**ALGORITMOS Y ESTRUCTURAS DE DATOS**
**SEMANA Nº 3**

**OBJETIVOS DE LA TERCER SEMANA**

**Clase teórica**
- Lenguaje C++.
- Elementos del lenguaje.
- Clases en C++ (Estructura general, declaración y definición, cuerpo de una clase. Instancias de una clase (objetos) (acceso a los miembros, la vida de un objeto), declaración de miembros de clases (modificadores de acceso a miembros de clases) y atributos.

**Clase práctica**
- Ejercicios de clases con código y diagramas de flujo.

*CLASE TEORICA*

# LENGUAJE C++

## INTRODUCCION

C++ es uno de los lenguajes más usados en el campo profesional. Heredando toda la potencia y flexibilidad del lenguaje C, C++ añade unas posibilidades de orientación a objetos que permiten simplificar el diseño de proyectos de gran complejidad, todo ello sin sacrificar las características de bajo nivel que permiten realizar llamadas a todo la API de Windows, utilizar punteros, y, en definitiva, llegar a los rincones más difíciles del sistema, aprovechando todos sus recursos.

C++, es seguramente el lenguaje más flexible y potente con el que cuentan actualmente los programadores profesionales, un lenguaje totalmente orientado a objetos y para el que no existen limitaciones. C++ se caracteriza por ser un compilador muy rápido y generar código directamente ejecutable, no pseudo-código que es necesario interpretar posteriormente durante la ejecución. Esto conlleva ventajas para el programador y el usuario, ya que el ejecutable es más rápido, se ejecuta sin necesidad de interpretación.

**Características principales**

- Estructurado y orientado a objetos.
- Compilador.
- Nivel bajo, medio y alto.
- Portable, se adapta a diferentes equipos.
- Eficiente.
- Tipos de datos básicos.
- Carencia de comprobaciones: Limites de vectores, compatibilidad de tipos, etc.
- Manejo de direcciones de memoria.

**CUERPO DE UN PROGRAMA EN C++**

Un programa en C++, en su versión más simple, está formado sólo por la función *main().* En esta función se pueden introducir las sentencias necesarias para realizar la acción que se desee, por ejemplo imprimir un mensaje. El siguiente fragmento es un programa completo, que no hace nada, ya que en el bloque so se ha introducido ninguna sentencia.

```
void main()
{
}
```

Las llaves que hay tras main(), son las que delimitan el cuerpo de la función en el cual se incluirán las sentencias necesaria. La palabra que hay delante del nombre de la función, void, indica el tipo del valor que devolverá la función, en este caso no devuelve nada.

Las sentencias dispuestas en la función principal del programa son las que tomarán el control en cuanto el programa se ejecute. Una sentencia puede ser una instrucción simple, una llamada a función, una sentencia de asignación, la evaluación de una expresión, etc.

## CONVENCIONES LEXICAS

**FINALIZADOR DE SENTENCIAS ;**

En c++ una sentencia puede ocupar una o varias líneas. También es posible escribir en una sola línea varias sentencias. Por lo tanto, para que el compilador sepa dónde termina una determinada sentencia tendremos que utilizar el indicador punto y coma.

Detrás de las llaves, la directiva include o la cabecera de la función no hay punto y coma, ya que dichas líneas no son en sí sentencias, no tienen código ejecutable.

```
x = y;
y = y + 1;
h=5;          g=10;
suma ();
```

**DELIMITADOR DE BLOQUE { }**

Los delimitadores de bloque se utilizan para delimitar el cuerpo de las funciones en el cual se incluirán las sentencias necesarias. Los bloques se delimitan con las llaves, cualquier sentencia o grupo de sentencias encerradas entre las llaves indican que pertenecen a ese bloque de código.

```
main()                    suma ()
{                         {
        ...                       ...
}                         }
```

**COMENTARIOS /* ……*/ //…….**

En C++ un comentario puede ocupar una o más líneas, ya que su inicio y fin estarán marcados mediante delimitadores. Para iniciar un comentario se usa la pareja de caracteres /*. Para cerrar dicho comentario utilizaremos el delimitador complementario, */. Si el comentario va al final de una línea, generalmente se utiliza la pareja de caracteres // para comenzar y el retorno de línea para cerrar el mismo. Cualquier carácter que se encuentre entre los delimitadores de inicio y fin del comentario no será tenido en cuenta por el compilador.

Por ejemplo:

```
void main()          //función principal
{
    /*     comentario en
           varias líneas
           ……                */
    // comentario en una línea.
}
```

**DIRECTIVAS DEL COMPILADOR**

Habitualmente antes de la función main() se disponen una o más directivas del compilador, mediante las cuales se indica, por ejemplo, la necesidad de incluir el código de un cierto archivo de cabecera en el código de nuestro programa a la hora de la compilación.

La directiva que más se utiliza es #include que incluye librerías que contienen funciones que se utilizan en el programa.

Por ejemplo:

```
#include <iostream.h>   //incluye la librería para manejar entradas y salidas estandar
```

**PALABRAS RESERVADAS**

Las siguientes palabras reservadas no se pueden utilizar como identificadores, ya que están reservadas para el uso exclusivo del lenguaje.

| | | | | | |
|---|---|---|---|---|---|
| auto | default | float | register | struct | volatile |
| break | do | for | return | switch | while |
| case | double | goto | short | typedef | |
| char | else | if | signed | union | |
| const | enum | int | sizeof | unsigned | |
| continue | extern | long | static | void | |

**CREACION DE APLICACIONES CON INTERFAZ DOS**

Para probar nuestras aplicaciones utilizaremos el programa turbo C++ 1.0 de Borland que es gratuito y se puede bajar desde la pagina de Borland: bdn.borland.com

El primer ejemplo es un programa sencillo que presenta un mensaje “Hola Mundo” en la pantalla. No es orientado a objetos porque no tiene clases pero sirve para que usted aprenda los pasos para crear una aplicación. Para ello debe crear un espacio de trabajo y el archivo de C++ necesarios para este programa.

El procedimiento de escritura de un programa C++ mediante Turbo C++ es sencillo y fácil. Siga estos pasos:

1. Inicie Turbo C++.
2. Escriba en la pantalla de edición el código que se describe a continuación.

### HolaMundo.cpp

El programa HolaMundo presenta HOLA MUNDO en pantalla. El listado siguiente contiene el código:

```
1: //Nombre del espacio de trabajo: Hola
2: // Nombre el programa: HolaMundo.cpp
3:
4: # include <iostream.h>
5:
6: void main ()
7: {
8:      cout << "HOLA MUNDO \n ";
9: }
```

En la ventana de HolaMundo.cpp teclee el código tal como aparece. No teclee los números de línea; son únicamente para referencia. C++ es sensible a mayúsculas y minúsculas, por lo que main no es lo mismo que MAIN, que a su vez no es igual que Main.

Para ejecutar el programa, siga estos pasos:

1. Grabe el archivo. Para ello seleccione del menú principal la opción File / Save. Luego escriba el nombre del archivo a grabar, tenga en cuenta que los archivos de código fuente de C++ tienen la extensión cpp, por esto cuando escriba el nombre del archivo debe colocar además del nombre la extensión mencionada, por ejemplo HolaMundo.cpp.
2. Compile el archivo. Para ello seleccione del menú principal la opción Compile/Make EXE File
3. Ejecute el archivo. Para ello seleccione del menú principal la opción Run/ Run

Turbo C++ compila y enlaza el programa, creando un archivo ejecutable. La ventana output reporta el éxito o fracaso de la compilación. Una compilación exitosa devuelve:

HolaMundo.exe – 0 error(s), 0 warning(s)

Si la compilación reportó errores, verifique que todas las líneas del programa se hayan tecleado exactamente como se muestra.

El programa cuando se ejecuta muestra en pantalla el texto Hola Mundo.

## ESTRUCTURA DE UN PROGRAMA EN C++

Utilizando la programación orientada a objetos en C++, los programas fuentes se estructuran de la siguiente forma:

1. Directivas del pre procesador
2. Declaración de clases
3. Implementación de los métodos de las clases, si fuera necesario
4. Función principal, main()

A continuación analizaremos la estructura un programa ejemplo, hay muchos conceptos que todavía no se han visto pero con una explicación breve podrá ver en general como tiene que codificar los programas:

```
#include <iostream.h>                    //directiva del preprocesador

class Cuadrado {
   private:
       float lado;
   public:
       Inizializa() {lado = 0; }
       void SetLado(float l)  { lado = l; }
       float GetLado() { return  lado;}
       float GetAreaCuadrado() { return (lado * lado);}
};

void main ()
{
       float n1;
       Cuadrado c1
       c1.Inicializa();
       cout << "Ingrese el valor del lado : ";     cin >>n1;
```

```
        c1.SetLado(n1);
        cout << "El área del Cuadrado es : "<< c1.GetAreaCuadrado();

    } //fin main
```

El programa anterior se almacena en el archivo cuadrado.cpp, que se llamará archivo fuente.

- La línea 1 es una instrucción #include al preprocesador de C, que incluye todas las declaraciones necesarias para utilizar las facilidades de entrada/salida, estas están contenidas en el archivo de cabecera iostream.h y dentro de las cuales se encuentra la función cout y cin que se utilizarán en el programa.
- En la línea 3 se define la clase cuadrado con la palabra class.
- En la línea 4 se especifica el acceso privado a los miembros que se listan a continuación.
- En la línea 5 se define el atributo lado.
- En la línea 6 se especifica el acceso público para los métodos que se listan a continuación.
- En la línea 7 se define el método Inicializa que asigna el valor 0 al lado.
- En la línea 8 se define el método SetLado que asigna un valor l, recibido por parámetro, al lado.
- En la línea 9 se define el método GetLado que retorna el valor del lado.
- En la línea 10 se define el método GetAreaCuadrado que calcula el área del cuadrado y retorna ese valor.
- En la línea 11 se cierra la definición de la clase.
- En la línea 13 la función main, que es la función principal, o mejor dicho la función que contiene el cuerpo del programa principal y es la función que se ejecuta primero. Debe aparecer una y solo una vez en cualquier programa en C.
- En la línea 14 y 22 se encuentran los delimitadores de bloques representados por los signos de llave y se usan para definir los limites del programa o bloques de sentencias. Los mandatos del programa van entre estas llaves.
- En la línea 15 se define una variable auxiliar n1 para cargar el lado del cuadrado.
- En la línea 16 se crea un objeto de tipo cuadrado c1.
- En la línea 17 se ejecuta el método Inicializar del objeto c1.
- En la línea 18 se pide el ingreso del lado del cuadrado usando la función cin, el cvalor se guarda en la variable n1. Es necesario aclarar que esta función está definida en otro archivo fuente y su declaración se encuentra en el archivo de cabecera iostream.h incluido en la primera línea de este programa.
- En la línea 19 se asigna el valor del lado del cuadrado invocando a la función setLado() de la clase cuadrado, el valor a asignar es el valor de n1.
- En la línea 20 se muestra el mensaje con la función cout, cuya definición se encuentra en el archivo de cabecera iostream.h incluido en la primera línea de este programa, el mensaje mostrado es el área del cuadrado y el valor del área calculado por la función GetAreaCuadrado() de la clase Cuadrado.

**TIPOS DE DATOS EN C++**

En C++ existen los siguientes tipos básicos:

- ***Entero:*** *Puede representar números enteros positivos y negativos.*

| Palabra clave | Longitud | Rango (Valores que puede asumir) |
|---|---|---|
| int | 2 byte | desde -32768 hasta 32767 |

- *Punto flotante : Puede representar números reales, tiene el punto decimal asociado a él y normalmente posee un gran número de dígitos significativos.*

| Palabra clave | Longitud | Rango (Valores que puede asumir) | |
|---|---|---|---|
| float | 4 bytes | desde 3.4E-38 | hasta 3.4E+38 |

- *Doble Punto flotante : Al igual que el tipo anterior puede representar números reales, pero con rango y precisión mayor.*

| Palabra clave | Longitud | Rango (Valores que puede asumir) | |
|---|---|---|---|
| double | 8 bytes | desde 1.7E-308 | hasta 1.7E+308 |

- *Carácter : Este tipo de dato es usado para representar caracteres, es decir el conjunto que forman todas las letras más los números del 0 al 9 y los caracteres especiales, como por ejemplo los signos de puntuación. Este está preparado para contener todos los símbolos incluidos en la tabla ASCII (American National Standard Code for Interchange of Information. Código Estándar Americano para el Intercambio de la Información), que contiene todos los caracteres que se encuentran en el teclado de nuestro ordenador y algunos otros que no se ven pero que surgen de presionar dos o tres teclas a la vez.*

| Palabra clave | Longitud | Rango (Valores que puede asumir) |
|---|---|---|
| char | 1 byte | -128 a 127 |

El tipo "char" es parecido al entero, excepto en que solo pueden asignársele números entre -128 y 127, ya que se almacenan en 1 solo byte de memoria.

- *Vacio : Este tipo de dato no existe en otros lenguajes, ocupa 0 bytes y su rango no tiene valores definidos.*

| Palabra clave | Longitud | Rango (Valores que puede asumir) |
|---|---|---|
| void | 0 | sin valores |

**MODIFICADORES DE TIPOS**

Excepto para void, los tipos de datos básicos tienen modificadores que los preceden. Se usa un modificador para alterar el significado de un tipo base. Estos modificadores son:

Signed Unsigned Long Short

La siguiente tabla muestra los tipos de datos intrínsecos de C++, junto con el rango de valores que pueden almacenar y la ocupación en memoria.

| Tipo | Largo | Rango |
|---|---|---|
| unsigned char | 8 bits | 0 to 255 |
| char | 8 bits | -128 to 127 |
| unsigned int | 16 bits | 0 to 65,535 |
| short int | 16 bits | -32,768 to 32,767 |
| int | 16 bits | -32,768 to 32,767 |
| unsigned long | 32 bits | 0 to 4,294,967,295 |
| long | 32 bits | -2,147,483,648 to 2,147,483,647 |
| float | 32 bits | 3.4 * (10**-38) to 3.4 * (10**+38) |
| double | 64 bits | 1.7 * (10**-308) to 1.7 * (10**+308) |
| long double | 80 bits | 3.4 * (10**-4932) to 1.1 * (10**+4932) |

**VARIABLES Y CONSTANTES**

**Definición**

**Vista matemática:**

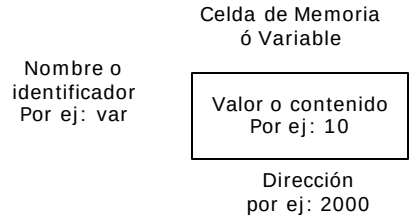
*Constante:* toma siempre el mismo valor, valores fijos que el programa no puede alterar.
*Variables:* Toman distintos valores, que pueden cambiar en el programa.

Ejemplo : longitud de la circunferencia

cf=2 pi r 2 y pi constante, r y cf variables

**Vista programación:**

Nombre simbólico que se asigna a las celdas en lugar de su dirección numérica. Es decir, una variable es una unidad de almacenamiento identificada por un nombre y un tipo de dato, tiene una dirección (ubicación) dentro de la memoria y puede almacenar información según el tipo de dato que tenga asociado. Si imaginamos la memoria del computador como un conjunto de celdas, entonces podemos decir que una variable es una de esas celdas que el usuario identifica con un nombre.




Según este ejemplo en la dirección **2000** de memoria se encuentra el identificador o variable **var** que tiene el valor **10**.

**VARIABLES EN C++**

**DECLARACION**

La declaración de una variable esta compuesta un identificador o nombre para esa variable, precedida por el tipo de dato que se puede almacenar en la misma. Los valores de las variables pueden cambiar en el programa.

En general:

Tipo Nombre o identificador;

- Los nombres o identificadores de las variables consisten en secuencias de letras y dígitos, donde el primer caracter debe ser siempre una letra.
- C establece una diferencia entre las letras mayúsculas y minúsculas.
  PRIMERA Primera primera
- Es necesario definir las variables de un programa antes de que puedan ser utilizadas.
- Ejemplos:

```
int xPos;                    short int acum;
double Precio, Cantidad;
char asterisco;
int Resultado;
```

**ASIGNACION**

La operación de asignación está compuesta de tres elementos: la variable que va a recibir el valor, el operador de asignación y el valor que se va a asignar. La variable sólo podrá ser una que haya sido declarada. El operador de asignación es el símbolo = y por último el valor asignar, que puede ser un literal otra variable, una expresión, etc., siendo la única condición que el resultado sea del mismo tipo que la variable que va a recibir el valor, o de algún tipo compatible.

En general:

```
variable = expresión
```

Por ejemplo:

```
int a;
a=10;
char c = ‘A’ ;
```

**AMBITO**

El ámbito de una variable determina la accesibilidad de la misma durante el programa. Es el período durante el cual la misma existe en un programa.

**ÁMBITO GLOBAL** cuando la misma existe durante toda la vida del programa y su contenido puede ser consultado o modificado desde cualquier parte del programa. Este tipo de ámbito no lo vamos a utilizar en nuestros ejemplos, ya que es un concepto usado en programación estructurada.

**ÁMBITO LOCAL** cuando la misma solo exista durante un breve período, generalmente asociado al inicio y fin de un bloque o a la visibilidad de una clase.

Ejemplo :

```
#include <iostream.h>                    //directiva del preprocesador

class Cuadrado {
   private:
       float lado;                 //lado es una variable local a la clase
                                   //cuadrado, tiene visibilidad en toda la clase
   public:
       Inizializa() {lado = 0; }
       void SetLado(float l)  { lado = l; }
       float GetLado() { return  lado;}
       float GetAreaCuadrado() { return (lado * lado);}
};

void main ()
{
       float n1;                   //n1 y c1 son variables locales a main, solo se
       Cuadrado c1                 // pueden usar en esta función

                                   //la variable lado definida en la clase cuadrado
                                   //no tiene visibilidad aquí, solo en la clase cuadrado.
       c1.Inicializa();
       cout << “Ingrese el valor del lado : ”;     cin >>n1;
       c1.SetLado(n1);
       cout << “El área del Cuadrado es : ”<< c1.GetAreaCuadrado();

} //fin main
```

**CONSTANTES**

Se declaran en forma similar a las variables, pero precediendo el identificador con la palabrea *const.* Las constantes pueden tener tipo, como las variables, o no tenerlo, en cuyo caso el tipo implícito es *int*. Podemos usar una constante para representar a cualquier valor que vayamos a utilizar directamente en el código, con el fin de clarificar su significado y facilitar un posterior mantenimiento. A diferencia de las variables, el valor de una constante no puede ser modificado posteriormente, es decir, en una operación de asignación el identificador que representa a una constante no puede aparecer a la izquierda.

**OPERADORES EN C++**

Un operador es un elemento del lenguaje, generalmente representado por un símbolo, cuya finalidad es generar un resultado manipulando uno o dos operandos. Si el operador trabaja sobre un solo operando se dice que es unario, mientras que si lo hace sobre dos se llama binario.

**OPERADORES ARITMETICOS**

Estos operadores son binarios, a excepción de (cambio de signo) que es unitario.

| | |
|---|---|
| + : Suma | - : Resta |
| * : Multiplicación | / : División |
| % : Resto de división entera | - : Cambio de signo |
| ++ : incremento | -- : decremento |

El orden de prioridades de estos operadores es como en las matemáticas : *, / y % tienen la prioridad mas alta.

Los operadores de incremento y decremento son unitarios y son los únicos que pueden se utilizados como pre y post fijos del operador.

Por ejemplo:

| **Operación** | **Funcionamiento** | |
|---|---|---|
| x++ | suma 1 a x equivalente a | x=x+1; |
| ++x | ídem | |

y = x++ Asigna a "y" el valor de "x" y luego suma 1 a "x"
y = ++x Suma 1 a "x" y luego asigna a "y" el nuevo valor de "x"
y = x-- Asigna a "y" el valor de "x" y luego resta 1 a "x"
y = --x Resta 1 a "x" y luego asigna a "y" el nuevo valor de "x"

Supongamos : x = 5 Al finalizar la operación

| **Operación** | **x** | **y** |
|---|---|---|
| y = x++ | 6 | 5 |
| y = ++x | 6 | 6 |
| y = x-- | 5 | 4 |
| y = --x | 4 | 4 |

**OPERADORES DE ASIGNACION**

La operación de asignación consiste en “copiar” el valor de una expresión en una variable. Cuando decimos el valor de una expresión, queremos decir que este valor puede ser el resultado de alguna operación matemática, o bien el valor de otra variable y también el resultado devuelto por una función o mandato, aunque sea definido por el usuario.

En general:

```
variable = expresión;
```

Existen, como se deja traslucir de esta regla sintáctica, tres partes en una operación de asignación, a saber: primero el receptor, que siempre es el nombre de una variable. Segundo, el operador de asignación en sí y por último la expresión de la que ya se habló con anterioridad y que es la parte transmisora del valor asignado al receptor.

El destino, o la parte izquierda, de la asignación debe ser una variable (no una función, ni una constante).

El resultado de una operación de asignación es que el valor de la expresión que se encuentra a la derecha del operador (“=“) es asignado a la variable que se encuentra a la izquierda del mismo.

Una expresión con un operador de asignación puede ser utilizada en una expresión como la siguiente:

```
valor = 5 ;                      //asigna 5 a la variable valor
valor1 = 8 * (valor2 = 5);       // primero se asigna 5 a valor2. Este se multiplica
                                 //por 8, por lo que el valor final de valor1 es 40.
```

**ASIGNACIONES MULTIPLES**

Se permite asignar a muchas variables el mismo valor utilizando asignaciones múltiples en una sola sentencia. En los programas a menudo se usa éste método para asignar valores comunes a las variables. Por ejemplo:

```
valor1 = valor2 = valor3 = 0 ;
```

**ASIGNACIONES EN LA DECLARACION DE VARIABLES O INICIALIZACIÓN**

C++ permite asignar valores a las variables en el momento en que estas están siendo declaradas. A esta operación se la denomina Inicialización y tiene el mismo sentido que cuando inicializamos una variable previamente declarada; en cualquier punto del programa. Solo que en este caso las variables toman su valor en el momento de compilación. Para llevar a cabo esta tarea, basta con declarar la variable y en esa misma línea agregar el operador de asignación y el valor que se le quiere asignar.

Por ejemplo:

```
int valor1=0;
```

Es muy importante hacer notar el hecho de que no pueden combinarse las características de asignación múltiple y de inicialización ya que este lenguaje no lo permite, y esto se basa en el simple hecho de que si estamos declarando una variable, recién después de que esta haya tomado su valor en memoria, puede tomar su valor, y la expresión de asignación múltiple nos exige, indicar el valor de la variable al final de la sentencia, y si analizamos esto, le estaríamos pidiendo al compilador que realice una tarea imposible, ya que primero debe buscar lugar en memoria para todas las variables declaradas en el orden en que se declararon, y luego debe retroceder para tomar el valor de la última y asignárselo a las demás en orden inverso al orden de declaración. Le dejamos un ejemplo para que Ud. piense y analice acerca de él, siempre haciendo hincapié en el hecho de que este ejemplo no es válido:

```
int valor1=valor2=valor3=10;
```

Además, y para finalizar, es de destacar el tema de que para definir muchas variables en una sola línea de programa, estas deben ir separadas por una “,” y no por un “=“.

**OPERADORES DE ASIGNACION COMPUESTOS**

Es un conjunto adicional de operadores de asignación que permiten expresar ciertos cálculos de modo más conciso.

| | | | |
|---|---|---|---|
| + = | Suma y Asignación | - = | Resta y Asignación |
| * = | Multiplicación y Asignación | / = | División y Asignación |
| % = | Resto Modulo y Asignación | | |

Como podemos observar la mayoría de ellos resultan de la combinación del operador "=" (asignación directa) con operadores Aritméticos.

donde la expresión: `x+=15`
es equivalente a: `x=x+15`

**OPERADORES RELACIONALES**

Sirven para obtener una expresión que no es un resultado numérico sino una afirmación o negación de una determinada relación entre dos operandos.

| | |
|---|---|
| > : Mayor que | < : Menor que |
| >= : Mayor o igual que | =< : Menor o igual que |
| == : Igual que (operador de igualdad) | |
| != : Distinto de (operador de igualdad) | |

**OPERADORES LOGICOS**

Permiten crear relaciones complejas. Se utilizan para vincular el resultado de distintas expresiones en donde intervienen operadores relacionales. Estos operadores siempre se evalúan de izquierda a derecha, deteniéndose la evaluación tan pronto como se produzca un fallo en la condición (para el and) o se cumpla (para el or). La prioridad de ejecución más alta corresponde a los operadores relacionales.

&& : y lógico (and ) || : o lógico (or) ! : negación (not)

**ESTRUCTURAS DE CONTROL EN C++**

*ESTRUCTURA SECUENCIAL :* Las acciones se ejecutan sucesivamente, incondicionalmente, en el orden en que están escritas en el algoritmo o estrategia.

```
instrucción 1;
instrucción 2;
...
instrucción n;
```

*ESTRUCTURA CONDICIONAL*

*SIMPLE :* La acción se ejecuta solamente cuando se cumple una determinada condición.

```
if (condición)
{
  acciones
}
```

*ALTERNATIVA :* Del cumplimiento de una determinada condición depende que se ejecute una acción u otra.

```
if (condición)
{
  acciones
}
else
{
  acciones
}
```

*DECISION MULTIPLE :* Cuando se presentan más de dos alternativas. De acuerdo al valor de un indicador se ejecuta una entre varias acciones.

```
swith (val)
{
        case 1 :
                acciones
                break ;
        case 2 :
                acciones
                break ;
        default :
                acciones
                break ;
}
```

*ESTRUCTURAS REPETITIVAS :* Una instrucción o conjunto de instrucciones se repite una cierta cantidad de veces.

*do .... while :* La iteración se ejecuta mientras se cumpla la condición establecida.

```
do
{
  acciones
} while (condición) ;
```

*while ... do:* Mientras se cumpla una determinada condición se repetirá el proceso.

```
while (condición)
{
  acciones
}
```

*for :* Dada una o más variables de control, que se inicializan al comienzo del ciclo, el mismo se ejecutará según una o más condiciones, y por cada vuelta de ciclo se incrementaran las variables de control.

```
for (inicialización ; condición ; incremento)
{
  acciones
}
```

**Sentencias especiales**

**break :** Rompe la correcta ejecución (paso a paso) de una estructura, obligando al programa a continuar con la primera instrucción que encuentre fuera de la estructura en la que se encontraba.

**Continue :** Esta sentencia funciona de manera similar a la sentencia break. Sin embargo, en vez de forzar a la terminación del bucle, continue fuerza la siguiente iteración y salta cualquier código entre medio.

**NOTA :** Cuando dentro de las estructuras se encuentra un sola instrucción o acción, no es necesario encerrarla entre llaves, excepto en la sentencia swith y do... while.

**ENTRADAS Y SALIDAS BASICAS EN C++**

- C++ proporciona la biblioteca **iostream**, implementada en POO, de funciones que realizan operaciones de E/S (Entrada/Salida).
- Para acceder a dicha biblioteca se debe incluir en cada programa el archivo de cabecera *iostream.h*, mediante la directiva del preprocesador #include:

```
#include <iostream>
```

- Los objetos de E/S son:

  *cin*: Dispositivo estándar de entrada, el dispositivo por defecto es el teclado.

```
cin  >> var;          //lee desde teclado un valor y lo guarda en la variable var.
cin  >> a >> b >>c;   // lee tres valores del teclado.
```

  *cout*: Dispositivo estándar de salida, el dispositivo por defecto es la pantalla.

```
cout << "mensaje";          //muestra una cadena de caracteres por pantalla.
cout << "mensaje"<< var;    //muestra una cadena de caracteres y el contenido
                            //de una variable por pantalla.
cout << var;                //muestra el contenido de una variable por pantalla.
```

- Manipuladores de salida:
  Endl ó "\n": Envía caracteres de fin de línea y limpia el buffer.
  `cout << endl;` ó `cout << "\n";`
- Lectura de línea de datos: Se utiliza la función getline() cuando se desea introducir una cadena de datos en un array de caracteres.

```
char nombre[15];
cin.getline(nombre,15);
cout << "\n" << nombre << "\n"
```

# CLASES EN C++

***INTRODUCCION***

Una clase es simplemente un modelo que se utiliza para describir uno o más objetos el mismo tipo.

Las clases soportan el conceptos de encapsulamiento de datos, que se produce cuando la representación interna de un objeto junto con sus operaciones, se encierran en la misma estructura (la clase).

Una clase es un marco que permite crear objetos de su misma estructura. Estos objetos, que pueden ser elementos de duración estática, local o dinámica, constan de:

- Variables de clase y de instancia (datos miembro en C++), que son los descriptores de los atributos de los objetos.
- Métodos (funciones miembro en C++), que definen las operaciones que pueden realizar esos objetos.

Un objeto se instancia a partir de la descripción de una clase: un objeto es dinámico, una clase puede tener una infinidad de objetos descritos por ella. Un objeto es un conjunto de datos presente en memoria.

Así, si define una clase Ventana, ésta podrá instanciarse tantas veces como lo desee. Pero todos los objetos creados tendrán la misma descripción y el mismo comportamiento.

¿Qué significa esto? La descripción de un objeto viene dada por sus atributos; en el caso de una Ventana, los atributos serán, por ejemplo, su título, sus coordenadas, su altura y su anchura:

```
class Ventana {
        String titulo;
        int coordenadax;
        int coordenadaY;
        int altura;
        int anchura;
}
```

Cada Ventana así creada tendrá sus propios atributos, pero todas estas Ventana compartirán esta descripción. Observemos que los atributos de un objeto pueden ser tipos simples (enteros, booleanos, etc.) o bien otros objetos como aquí la cadena de caracteres que contiene el nombre de la ventana.

En C++, se pueden crear los objetos en forma estática o dinámica, como cualquier variable.

```
Ventana ven1;                   //creación estática
Ventana *ven2;
ven2 = new Ventana;             //creación dinámica
...
delete Ventana;
```

**DESCUBRIMIENTO DE LAS CLASES**

Una de las primeras decisiones que se debe tomar al crear una aplicación orientada a objetos es la selección de clases. Las clases en la POO pueden tener tipos diferentes de responsabilidades, en UML (lenguaje unificado de modelo) existen tres estereotipos y se utilizan para ayudar a los desarrolladores a distinguir el ámbito de las diferentes clases. Ellas son:

- Clases **de entidad**. Las clases de entidad se utilizan para modelar información que posee una vida larga y que es a menudo persistente. Son clases cuya responsabilidad principal es mantener información de datos o de estado. Con frecuencia se reconocen como los sustantivos en la descripción de un problema y generalmente son los bloques de construcción fundamentales de un diseño. Estas clases modelan la

información y el comportamiento asociado a algún concepto, como una persona, un objeto del mudo real.

- Clases **de interfaz**. Las clases de interfaz se utilizan para modelar la interacción entre el sistema y sus actores (es decir usuarios y sistemas externos). Esta interacción a menudo implica recibir (y presentar) información y peticiones de (y hacia) los usuarios y los sistemas externos. