemos reservada, lo podemos llevar a cabo mediante la función *realloc*, pero debemos tener especial cuidado puesto que es una función lenta y en procesos en los que se repita este redimensionamiento, obtendremos grandes retardos.

2.8 Punteros a funciones

Tal y como sucede en los arrays, el nombre de una función representa la dirección de memoria donde se localiza esa función. Esto quiere decir que la dirección de una función puede pasarse como argumento de otra función, almacenarla, etcétera. La sintaxis es la siguiente:

```
tipo(*pfn)();
```

Donde “tipo” es el tipo del valor devuelto por la función y “pfn” es el nombre de una variable de tipo puntero. En este puntero tendremos la dirección de comienzo de una función. Si “pfn” es el puntero a la función, *pfn será la función en sí:

```
int (*pfn)(int)
```

La función *pfn devuelte un entero y recibe como argumento un entero. La llamada a la función será pfn(int).

En el siguiente ejemplo se puede ver un uso de los punteros a funciones, en el que primero se le asigna al puntero “pfn” la dirección de la función *cuadrado* y se utiliza el puntero para llamar a la función. Después se asigna la dirección de la función *suma* y se vuelve a utilizar el mismo puntero para llamar a la función.

```
int suma(int a, int b)
{
      return (a+b);
}
int cuadrado (int a)
{
      return (a*a);
}
void main()
{
      int (*pfn)();
      int x = 5, y = 3; z = 0;

      pfn = cuadrado;
      z = (*pfn)(x); /* Igual que poner z = pfn(x) */

      pfn = suma;
      z = (*pfn)(x,y); /* Igual que poner z = pfn(x,y) */
}
```

Podemos utilizar el puntero “pfn” para llamar a distintas funciones que tenga un número y tipo de argumentos distintos. Pero no es posible para funciones que devuelvan distintos tipos.

3 ESTRUCTURAS DINÁMICAS DE DATOS

3.1 Listas abiertas (listas enlazadas)

Hasta ahora hemos estudiado cómo crear vectores estáticos y vectores cuyo tamaño se calcula en tiempo de ejecución. En el primer caso, la declaración del vector incluye el tamaño exacto del mismo que permanece constante durante toda la ejecución del programa:

```
unsigned int vector[100];
```

En el caso de que el tamaño del vector no sea conocido en tiempo de compilación, podemos utilizar la función malloc para reservar el espacio de memoria necesario durante la ejecución del programa:

```
unsigned int * vector;
vector = (unsigned int *)malloc( numElem * sizeof(unsigned int) );
…
free(vector);
```

El segundo caso nos permite cambiar las dimensiones del vector si nos hemos quedado cortos inicialmente. Para ello disponemos de la función realloc(). Sin embargo, como ya se ha mencionado antes, realloc() puede necesitar crear un vector nuevo y copiar todos los elementos del antiguo sobre éste, con la consiguiente sobrecarga en tiempo de ejecución.

Otra de las limitaciones más importantes de los vectores es que si se necesita insertar un elemento en mitad de los mismos es necesario desplazar todos los elementos que estaban emplazados a continuación. Además, no es posible distinguir una posición del vector libre de una ocupada si no se utiliza un vector de ocupación adicional. La alternativa utilizada generalmente en estos casos es la lista enlazada (lista abierta).

En una lista enlazada no se reserva espacio inicialmente para un número de elementos sino que éste se reserva y se libera dinámicamente según se van creando y destruyendo elementos durante la ejecución del programa. La idea básica es reservar espacio para el nuevo elemento cuando éste se va a crear y, dado que los elementos ya no tienen por qué ocupar posiciones contiguas en memoria, mantener un enlace a la posición en la que reside el siguiente. Generalmente, se declara una estructura que contiene los campos de datos de los elementos que se van a almacenar y un puntero a la propia estructura (recursión estructural) que permite enlazar con el resto de nodos:

```
typedef struct tnodo
{
      int dato;
      struct tnodo * siguiente;
} TNodo;
```

La lista completa puede ser referenciada mediante un puntero al primer elemento en ella. El último nodo de la lista tiene como puntero al siguiente elemento el valor especial NULL. Por tanto, una lista vacía consiste en un único puntero cuyo valor es NULL. A

continuación se muestra un resumen de las operaciones más comunes que pueden realizarse sobre la lista.

3.1.1 Inserción

En una lista abierta lo más fácil es insertar los elementos por el principio. Para ello, simplemente se crea el nuevo nodo con la información que debe almacenarse, se hace que el puntero de éste apunte al nodo que era hasta ahora el primero de la lista y se pasa a utilizar como cabeza de la lista el puntero al nuevo nodo:

```
TNodo * InsertarNodo(TNodo * lista, int NuevoValor)
{
        TNodo * nuevo = (TNodo *)malloc( sizeof(TNodo) );
        if (nuevo != NULL )
        {
              nuevo->siguiente = lista;
              nuevo->dato = NuevoValor;
        }
        return nuevo;     // Nueva cabeza de la lista.
}
```

Ejercicio 1. Escribir una función InsertarNodoFin() que inserte un nodo al final de la lista.
Ejercicio 2. Escribir una función InsertarNodoOrden() que inserte un nodo en la posición de la lista que le corresponda según el valor del dato que almacene.

(*enviar consultas o sugerencias a*: academia@cartagena99.com.es)

3.1.2 Borrado

Cualquier elemento de la lista puede ser eliminado con la misma facilidad con que fue creado. Como ejemplo se presenta a continuación una función que borra el primer elemento de la lista. Es especialmente importante recordar que el espacio de memoria ocupado por los nodos destruidos debe ser devuelto al sistema mediante una llamada a la función free(). Además, una vez destruido un nodo, no debe utilizarse ningún puntero para acceder a él.

Ejercicio. Realizar la función BorrarNodo para eliminar un nodo de una lista.
(*enviar consultas o sugerencias a*: academia@cartagena99.com.es)

3.1.3 Recorrido

A veces es necesario recorrer todos los elementos de una lista para buscar uno en concreto o aplicar una operación sobre todos ellos.

Ejercicio. Implementar una función que recorra una lista buscando un elemento en concreto (*enviar consultas o sugerencias a*: academia@cartagena99.com.es).

3.1.4 Inconvenientes del uso de las listas enlazadas

La flexibilidad de uso que proporcionan las listas enlazadas tiene como contrapartida los siguientes inconvenientes que deben ser cuidadosamente evaluados:

- Uso de más memoria: además de los datos se almacenan los punteros a los siguientes elementos. Este factor es más llamativo cuanto más pequeño sea el tamaño de los datos almacenados en cada nodo.
  Ejercicio: Utilizando un vector como estructura de datos, es posible almacenar 100 elementos de tipo int en 400 bytes. ¿Cuántos bytes son necesarios para almacenar los mismos 100 elementos en la lista dinámica presentada anteriormente? ¿Cuáles serían los valores respectivos si el dato almacenado en cada nodo fuese una estructura de 128 bytes? (*enviar consultas o sugerencias a*: academia@cartagena99.com.es)
- Mayor número de accesos a memoria: cada vez que se accede a un dato mediante un puntero se realizan más accesos a memoria.
- Pérdida del acceso secuencial a los elementos. En un vector es posible acceder directamente a cualquier elemento en cualquier posición del mismo. En el caso de las listas enlazadas, como sólo se mantiene el puntero al primer elemento, es necesario recorrer todos los nodos anteriores a la posición buscada.
- Peor rendimiento del sistema debido a la pérdida de localidad espacial en la memoria cache del procesador.
- Sobrecarga debida al gestor de memoria dinámica. El sistema necesita mantener un mapa con las zonas de memoria que están disponibles y ocupadas en cada momento. La gestión de este mapa requiere que se realicen ciertas operaciones después de cada llamada a malloc() o free(). Este trabajo adicional puede llegar a repercutir significativamente en el tiempo de ejecución de la aplicación.

3.2 Pilas

Una pila es un caso especial de lista en la que tanto la inserción como el borrado se producen por un mismo extremo, la “cima de la pila”. También se conocen como listas LIFO (last in, first out). La operación de inserción de un elemento se conoce generalmente como “push” y la de extracción, como “pop”.

Ejercicio 1. Implementar las funciones Push() y Pop() de una pila.

```
void Push(TNodo ** lista, int dato);
int Pop(Tnodo ** lista);
```

Ejercicio 2. Implementar una función Peek() que devuelva el elemento que ocupa la cima de la pila pero sin eliminarlo de ésta.

```
int Peek(TNodo * lista);
```

Ejercicio 3. Implementar las operaciones de Push() y Pop() sobre un vector de modo que el tamaño máximo de la pila sea conocido.
(*enviar consultas o sugerencias a*: academia@cartagena99.com.es)

3.3 Colas

Las colas son un tipo de listas enlazadas en las que la inserción se realiza por un extremo y la extracción por el otro (extracción implica recuperación y borrado en este caso). Típicamente se utilizan en casos donde se requiere espacio intermedio para almacenar los elementos producidos en un proceso productor hasta que pueden ser procesados, en orden, por un proceso consumidor. Se habla, por tanto, de listas FIFO (first in, first out).

Ejercicio 1. ¿Existe alguna diferencia entre el método de inserción en una cola y el de una lista abierta?
Ejercicio 2. Desarrollar el método de extracción de una cola. Supóngase que se dispone de un puntero al último elemento para hacer más eficientes las operaciones.
(*enviar consultas o sugerencias a*: academia@cartagena99.com.es)

3.4 Listas circulares

Una lista circular es una lista lineal en la que el último elemento se enlaza con el primero. Desde cualquier punto dado, podemos acceder a cualquier punto de la lista. Se evitan casos especiales ya que no existe cabecera ni final de la lista.

3.5 Listas doblemente enlazadas

En ocasiones es necesario recorrer una lista no sólo en sentido de avance sino también en sentido inverso, desde el último elemento al primero. En una lista enlazada normal no es posible realizar esta operación sin recorrer en cada paso la lista desde el principio avanzando un paso menos cada vez. Como una solución para estos casos se dispone de las listas doblemente enlazadas cuyos nodos almacenan no solamente el puntero al siguiente nodo sino también al precedente.

Una lista doblemente enlazada se identifica mediante un puntero al primer y al último elemento:

```
typedef struct tnodo
{
      int dato;
      tnodo * siguiente, * anterior;
} TNodo;
```

El atributo siguiente del último nodo de la lista vale NULL. Asimismo, el atributo anterior del primer nodo de la lista vale NULL también.

Ejercicio. Dado el puntero a cualquier nodo intermedio de una lista doblemente enlazada, escribir la función que elimine ese nodo y reconstruya la lista adecuadamente (*enviar consultas o sugerencias a*: academia@cartagena99.com.es).

3.6 Árboles

Un árbol es una estructura no lineal formada por un conjunto de nodos y un conjunto de ramas.

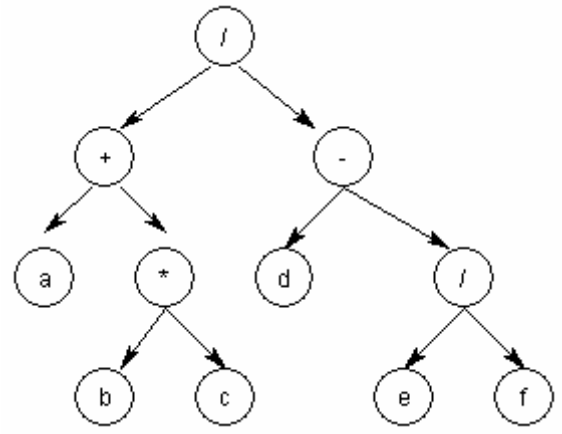
Tenemos un nodo especial llamado *raíz*. Un nodo del que sale una rama se llama *nodo de bifurcación* o *nodo rama*. Un nodo que no tiene ramas se llama *nodo terminal* o *nodo hoja.*

De cada nodo cuelgan nuevos nodos que constituyen un nuevo árbol denominado *subárbol de la raíz.* El número de ramas de un nodo recibe el nombre de *grado* del nodo. Si la raíz tiene nivel 0, el *nivel* de un nodo es igual a la distancia de ese nodo al nodo raíz. El nivel máximo se llama *profundidad* o *altura* del árbol.

3.7 Árboles binarios

Si limitamos los árboles en el sentido de que cada nodo sólo pueda ser de grado 2, tendremos *árboles binarios*, en los que podemos distinguir *árbol derecho* y *árbol izquierdo*. A su vez, cada subárbol es un árbol binario.

Ejemplo. Las fórmulas algebráicas, debido a que los operadores que intervienen son operadores binarios, nos dan un ejemplo de estructura de árbol binario. Por ejemplo:
(a+b*c)/(d-e/f)



Dada la definición de árbol binario, resulta sencilla su representación en un ordenador. Dispondremos de una variable *raíz* para referenciar al árbol y cada nodo contendrá dos enlaces: *izquierdo* y *derecho*:

```
 typedef struct datos nodo;
 struct datos
{
 nodo * izquierdo;
 nodo * derecho;
};
```

3.7.1 Recorrido de árboles binarios

Esencialmente se puede recorrer un árbol binario de tres formas:

- Preorden: raíz, subárbol izquierdo, subárbol derecho.
- Inorden: subárbol izquierdo, raíz, subárbol derecho.
- Postorden: subárbol izquierdo, subárbol derecho, raíz.

Ejercicio. Construir las funciones que recorren un árbol de las tres formas descritas, pasándoles como argumento un puntero al nodo raíz:

```
void preorden (nodo *a);
void inorden (nodo *a);
void postorden (nodo *a);
```

(*enviar consultas o sugerencias a*: academia@cartagena99.com.es)
NOTA: construir estas funciones de forma recursiva.

3.7.2 Árboles binarios de búsqueda (ABB)

Un árbol binario de búsqueda es un árbol ordenado, es decir, las ramas de cada nodo están ordenadas de acuerdo con las siguientes reglas:

- para todo nodo, todas las claves del subárbol izquierdo son menores que la clave del nodo.
- para todo nodo, todas las claves del subárbol derecho son mayores que la clave del nodo.

Ejercicio 1. Realizar una función que, dado un puntero a la raíz de un árbol binario de búsqueda, localice un número entero de la forma más eficiente posible (ambos datos pasados como parámetros a la función). Cuando encuentre el número, devolverá un puntero al nodo que buscamos. Si no se encuentra el número que buscamos, la función devolverá un puntero a NULL.

```
void buscar (int x, nodo ** raiz);
```

(*enviar consultas o sugerencias a*: academia@cartagena99.com.es)

Ejercicio 2. Reutilizar la función del ejercicio anterior para realizar un programa que, dada una secuencia de números enteros, busque cada uno de ellos en un árbol inicialmente vacío: si lo encuentra, incrementará el valor del **contador**; si no lo encuentra, añadirá el nodo con la **clave** e incrementará el contador. Véase la estructura de un nodo:

```
typedef struct datos nodo;
struct datos
{
      int clave;
      int contador;
      nodo * izquierdo;
```

```
        nodo * derecho;
    }
```

Finalmente, se propone que se muestre el contenido del árbol por pantalla recorriéndolo de la forma *inorden*.

(*enviar consultas o sugerencias a*: academia@cartagena99.com.es)

Ejercicio 3. Realizar una función recursiva que elimine un subárbol a partir de un nodo dado.

```
    void borrar_arbol (nodo *a);
```

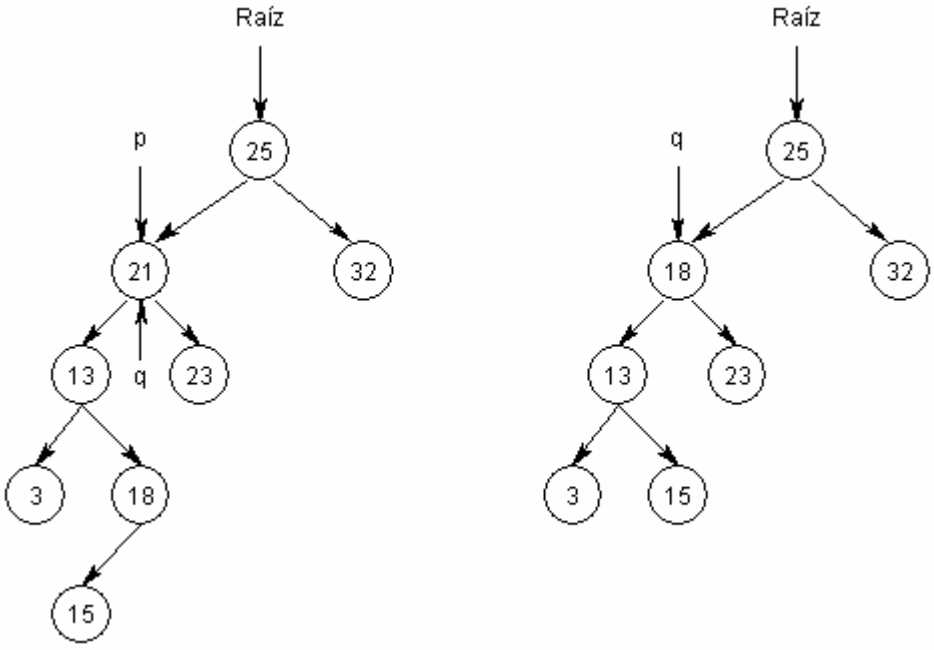
(*enviar consultas o sugerencias a*: academia@cartagena99.com.es)

3.7.3 Borrado en árboles binarios de búsqueda

Se propone buscar un nodo con una clave "x" en un árbol binario cuyas claves estén ordenadas. Si el nodo es un nodo terminal es un ejercicio trivial. Si el nodo que queremos borrar tiene un sólo descendiente, el problema se complica. Pero el caso más complicado es en el que el nodo tiene dos descendientes. En este caso, el nodo que se quiere borrar debe ser reemplazado por una de las siguientes opciones:

- el nodo más a la derecha del subárbol izquierdo.
- el nodo más a la izquierda del subárbol derecho.

En el siguiente gráfico se puede ver el esquema del borrado de un nodo con dos hijos:



El nodo que se quiere borrar es el que tiene la clave 21. Una vez que localizamos el nodo que contiene la clave que queremos borrar, mediante el puntero "p", apuntamos con el puntero "q" a ese mismo nodo. En este ejemplo, sustituimos el nodo que queremos borrar por el nodo más a la derecha del subárbol izquierdo del nodo apuntado por "q".

