PRIMERA CLASE

CONCEPTOS DE LA PROGRAMACIÓN ORIENTADA A OBJETOS

- ***La programación orientada a objetos es un estilo de programación que utiliza objetos como bloque esencial en su construcción. Las ventajas son que mejora la fiabilidad del proceso, mejora el desarrollo del programa y facilita el mantenimiento de los programas.***
- ***Un objeto es una unidad que contiene datos y las funciones que manejan esos datos. A los datos que forman parte del objeto se les conoce como datos miembro y a las funciones se las conoce como funciones miembro o métodos. Los datos quedan ocultos al programador y únicamente dispondrá de las funciones para acceder a ellos.***
- ***Una clase es un tipo de dato definido por el usuario que determina las estructuras de datos que lo forman y las funciones asociadas con él, es decir, es un modelo con el que se construyen los objetos y un objeto es la instancia de una clase.***
- ***La herencia es la propiedad por la cual unos objetos se pueden construir a partir de otros. Si una clase sólo recibe propiedades de otra clase lo llamamos herencia simple; si una clase recibe propiedades de más de una clase entonces hablamos de herencia múltiple.***
- ***El polimorfismo es la característica por la cual una operación puede ser interpretada de formas diferentes por diferentes objetos.***

TIPOS DE DATOS

Básicos:

- ***char! va a representar un carácter y un dato de este tipo; ocupará 1 byte. Este carácter se puede representar con su código ASCII o con el carácter entre comillas simples. Formando parte de estos caracteres hay unos especiales:***

` \a ` *! genera un pitido*

` \b `*! carácter de retroceso del cursor*

` \f `*! carácter de avance de página (salto de página)*

` \n `*! carácter de nueva línea*

` \r `*! retorno de carro*

` \t `*! tabulación horizontal*

` \v `*! tabulación vertical*

` \\ ` *! carácter* \

` \´ `*! carácter apóstrofe (´)*

` \ `*! carácter*

` \nnn `*! carácter representado en octal donde nnn son tres cifras octales*

` \xnn `! carácter representado en hexadecimal

- ***int! estos datos representan un número entero comprendido entre −32768 y 32767. Este dato ocupa 2 bytes. Se les puede aplicar los modificadores short y long. Aplicando el primero obtendré un entero comprendido entre −128 y 127 ocupando sólo 1 byte (0 signo +, 1 signo −). Aplicando el segundo obtendremos un entero comprendido entre más menos 10 −37 y más menos 10 38 ocupando 4 bytes.***
- ***Float ! representa un número en coma flotante que ocupa 4 bytes. Este tipo de dato permite hasta 6 cifras decimales. A este dato no le podemos aplicar ni el long ni el short.***
- ***Double ! representa un número en coma flotante de 8 bytes de tamaño. Le podemos aplicar el modificador long y obtendremos un dato de 16 bytes. El double permite hasta 12 cifras decimales.***
- ***Void ! representa un dato sin valor. Se utiliza en funciones que no devuelven valores o en punteros a objetos de tipo desconocido.***

Variables: representa un valor almacenado en la memoria. Este valor puede variar a lo largo del programa. Algunos conceptos de las variables:

- ***Nombre de la variable: también llamado identificador y permite diferenciar una variable de otra. Tiene una serie de reglas: el nombre es una secuencia arbitraria de letras y dígitos; el primer carácter debe ser una letra; el carácter de subrayado se trata como una letra; los nombres que comienzan con dos subrayados están reservados para el uso interno del compilador; C++ es sensible al tamaño de las letras, es decir, diferencia entere mayúsculas y minúsculas; no se pueden usar palabras reservadas como nombres de variables.***
- ***Declaración de una variable: se utiliza para informar al compilador de que se va a usar una variable con un determinado nombre pero no se le asigna nombre. El formato de una declaración es:***

Tipo identificador;

Tipo identificador1, identificador2...;

- ***Definición de una variable: actúa como una declaración y además asigna memoria a la variable. Simplemente hay que dar un valor inicial a la variable.***

Los tipos de variables son:

- ***Locales: se definen dentro de una función y conservan su valor mientras se ejecuta ésta; además, sólo pueden usarse en esa función; sólo pueden ser referenciadas con instrucciones contenidas en esa función; es aconsejable utilizar este tipo de variables puesto que sólo ocupan memoria mientras se ejecuta la función a la que pertenece.***
- ***Globales: se definen fuera de cualquier función, incluso fuera de la función main (); son reconocidas desde cualquier parte del programa y las puede utilizar cualquier función; ocupan memoria durante toda la ejecución del programa.***
- ***Parámetros formales: se utiliza cuando una función emplea argumentos de otra; deben ser del mismo tipo que los argumentos que se utilizan en la llamada; para la función en que son utilizadas funcionarán como variables locales. Existen dos formas de definirlos, la primera es hacerlo dentro de la función:***

Tipo nombre función (tipo identificador1, tipo identificador2...)

{

(líneas de código)

}

La otra forma es después del nombre y antes del código:

Tipo nombre función (identificador1, identificador2...)

Tipo identificador1;

Tipo identificador2;

(líneas de código)

{

(líneas de código)

}

Modificadores:

- ***Signed ! utiliza el primer bit del dato como signo y no como dígito. Por defecto, todos los datos son de tipo signed menos el dato tipo char (signed char = short int).***
- ***Unsigned ! el bit más significativo de un dato será tratado como dígito y no como signo. Los datos tipo char son por defecto unsigned consiguiendo aumentar el tamaño del número que vamos a contener.***
- ***Static ! una variable estática conserva su valor durante todo el programa. Habrá dos tipos de variables con static:***

- ***Variables estáticas locales: guardan el valor durante la ejecución del programa pero sólo pueden utilizarse en la función donde han sido definidas.***
- ***Variables estáticas globales: guardan el valor por ser global pero sólo pueden utilizarse en el archivo en que fue definida.***

Constantes: son datos cuyo valor no se pueden modificar a lo largo del programa. Para indicar que un dato es una constante se coloca el modificador const delante de la definición del dato:

Const tipo identificador = valor;

Const float pi = 3.14159

Los tipos de constantes son:

- ***Enteras: 44, 0, −31, etc.***
- ***Enteras largas: se impone una L (mayúscula o minúscula) al final de la constante.***
- ***Octales o hexadecimales: en las octales su formato es 0nn y en las hexadecimales es 0xnn.***
- ***Reales: son las de coma flotante. Hay que poner el sufijo F (mayúscula o minúscula).***
- ***Carácter: `a', `z', etc.***
- ***De cadena: secuencia de caracteres entre comillas dobles.***

OPERADORES

Símbolos que tienen una determinada función. Pueden ser:

<u>Aritméticos:</u> +, −, /, *, % (resto), ++ (incremento), −− (decremento), − (signo menos). Si utilizamos la división con datos de tipo int trunca los decimales, es decir, no aparecen y la posición de los operadores incremento y decremento varía su función. Si se colocan delante de la variable, primer realizan el incremento o decremento y después asignan el valor; si se sitúan detrás, primero asignan el valor y después efectúan el incremento o decremento.

Int a = 3;

Int c;

Int d;

C = ++a ! c = 4 y a = 4

D = a++ ! d = 3 y a = 4

<u>Relacionales:</u> se utilizan para comparar expresiones. Podemos obtener como resultado 0 = falso o un valor distinto de cero que significará verdadero. Los relaciones son: <, >, <=, >=, == (igual que), != (distinto de).

<u>Lógicos:</u> nos permiten conectar varias comparaciones entre sí y suelen ir asociadas con los relacionales. Son: ! (Not, no lógico), && (And, y lógico), || (Or, ó lógico)

PUNTEROS Y REFERENCIAS

Un puntero es la dirección de memoria que ocupa una variable. Una variable puntero es una variable que contiene la dirección de otra variable. Para crear una variable puntero se ha de poner su nombre precedido de un asterisco:

Tipo *identificador;

Variable puntero = &variable (indicar donde está la variable a la que voy a apuntar)

El tipo de la variable puntero tiene que ser el mismo que el tipo de la variable a la que apuntará.

C = *a (c = a lo que apunta una variable a)

****a = 56 (el contenido de a igual a otro valor, lo que está apuntando la variable puntero tome un valor nuevo)***

Una referencia es el alias de una variable, es un nuevo nombre para una variable. Hay que preceder a la referencia con el & (operador de direccionamiento) y siempre será inicializada (habrá que decir de quién es alias).

Tipo &referencia = identificador;

Int val = 50;

Int &refval = val;

CONVERSIONES DE TIPOS Y MOLDES

Es posible convertir un tipo de dato a otro. Esto se puede hacer de forma implícita o explícita. De forma implícita lo realizará automáticamente el compilador. Se podrá convertir un tipo de dato a otro si tiene un tamaño de almacenamiento igual o superior.

De forma explícita es el programador quien fuerza la conversión mediante moldes existiendo dos formas de utilizar un molde. La primera es con el tipo entre paréntesis:

Double a = 3.18;

Int b = (int) a;

La segunda manera colocando entre paréntesis el dato a convertir (notación funcional):

Double a = 3.18;

Int b = int (a);

Los moldes nos permiten también forzar el resultado de una operación:

(Sin molde)

int a = 7, b = 2;

float c;

c = a/b;

(Con molde)

int a = 7, b = 2;

float c;

c = ((float) a/b)

ESTRUCTURA DE UN PROGRAMA

***Comentarios:** existen dos tipos de comentarios en C++. Se puede poner una sola línea cuando se precede de dos barras // o encerrando todo un bloque de código (más de una línea) entre /*...*/.*

***Función main ():** un programa de C++ está divido en funciones; además, una función es un conjunto de instrucciones agrupadas. Para que un programa pueda funcionar debe tener como mínimo una función llamada main () (ejecución principal del programa). El formato de esta función es:*

Tipo main (argumentos)

{

(líneas de código)

}

pondremos el tipo de dato que va a devolver, el nombre main y entre paréntesis los argumentos que hay que pasarle. Normalmente se usa:

void main ()

{

(líneas de código)

}

SEGUNDA CLASE

<u>Archivos de cabecera:</u> el compilador dispone de una biblioteca estándar de funciones. Estas funciones están en lo que denominamos archivos de cabecera. Para incluir un archivo de cabecera, utilizaremos el siguiente formato:

#include <nombre archivo cabecera.h>

Este formato se usará si el archivo se encuentra en el directorio por defecto. Si no, se utiliza este otro formato:

#include nombre archivo cabecera.h

Un archivo de cabecera está compuesto por las definiciones de una serie de funciones y constantes. Algunos archivos de cabecera son:

Clrscr ()! sirve para borrar la pantalla y sitúa el cursor en las coordenadas (1,1). Esta función utiliza el archivo de cabecera #include <conio.h>.

Gotoxy ()! sirve para situar el cursor en una posición determinada de la pantalla. Utiliza el mismo archivo de cabecera. Hay 25 filas y 80 columnas.

ENTRADA Y SALIDA DE DATOS

<u>Salida:</u> para realizar la salida en C++ se puede utilizar cualquiera de las funciones ya presentes en C. El flujo de salida se representa mediante el identificador cout (es un objeto). Se combina con el uso del operador de inserción <<. El operador de inserción dirige el valor de la variable que hay a su derecha con el objeto que hay a su izquierda. El archivo de cabecera que utiliza el cout es <iostream.h>.

<u>EJEMPLO:</u>

#include <iostream.h>

void main (){

cout << Este es un programa de prueba \n;

}

Con el cout también se pueden visualizar múltiples datos:

EJEMPLO:

```
#include <iostream.h>

void main () {

int precio = 1234;

cout << El valor del precio es:;

cout << precio;

cout << `.';

}
```

También se puede combinar con múltiples operadores de inserción. Únicamente hay que poner un operador de inserción por cada dato de tipo diferente.

EJEMPLO:

```
#include <iostream.h>

void main () {

int precio = 1234;

cout << El valor del precio es: << precio << `.';

}
```

Entrada: ***en C++ se puede utilizar cualquiera de las funciones de entrada que disponíamos en C. Aparte de esto tenemos un flujo de entrada que se hace con el objeto cin. Este flujo lo utilizaremos con el operador de extracción >>. Este operador toma el valor del objeto situado a su izquierda y lo sitúa en la variable de su derecha. También utiliza el archivo <iostream.h>.***

EJEMPLO:

```
#include <iostream.h>

void main () {

int num;

cout << Introduzca un número:;

cin >> num;

cout << El número introducido es: << num << ` \n';

}
```

SENTENCIAS DE CONTROL

<u>Condicionales</u>:

<u>If</u> ! nos sirve para evaluar si una condición simple o compuesta es verdadera (distinta de cero) o falsa (igual a cero). Para ello utilizaremos dos tipos de formatos. El primero es:

If (expresión) sentencia 1;

[else sentencia 2];

Y el segundo es:

if (expresión)

{

sentencias;

}

[else

{

sentencias;

}]

Los dos formatos se pueden combinar. Si la expresión es falsa y no he puesto else, el programa seguirá su secuencia normal.

<u>EJEMPLO:</u>

Void main () {

Clrscr ();

Cout << Introduzca un número:;

Int x;

Con >> x;

If (x= = 0) cout << el número es cero;

Else cout << No es el cero;

}

También se puede utilizar como expresión, una compuesta:

Void main () {

Clrscr ();

Int a ,b, c;

Cin >> a >> b >> c;

If (a>b && a>c) cout << El mayor es << a;

Else cout << El mayor no es << a;

}

Los if también pueden ir anidados en cuyo caso cada else irá asociado al último if evaluado.

If (expresión)

{

if (expresión) sentencia 1;

else sentencia 2;

}

if (expresión)

{

if (expresión) sentencia 1;

}

else sentencia 2;

<u>Switch</u>! esta sentencia de control se utiliza para crear decisiones múltiples. Principalmente pretende sustituir anidaciones de if. El formato es el siguiente:

Switch (expresión) {

Case constante 1:

Sentencias;

Break;

Case constante 2:

Sentencias;

Break;

[default :

sentencias];

}

Si constante 1 coincide con la expresión, se ejecutarán las sentencias que hay hasta el break. Hay unas reglas básicas en el uso del switch:

Dos constantes no pueden tener el mismo valor dentro de un mismo switch

El switch sólo compara igualdades

Si en un case no escribimos la sentencia break también se ejecutarán las sentencias del siguiente case.

<u>EJEMPLO:</u>

Void main () {

Clrscr ();

Cout << Introduzca el número de mes:;

Int x;

Cin >> x;

Switch (x); {

Case 1:

Case 2:

Case 3:

Cout << Invierno;

Break;

Case 4:

Case 5:

Case 6:

Cout << Primavera;

Break;

Case 7:

Case 8:

Case 9:

Cout << Verano;

Break;

Case 10:

Case 11:

Case 12:

Cout << Otoño;

Break;

Default:

Cout << Error al introducir el número de mes;

}

}

Los switch también se pueden anidar en cuyo caso se coloca el nuevo switch como tratamiento de una orden case.

Switch (expresión) {

Case 1:

Switch (expresión 2) {

Case 1:

Sentencias;

Break;

Case 2:

Sentencias;

Break;

Break;

Case 2:

Sentencias;

Break;

}

<u>Bucles:</u> un bucle es una estructura de control que permite repetir varias veces el mismo proceso. Hay tres tipos de sentencias:

<u>For</u> ! nos permite ejecutar un bucle mientras la condición sea verdadera. Hay dos formatos. El primero es:

For (inicialización; condición; incremento) sentencia;

Y el segundo es:

For (inicialización; condición; incremento)

{

sentencias;

}

Lo primero que hace es inicializar la/s variable/s; a continuación se evalúa la condición planteada, si es verdadera se ejecuta la/s sentencia/s que forman el bucle y finalmente se realiza el incremento. Este proceso se repetirá hasta que deje de cumplirse. Si la condición es falsa la primera vez el bucle no se ejecutará. Existe la posibilidad de inicializar más de una variable, de utilizar condiciones compuestas, de realizar más de una operación en el incremento e incluso existe la posibilidad de omitir algunos de los parámetros. Si se realiza más de una operación en alguno de estos parámetros se separan por comas y si se omite un parámetro se deja vacío.

<u>EJEMPLOS:</u>

Void main () {

Clrscr ();

For (int x = 1; x < 101; x++)

{

cout << x;

cout << `\n';

}

{

<u>INTRODUCIR NÚMEROS Y HALLAR LA SUMA DE LOS COMPRENDIDOS ENTRE 75 Y 100. SI NO, FINALIZA</u>.

Void main () {

Clrscr ();

Int s, x;

For (x = 0, s = 0, x < 75 || x > 100)

{

s+ = x;

cout << Introduce un número:;

cin >> x;

}

cout << La suma es: << s;

}

Existe otra variante del bucle for y es el bucle infinito. Es un bucle for donde ha sido omitida la condición (habitualmente también se omiten los otros parámetros). Hay dos formatos. El primero es:

For (;;) sentencia;

Y el segundo es:

For (;;)

{

sentencias;

}

Normalmente, el bucle for infinito poseerá una sentencia if con unas condiciones o sentencias y un break para salir del mismo.

<u>While</u> ! esta sentencia permite la repetición varias veces de un mismo proceso mientras la condición sea verdadera. No necesita ni inicialización ni incremento. Tiene dos formatos. El primero es:

While (condición) sentencia;

Y el segundo es:

While (condición)

{

sentencias;

}

EJEMPLOS:

INTRODUCIR UN NÚMERO POR TECLADO Y HALLAR SU CUBO. EL PROGRAMA FINALIZA CUANDO SE INTRODUCE −1

Void main () {

Clrscr ();

Int x = 0, r;

While (x! = −1) {

Cout << Introduce un número:;

Cin >> x;

R = x * x * x;

Cout << Su cubo es: << r;

}

}

El while también tiene un formato de bucle infinito. La condición se tendrá que cumplir siempre. El primer formato es:

While (1) sentencias;

Y el segundo es:

While (1)

{

sentencias;

}

Para salir del bucle se pone también la sentencia break.

Do ... while ! esta sentencia permite realizar varias veces un proceso mientras la condición se cumpla. La diferencia con las sentencias anteriores es que en este caso la condición se evalúa al final. De esta forma, aunque desde el principio de la condición ea falsa, el bucle se ejecuta al menos una vez. Hay dos formatos. El primero es:

Do sentencia while (condición);

Y el segundo es:

Do {

Sentencias;

} while (condición);

<u>EJEMPLOS:</u>

Void main () {

Clrscr ();

Int a;

Do {

Cout << Introduce un número:;

Cin >> a;

If (a > 0) cout << El número es mayor que cero;

If (a < 0) cout << El número es menor que cero;

} while (a! = 0);

}

SENTENCIA BREAK

Tiene dos funciones:

- ***En una orden case da por terminada la instrucción o la sentencia switch saltándose todas las instrucciones que aparezcan a continuación dentro de ese switch .***
- ***En un bucle provoca la salida del mismo (en bucles anidados provoca la salida al bucle inmediatamente exterior).***

SENTENCIA CONTINUE

Fuerza una nueva ejecución del bucle saltándose todas las instrucciones que faltasen para la finalización del mismo y yendo a evaluar directamente la condición. En un bucle for incrementará y evaluará la condición.

SENTENCIA EXIT

Se utiliza para finalizar un programa y devolver el control al sistema operativo. Se suele utilizar en casos especiales. Se utiliza poniendo exit (n) donde n es un número que puede ser cero si el programa finaliza sin errores o el 1 si el programa finaliza debido a que se ha producido un error.

USO DEL COMPILADOR

Las etapas desde la creación hasta la ejecución del programa son:

- ***Codificación del programa.***
- ***Compilación del programa fuente.***
- ***Depuración de los posibles errores aparecidos.***
- ***Linkar o enlazar los componentes del programa objeto.***
- ***Ejecución del programa.***

Uso de teclas rápidas en el compilador:

- ***F1: ayuda.***
- ***Ctrl + F1: ayuda de la instrucción que está sobre el cursor.***
- ***F2: graba el archivo de la ventana activa.***
- ***F3: abre un archivo.***
- ***Alt + F3: cierra la ventana activa.***
- ***F4: ejecuta el archivo hasta el cursor.***
- ***F5: zoom.***
- ***F6: cambia de ventana.***
- ***F7: ejecuta paso a paso entrando en funciones.***
- ***F8: ejecuta paso a paso sin entrar en funciones.***
- ***Alt + F9: compila el programa.***
- ***F9: linka el programa.***
- ***Ctrl + F9: ejecuta el programa.***

EJERCICIOS DE LA SEGUNDA CLASE:

- ***Codificar un programa que calcule y visualice los 100 primeros números pares y su media aritmética.***
- ***Codificar un programa que mediante un menú de opciones permita sumar, restar, multiplicar o dividir dos números.***
- ***Codificar un programa que introduciendo 10 números visualice el mayor de todos ellos.***

```
// Programa que, mediante un menú de opciones permita sumar, restar, multiplicar y

// dividir

#include <iostream.h>

#include <conio.h>

void main ()

int a, b;

char opcion;

do

{

clrscr ();

gotoxy (20,5);
```

```
cout << CALCULADORA;

gotoxy (25,8);

cout << 1. Sumar;

gotoxy (25,10);

cout << 2. Restar;

gotoxy (25,14);

cout << 3. Multiplicar;

gotoxy (25, 14);

cout << 4.Dividir;

gotoxy (25,16);

cout << 5. Salir;

gotoxy (20,20);

cout << Elige una opción ( 1 − 5):;

cin >> opcion;

clrscr ();

if (opcion!='5')

{

gotoxy (25,8);

cout << Introduce primer número:;

cin >> a;

gotoxy (25,10);

cout << Introduce segundo número:;

cin >> b;

}

gotoxy (50,9);

switch (opcion);
```

```
{
case `1' : cout << La suma es : << a+ b;
break;
case `2': cout << La resta es: << a − b;
break;
case `3': cout << El producto es : << a * b;
break;
case `4': if (b = = 0) cout << Imposible dividir por cero;
else cout << El cociente es : << (float) a/b;
break;
}
getch ();
} while (opcion ¡= `5');
// Programa que, introduciendo 10 números por teclado, visualice el mayor de todos
// ellos
#include <iostream.h>
#include <conio.h>
void main () {
int num, mayor = 0;
clrscr();
for (int i = 0; i < 10; i++) {
cout << Introduce el  << i + 1 << número: ;
cin >> num;
if (num > mayor) mayor = num;
}
cout << \nEL mayor de todos es el  << mayor;
```

```
getch ();

}

// Programa que calcula y visualiza los 100 primeros números pares y su media

// aritmética

#include <iostream.h>

#include<conio.h>

void main () {

int n = 0, suma = 0;

float media = 0;

for (int c = 0; c > 100; c++) {

n+ = 2;

cout << c +1 << º número par −> << n << `\n';

suma+ = n;

getch ();

}

media = (float)suma/100;

cout << La media es −> << media;

}
```

TERCERA CLASE

ARRAYS

Son conjuntos de datos del mismo tipo almacenados en memoria bajo un mismo nombre. Pueden tener una dimensión (vectores), dos dimensiones (matrices) o pueden ser multidimensionales.

Declaración de los arrays

<u>De una dimensión:</u> tipo nombre array [tamaño];

Para averiguar el tamaño del array multiplicamos el número de elementos por el tamaño que ocupan. El acceso se va a realizar de dos métodos: por indexación o por punteros. Con la indexación se va a utilizar un número llamado índice para acceder a cada elemento. El primer elemento de cualquier array es el que está en la posición cero.

EJEMPLO:

```
Void main () {

Int array [10];

For (int = 0; x<10; x++) {

Cout << Introduce el << x+1 º número;

Cin >> array [x];

}

For (x = 0; x<10; x++) {

if (array [x]/2 == (float) array [x]/2)

cout << array [x];

{

{
```

En C++ los arrays son tratados como punteros. El nombre del array es en realidad un puntero a la primera posición (a = * números ! así sólo apuntaría al primero). Si quiero obtener todos los datos del array usaremos la aritmética de punteros. Cuando a un puntero se le suma uno lo que se consigue es que el puntero apunte al siguiente dato de ese mismo tipo (a = * (número + 1))

De dos dimensiones: ***tipo nombre array [tamaño 1] [tamaño 2]***

Para acceder a las posiciones de un array de dos dimensiones hay dos métodos: por indexación o por punteros. Por indexación debo poner el índice de cada dimensión array [x] [y]; Por punteros se pone * (tantas filas que me salto * número de elementos de cada fila + columnas que me salto)); debido a que las matrices se representan en memoria como una sucesión de datos.

Multidimensionales: ***tipo nombre array [tamaño 1] [tamaño 2] ...;***

Se pueden acceder también por indexación o usando los punteros

Arrays de caracteres: ***funcionan exactamento igual que cualquier otro array. Se declara y se accede igual que los demás. La diferencia es que se pueden introducir todos sus elementos en una instrucción y visualizarlos también en una instrucción.***

EJEMPLO:

```
Char nombre [15];

Cin >> nombre;

Cout << nombre;
```

Otra característica es que están limitados por un carácter llamado nulo o carácter de fin de cadena `\0'. Deben tener una posición más del tamaño que vamos a guardar.

Iniciación de arrays:

<u>De una dimensión:</u> tipo nombre array [tamaño] = {lista de argumentos}

Los argumentos son los valores que queremos que tomen inicialmente cada uno de los elementos del array int n[9] = {2,4,6,8,10,12,14,16} Si no se realiza una inicialización no es necesario declarar el tamaño del array int n [] = {2,4,6,8,10,12,14,16}

<u>De más de una dimensión:</u> sólo se puede omitir el tamaño de la primera dimensión

int n [3] = {1,2,5,9,12,3};

int n[] [2] = {1,2,5,9,12,3};

<u>Arrays de caracteres:</u>

Char nombre [7] = {`M', `A', `N', `O', `L', `O', `\0'};

Char nombre [7] = Manolo;

Exploración de un array:

<u>Secuencial:</u> consiste en ir comparando todos los elementos con el elemento buscado hasta encontrarlo.

<u>EJEMPLO:</u>

```
Void main () {

Clrscr ();

Int array [10];

For (int x = 0; x<10; x++) {

Cout << Introduce un número:;

Cin >> array [x];

}

Int num;

Cin >> num;

Int sw = 0;

For (x = 0; x<10; x++) {

If (array [x] = = num) {
```

Cout << El número está en la posición << x;

Sw = 1;

Break;

}

}

if (sw = = 0) cout << Ese número no está;

getch ();

}

<u>Dicotómica:</u> (binaria). Sólo se puede realizar en arrays ordenados. Consiste en buscar por mitades.

<u>EJEMPLO:</u>

Void main () {

Clrscr ();

Int array [10];

For (int x = 0; x<10; x++) {

Cout << Introduce un número:;

Cin >> array [x]; // Deben introducirse ordenados

}

Int num;

Cin >> num;

Int sw = 0, m;

For (int i = 0; f = 9; f>= i;) {

M = (i + f)/2;

If (array [m] = = m) {

Cout << El número está en la posición << m;

Sw = 1;

Break;

{

if (array [m]>num) f = m − 1;

else i = m + 1;

}

if (sw = = 0) cout << El número no está en el array;

getch ();

}

Ordenación de arrays:

<u>Método de la burbuja:</u>

Void main () {

Clrscr ();

Int array [10];

For (int x = 0; x<10; x++) {

Cout << Introduce un número:;

Cin >> array [x];

}

cout << Array desordenado;

For (x = 0; x < 10; x++) cout array [x] << −;

For (x = 0; x < 9; x++) {

if (array [x] > array [x + 1]) {

int aux = array [x + 1];

array [x + 1] = array [x]

array [x] = aux;

x = −1;

}

}

cout << Array ordenado;

For (x = 0; x < 10; x++) cout << array [x] << −;

Getch ();

}

<u>Método de selección directa:</u> compara el primer dato con todos los demás

Void main () {

Clrscr ();

Int array [10];

For (int x = 0; x<10; x++) {

Cout << Introduce un número:;

Cin >> array [x]; // cin >> * (array + x)

}

cout << Array desordenado;

for (x = 0; x < 10; <++) cout array[x] << −;

int y;

for (x = 0; x < 9; x++) {

for (y = x + 1; y < 10; y++) {

if (array [x] > array [y]) {

int aux = array [x];

array [x] = array [y];

array [y] = aux;

}

}

}

cout << Array ordenado;

for (x = 0; x < 10; x ++) cout << array [x] << −;

getch ();

}

LA MEMORIA DINÁMICA

La estructura de la memoria de un programa en C++ va a ser la siguiente:

- ***Zona de código:* donde se almacena el programa y permanece fija durante toda la ejecución.**
- ***Zona de datos estáticos:* en esta zona se guardan los datos que no cambian durante la ejecución del programa.**
- ***Zona de pila (stack):* donde se almacenan la gran parte de las variables. Se dice que es dinámica.**
- ***Zona de datos dinámicos (montón):* se utiliza para reservar memoria de forma dinámica.**

Asignación dinámica consiste en reservar una serie de posiciones de memoria durante la ejecución del programa. Para hacer la asignación dinámica se utilizan los operadores:

***New* ! permite reservar memoria (equivale a la función malloc de C). Se puede utilizar con dos formatos:**

Variable puntero = new tipo;

Variable puntero = new tipo [tamaño];

Reserva memoria para poder almacenar un dato del tipo que se indique o para poder almacenar un array de datos de ese tipo. Devuelve la dirección de memoria donde comienza esa memoria reservada. Si no puede reservar memoria devuelve un cero. La variable será una variable puntero.

EJEMPLOS:

Int *p;

P = new int;

Int *p;

P = new int [s];

***Delete* ! permite liberar la memoria reservada con el operador new. También posee dos formatos:**

Delete variable puntero;

Delete [] variable puntero;

EJEMPLO:

```
Void main() {

Int *array, a;

Clrscr ();

Cout << Introduce la dimensión del array;

Cin >> n;

Array = new int [n];

If (array = = 0) {

Cout << No hay suficiente memoria;

Exit (1);

}

For (int x = 0; x < n; x++) {

cout << Introduce un valor;

cin >> array [x];

}

Delete [] array;

}
```

EJERCICIOS DE LA TERCERA CLASE

- ***Programa que permita introducir cinco palabras y que después indique el número de vocales que tenga cada una de ellas.***
- ***Programa que dado un array de 5 datos tipo int y otro array de 5 punteros a datos tipo int asigne a cada puntero uno de los datos del array de int y realice la ordenación de los datos a través de los punteros.***

```
// Programa que permite introducir 5 palabras y que después indique el número de

// vocales que tiene cada una de ellas

#include <conio.h>

#include <stdio.h>
```

```
void main () {
char vocales [] = aeiouAEIOU;
char palabras [5][15];
int num = 0, fil, col, x;
clrscr ();
for (fil = 0; fil < ; fil++) {
printf (\nIntroduce la %d palabra:, fil + 1);
scanf (%s, palabras [fil]);
}
for (fil = 0; fil<5; fil++) {
col = 0;
while (palabras [fil] [col]! = `\0') {
for ( x = 0; x<10; x++) {
if (palabras [fil] [col] = = vocales [x]) num++;
}
col ++;
}
printf (\n%s tiene %d vocales, palabras [fil], num);
num = 0;
}
getch ();
}
// Programa, que dispone de un array de 5 enteros y otro array de 5 punteros a entero
// y realiza la ordenación de los enteros mediante los punteros.
#include <conio.h>
#include <stdio.h>
```

void main () {

int enteros [5];

int *punteros [5];

int i, *aux;

clrscr ();

printf (Introdyce 5 números enteros:\n);

for (i = 0; i<5; i++) scanf (%d, &enteros [i]);

for (i = 0; i<5; i++) punteros [i] = &enteros [i];

for (i = 0; i<4; i++) {

if ((*punteros[i] > (*punteros[i + 1])) {

aux = punteros [i];

punteros [i] = punteros [i + 1];

punteros [i + 1];

i = −1;

}

}

printf (El orden de los números es:);

for (i = 0; i <5; i++) printf (%d, *punteros[i]);

getch ();

}

CUARTA CLASE

Con la función set_new_handler () evitamos tener que poner el if para controlar si hay o no hay memoria dinámica. Se ejecutará esta función cuando el new falle. Se encuentra en el archivo de cabecera new.h.

El único parámetro va a ser la función que utilizaremos para el tratamiento de los errores.

<u>EJEMPLO:</u>

Void error () {

Cout << Error, no hay memoria suficiente;

Getch ();

Exit (1);

}

void main () {

set_new_handler (error);

int *array, n;

Clrcsr ();

For (;;) {

Cout << Introduce la dimensión del array;

Cin >> n;

Array = new int [n];

For (int = x; x<n; x++) array [x] = x;

Cout << Contenido del array;

For (x = 0; x<n; x++) cout << array [x] << −;

Getch ();

}

}

FUNCIONES DE USUARIO

Podemos dividir un programa de C++ en el número de funciones que queramos. Se pueden situar en cualquier parte del código fuente o incluso se pueden situar en archivos de diferentes del archivo en el que está el programa fuente. Es necesario declarar las funciones antes de utilizarlas.

<u>Definición de una función:</u> se refiere a dos aspectos:

- ***Descripción de la cabecera de la función (también llamada prototipo)***
- ***Descripción del cuerpo de la función.***

<u>EJEMPLO:</u>

Float media_tres (float num1, float num2, float num3) {

Float media;

Media = (num1 + num2 + num3)/3;

Return media; // devuelve un valor

}

<u>Declaración de funciones</u> (prototipos)

La declaración indica al compilador el tipo de valor que va a devolver la función y el número y tipo de sus parámetros. Lo indicamos básicamente para la comprobación de tipos. El formato del prototipo es el siguiente:

Tipo nombre función (tipo parámetro1, tipo parámetro2,...)

Si una función no devuelve ningún valor puede ser declarada del tipo void. Si una función no tiene parámetros se puede declarar sin parámetros o con void entre paréntesis.

<u>EJEMPLOS:</u>

Void fact (int);

Int fact (); // equivalentes // int fact (void);

También se puede omitir el tipo de dato devuelto en cuyo caso se considerará que es tipo int. En C++ es necesario declarar una función antes de utilizarla. Lo podemos hacer de dos formas:

- ***Definiendo el prototipo***
- ***Definiendo la función completa***

<u>EJEMPLOS:</u>

Void visualizar (char *s);

Void main () {

Visualizar (Hola, mundo);

}

void visualizar (char *s) {

cout << s << \n;

void visualizar (char *n) {

cout << s << \n;

}

void main ();

visualizar (Hola, mundo);

}

}

Notas relativas a las funciones

- ***El prototipo de una función se debe corresponder en el número y tipo de parámetros con la llamada y con la definición de la función.***
- ***El tipo devuelto en la definición de la función debe corresponderse con el indicado en el prototipo.***
- ***Si no concuerdan lo tipos, el compilador tratará de realizar conversiones y si no lo consigue se genera un error.***

Devolución de valores

Una función puede devolver cualquier tipo básico de dato, punteros, referencias, objetos, etc.

EJEMPLO:

Int sumar (int, int);

Void main () {

Int a, b;

Gotoxy (10,10);

Cout << Introduce primer número;

Cin >> a;

Cout << Introduce segundo número;

Cin >> b;

Int c;

C = sumar (a,b);

Cout << El resultado es << c;

}

int sumar (int a, int b) {

return (a + b);

}

Para forzarla devolución de un valor se utiliza la instrucción return indicando el dato a devolver a continuación. En una función puede existir más de una instrucción return y no tiene por qué ser la última

instrucción de la función.

<u>Paso de argumentos</u>

Existen dos formas de realizar el paso de argumentos:

<u>Por valor</u> ! cuando se realice la llamada lo que se hace es una copia de los datos para utilizarlos en la función. Con este sistema los valores originales no se pueden modificar desde la función.

<u>Por referencia</u> ! lo que se pasa es la dirección de memoria del dato utilizado como parámetro. De esta forma ambas variables compartirán la misma zona de memoria. Si cambia el valor de una, cambia el valor de la otra.

<u>EJEMPLO:</u>

```
Void sumar (int, int); // por valor

Void restar (int *, int *); // por referencia

Void main () {

Int a, b;

Gotoxy (10,10);

Cout << Introduce primer número;

Cin >> a;

Cout << Introduce segundo número;

Cin >> b;

Sumar (a,b);

Reatar (&a, &b);

}

void sumar (int i, int d) {

int result = c+ d;

cout << La suma es << resul;

}

void restar (int *c, int *d) {

int resul = *c − *d;

cout << La resta es << resul;
```

}

<u>Argumentos por omisión</u> (parámetros por omisión)

Al llamar a una función se pueden omitir argumentos o parámetros inicializándolos a un valor por defecto. Ese valor será el que tenga el parámetro si es omitido en la llamada. El valor por defecto se indica en el prototipo de la función. El formato es:

Tipo nombre función (tipo parámetro = valor, tipo parámetro = valor,...)

<u>Reglas de utilización de argumentos por defecto</u>

- ***Se pasan por valor***
- ***Sólo pueden ser valores literales o constantes. No pueden ser valores***
- ***Todos los argumentos por defecto deben estar situados al final del prototipo***
- ***Si el primer argumento es por defecto, los argumentos posteriores tendrán que ser por defecto***

<u>EJEMPLO:</u>

```
Int suma (int a = 0, int b = 0, int c = 0);

Int resta (int a; int b = 0, int c = 0);

Void main () {

Clrscr ();

Cout << Para suma (1,2,3) = << suma (1,2,3);

Cout << Para suma (1,2,) = << suma (1,2);

Cout << Para suma (1) = << suma (1);

Cout << Para suma () = << suma ();

Cout << Para resta (1,2,3) = << resta (1,2,3);

Cout << Para resta (1,2) = << resta (1,2);

Cout << Para resta (1) = << resta (1);

Cout << Para resta () = << resta (); // Error

}

int suma (int a, int b, int c) {

return ( a + b + c);

}

int resta (int a, int b, int a) {
```

return (a − b − c);

}

FUNCIONES DE LIBRERIA ESTÁNDAR

Son funciones incluidas en los archivos de cabeceras.

Funciones trigonométricas. (math.h)

Sin ()! devuelve el seno de un ángulo. Su formato es:

double sin (double ángulo);

Cos ()! devuelve el coseno de un ángulo. El formato es:

double cos (double ángulo);

Tan ()! devuelve la tangente del ángulo indicado. El formato es:

double tan (double ángulo);

Atan ()! devuelve el arcotangente del ángulo especificado. El formato es:

Double atan (double ángulo);

Funciones logarítmicas y exponenciales. (math.h)

Log ()! devuelve el logaritmo neperiano de un número. El formato es:

Double log (double número);

Log10 ()! devuelve el logaritmo decimal de un número. El formato es:

Double log10 (double número);

Exp ()! devuelve el exponente neperiano de un número. El formato es:

Double exp (double número);

Sqrt ()! devuelve la raíz cuadrada de un número. El formato es:

Double sqrt (double número);

Pow ()! devuelve el resultado de una potencia. Debemos indicar la base y el exponente. El formato es:

Double pow (double base, double exponente);

EJERCICIOS DE LA CUARTA CLASE

- ***Programa que introduciendo dos números por teclado y mediante un menú de opciones permita realizar la suma, producto y cociente utilizando una función para cada una de estas operaciones***

con llamada por valor.

- *Programa que introduciendo un número por teclado permita calcular su factorial utilizando una función llamada por referencia.*

```
// Programa que, introduciendo dos números por teclado, mediante un menú de

// opciones permita realizar su suma, producto y cociente, utilizando una función

// para cada una de estas operaciones, con paso por valor.

#include <conio.h>

#include <iostream.h>

void main () {

int a, b;

char opción;

void entrarnúmeros (int *, int *);

void suma 8int, int), producto 8int, int), cociente (int, int);

do {

clrscr ();

cout << \n\n 1. − Sumar;

cout << \n\n 2. − Producto;

cout << \n\n 3. − Salir\n;

cin >> opción;

switch (opción) {

case `1' : entrarnúmeros (&a, &b);

suma (a,b);

break;

case `2' : entrarnúmeros (&a, &b);

producto (a,b);

break;

case `3' : entrarnúmeros (&a, &b);
```

```
cociente (a,b);
break;
case `4' : clrscr ();
cout << \n\nFin del Programa;
break;
default : cout << \nOpción incorrecta;
getch ();
}
} while (opción !='4');
getch ();
}
void entrarnúmeros (int *c, int *d) {
clrscr ();
cout << \nIntroduce el primer número:;
cin >> *c;
cout <<\nIntroduce el segundo número:;
cin >> *d;
}
void suma (int op1, int op2) {
int res = op1 + op;
cout << \n\n La suma es : << res;
getch ();
}
void producto (int op1, int op2) {
long int res = op1 * op2;
cout << \n\nEl producto es : << res;
```

```
getch ();
}
void cociente 8int op1, int op2) {
if (op2 == 0) cout << \n\n No se puede dividir por 0;
else {
float res = (float)op1 / op2;
cout << \n\n El cociente es: << res;
}
getch ();
}
// Programa que, introduciendo un número por teclado, permite calcular su factorial
// utilizando una función llamada por referencia
#include <conio.h>
#include <iostream.h>
void main() {
int num;
long int factorial (in *);
clrscr ();
cout << \n\n Introduce un número:;
cin >> num;
cout << \n\n Su factorial e: << factorial (&num);
getch();
}
long int factorial (int *a) {
long int fact = 1;
for (; a<0 ; (*a) −−) fact = fact * (*a);
```

return fact;

}

QUINTA CLASE

CONCEPTO DE CLASE

Una clase es un tipo de dato definido por el usuario que engloba datos y las funciones que trabajaban con esos datos. A los datos les llamamos datos miembro y a las funciones las llamamos funciones miembro o métodos. Una clase tiene dos partes:

- ***Nombre de la clase precedido por las palabras struct o class.***
- ***Cuerpo de la clase entre llaves y terminado en punto y coma. En el cuerpo es donde se definen los datos que pueden ser de cualquier tipo y las funciones miembro. Normalmente de las funciones sólo se pone el prototipo.***

El formato es:

Class nombre clase {

Cuerpo de la clase;

(sentencias);

};

<u>EJEMPLO:</u>

Class número {

Int entero;

Float decimal;

Void sumar ();

Void restar ();

};

Una vez se ha definido la clase ya se pueden crear objetos de esa clase. Los datos miembro no pueden ser incializados en la clase.

En la clase sólo se pone el prototipo y el cuerpo se especifica más adelante fuera de la clase. El formato para definir el cuerpo es:

Tipo nombre clase :: nombre función (parámetros) {

Cuerpo de la función;

}

Al símbolo de : : se le llama operador de ámbito

EJEMPLO:

```
Class número {

Int entero;

Void sumar (float);

};

Void número :: sumar (float int) {

entero = entero + a;

}
```

Una función miembro puede utilizar cualquier dato miembro de la clase libremente.

CONCEPTO DE OBJETO

Un objeto es un elemento creado según un tipo de clase (también se dice que un objeto es una instancia de una clase. El objeto contendrá los datos que se han definido en la clase y podrá utilizar sus funciones miembro. El formato de definición de un objeto a partir de una clase es:

Nombre clase nombre objeto;

EJEMPLO:

```
Struct número {

Int a, b;

Float c;

Void sumar ();

Void restar ();

Void visualizar ();

};

Void número :: sumar () {

a = 3; b = 7; c = 0;

b = b − a;

c = c + b;
```

}

Void número :: restar () {

a = 5;

c = a − b;

}

Void número :: visualizar () {

cout << a: << a << b: << b << c: << c;

}

Void main () {

Número obj1, obj2;

Obj1.sumar ();

Obj2.restar ();

Obj1.visualizar ();

Obj2.visualizar ();

}

Un dato miembro se puede declarar estático. Si es así, ese dato será compartido por todos los objetos, es decir, habrá una única copia de él. También se puede declarar como dato miembro, un objeto de otra clase.

EJEMPLO:

Class número {

(sentencias);

};

class letra {

número obj1;

};

class número;

class letra {

número obj1;

(sentencias);

};

class número {

(sentencias);

};

Un objeto de una clase no puede ser miembro de ella misma pero sí un puntero a un objeto.

<u>EJEMPLO:</u>

Class letra {

Letra a; //error. Daría un dato infinito.

Letra *b;

};

ÁMBITO DE UNA CLASE

Se refiere al acceso que podemos tener a los datos miembro. Para acceder a un dato o a un función tenemos tres formas:

A través de un objeto de su clase con el operador .

A través de un puntero a un objeto de su clase con el siguiente operador − >

Con el nombre de la clase utilizando el operador de ámbito :: siempre que hay asociado un objeto.

<u>EJEMPLO:</u>

Struct número {

Int num;

Float decimal;

Void suma ();

Void resta ();

};

void número :: suma () {

float resul = num + decimal;

cout << resul;

número :: resta (); // 3ª forma

}

void número :: resta () {

float resul = decimal − num;

cout << resul;

}

void main () {

número *objeto ();

objeto −> suma (); // 2ª forma

número objeto2;

objeto2.suma (); //1ª forma

}

CONTROL DE ACCESO

Para controlar el acceso a los miembros de una clase se utilizan las palabras clave: public, protected, private. Estas palabras se anteponen a los datos miembros que queramos que sean públicos, protegidos o privados.

EJEMPLO:

Class letra {

Private:

Int a;

Protected:

Char b;

Public:

Float c;

};

<u>public:</u> se puede acceder a la parte pública de una clase desde cualquier lugar del programa.

<u>Private:</u> se puede acceder a la parte privada de una clase únicamente desde funciones miembro de la clase (también funciones amigas). No se puede acceder a los datos miembros de la clase desde funciones de una clase derivada.

<u>Protected:</u> es tratado como privado pero funciona como público para las funciones de clase derivadas.

DIFERENCIA ENTRE CLASS Y SRUCT

Si indico class, por defecto todos los datos y funciones miembro son privados. Si indicamos struct todos los datos y funciones miembro son por defecto públicos.

Habitualmente los datos miembro se ponen privados o protegidos si vamos a hacer herencia y las funciones miembro se ponen públicos.

OBJETOS CONSTANTES

Definir un objeto como constante significa dar valores iniciales a los datos miembro del objeto y no modificarlos durante la ejecución del programa. Un objeto se declara como constante con el siguiente formato:

Const nombre clase nombre objeto;

Las funciones que utiliza un objeto constante deben ser definidas como funciones constantes. Una función constante es una función que no va a modificar el valor de los datos miembro.

<u>EJEMPLO:</u>

Class fecha {

Int día, mes, anyo;

Public:

Void asignarfecha (fecha &) const;

};

void fecha :: asignarfecha (fecha &a) const {

(sentencias);

}

void main () {

const fecha hoy, ayer;

hoy.asignarfecha (ayer);

(sentencias);

}

ARRAYS DE OBJETOS

Un array de objetos se crea exactamente igual que un array de cualquier otro tipo de datos. El formato es:

Nombre clase nombre objeto [tamaño];

<u>EJEMPLO:</u>

Class fecha {

(sentencias);

};

void main () {

fecha array [100];

(sentencias);

}

FUNCIONES DE LIBRERÍA ESTÁNDAR (II)

<u>Otras funciones matemáticas.</u> (math.h)

Abs () devuelve el valor absoluto de un número entero. El formato es:

Int abs (int número);

Fabs () devuelve el valor absoluto de un número de tipo double. El formato es:

Double fabs (double número);

Labs () devuelve el valor absoluto de un dato long int. El formato es:

Long int labs (long int número);

Cabs () devuelve el valor absoluto de un número complejo. El dato de tipo complejo está definido como estructura:

Struct complex {

Double x, y;

}

El formato de esta función es: double cabs (struct complex número);

Ceil () devuelve la parte entera de un número redondeándolo al entero mayor. El formato es:

Double ceil (double número);

Floor () devuelve la parte entera de un número redondeándolo al entero inferior. El formato es:

Double floor (double número);

Fmod () devuelve el resto de la división entre dos números. El formato es:

Double fmod (double dividendo, double divisor);

Modf () descomponen un número en su parte decimal y entera devolviendo la decimal y guardando la entera donde se indique. El formato es:

Double modf (double num, double *ent);

EJERCICIOS DE LA QUINTA CLASE:

- ***Codificar un programa que maneje una agenda de teléfonos. La agenda será un array de objetos de la siguiente clase:***

```
Class agenda {

Char nombre [20];

Char apellidos [40];

Char teléfono [10];

Public:

Void asignar datos ();

Void visualizar ();

};

//Programa que maneja una agenda de 10 teléfonos, mediante un array de objetos de la

// siguiente clase

#include <conio.h>

#include <iostream.h>

class agenda {

char nombre [20];

char apellidos [40];
```

```
char teléfono [10];
public:
void introducir ();
void visualizar ();
};
void agenda :: introducir () {
clrscr ();
gotoxy (10,10);
cout << Nombre:;
gotoxy (10,12);
cout << Apellidos:;
gotoxy (10,14);
cout << Teléfono: ;
gotoxy (22,10);
cin >> nombre;
gotoxy (22,12);
cin >> apellidos;
}
void agenda :: visualizar () {
cout << `\t' << nombre `\t' << apellidos << `\t' << teléfono << `\n;
}
void main () {
agenda agen [10];
int índice = 0, cont;
char op;
do {
```

```
clrscr ();
gotoxy (10,10);
cout << AGENDA DE TELÉFONOS;
gotoxy (14,14);
cout << 1. Introducir teléfono;
gotoxy (14,16);
cout << 2. Visualizar teléfonos;
gotoxy (14,18);
cout << 3. Salir;
gotoxy (14,20);
cout << Elige una opción;
cin >> op;
switch (op); {
case `1': if (índice < 9) {
agen [índice].introducir ();
índice ++;
}
else {
gotoxy (50, 20);
cout << Agenda llena;
}
break;
case `2': clrscr ();
for (cont = 0; cont < índice; cont ++) {
agen [cont].visualizar ();
}
```

getch ();

break;

case `3': cout << FIN;

}

}while (op! = `3');

}

SEXTA CLASE

CONSTRUCTORES

Son funciones que sirven para inicializar los datos miembro de los objetos que se crean. Estas funciones se llaman automáticamente cuando se crea el objeto. Sus características son:

- ***Tiene el mismo nombre que la clase a la que pertenece.***
- ***El constructor no se hereda, no devuelve valores y no se puede declarar ni como const, virtual o static.***
- ***Puede admitir parámetros.***
- ***Puede existir más de un constructor e incluso no existir. En el caso de que no haya constructor se crea uno automáticamente por omisión: (datos miembro valor cero, punteros valor null).***

EJEMPLO:

Class clase {

Int a, b, c;

Public:

Clase (); // constructor. No devuelve valor

Void visualizar ();

};

clase :: clase () {

a = 5; b = 3; c = 2;

}

void clase :: visualizar () {

gotoxy (10,10);

cout << a << b << c;

}

void main () {

clrscr ();

clase obj;

obj.visualizar ();

}

TIPOS DE CONSTRUCTORES

***Constructor por defecto** es el que se crea automáticamente si no se especifica un constructor. No acepta argumentos o parámetros. Da valores iniciales cero y null.*

***Constructor con argumentos** es un constructor que admite argumentos que serán enviados al crear el objeto. El formato será:*

Nombre clase nombre objeto (argumentos);

***Constructor con argumento por defecto** los argumentos por defecto se especifican en el prototipo del constructor. Los datos miembro de la clase se inicializarán con estos valores por defecto si no se especifica argumento. Se utilizan igual que en cualquier función con argumentos por defecto.*

***Constructores sobrecargados** cuando una clase tiene más de un constructor se dice que están sobrecargados. Todos los constructores tendrán el mismo nombre y se diferenciarán por el número y tipo de sus parámetros.*

EJEMPLO:

Class número {

Int num;

Float decimal;

Public:

Número ();

Número (int a, float b = 45);

Void sumar ();

};

número :: número () {

cin >> num;

cin >> decimal;

}

número :: número (int a, float b) {

num = a;

decimal = b;

}

void número :: sumar () {

float resul = num + decimal;

cout << Resultado << resul;

}

void main () {

clrscr ();

número uno; // llamará al constructor sin parámetros

uno.sumar () // suma de los valores introducidos

número dos (s);

dos.sumar (); // resultado 9.5

número tres (7,13.2);

tres.sumar (); //resultado 20.5

getch ();

}

Constructores copia sirven para crear un nuevo objeto a partir de otro ya existente. Se pretende que los datos miembro del nuevo objeto tengan los mismos valores que los datos miembro del objeto ya existente. Para eso, el constructor copia tendrá un argumento que será una referencia a un objeto constante de esa clase. El formato será:

Nombre clase:: nombre clase (const nombre clase &);

EJEMPLO:

Class fecha {

Int día, mes, anyo;

```
Public:

Fecha ();

Fecha (int, int, int);

Fecha (const fecha &);

Void visualizar ();

};

fecha :: fecha () {

día = 8;

mes = 9;

anyo = 1999;

},

fecha :: fecha (int d, int m, int a) {

día = d;

mes = m;

anyo = a;

}

fecha :: fecha (const fecha & x) {

día = x.día;

mes = x.m,es;

anyo = x.anyo;

}

void visualizar () {

cout << día << `/' << mes << `/' << anyo;

}

void main () {

clrscr ();
```

fecha hoy (29,9,98);

fecha futuro;

fecha otra fecha (hoy);

hoy.visualizar ();

otra fecha.visualizar ();

futuro.visualizar ();

}

Si no creamos un constructor copia, el compilador crea uno por defecto que podremos utilizar con el operador igual (=). Todos los datos miembro del objeto de la izquierda tomarán los valores de los datos miembro del objeto de la derecha.

EJEMPLO: *fecha otrafecha = hoy;*

Esto va a dar problemas si entre los datos miembro hay punteros o arrays. Tendríamos que decirle, por ejemplo, que copie el contenido de la dirección porque el constructor por defecto copia la dirección.

DESTRUCTORES

Un destructor es una función que permite destruir cada objeto construido. Lo que hace es liberar la memoria que ocupa. Cuando el proceso de ejecución del programa sale del ámbito donde el objeto fue construido, automáticamente se llama al destructor. Sus características son:

- ***El nombre del destructor es el mismo que el de su clase precedido de este símbolo ~ (ALT + 126).***
- ***No puede devolver ningún tipo de dato.***
- ***Cada clase sólo puede tener un destructor; si no indicamos destructor crea uno por defecto aunque no será válido cuando reservemos memoria dinámica.***
- ***No se puede heredar ni ser declarado const ni static pero sí virtual (es recomendable).***

EJEMPLO:

Class cadena {

*Char *datos;*

Public:

Cadena ();

Cadena (int s);

~cadena ();

void visualizar ();

void entrar ();

```
};
cadena :: cadena () {
int a;
cout << Introduce longitud;
cin >> a;
datos = new char [a];
}
cadena :: cadena (int s) {
datos = new char [s];
}
cadena :: ~cadena () {
delete [] datos; // libera la memoria de lo que apunta el puntero
}
void cadena :: entrar () [
cout << Entre una palabra;
cin >> datos;
}
void cadena :: visualizar () {
cout << Has entrado << datos;
}
void main () {
cadena string;
string.entrar ();
string.visualizar ();
getch ();
}
```

EL PUNTERO IMPLÍCITO THIS

Es un puntero que siempre apunta al objeto con que hemos llamado a una función miembro. El puntero this se utiliza en:

- ***Para hacer una copia del objeto asociado o para asignar un nuevo valor al objeto.***

<u>EJEMPLO:</u>

```
Class fecha {

Int día, mes, anyo;

Public:

Void asignar fecha ();

};

void fecha :: asignar fecha () {

fecha fecha2;

fecha2 = *this; // Querría hacer fecha2 = fecha1 pero no puedo ponerlo porque // fecha1 es local de la main

}

void main () {

fecha fecha1;

fecha1.asignar fecha ();

}
```

- ***Al llamar a una función si queremos pasarle como argumento el objeto con que se llamó a la función miembro.***

<u>EJEMPLO:</u>

```
Void clase :: asignar fecha () {

Visualizar (this);

}
```

- ***Para devolver una referencia al objeto asociado con la llamada a la función miembro***

<u>EJEMPLO:</u>

```
Class pareja {
```

```
Char c1, c2;
Public:
Pareja (char, char);
Pareja & incrementar (); // pareja es el dato que devuelve
Void imprimir ();
};
pareja :: pareja (char a, char b) {
c1 = a;
c2 = b;
}
pareja & pareja :: incrementar () {
c1 ++; // suma 1 a su código ASCII
c2 ++;
return (*this);
}
void pareja :: imprimir () {
cout << c1 << c2; // cout. Visualizar (c1)
}
void main () {
pareja a (`A', `B');
pareja b (`C', `D');
pareja c (`E', `F');
a.imprimir ();
b.imcrementar (). Imprimir ();
c.imprimir ();
}
```

No se podría poner b.incrementar ().imprimir () si imprimir no devuelve un dato de tipo pareja.

- ***Para referenciar explícitamente un miembro oculto debido a una declaración***

<u>EJEMPLO:</u>

```
Class x {
Int longitud;
Public:
Void función (int);
};
void x :: función (int longitud) {
this −> longitud = longitud;
}
```

EJERCICIOS DE LA SEXTA CLASE

Desarrollar la siguiente clase:

```
Class racional {
Int num, den;
Public:
Racional (); //constructor
Racional sumar (racional &);
Racional restar (racional &);
Void visualizar ();
};
// Desarrollo de una clase par números racionales y ejemplo de su uso
#include <iostream.h>
·include <conio.h>
class racional {
int num, den;
```

```
public:
racional ();
racional (int, int);
racional sumar (racional &);
racional restar (racional &);
void visualizar ();
};
racional :: racional () {
cout << Numerador:;
cin >> num;
do {
cout << Denominador:;
cin >> den;
} while (den == 0);
}
racional :: racional (int x, int y) {
if (y == 0) return;
num = x;
den = y;
}
racional racional :: sumar (racional) &rac) {
racional res (0,1);
res.num = (num * rac.den) + (den * rac.num);
res.den = (den * rac.den);
return res;
}
```

```
racional racional :: restar (racional &rac);

racional res (0,1);

res.num = (num * rac.den) − (den * rac.num);

res.den = (den * rac.den);

return res;

}

void racional :: visualizar () {

cout << num << `/' << den;

}

void main () {

clrscr ();

racional rac1, rac2, resul (0,1);

resul = rac1, sumar (rac2);

cout << \nEl resultado de la suma es de:;

resul. Visualizar ();

resul = rac1.restar (rac2);

cout << \n El resultado de la resta es de:;

resul.visualizar ();

getch ();

}
```

SÉPTIMA CLASE

FUNCIONES AMIGAS

Una función amiga de una clase es una función no miembro que puede acceder a los datos miembro privados y protegidos de la clase. Para declarar una función amiga simplemente se antepone la palabra friend al prototipo de la función. El formato es:

```
Class nombre clase {

Friend función amiga ();
```

(sentencias);

};

<u>EJEMPLO:</u>

```
Class dat {
Int dato;
Public:
Friend void cargar (dat &dat, int x);
Void visualizar ();
};
void dat :: visualizar () {
clrscr ();
cout << dato;
}
void cargar (dat &t, int x) { //pasa un objeto de tipo dat
t.dato = x;
}
void main () {
dat t;
cargar (t, 50);
t.visualizar ();
}
```

Con esto se puede declarar también una función miembro de una clase como amiga de otra clase.

<u>EJEMPLO:</u>

```
Class c1;
Class c2 {
Int nc2;
```

```
Public:
Void asignar dato (int n) {nc2 = n;} // función inline
Int obtener dato (const c1&);
};
class c1 {
friend int c2 :: obtener dato (const c1&);// pasa un dato de tipo c1
int nc1;
public:
void asignar dato (int n) {nc1 = n;}
};
int c2 :: obtener dato (const c1 &obj) {
return obj.nc1 + nc2;
}
void main () {
c1 objeto 1;
c2 objeto 2;
int dato;
cout << Entre un número:;
cin >> dato;
objeto1.asignar dato (dato);
cout << Entre otro número:;
cin >> dato;
objeto2.asignar dato (dato);
cout << Resultado;
cout << objeto2.obtener dato (objeto1);
}
```

FUNCIONES INLINE

Declarar una función inline significa que el compilador tiene la facultad de reemplazar cualquier llamada a la función en el programa fuente por el cuerpo de la función. La desventaja es que el código va a aumentar pero el programa ganará en rapidez porque no hará la llamada. Hay dos formas de declarar una función inline:

- ***La primera es poniendo la palabra inline delante de la definición de la función.***

<u>EJEMPLO:</u>

```
Inline float librasKgm (float libras) {

Return 0.453 * libras;

}

void main () {

clrscr ();

float lbs;

cout << Introduzca su peso en libras:;

cin >> lbs;

cout << Su peso en KG es: << librasKgm (lbs);

}
```

- ***La segunda es cuando la función sea miembro de una clase incluyendo el cuerpo de la función dentro de la declaración de la clase.***

<u>EJEMPLO:</u>

```
Class reloj {

Int hora, minutos, segundos;

Public:

Reloj (int, int, int); // constructor

Reloj (); // constructor

Void visualizar () {

Cout << Hora: << hora << Minutos: << minutos << Segundos << segundos;

}
```

};

SOBRECARGA DE OPERADORES

Es la facultad que tiene un operador de funcionar de forma diferente según el tipo de dato con el que se utilice, es decir, hacer que el operador funcione con los datos que yo cree. El formato de la sobrecarga de un operador si la función no es miembro es:

Tipo operator operador (argumentos) {

(sentencias);

}

Si la función es miembro de una clase el formato es:

Tipo clase :: operator operador (argumentos) {

(sentencias);

}

En el cuerpo de la función habrá que poner las operaciones que queramos que realice el operador.

<u>EJEMPLO:</u>

```
Class clase {
Int x, y;
Public:
Clase (int x = 0; int y = 0);
Clase operator + (clase&);
Void visualizar ();
};
clase :: clase (int m, int n) {
x = m;
y = n;
}
clase clase :: operator +(clase &obj) { // pasa un objeto de tipo clase
clase aux;
```

aux.x = x + obj.x // x del objeto más x que pasa

aux.y = y + obj.y;

return aux;

}

void clase :: visualizar () {

cout << x << x << y << y;

}

void main () {

clase a (5,7);

clase b (3,4);

clase c;

c = a + b; // x de a más x de b; y de x más y de b

c.visualizar (); // c = a.operator + (b);

}

Las restricciones que hay en la sobrecarga de operadores son:

- ***No se pueden crear nuestros propios operadores. Sólo se pueden sobrecargar los ya existentes.***
- ***La sobrecarga de operadores sólo funciona con objetos de una clase.***
- ***No se puede cambiar la prioridad de los operadores.***

<u>Sobrecarga de los operadores de inserción y extracción</u> (<< , >>)

Es los mismo poner cout << a que poner cout.operator << (a);. También es lo mismo poner cou t<< a << b; que poner (cout.operator << (a)).operator << (b);

La sobrecarga de estos operadores se realizará mediante funciones amigas. Al declarar la sobrecarga habrá que poner:

Friend ostream & operator << (ostream & os, const clase &c);

Friend istream & operator >> (istream & is, const clase &c);

<u>EJEMPLO:</u>

Class complejo {

Int real, imag;

```
Public:
Complejo (int r = 0; int i = 0);
Friend ostream & operator << (ostream &, complejo &);
Friend istream & operator >> (istream &, complejo &);
};
complejo :: complejo (int r, int i) { //complejo :: complejo (int r, int i): real (r) imag (r) {}
real = r;
imag = i;
}
ostream & operator << (ostream & os, complejo &c) {
os << c.real << + << c.imag << i;
return os; // os es de tipo ostream
}
istream & operator << (istream & is, complejo &c) {
cout << Introduce parte real;
is >> real;
cout << Introduce parte imaginaria;
is >> imag;
return is; // is es de tipo istream
}
void main () {
complejo c1, c2 (7,5);
cout << c1 << c2;
cin >> c1;
cout << Nuevo c1 << c1;
}
```

EJERCICIOS DE LA SÉPTIMA CLASE

Desarrollar la siguiente clase sobrecargando los operadores indicados:

Class racional {

Int num, den;

Public:

Racional (int x = 0, int y = 1);

Friend ostream & operator << (ostream &, racional &);

Friend istream & operator >> (istream &, racional &);

Racional operator + (racional);

Racional operator − (racional);

};

//Desarrollo de una clase para números racionales, incluida sobrecarga de operadores // + − << y >>, y ejemplo de su uso

#include <iostream.h>

#include <ostream.h>

Class racional {

Int num, den;

Public:

Racional (int x = 0, int y = 1);

Friend ostream & operator << (ostream &, racional &);

Friend istream & operator >> (istream &, racional &);

Racional operator + (racional);

Racional operator − (racional);

};

racional :: racional (int x, int y) {

num = x;

if (y == 0) y = 1;

```
den = y;
}
racional racional :: operator + (racional rac) {
racional res;
res.num = (num * rac.den) + (den * rac.num);
res.den = den * rac.den;
return res;
}
racional racional :: operator − (racional rac) {
racional res;
res.num = (num * rac.den) − (den * rac.num);
res.den = den * rac.den;
return res;
}
ostream & operator << (ostream &os, racional &rac) {
os << `\t' << rac.num;
os << \n\t−−\n;
os << `\t' << rac.den;
return os;
}
istream & operator >> (istream &is, racional &rac) {
cout << `\t';
is >> rac.num;
cout << \t;
cout << \n\t;
is >> rac.den;
```

```
return is;

}

void main () {

char op;

do {

racional a,b,c;

clrscr ();

cout << Introduzca dos números racionales :;

gotoxy (2,3);

cin >> a;

gotoxy (1,8);

cin >> b;

clrscr ();

c = a + b;

cout << \nEl resultado de la suma es:\n << c;

c = c − b;

cout << \nEl resultado de la resta es:\n << c;

gotoxy (50,20);

cout << ¿Otra operación (s/n)?:;

cin >> op;

}while(op ! = `N' && op! = `n');

}
```

OCTAVA CLASE

HERENCIA

Es la propiedad que permite a los objetos crearse a partir de otros objetos. Cada subclase comparte características comunes con la clase de la que deriva. La clase original la llamamos clase base y las nuevas clases creadas a partir de ellas, clases derivadas. Si una clase recibe características de más de una clase se llama herencia múltiple. La herencia la vamos a crear al declarar la clase derivada. El formato será:

Class nombre derivada : acceso nombre base1 [,acceso nombre base2...] {

(sentencias)

};

Según el acceso, vamos a tener:

- ***<u>Derivación pública</u> (public) los miembros públicos de la clase base serán miembros públicos en la clase derivada. Los miembros protegidos de la clase base serán protegidos en la derivada. Los miembros privados de la clase base serán inaccesibles desde la derivada.***
- ***<u>Derivación protegida</u> (protected) todos los miembros públicos y protegidos de la clase base son protegidos en la derivada y los privados serán inaccesibles.***
- ***<u>Derivación privada</u> (private) todos los miembros públicos y protegidos de la clase base serán privados en la derivada. Los privados seguirán siendo inaccesibles en la derivada.***

<u>EJEMPLO:</u>

Class base {

Int a;

Protected:

Int b;

Public:

Int c;

};

class derivada : public base {

private:

int d;

protected:

int e;

public:

int f;

};

void main () {

clrscr ();

derivada der;

der.a = 1; //Error. Inaccesible

der.b = 2;//Error. Sólo las uso desde funciones miembro.

der.c = 3;

der.d = 4; //Error. Inaccesible.

der.e = 5;//Error;

der.f = 6;

}

Lo normal es que todos los datos miembro de la clase base sean protegidos.

EJEMPLO:

Class base {

Int a;

Protected:

Int b;

Public:

Int c;

};

Class derivada : public base {

Int d;

Protected:

Int e;

Public:

Int f;

Void darA (int x) {a = x;}

Void darB (int x) {b = x;}

Void darC (int x) {c = x;}

```
Void darD (int x) {d = x;}

Void darE (int x) {e = x;}

Void darF (int x) {f = x;}

};

void main () {

derivada der;

der.darA (1); //Error. Inaccesible

der.darB (2);

der.darC (3);

der.darD (4);

der.darE (5);

der.darF (6);

}
```

Algunas características de la herencia son:

- ***Una clase derivada puede ser a su vez una clase base dando lugar a una jerarquía de clases.***
- ***La clase derivada hereda todos los datos y funciones miembro.***
- ***Una clase derivada puede añadir a los miembros heredados sus propios datos y funciones miembro.***

<u>Constructores y destructores en la herencia</u>

Cuando creemos un objeto de una clase derivada debemos llamar al constructor de la clase base y al de la clase derivada. La clase derivada debe definir un constructor que además de dar valores iniciales a los datos miembro de esta clase también llame al constructor de la clase base. El orden de ejecución será:

- ***Llamando al constructor de la clase derivada.***
- ***Llamada del constructor de la clase derivada al constructor de la clase base.***
- ***Ejecución del cuerpo del constructor de la clase base.***
- ***Ejecución del cuerpo del constructor de la clase derivada.***

Cuando el constructor de la clase base no tenga argumentos este proceso se realiza automáticamente. Si el constructor de la clase base tiene argumentos, debe indicarse explícitamente la llamada al constructor desde el constructor de la clase derivada. El formato de la llamada a la clase del constructor base desde el constructor de la derivada será:

Nombre clase derivada :: nombre clase derivada (argumentos) : clase base (argumentos) {

(sentencias)

}

Los destructores funcionan en la herencia igual que los constructores pero con el orden de ejecución invertido. Los destructores siempre se van a llamar de forma implícita porque el destructor no lleva argumentos.

<u>EJEMPLO:</u>

```
Class base {

Protected:

Int a;

Public:

Base (int);

~ base ();

int doble();

};

base :: base (int x) {

a = x;

cout << Constructor base;

}

base :: ~ base { // Sólo para punteros para datos miembro

cout << Destructor base;

}

int base :: doble () {

return a*2;

}

class derivada :: public base {

int b;

public:

derivada (int);
```

```
~ derivada ();

void mostrar ();

};

derivada :: derivada (int y) : base (y) {

b = y;

cout << Constructor derivada;

}

derivada :: ~ derivada () {

cout << Destructor derivada;

}

void derivada :: mostrar () {

cout << El doble de << a << es << doble ();

cout << El triple de << b << es << b*3;

}

void main () {

clrscr ();

derivada d(5);

d.mostrar ();

}
```

Anulación y redefinición de funciones

Es posible que un dato o función miembro de una clase derivada tenga el mismo nombre que un dato o función miembro de la clase base. En ese caso, el dato o función miembro de la clase base queda anulado (oculto pero no destruido). Para utilizar un dato o función miembro anulado se utiliza el operador de ámbito.

EJEMPLO:

```
Class base {

Protected :

Int i;
```

Public:

Base (int x) { i = x;}

};

clñass derivada : public base {

int i;

public:

void imprimir () {int total = i + base :: i;

cout << El total es << total;}

};

void main () {

clrscr ();

derivad der (4,5);

der.imprimir ();

}

Herencia múltiple

Es la propiedad por la cual una clase derivada recibe características de más de una clase base. Será necesario usar a veces clases base virtuales debido a que en una clase derivada tengamos dos datos del mismo tipo al hacer herencia múltiple. Cuando utiliza clases base virtuales las crearemos igual que si no lo fueran pero en la derivación debemos especificar la palabra virtual:

Class clase derivada : public virtual class base {

(sentencias);

};

Con esto evitaremos que una clase sea derivada dos o más veces.

POLIMORFISMO

Capacidad de que objetos diferentes respondan a órdenes similares de forma diferentes. También se puede decir como la capacidad de que una orden realice funciones diferentes en cada momento. La sobrecarga de funciones es una especie de polimorfismo. El verdadero polimorfismo lo vamos a utilizar mediante funciones virtuales. Una función virtual es una función miembro de una clase que puede se redefinida en cada una de las clases derivadas y una vez redefinida se puede acceder a ella mediante un puntero o una referencia a la clase base resolviéndose la llamada en función, del tipo de objeto apuntado. Para declarar una función virtual anteponemos la palabra virtual a su declaración. Su redefinición en las clases

derivadas será por defecto virtual sin necesidad de especificarlo.

<u>EJEMPLO:</u>

```
Class base {

Public:

Void mostrar () { cout << Clase base;}

};

class derivada : public base {

public:

void mostrar () {cout <<Clase derivada;}

};

void main () {

derivada dv;

bnse *ptr;

dv.mostrar ();

ptr = &dv;

ptr −> mostrar (); //Clase base

}
```

Con virtual mira a qué tipo de dato está apuntando el puntero, no las resuelve las funciones miran el tipo de dato del puntero.

NOVENA CLASE

FICHEROS

El poder utilizar clases nos permite realizar entradas y salidas de datos aplicable a un disco. El archivo de cabeceras donde están todas las funciones es fstream.h diviéndose en ifstream.h y ofstream.h. Para poder leer un fichero primero hay que abrirlo ya sea para lectura, para escritura o para ambas. Los ficheros los tenemos de texto o binario. El de texto sólo puede grabar textos y en el binario grabaremos cualquier tipo de dato.

La operación de apertura de un archivo supone la creación de un objeto ostream para grabar e istream para leer. Los modos de apertura están en la clase ios.

Ios :: in abierto para leer

Ios :: out abierto para escribir

Ios :: ate abre y salta al final

Ios :: app añade por el final (el más usado)

Ios :: trunc si el fichero existe lo destruye y lo vuelve a crear desde el principio.

Ios :: binary fichero binario

<u>EJEMPLO:</u>

Ios :: binary | ios :: out

Un fichero lo podemos abrir de dos maneras:

- ***Definiendo un objeto y abrirlo a través de un constructor:***

Ofstream, ifstream, fstream nombre objeto (nombre archivo, acceso)

- ***Abrirlo directamente con open:***

Ifstream, ofstream, fstream nombre objeto

Nombre objeto.open(nombre archivo, acceso)

Para cerrarlo en ambos casos vamos a utilizar la función clase:

Nombre objeto.close();

<u>EJEMPLO:</u>

Ifstream fichero (a:\\Agenda.dat, ios :: binary | ios :: in); //constructor

<u>EJEMPLO:</u>

Ifstream fichero;

fichero.open(a:\\Agenda.dat, ios :: binary | ios :: in);

Siempre hay que controlar que esté bien abierto el fichero. Para ello se procede de la siguiente manera:

Fstream fichero;

Fichero.open(a:\\agenda.dat, ios :: binary | ios :: out);

If(!fichero){

Cout << El fichero no está bien abierto << endl; //equivale a \n

Exit (1);

}

Ficheros de texto

Para enviar datos a un archivo vamos a utilizar <<. Para recibir datos utilizaremos >>. También podemos utilizar las funciones put () y get ()(introducir y sacar datos).

EJEMPLO:

#include <conio.h>

#include<iostream.h>

#include<stdlib.h>

#include<fstream.h>

void main (9 {

clrscr();

ofstream salida;

salida.open(a:\\Agenda.dat, ios :: out);

if(!salida) {

cout << fichero mal abierto;

}

char car;

while (cin.get(car)) {

salida.put(car);

};

getch();

}

Ficheros binarios

Los archivos binarios me van a permitir grabar cualquier tipo de dato en ellos. Se utilizan unas funciones especiales. Dentro de la clase istream tenemos funciones miembro seekp y tellg. Mueven un puntero a distintas posiciones del fichero:

Ios :: beg al principio

Ios :: cur al sitio actual

Ios :: end al final del fichero

Al igual que llos de texto vamos a poder utilizar get() y put(). La impresora se tratará como un archivo.

<u>EJEMPLO:</u>

Ofstream impresora;

Impresora.open(prn, ios :: out);

If(!impresora) {

Cout << Error al abrir impresora;

Exit(1);

};

ENTRADA Y SALIDA AVANZADA (FLUJO)

El sistema de E/S en C++ está definido para trabajar con diferentes dispositivos. Cada uno de estos dispositivos se convierte en un dispositivo lógico que se llama flujo. Éstos, forman una interfaz entre el programa, el dispositivo y el usuario. Este sistema se le conoce como buffers. El buffer de entrada es cin y para sacar es cout. El archivo de cabecera iostream.h va a ser el que va a definir el interfaz con la biblioteca de flujos.

Cin toma caracteres de la entrada estándar y los va a almacenar en el buffer. A continuación los pasará a las variables elegidas. Se utiliza >> para introducir datos y << para la inserción de los datos.

Cout pone caracteres en la salida estándar y los almacena en el buffer. Más tarde los visualiza.

Las funciones de salida son put () para caracteres y write () para cadenas. Las funciones de entrada son get () para recuperar una cadena completa de texto incluyendo los espacios y getline () que sirve para introducir cadenas de caracteres.