Ejercicio. Realizar una función que elimine el nodo de un árbol binario de búsqueda correspondiente a la clave que se le pasa como argumento. Habrá que tener en cuenta los tres posibles casos siguientes:

- No hay un nodo con clave igual a “x”.
- El nodo con clave “x” tiene un único descendiente.
- El nodo con clave “x” tiene dos descendientes.

```
void borrar (int x, nodo ** raiz);
```

NOTA: Construir esta función de forma recursiva.
(*enviar consultas o sugerencias a*: academia@cartagena99.com.es)

El pseudocódigo de la función podría ser el siguiente:

- Obtener un puntero “p” a la raíz (argumento de la función).
- Descender por el árbol para buscar el nodo que se quiere borrar.
  i. Si el puntero “p” es NULL, acabar (árbol o subárbol vacío).
  ii. Si la clave del nodo apuntado por “p” es mayor que la clave buscada, llamar a la función “borrar” con el subárbol izquierdo.
  iii. Si la clave del nodo apuntado por “p” es menor que la clave buscada, llamar a la función “borrar” con el subárbol derecho.
  iv. Si no se cumple ninguna de las condiciones anteriores, borrar el nodo. Apuntar “q” al nodo “p”.
    a) Si sólo tiene subárbol izquierdo, apuntar el puntero “p” al subárbol izquierdo y eliminar “q”.
    b) Si sólo tiene subárbol derecho, apuntar el puntero “p” al subárbol derecho y eliminar “q”.
    c) Si tiene ambos subárboles, descender al nodo más a la derecha del subárbol izquierdo, o al nodo más a la izquierda del subárbol derecho. Sustituir el nodo que se quiere borrar por el nodo a la derecha del subárbol izquierdo o por el nodo a la izquierda del subárbol derecho.

3.8 Árboles AVL

Un árbol AVL o *árbol perfectamente equilibrado* es aquel en el que, para todo nodo, el número de nodos en el subárbol izquierdo y en el derecho, difieren como mucho en una unidad.

Una forma sencilla de construir un árbol AVL es, partiendo del nodo raíz, obtener el número de nodos que tenemos en el subárbol derecho y en el izquierdo. Por ejemplo, llamemos “ni” al número de nodos del subárbol izquiero y “nd” al número de nodos del subárbol derecho. La mitad de los nodos los pondremos en el árbol izquierdo y los sobrantes menos uno en el derecho. Esto es:

$$ni = \frac{n}{2}; nd = n - ni - 1$$

La función que construye el árbol será, entonces, recursiva.

Ejercicio. Construir un árbol AVL a partir de un array de enteros introducidos por teclado. Visualizar el árbol una vez construido.

```
nodo * contruir_arbol (int n);
void visualizar_arbol (nodo *a, int n);
```

NOTA: ambas funciones se deben implementar de forma recursiva (*enviar consultas o sugerencias a*: academia@cartagena99.com.es).

4 MANEJO DE FICHEROS EN C

4.1 Funciones C para ficheros

Para dar soporte para el trabajo con ficheros, la biblioteca de C proporciona varias funciones de entrada y salida que permite leer y escribir datos en ficheros y dispositivos. La característica fundamental de las funciones de entrada y salida es que, en el procesamiento de ficheros, se realiza esta entrada y salida de datos a través de un buffer o memoria intermedia.

4.1.1 Funciones y tipos C

Para poder leer o escribir en un fichero, hay que abrir el mismo. Hay tres opciones para abrir un fichero:

- Lectura.
- Escritura.
- Lectura y escritura.

Abrir un fichero significa definir un *stream* que permite el acceso al fichero. Esta definición lleva implícita la definición de un buffer para conectar el *stream* con el fichero en disco. Un *stream* no es más que un puntero a una estructura de tipo FILE.

Abrir un fichero

La función que se utiliza para abrir un fichero en C es *fopen*. Cuando un programa comienza su ejecución, automáticamente se abren tres ficheros:

- stdin: dispositivo de entrada estándar (teclado).
- stdout: dispositivo de salida estándar (pantalla).
- stderr: dispositivo de error estándar (pantalla).

Estos *streams* pueden ser utilizados como argumento de cualquier función que requiere un puntero a una estructura de tipo FILE.

El prototipo de la función *fopen* es:

```
FILE * fopen (const char * nombre_fichero, const char *modo);
```

Si la operación no se realiza con éxito, se devuelve un puntero nulo.

Los modos que podemos utilizar son los siguientes:

- "r": abrir un fichero para leer. Si el fichero no existe o no se encuentra, se obtiene un error.
- "w": abrir un fichero para escribir. Si el fichero no existe, se crea; si existe, su contenido se destruye para ser creado de nuevo.

- "a": abrir un fichero para añadir información al final del mismo. Si el fichero no existe, se crea.
- "r+": abrir un fichero para leer y escribir. El fichero debe existir.
- "w+": abrir un fichero para leer y escribir. Si el fichero no existe, se crea; si existe, su contenido se destruye para ser creado de nuevo.
- "a+": abrir un fichero para leer y añadir. Si el fichero no existe, se crea.

Además, a continuación de la letra que indica el modo, se puede añadir la letra "b": "rb", "wb", "ab"... Esto se utiliza para indicar que el fichero debe abrirse en modo *binario*. En sistemas operativos UNIX, un fichero de texto realiza el cambio de línea con el carácter '\n', pero en sistemas MS-DOS se realiza con CR+LF. Por lo tanto, en sistemas MS-DOS cuando se abre un fichero de texto, los caracteres CR+LF se sustituyen por '\n'. El problema aparece cuando esos caracteres no se corresponden con texto sino, por ejemplo, con un valor numérico que estemos recuperando de un fichero de disco y 2 bytes de ese valor coincidan con la representación de los caracteres CR+LF. Por ello se utiliza el modo binario, en el que no se realizan estas conversiones.

Función freopen

La función *freopen* cierra el fichero actualmente asociado con el puntero *stream* y reasigna *stream* al fichero indicado. La utilidad principal es redireccionar *stdin, stdout* o *stderr*, a ficheros especificados.

```
FILE * freopen (const char * nom_fichero, const char * modo, FILE *
stream);
```

Cerrar un fichero

Cuando finalizamos el trabajo con un fichero, debemos cerrarlo. Automáticamente se cierran los ficheros al finalizar el programa, pero es recomendable hacerlo de forma explícita porque el número de ficheros abiertos al mismo tiempo es limitado.

La función que se utiliza es *fclose*:

```
int fclose (FILE * pf);
```

Esta función devuelve un cero si se realiza la operación con éxito; si se produce algún error, devuelve EOF.

Final de un fichero

Cuando se crea un fichero, automáticamente se añade la marca de final de fichero. Si se intenta leer más allá de dicha marca, se activa un indicador que podemos consultar con la función *feof*:

```
int feof (FILE *pf);
```

Esta función devuelve un valor distinto de cero cuando se intenta leer del fichero y encontramos un *eof* ("end of file").

Funciones de entrada y salida

- Entrada y salida carácter a carácter: *fputc, fgetc*.
- Entrada y salida de cadenas de caracteres: *fputs, fgets*.
- Entrada y salida con formato: *fprintf, fscanf*.
- Entrada y salida utilizando registros. Se almacenan los datos en formato binario (no confundir con el modo binario para abrir los ficheros).

```
size_t fread (void * buffer, size_t n, size_t c, FILE *pf);
size_t fwrite (const void * buffer, size_t n, size_t c, FILE *pf);
```

Ambas devuelven el número de elementos leídos/escritos. "n" indica el tamaño de cada uno de los registros y "c" el número de registros.

Ejercicio 1. Realizar un programa que obtenga cadenas de caracteres por teclado y las almacene en un fichero de texto. Además, debe ser capaz de visualizar las cadenas de caracteres que se han almacenado en el fichero de texto.
Ejercicio 2. Realizar un programa que copie un archivo en otro.
Ejercicio 3. Dada la siguiente estructura:

```
typedef struct _ficha_
{
      char nombre[20];
      char direccion[40];
      int edad;
      long telefono;
}tficha;
```

Realizar un programa que gestione una pequeña base de datos en un fichero con este tipo de información. La información se introducirá por teclado y se almacenará en un fichero en formato binario, no en texto claro.
(*enviar consultas o sugerencias a*: academia@cartagena99.com.es)

4.1.2 Funciones C para ficheros de acceso aleatorio

Existen dos formas para acceder a los ficheros. Hasta el momento, estábamos utilizando funciones que acceden de forma secuencial a los mismos. Si no queremos leer consecutivamente los registros almacenados en un fichero, podemos utilizar funciones especiales para ello.

La función *fseek* mueve el puntero de lectura y escritura asociado con el fichero apuntado por *pf*, a una nueva localización desplazada *desp* bytes de la posición dada por el argumento *pos*. Podemos tener un desplazamiento positivo o negativo.

```
int fseek (FILE *pf, long desp, int pos);
```

El argumento *pos* puede ser alguno de los siguientes valores:

- SEEK_SET: principio del fichero.
- SEEK_CUR: posición actual del puntero de L/E.
- SEEK_END: final del fichero.

Disponemos de dos funciones más para el acceso aleatorio en ficheros:

```
long ftell (FILE *pf);
void rewind (FILE *pf);
```

La función *ftell* da como resultado la posición actual del puntero de L/E relativa al principio del fichero.

La función *rewind* pone el puntero de L/E al comienzo del fichero.

Ejercicio. Reutilizar el ejercicio 3 del apartado anterior e implementar una función que busque la ficha de una persona de forma aleatoria. Como argumento se le indicará la posición que ocupa la ficha que queremos buscar.
(*enviar consultas o sugerencias a*: academia@cartagena99.com.es)

4.2 Archivos secuenciales

En estos archivos, la información sólo puede leerse y escribirse empezando desde el principio del archivo.

Los archivos secuenciales tienen algunas características que hay que tener en cuenta:

1. La escritura de nuevos datos siempre se hace al final del archivo.
2. Para leer una zona concreta del archivo hay que avanzar siempre, si la zona está antes de la zona actual de lectura, será necesario "rebobinar" el archivo.
3. Los ficheros sólo se pueden abrir para lectura o para escritura, nunca de los dos modos a la vez.

Esto es en teoría, por supuesto, en realidad C no distingue si los archivos que usamos son secuenciales o no, es el tratamiento que hagamos de ellos lo que los clasifica como de uno u otro tipo.

Pero hay archivos que se comportan siempre como secuenciales, por ejemplo los ficheros de entrada y salida estándar: stdin, stdout y stderr.

Tomemos el caso de stdin, que suele ser el teclado. Nuestro programa sólo podrá abrir ese fichero como de lectura, y sólo podrá leer los caracteres a medida que estén disponibles, y en el mismo orden en que fueron tecleados.

Lo mismo se aplica para stdout y stderr, que es la pantalla, en estos casos sólo se pueden usar para escritura, y el orden en que se muestra la información es el mismo en que se envía.

Trabajar con archivos secuenciales tiene algunos inconvenientes. Por ejemplo, imaginemos que tenemos un archivo de este tipo en una cinta magnética. Por las características físicas de este soporte, es eviente que sólo podemos tener un fichero abierto en cada unidad de cinta. Cada fichero puede ser leído, y también sobrescrito, pero en

general, los archivos que haya a continuación del que escribimos se perderán, o bien serán sobreescritos al crecer el archivo, o quedará un espacio vacío entre el final del archivo y el principio del siguiente.

Lo normal cuando se quería actualizar el contenido de un archivo de cinta añadiendo o modificando datos, era abrir el archivo en modo lectura en una unidad de cinta, y crear un nuevo fichero de escritura en una unidad de cinta distinta. Los datos leídos de una cinta se editan o modifican, y se copian en la otra secuencialmente.

Cuando trabajemos con archivos secuenciales en disco haremos lo mismo, pero en ese caso no necesitamos dos unidades de disco, ya que en los discos es posible abrir varios archivos simultáneamente.

En cuanto a las ventajas, los archivos secuenciales son más sencillos de manejar, ya que requieren menos funciones, además son más rápidos, ya que no permiten moverse a lo largo del archivo, el punto de lectura y escritura está siempre determinado.

En ocasiones pueden ser útiles, por ejemplo, cuando sólo se quiere almacenar cierta información a medida que se recibe, y no interesa analizarla en el momento. Posteriormente, otro programa puede leer esa información desde el principio y analizarla. Este es el caso de archivos "log" o "diarios" por ejemplo, los servidores de las páginas WEB pueden generar una línea de texto cada vez que alguien accede al una de las páginas y las guardan en un fichero secuencial.

4.2.1 Ordenar archivos secuenciales

Si los ficheros que queremos ordenar son pequeños, lo más sencillo es leerlos enteros a memoria y ordenarlos como si se tratara de un array de registros. El problema comienza cuando los archivos son grandes y no pueden ser tratados enteros en memoria.

Por ello, típicamente se utiliza un algoritmo llamado *mezcla natural*. Este algoritmo, en cada iteración o “pasada”, realiza dos funciones:

- Fase de distribución.
- Fase de mezcla.

Inicialmente, partimos del fichero origen *c*. Generamos entonces dos ficheros auxiliares: *a* y *b*. Se distribuyen los datos del fichero *c* en los ficheros *a* y *b*, según divisiones por tramos. Después, se van mezclando en *c*, de nuevo, los datos de *a* y *b* ordenándolos por tramos.

Ejemplo. En el fichero *c* tenemos los datos iniciales, que separaremos por tramos. Dicha separación se indica con un guión “-“.

```
c: 18 32 – 10 60 – 14 42 44 68 – 12 24 30 48
```

En la primera fase, se realiza la distribución por tramos en los ficheros *a* y *b*:

```
a: 18 32 – 14 42 44 68
b: 10 60 – 12 24 30 48
```

En la segunda fase, se leen los datos por tramos de forma ordenada y se almacenan en el fichero *c*:

```
c: 10 18 32 60 – 12 14 24 30 42 44 48 68
```

Repetimos ambas fases y obtenemos:

```
a: 10 18 32 60
b: 12 14 24 30 42 44 48 68
c: 10 12 14 18 24 30 32 42 44 48 60 68
```

El algoritmo se repite hasta que el número de tramos es 1. El número de iteraciones que necesitaremos hacer es la mitad del número de tramos, si el número de tramos es par, y la mitad del número de tramos más uno, si es impar.

Ejercicio. Implementar el algoritmo *mezcla natural* para poder ordenar de mayor a menor un fichero de acceso secuencial que contenga una secuencia de números enteros. Para ello, dividir el problema en las siguientes funciones:

```
void mezcla_natural (FILE * pf);
int distribuir (FILE * pf, FILE *fa, FILE *fb);
int mezclar (FILE *fa, FILE *fb, FILE *pf);
```

NOTA: la función *mezcla_natural* utiliza las funciones *distribuir* y *mezclar*, las cuales devuelven el número de tramos que quedan.

(*enviar consultas o sugerencias a*: academia@cartagena99.com.es)

4.3 Archivos de acceso aleatorio

Los archivos de acceso aleatorio son más versátiles, permiten acceder a cualquier parte del fichero en cualquier momento, como si fueran arrays en memoria. Las operaciones de lectura y/o escritura pueden hacerse en cualquier punto del archivo.

En general se suelen establecer ciertas normas para la creación, aunque no todas son obligatorias:

1. Abrir el archivo en un modo que te permita leer y escribir. Esto no es imprescindible, es posible usar archivos de acceso aleatorio sólo de lectura o de escritura.
2. Abrirlo en modo binario, ya que algunos o todos los campos de la estructura pueden no ser caracteres.
3. Usar funciones como fread y fwrite, que permiten leer y escribir registros de longitud constante desde y hacia un fichero.
4. Usar la función fseek para situar el puntero de lectura/escritura en el lugar apropiado de tu archivo.

Ejercicio 1. Realizar una función que calcule el tamaño en bytes de un fichero.
(*enviar consultas o sugerencias a*: academia@cartagena99.com.es)

4.3.1 Ordenar archivos de acceso aleatorio

Como hemos dicho, tratar un fichero de acceso aleatorio es semejante a tratar elementos de un array. Por lo tanto, podremos utilizar cualquiera de los métodos de ordenación comunes conocidos.

Método de la burbuja

Es uno de los algoritmos que más tiempo de ejecución requieren, es decir, es de complejidad elevada. Partiendo de una lista de "n" elementos, el algoritmo sería el siguiente:

1.- Comparamos el primer elemento con el segundo, el segundo con el tercero, el tercero con el cuarto, etcétera. Cuando el resultado de una comparación sea "mayor que", se intercambian los valores de los elementos comparados. Con esto conseguimos llevar el valor mayor a la posición "n".

2.- Repetimos el punto 1, ahora para los n-1 primeros elementos de la lista. Con esto conseguimos llevar el valor mayor de éstos a la posición "n-1".

3.- Repetimos el punto 1, ahora para los n-2 primeros elementos de la lista y así sucesivamente.

4.- El proceso termina después de repetir el proceso n-1 veces, o cuando al finalizar la ejecución de la iteración i-ésima no haya habido ningún cambio.

Método de inserción

Este algoritmo es de complejidad muy inferior al método de la burbuja. El procedimiento es el siguiente:

1.- Inicialmente, se ordenan los dos primeros elementos del array.

2.- Se inserta el tercer elemento en la posición correcta con respecto a los dos primeros. A continuación se repite con el cuarto elemento, y así sucesivamente.

3.- El proceso termina cuando se ha insertado el último elemento del array.

Ejercicio. Realizar una función que, dado un array de números enteros y el número de elementos del mismo, los ordene de forma ascendente mediante el método de inserción.
(*enviar consultas o sugerencias a*: academia@cartagena99.com.es)

Método quicksort

Está considerado como el mejor método de ordenación. El algoritmo es el siguiente:

1.- Se selecciona un valor perteneciente al rango de valores del array. Este valor se puede escoger aleatoriamente, o haciendo la media de un pequeño conjunto de

valores tomados del array. El valor óptimo sería la mediana (el valor que es menor o igual que los valores correspondientes a la mitad de elementos del array y mayor o igual que los valores correspondientes a la otra mitad). No obstante, incluso en el peor de los casos (el valor escogido está en un extremo), el método quicksort funcionará correctamente.

2.- Se divide el array en dos partes: una con todos los elementos menores que el valor seleccionado y otra con todos los elementos mayores o iguales.

3.- Se repiten los puntos 1 y 2 para cada una de las partes en la que se ha dividido el array, hasta que esté ordenado.

4.4 Ficheros de índices

Mantener grandes ficheros de datos ordenados es muy costoso, ya que requiere mucho tiempo de procesador. Afortunadamente, existe una alternativa mucho mejor: utilizar índices.

Para indexar un archivo normalmente se suele generar un archivo auxiliar de índices. Existen varios métodos. El más sencillo es crear un archivo plano que sólo contenga registros con dos campos: el campo o la expresión por la que queremos ordenar el archivo, y un campo con un índice que almacene la posición del registro indicado en el archivo de datos.

Por ejemplo, supongamos que tenemos un archivo de datos con la siguiente estructura de registro:

```
struct stRegistro {
  char nombre[32];
  char apellido[2][32];
  char telefono[12];
  char calle[45];
  int numero;
  char ciudad[32];
      char fechaNacimiento[9]; // formato AAAAMMDD: Año, mes y
                               //día
  char estadoCivil;
  int hijos;
}
```

Imaginemos que necesitamos buscar un registro a partir del número de teléfono. Si no tenemos el archivo ordenado por ese campo, estaremos obligados a leer todos los registros del archivo hasta encontrar el que buscamos, y si el número no está, tendremos que leer todos los registros que existan.

Si tenemos el archivo ordenado por números de teléfono podremos aplicar un algoritmo de búsqueda. Pero si también queremos hacer búsquedas por otros campos, estaremos obligados a ordenar de nuevo el archivo.

La solución es crear un fichero de índices, cada registro de este archivo tendrá la siguiente estructura:

```
struct stIndiceTelefono {
  char telefono[12];
  long indice;
```

}

Crearemos el fichero de índices a partir del archivo de datos, asignando a cada registro el campo "telefono" y el número de registro correspondiente. Veamos un ejemplo:

```
000: [Fulanito] [Pérez] [Sanchez] [12345678] [Mayor] [15] [Lisboa]
     [19540425] [S] [0]
001: [Fonforito] [Fernandez] [López] [84565456] [Baja] [54]
     [Londres] [19750924] [C] [3]
002: [Tantolito] [Jimenez] [Fernandez] [45684565] [Alta] [153]
     [Berlin] [19840628] [S] [0]
003: [Menganito] [Sanchez] [López] [23254532] [Diagonal] [145]
     [Barcelona] [19650505] [C] [1]
004: [Tulanito] [Sanz] [Sanchez] [54556544] [Pez] [18] [Dublín]
     [19750111] [S] [0]
```

Generamos un fichero de índices:

```
[12345678][000]
[84565456][001]
[45684565][002]
[23254532][003]
[54556544][004]
```

Y lo ordenamos:

```
[12345678][000]
[23254532][003]
[45684565][002]
[54556544][004]
[84565456][001]
```

Ahora, cuando queramos buscar un número de teléfono, lo haremos en el fichero de índices, por ejemplo el "54556544" será el registro número 3, y le corresponde el índice "004". Con ese índice podemos acceder directamente al archivo de datos, y veremos que el número corresponde a "Tulanito Sanz Sanchez".

Podemos tener más ficheros de índices para otros campos. El mayor problema es mantener los ficheros de índices ordenados a medida que añadimos, eliminamos o modificamos registros. Pero al ser los registros de índices más pequeños, los ficheros son más manejables, pudiendo incluso almacenarse en memoria en muchos casos.

4.4.1 Búsqueda binaria

La búsqueda binaria es un método muy eficiente de búsqueda, al contrario que la búsqueda secuencial. Puede aplicarse a los arrays clasificados, es decir, sus elementos deben estar ordenados ascendentemente. El algoritmo es el siguiente:

1.- Se selecciona el elemento del centro o aproximadamente del centro del array. Si el valor del elemento que se quiere buscar no coincide con el elemento seleccionado, continuar por el paso 2.

2.- Si el valor del elemento que se busca es mayor que el elemento seleccionado, se busca en la segunda mitad del array.

3.- Si, por el contrario, el valor del elemento que se busca es menor que el elemento seleccionado en el paso 1, se busca en la primera mitad del array.

4.- La búsqueda en cualquiera de las dos mitades del array se realiza calculando un nuevo elemento central de la mitad del array, de tal forma que nuevamente tenemos la mitad del array dividida en dos partes.

5.- Repetiremos los pasos hasta que se encuentra el valor que queremos o hasta que el intervalo de búsqueda sea nulo (el elemento buscado no existe en el array).

Ejercicio. Dado un array de números enteros y un valor que se quiera buscar, realizar una función que implemente la búsqueda binaria de ese valor en ese array.

(*enviar consultas o sugerencias a*: academia@cartagena99.com.es)

5 C++ CONCEPTOS BÁSICOS. PROGRAMACIÓN ORIENTADA A OBJETOS

5.1 Programación Orientada a Objetos (POO)

C++ es un lenguaje orientado a objetos que tiene la peculiaridad de ser compatible con C, es decir, que podemos usar nuestro compilador de C++ para compilar programas escritos en C, e incluso, podemos mezclar en el mismo programa ambas filosofías. Además incorpora otra serie de mejoras respecto a C, algunas de ellas las veremos a continuación.

Más adelante se define qué es un objeto, una clase, etc. pero para poder hacernos una idea provisional, podríamos decir que una clase es algo así como una estructura con funciones, y un objeto es una variable de esa estructura. Recuérdese: La clase es la definición y el objeto la declaración de una variable, que también se llama instanciación.

Esta idea genérica que hemos dado de objeto como una estructura con funciones podría sugerir que tan sólo existe esta pequeña diferencia entre la programación estructurada y la orientada a objetos; sin embargo, los lenguajes orientados a objetos incorporan toda una serie de características para potenciar el manejo de clases, de modo que, cuando se programa en un lenguaje orientado a objetos todo son clases y objetos. La filosofía de programación cambia bastante entre un tipo de programación y el otro.

Se dice que, mientras en programación estructurada el programador debe plantearse "*¿qué tiene que hacer el programa?*", en programación orientada a objetos el programador debe plantearse "*¿de qué trata el programa?*". En el primer caso la respuesta para un ejemplo clásico, sería: *Un programa que gestiona las notas de alumnos y que te dice la nota media de cada alumno, el número de suspensos, etc.* En el segundo caso sería: *Un programa que trata de alumnos, cada objeto "alumno" tendrá cinco notas y deberá calcular su nota media, su número de suspensos, etc.*

Para entender la programación orientada a objetos debemos olvidar en cierta medida la programación estructurada y pensar en objetos, entendiendo por *objeto* una encapsulación de un conjunto de datos y de los métodos para manipular éstos. Es decir, que un objeto, es como un conjunto de datos capaz de además hacer cosas con esos datos. Un objeto puede guardar una serie de números y además puede tener definidas funciones que calculen la media, el mayor elemento, la suma de todos, etc.

Los objetos pueden establecer una comunicación entre ellos. Los eventos que suceden (una pulsación del ratón en un botón) hacen que los objetos sobre los que se producen generen mensajes para otros objetos que reaccionarán, según su implementación. Por lo tanto, la POO es una forma de programación que utiliza objetos que responden a sucesos. Los mensajes entre los objetos originan cambios en el estado del objeto que recibe el mensaje.

Las características de la POO son: abstracción, encapsulamiento, herencia y polimorfismo.

Los mecanismos básicos de la POO son, por tanto: objetos, mensajes, métodos, clases y subclases.

Objetos

Un programa tradicional se compone de prodecimientos y datos. Un programa orientado a objetos se compone sólo de objetos. Un objeto es una encapsulación genérica de datos y de los procedimientos para manipularlos. Esos datos se denominan *atributos* y los procedimientos *métodos*.

Los métodos definen el comportamiento del objeto. Los atributos definen el estado del mismo y son manipuladas por los métodos.

Mensajes

Los objetos de un programa orientado a objetos reciben, interpretan y responden a mensajes de otros objetos. El conjunto de mensajes a los que un objeto puede responder se denomina *protocolo*.

En C++, un mensaje está asociado con el prototipo de una función miembro de tal forma que, cuando se produce ese mensaje, se ejecuta la correspondiente función miembro de la clase a la que pertenece el objeto. Es decir, el envío de un mensaje equivale en C++ a llamar a la función miembro.

Métodos

Un método, que en C++ se denomina *función miembro*, se implementa en una *clase*, y determina cómo tiene que actuar el objeto cuando recibe un mensaje. La descripción de un método se denomina *operación*.

Clases

Una clase se puede considerar una plantilla para crear objetos de esa clase o tipo. Describe los métodos y atributos que definen las características comunes a todos los objetos de esa clase. Se trata entonces de abstraer los métodos y atributos comunes a un conjunto de objetos y encapsularlos en una clase.

Por lo tanto, un objeto es la concreción de una clase.

Subclases

Una característica primaria que define un sistema orientado a objetos es la manera en que trata las relaciones estructurales y semánticas entre clases de objetos y elimina la redundancia de almacenar el mismo atributo o método más veces de lo necesario. La clase hija o subclase hereda los atributos y métodos de su clase padre. A la clase padre también se le llama *clase base* y a la hija, *clase derivada*.

5.2 Aportaciones de C++

Datos miembro *static*

Cada objeto mantiene una copia de los datos miembro de la clase, pero no de las funciones miembro, de las cuales sólo existe una copia para todos los objetos de la case.

Sin embargo, cuando un dato miembro se declara *static*, significa que sólo habrá 1 copia del dato para todos los objetos de esa clase.

Ejemplo:

```
class C
{
      int a, b, c;
      static int d;
};

C ob1, ob2, ob3;
```

En el ejemplo tenemos tres objetos de la clase C. Cada objeto tiene una copia de los datos miembro a, b y c. Pero los tres objetos comparten la copia del dato miembro d.

5.2.1 Argumentos por omisión en funciones

En C++ podemos declarar parámetros por omisión en la declaración de las funciones. Se realizar esto desde un determinado parámetro hasta el final.

Ejemplo.

```
#include <iostream>
using namespace std;

void visualiza (int a = 1 , int b = 2, int c = 3, int d = 4)
{
      cout << "a = " << a;
      cout << "b = " << b;
      cout << "c = " << c;
      cout << "d = " << d;

}
void main()
{
      visualiza();
      visualiza(2);
      visualiza(2,1);
      visualiza(2,1,4,5);
}
```

Omitir un argumento en la llamada implica omitir todos los argumentos que le siguen, y especificar los que le preceden.

La inicialización se hace sobre el prototipo de la función.

5.2.2 Funciones sobrecargadas

En C++ se permite sobrecargar funciones analizando el tipo de parámetros que se le pasan y el tipo devuelto. Así, podemos tener:

```
int pot (int, int);
double pot (double, double);
double pot (int, double);
double por (double, pot);
```

El compilador resuelve cuál de las funciones con el mismo nombre es invocada comparando los parámetros reales con los formales. Si no encuentra la función que corresponde exactamente, realizaría las conversiones permitidas sobre los parámetros reales para buscar una función.

No se pueden declarar dos funciones que difieran sólo en el valor devuelto.

5.2.3 Ambigüedades

Tanto la sobrecarga de funciones como los argumentos por omisión en funciones pueden producir algunas ambigüedades. Por ejemplo:

```
int fn (int, int);
int fn (int, int, int = 1);
```

Si llamamos a la función: `fn (3, 4);`

Se producirá un error por ambigüedad, ya que el compilador no sabe si se está llamando a la función con el primer prototipo, indicándole todos sus parámetros, o se está llamando a la segunda, con el tercer argumento por omisión. Si el segundo prototipo fuera:

```
int fn (int, int, int);
```

No habría problema con la ambigüedad.

5.2.4 Referencias

Una referencia es un nombre alternativo (sinónimo) para un objeto. Por ejemplo:

```
int y = 10;
int &x = y;
```

Tenemos declarado un entero “y” con valor 10 y una referencia “x” que indica al compilador que la variable “y” tiene otro nombre también que es “x”. Una referencia no es una copia de la variable referenciada, sino que es la misma variable con otro nombre.

Ejemplo.

```
int conta = 0;
int &cont = conta;

con++;
cout << conta << endl;
cout << con << endl;
```

En el ejemplo podemos ver cómo se muestra por pantalla el mismo valor: 1.

Una referencia tiene que ser inicializada al declararla. Su valor no puede ser alterado después de haberla inicializado, así que siempre se referirá al mismo objeto. Tampoco puede ser desreferenciada con el operador *.

Ejercicio. Dado el siguiente código, ¿qué se mostrará por pantalla? (*enviar consultas o sugerencias a*: academia@cartagena99.com.es)

```
int a[5] = {10, 20, 30, 40, 50};
int &rUltimo = a[4];
int &rElemento = a[0];
int * p;

while (rElemento <= rUltimo)
{
      p = &rElemento;
      cout << *p << ", ";
      rElemento++;
}
```

-Paso de parámetros por referencia

Para pasar una variable por referencia en los argumentos de una función, podemos hacerlo de dos formas:

1.- Pasar la dirección del parámetro real a su correspondiente parámetro forma, el cual tiene que ser un puntero. Por ejemplo:

```
void intercambio (int * x, int * y);

void main ()
{
      int a = 20, b = 30;
      intercambio (&a, &b);
      printf("a vale %d y b vale %d\n", a, b);
}

void intercambio (int *x, int *y)
{
      int z = *x;
      *x = *y;
      *y = z;
}
```

2.- Declarar el parámetro formal como una referencia al parámetro real que se quiere pasar por referencia. Por ejemplo:

```
void intercambio (int &, int &);

void main ()
{
      int a = 20, b = 30;
      intercambio (a, b);
      printf("a vale %d y b vale %d\n", a, b);
}

void intercambio (int &x, int &y)
{
      int z = x;
      x = y;
      y = z;
}
```

-Referencia como valor devuelto

Una función puede declararse para que devuelva una referencia. Hay dos razones para hacerlo:

1.- La información que se devuelve es un objeto grande y es más eficiente devolver una referencia que una copia del objeto. Cuando una función devuelve una referencia, el objeto referenciado debe ser *static* dentro de la función o estaremos referenciando a un objeto que ha desaparecido al salir de la función.

2.- El tipo de la función debe ser un *l-value*. Por ejemplo, operadores sobrecargados. Las expresiones que se refieren a ubicaciones de memoria se llaman expresiones "l-value" (*left value*). Representa una región de almacenamiento que puede aparecer a la izquierda del signo igual (=). Sin embargo, un *r-value* se utiliza a veces para describir el valor de una expresión y para distinguirlo de un *l-value*. Todo "l-value" es un "r-value" pero no al revés.

Ejemplo.

```
struct punto
{
      int x;
      int y;

      int &cx()
      {
            return x;
      }
      int & cy()
      {
            return y;
      }
};

void main()
{
      punto origen;

      origen.cx() = 60;
      origen.cy() = 80;

      cout << "x = " << origen.cx() << endl;
      cout << "y = " << origen.cy() << endl;
}
```

El valor devuelto por la función cx es una referencia al miembro "x" de la estructura "punto". El resultado es que cx actúa como un nombre alternativo para "x". Esto significa que la llamada a la función puede aparecer a la izquierda o a la derecha de un operador de asignación.

5.2.5 Operadores *new* y *delete*

Se utilizan para llevar a cabo la asignación y liberación de una zona de memoria de forma dinámica. Se utilizan así:

```
puntero = new tipo-del-objeto;
puntero = (new tipo-del-objeto);

delete puntero;
delete [] puntero;
```

Ejemplo.

```
long * pl = new long;
float *pf = new (float);

int n_elementos;

cin >> n_elementos;

int *a = new int[n_elementos];

int d1 = 5, d2 = 10;
int ** d = new double *[d1*2];
for (int i = 0; i < d1; i++)
      d[i] = new double[d2];

delete pl;
delete [] a;

for (i = 0; i< d1; i++)
      delete [] d[i];
delete [] d;
```

5.2.6 Otros

- Conversión explícita del tipo de una expresión:
  (nombre-de-tipo)expresion
  nombre-de-tipo(expresion)
- Conversión del tipo void *: en ANSI C la conversión a tipo (void *) a otro tipo se puede hacer de forma implícita. En C++ hay que hacerla de forma explícita.
- Identificadores y estructuras. En C++, el nombre de una estructura definida dentro de un bloque oculta la visibilidad de un identificador con el mismo nombre definido fuera de este bloque. Además, una estructura define su propio ámbito.
- El operador de resolución de ámbito (::). Permite acceder a una variable global cuya visibilidad ha sido ocultada por una variable local. También se usa para especificar a qué clase pertenece una determinada función miembro.

- Entrada y salida estándar. (#include <iostream.h>). Disponemos de diferentes *streams* para la entrada y salida estándar: cin, cout, cerr. (Operador de extracción (>>) y operador de inserción (<<).

  Ejemplo:

```
int numero;
cout << "Hola mundo desde C++.";
cout << "Introduzca un numero";
cin >> numero;
```

5.3 Trabajando con clases

5.3.1 Clases

Si utilizamos programación estructurada, se nos presentan los siguientes inconvenientes:

- No hay ningún mecanismo que prevenga la manipulación directa de los miembros de las estructuras de datos. Se les puede asignar datos inválidos por ejemplo.
- Al no organizar el desarrollo alrededor de los datos manipulados, sino alrededor de las funcionalidades, un cambio en la estructura de datos obligaría a modificar los programas que utilizan esa estructura.

En el diseño orientado a objetos, nos centramos en primer lugar en los datos, a los que se asocian posteriormente los procedimientos o interfaz de acceso a los datos. El resultado es un tipo de objeto que en C++ se denomina clase.

5.3.2 Definición de clases

La definición de una clase consta de dos partes: nombre de la clase y cuerpo de la clase.

```
class nombre_clase
{
      cuerpo de la clase
} [lista de declaraciones];
```

En el cuerpo de la clase se suelen poner especificadores de acceso (public, protected y private), atributos y métodos.

Ejemplo.

```
class Test
{
      private:
            int n_entero;
            float n_real;
```

```
        protected:
              char *p;
              int EsNegativo(void *, char);
        public:
              Test (int i, float r);
              void Asignar (char *c);
              void Visualizar (int a, float b, char *c);
};
```

Por omisión, los datos miembro de una clase son privados.

Podemos declarar como datos miembro, otra clase.

Para definir una función miembro fuera del cuerpo de la clase, hay que indicar a qué clase pertenece. Se hace de la siguiente forma:

```
void Test::Asignar (char *c )
{
      //cuerpo de la función
}
```

Si una función miembro se define dentro del cuerpo de la clase, no hace falta esta última indicación, y se considera como una función *inline*.
Los datos miembro de una clase pueden ser:

- public: es accesible desde cualquier parte del programa donde el objeto de la clase sea accesible.
- private: puede ser accedido sólo por funciones miembro de la misma clase o por funciones amigas (*friend*) de su clase. No puede ser accedido por funciones globales ni por funciones de una clase derivada.
- protected: se comporta igual que uno privado para las funciones globales, pero actúa como un miembro público para las funciones miembro de una clase derivada.

Ejercicio. Definir una clase CFecha cuyos atributos sean: dia, mes y año. Definir como funciones miembro una función para asignar la fecha y otra para obtenerla. A continuación realizar un programa que declare un objeto de esa clase y lo utilice. (*enviar consultas o sugerencias a*: academia@cartagena99.com.es)

5.3.3 El puntero implícito this

Cada objeto de una determinada clase mantiene su propia copia de los datos miembro de la clase, pero no de las funciones miembro, de las cuales sólo existe una copia para todos los objetos de esa clase. Por lo tanto, para que una función miembro conozca la identidad del objeto particular para el cual dicha función ha sido invocada, C++ proporciona un puntero al objeto denominado *this*. Por ejemplo, si declaramos un objeto CFecha fecha1 y a continuación le enviamos el mensaje *AsignarFecha*:

```
fecha1.AsignarFecha();
```

C++ define un puntero *this* para permitir referirse al objeto *fecha1* en el cuerpo de la función que se ejecuta como respuesta al mensaje. La definición es:

```
CFecha *const this = &fecha1;
```

5.3.4 Funciones miembro y objetos constantes

Declarar un objeto *const* hace que cualquier intento de modificar dicho objeto sea detectado durante la compilación. Si declaramos un objeto:

```
const CFecha cumpleanyos;
```

Sería un error que las funciones miembro invocadas cuando se envía un mensaje a este objeto no fueran también constantes. Para declarar las funciones miembro constantes, se hace así:

```
int FechaCorrecta() const;
```

Y su definición:

```
int CFecha::FechaCorrecta() const
{
      //Cuerpo de la función
}
```

Esto supone que el puntero *this* quede implícitamente declarado constante a un objeto constante:

```
const CFecha *const this;
```

Si en la función miembro *FechaCorrecta* se invoca a otra función miembro que no sea constante, habrá que hacer la conversión cast:

```
((CFecha *)this)->Bisiesto();
```

5.3.5 Constructores

Hasta ahora, cuando declarábamos una variable en un programa, el compilador de forma automática reserva la memoria para la variable y, si es una variable global o *static*, inicializa su valor a cero en el caso de variables numéricas o a NULL en el caso de punteros o caracteres; si se trata de una variable local, no se conoce el valor inicial.

Además de ocurrir esto con tipos de datos predefinidos (enteros, reales…), también ocurre lo mismo con tipos definidos por el usuario: estructuras, tipos enumerados…

```
typedef struct _ficha_
{
      char nombre[20];
      int edad;
};
```

El compilador reservará, al declarar una variable del tipo de la estructura de datos, memoria para cada uno de los elementos de la estructura.

Esto se realiza porque existe un *constructor* por defecto u omisión que realiza estas tareas. Sin embargo, si quisiéramos inicializar los valores de las variables locales al declararlas o la estructura tuviera datos miembro que fueran punteros, no estaríamos reservando memoria automáticamente con este constructor por omisión.

En C++, en el cuerpo de una clase podemos declarar un constructor que inicialice los atributos del objeto en el momento de declararlo, y además que reserve memoria dinámica para los atributos que sean punteros. Además, el constructor por omisión de C++ puede inicializar los atributos.

Ejemplo.

```
class CComplejo
{
      public:
            double real, imag;
            void AsignarComplejo (double, double);
            void ObtenerComplejo (double *, double *);
};

CComplejo c1 = {1.5, -2};
```

En el ejemplo, creamos un objeto c1 de la clase CComplejo y se inicializan los valores de sus atributos automáticamente mediante el constructor por omisión. Esto tiene las siguientes restricciones:

- La clase no debe tener constructores definidos por el usuario.
- No debe ser una clase derivada.
- No puede tener datos miembro privados ni protegidos.
- No puede tener funciones virtuales.

Por lo tanto, interesa escribir nosotros mismos como método de la clase un constructor. Se llamará automáticamente al crear un objeto de la clase. Se puede utilizar de las formas siguientes:

- Declarando un objeto global.
- Declarando un objeto local.
- Invocando al operador *new*.
- Llamando explícitamente al constructor.

Ejercicio. Dada la clase CFecha declarada a continuación:

```
class CFecha
{
      private:
            int dia, mes, anyo;
      protected:
            int Bisiesto () const;
      public:
            CFecha (int dd, int mm, int aa);
            void AsignarFecha ();
            void ObtenerFecha (int *, int *, int *) const;
```

```
                int FechaCorrecta () const;
};
```

Escribir el constructor CFecha para inicializar los atributos de la clase (*enviar consultas o sugerencias a*: academia@cartagena99.com.es).

La llamada al constructor CFecha puede realizarse de dos formas que equivalen a lo mismo:

```
CFecha hoy (10, 2, 1997);
CFecha hoy = CFecha (10, 2, 1997);
```

Sería un error ahora crear un objeto de la siguiente forma:

```
CFecha hoy;
```

Puesto que el constructor ahora requiere unos argumentos. Si queremos seguir manteniendo un constructor por omisión, tendremos que escribir en el cuerpo de la clase un constructor por omisión también:

```
class CFecha
{
      ....
      public:
            CFecha();
            CFecha (int dd, int mm, int aa);
};
```

Constructor copia

Otra forma de inicializar un objeto es asignándole otro objeto de la misma clase en el momento de su creación.

```
CFecha hoy (10, 2, 1997);
CFecha otrodia = hoy;
```

El constructor copia tiene un solo argumento, que es el objeto a partir del cual se va a crear el nuevo objeto. Si no se especifica un constructor copia, se utiliza uno por omisión, copiándose miembro a miembro de un objeto a otro como ocurre con las estructuras de datos. Tenemos, por tanto, el mismo problema que en estructuras si alguno de los miembros es un puntero (se copiaría la dirección pero no los datos, así que tendríamos 2 objetos con un miembro que referencia al mismo espacio de memoria).

La sintaxis del constructor copia es la siguiente:

Declaración: `CFecha (const CFecha &);`
Definición: `CFecha::Cfecha (const CFecha & ObFecha);`

Asignación de objetos

Al igual que ocurre con el constructor y el constructor copia, disponemos de un operador de asignación por omisión que copiará miembro a miembro un objeto en otro. Sin embargo, podemos definirlo nosotros para cambiar la operación del mismo o por la existencia de punteros como miembros de una clase.

Declaración: `CFecha & operator=(const CFecha &);`
Definición: `CFecha & CFecha::operator=(const CFecha &ObFecha);`

En el cuerpo de la definición del operador igual (=), tendremos los siguiente:

```
CFecha & CFecha::operator=(const CFecha &ObFecha)
{
      dia = ObFecha.dia;
      mes = ObFecha.mes;
      annyo = ObFecha annyo;
      return * this;
}
```

Como podemos ver, en la asignación de un objeto a otro, los dos objetos existen ya, a diferencia del constructor copia. Por lo tanto, el operador de asignación tiene que devolver una referencia al objeto sobre el que se están haciendo las asignaciones. La referencia la podemos obtener con el puntero *this*.

NOTA: El hecho de devolver una referencia al objeto asignado, permite realizar asignaciones múltiples, como por ejemplo: a = b = c;

5.3.6 Destructores

Al igual que existe un constructor por omisión que reserva la memoria necesaria para variables de tipos tanto predefinidos como definidos por el usuario, el compilador libera la memoria reservada de esas variables declaradas al acabar el programa o la función donde estén definidas. Es decir, un objeto es destruido al salir del ámbito donde ha sido definido.

Sin embargo, si tenemos estructuras de datos que contienen punteros o punteros para los que hemos hecho una reserva de memoria dinámicamente, tendremos que liberar manualmente dicha memoria.

El destructor tiene la siguiente sintaxis:

Declaración: `~CFecha ();`
Definición: `CFecha::~CFecha();`

La llamada al destructor se realiza al salir del ámbito donde un objeto ha sido definido o llamando al operador *delete*.

Un destructor no puede ser declarado ni const, ni static, pero sí que puede ser declarado virtual, de tal forma que utilizando destructores virtuales podemos destruir objetos sin conocer su tipo.

Ejercicio. Dada la clase CHora:

```
class CHora
{
      private:
            int Horas;
            int Minutos;
            int Segundos;
            char * formato;

      int Formato () const;
      //Devuelve 1 si está en formato 12 horas
      //Devuelve 0 si está en formato 24 horas

      public:
            void SetHora ();
            // Esta función toma cuatro elementos de la
            // entrada estándar y los almacena en las
variables adecuadas.

            void GetHora (int & nHoras,
                        int & nMinutos,
                        int & nSegundos,
                        char * formato);
};
```

Escribir la función *main()*, que establecerá la hora y la visualizará. Para visualizar la hora se utilizará una función con el siguiente prototipo:

```
      void EscribirHora (const CHora & Hora);
```

Añadir las funciones miembro a la clase CHora necesarias para la reserva y liberación de memoria para el miembro *formato* (*enviar consultas o sugerencias a*: academia@cartagena99.com.es).

5.3.7 Funciones amigas de una clase

A los datos miembro privados de una clase sólo podemos acceder mediante las funciones miembro de la clase. Si queremos acceder desde una función no miembro de la clase, tendremos que declarar esta función miembro como amiga de la clase. Esto se hace de la siguiente forma:

```
class CFecha
{
      friend VisualizaFecha (const CFecha & fecha);
      private:
            int dia, mes, anyo;
      public:
            void IntroducirFecha (int dd, int mm, int aa);
```

```
                void ObtenerFecha (int *dd, int *mm, int *aa);
        }
```

Se puede declarar en cualquier punto del cuerpo de la clase y no se ve afectada por las palabras private, public ni protected. Ahora podríamos definir nuestra función *VisualizaFecha* que tiene acceso a los atributos de la clase.

Se puede declarar como función amiga de una clase, una función miembro de otra clase.

5.3.8 Operadores sobrecargados

Un operador sobrecargado es un operador que es capaz de desarrollar su función en varios contextos diferentes sin necesidad de otras operaciones adicionales.

La sintaxis es la siguiente:

```
tipo operator operador([argumentos]);
```

La palabra *operator* más el operador forman el nombre de la función, que debe tomar de forma implícita o explícita al menos un argumento, que se corresponde con un objeto de una clase.

Ejemplo.

```
class CComplejo
{
      private:
            float real;
            float imag;
      public:
            CComplejo (float r, float i);
            ~CComplejo ();
            CComplejo operator+(CComplejo x);
}

CComplejo a, b, c;
c = a + b;  //equivale a: c = a.operator+(b);
```

En este caso, al ser *operator+* una función miembro, debe invocarse a través de un objeto de su clase, que en este caso es el objeto que figura a la izquierda de la operación.

Tenemos 2 tipos de operadores: unarios y binarios. Los operadores unarios toman 1 sólo argumento de forma implícita; los binarios, toman uno de forma implícita y otro de forma explícita:

```
C operator-();
C operator-(C);
```

Si la función sobrecargada no fuera una función miembro, tendría todos los argumentos explícitos.

Por último, un operador sobrecargado no puede tener argumentos por omisión.

Ejercicio 1. Dada la clase CComplejo:

```
class CComplejo
      {
            private:
                  float real;
                  float imag;
            public:
                  CComplejo (float r, float i);
                  ~CComplejo ();
                  //Resto de funciones miembro
      }
```

Sobrecargar el operador de suma y resta para poder realizar estas operaciones sobre objetos de la clase CComplejo. Además, sobrecargarlo de forma similar al operador de asignación para poder realizar sumas y restas encadenadas: d = a + b + c;
(*enviar consultas o sugerencias a*: academia@cartagena99.com.es)

NOTA: devolver una referencia permite que no se llame al constructor copia para copiar el objeto pasado, y una llamada al destructor para eliminar el objeto local a la función cuanod ésta finalice. Declarar el objeto constante que se le pasa como argumento al operador, impide que éste se modifique, al pasarse usualmente por referencia. Habrá que devolver el valor del puntero *this, que es el objeto para el cual fue invocada la función, pero con los nuevos valores.

Ejercicio 2. Dada la clase CVector:

```
class CVector
{
      private:
            int *pVector;
            int nElementos;
      protected:
            void ErrorMem();
      public:
            CVector();
            CVector (int ne);
            CVector (int [], int );
            CVector (const CVector &);
            ~CVector () {delete [] pVector; }
            CVector operator=(const CVector &);
            int & ElementoI (int i) const {return pVector[i]; }
            int ObtenerNumElementos () const {return
                                                nElementos;}
};
```

Sobrecargar el operador de asignación. Nótese que habrá que liberar la memoria del puntero pVector antes de realizar la copia para poder volver a reservar memoria. Sobrecargar el operador de indexación [], y sustituir a la función *ElementoI*.
(*enviar consultas o sugerencias a*: academia@cartagena99.com.es)

5.3.9 Herencia. Clases derivadas

Se utiliza la derivación de clases cuando necesitamos ampliar una clase existente. Gracias a la herencia, no tenemos que reescribir el código que aporta la clase base. Una clase derivada puede serlo de múltiples clases base (derivación múltiple). Una clase derivada, puede ser una clase base a su vez de otras clases derivadas.

La clase derivada hereda todas las propiedades de su clase base: datos y funciones miembro. La derivación puede ser: privada, protegida o pública. Por omisión, es privada.

- En la derivación privada, los miembros public y protected de la clase base son private en la clase derivada. Los miembros private son inaccesibles por la clase derivada.
- En la derivación protegida, los miembros public y protected de la clase base son protegidos en la clase derivada, y los miembros private son inaccesibles por la clase derivada.
- En la derivación pública, los miembros public de la clase base son públicos en la clase derivada, los miembros protected son protegidos y los privados, inaccesibles desde la clase derivada.

Ejemplo.

```
class CPersona
{
      private:
            char nombre[20];
            int edad;
      public:
            SetEdad (int);
            int GetEdad ();
            void SetNombre (char *);
            void GetNombre (char *);
            void Visualizar ();
};

class CAlumno : public CPersona
{
      private:
            int notas[20];
            char centro[20];
      public:
            void SetNota(int, int);
            int GetNota (int);
            void SetCentro (char *);
            void GetCentro (char *);
            void Visualizar ();
};
```

Podemos redefinir en la clase derivada una función que aparezca en la clase base. Además, podemos aprovechar el código que tuviéramos si lo que queremos es añadir funcionalidad:

```
void CPersona::Visualizar ()
{
      cout << nombre << "\t" << edad << endl;
}

void CAlumno::Visualizar ()
{
      CPersona::Visualizar();
      cout << centro;
      for (int i = 0; i <20; i++)
            cout << "Nota " << i << " = " << notas[i] << endl;
}
```

Constructor

Los constructores de la clase base no son heredados por la clase derivada. Así, el constructor de la clase derivada tiene que invocar, implícita o explícitamente, al constructor de la clase base, puesto que la clase derivada contiene todos los miembros de la clase base, que deben ser inicializados. La sintaxis es la siguiente:

```
CDerivada::CDerivada (int a, char *b, char *c, float d) :
      CBase1 (a,b)
{
      //Cuerpo del constructor de la clase derivada
}
```

No es necesario indicar la lista de inicializadores en la clase base si ésta no tiene definida un constructor.

Destructor

El destructor de la clase base no se hereda. No requieren una sintaxis especial porque no reciben argumentos. El orden de ejecución de los destructores es el inverso al de los constructores: primero el destructor de la clase derivada y luego el de la clase base.

Funciones amigas

Una función *friend* de una clase base puede acceder sólo a los miembros públicos, protegidos y privados de la clase derivada que fueron heredados de la clase base, además de a los miembros públicos de la clase derivada. Es decir, la amistad no se hereda.

Si la función *friend* es de la clase derivada, puede acceder a todos los miembros públicos y protegidos de la clase base, que han sido heredados, y por supuesto a todos los miembros de la clase derivada.

5.3.10 Punteros a objetos de una clase derivada

Se declaran y se manipulan de la misma forma que los punteros a objetos de una clase cualquiera. Los punteros y referencias a clases derivadas pueden ser convertidos implícitamente a punteros y referencias a sus clases base. De tal forma que si tenemos una clase base *CFicha*, una clase derivada *CFichaLibro* y otra derivada a su vez de ésta *CFichaVolumen*, se realiza la conversión implícita al ejecutar el siguiente código:

```
CFicha * p;
CFichaLibro libro1;
CFichaVolumen libro2;
p = &libro1;      // convierte de CFichaLibro * a CFicha *
p = &libro2;      // convierte de CFichaVolumen * a CFicha *
```

Cuando accedemos a un objeto por medio de un puntero, el tipo del puntero determina qué función miembro puede ser llamada. Si accedemos a un objeto de una clase derivada a través de un puntero de la clase base, el estado de ese objeto sólo puede ser manipulado por funciones miembro de su clase base.

Además, se puede hacer la conversión contraria, de un puntero a la clase base a un puntero a su clase derivada, pero hay que hacerlo de forma explícita.

Este hecho es la base del polimorfismo.

5.4 Funciones virtuales

Cuando se invoca a una función miembro que está definida en la clase base y redefinida en las clases derivadas, la versión que se ejecuta depende del tipo del objeto o del puntero que se utilice para invocar a la misma. En el siguiente ejemplo se puede ver un uso muy útil. Imaginemos que disponemos de una clase base *CFicha*, una clase derivada *CFichaLibro* y una clase derivada de ésta *CFichaVolumen*.

```
void main()
{
      const int N = 4;
      CFicha *p[N];

      CFichaLibro libro1;
      p[0] = &libro1;
      CFichaVolumen libro2;
      p[1] = &libro2;
      CFichaVolumen libro3;
      p[2] = &libro3;
      CFichaLibro libro4;
      p[4] = &libro4;

      for (int i = 0; i<N; i++)
      {
```

```
                p[i]->VisualizarFicha();
                cout << endl;
        }
}
```

Si suponemos que hemos definido la función VisualizarFicha () en todas las clases, el resultado es que se invocará siempre a la función miembro de la clase base *CFicha*, ya que el tipo del puntero es de la clase base. Por ello, necesitamos que la función invocada pertenezca a la misma clase que el objeto referenciado. La solución está en que el mismo sistema sea el que se encargue de la identificación en tiempo de ejecución de la clase de objetos apuntados. Para ello, en C++ se proporciona el mecanismo de *función virtual*.

Una función virtual es una función miembro de una clase base que puede ser redefinida en cada una de las clases derivadas de ésta, y una vez redefinida puede ser accedida mediante un puntero o referencia a la clase base, resolviéndose la llamada en función del tipo del objeto apuntado.

Se declara así:

```
class CFicha
{
        ...
        ...
        virtual void VisualizarFicha ();
        ...
};
```

Las redefiniciones en las clases derivadas no necesitan llevar la palabra *virtual* ya que se declaran como funciones virtuales implícitamente.

5.5 Polimorfismo

La utilización de clases derivadas y funciones virtuales es frecuentemente denominada *programación orientada a objetos*. Además, la facultad de llamar a una variedad de funciones utilizando exactamente el mismo medio de acceso, proporcionada por funciones virtuales, es a veces denominada *polimorfismo*.

Polimorfismo es, por tanto, la facultad de llamar a una variedad de funciones, una declarada virtual en una clase base y múltiples formas de ésta declaradas en las clases derivadas directas o indirectas, utilizando exactamente el mismo medio de acceso: un puntero a la clase base.

Ejercicio. Escribir tres clases con la siguiente jerarquía y atributos:

- Clase Ficha CFicha: clase base de todas las demás. Sus atributos son: título y referencia.
- Clase Libro CLibro: clase derivada de CFicha (derivación pública), que añadirá como atributos: autor y edición.

- Clase Revista CRevista: clase derivada de CFicha (pública), que añadirá como atributo: número de edición.

Las tres clases definen una función Visualizar() que será virtual. Implementar las funciones miembro mínimas que se consideren oportunas. Hacer uso del polimorfismo proporcionado en C++ para visualizar los datos de objetos de cada clase de forma correcta.
(*enviar consultas o sugerencias a*: academia@cartagena99.com.es)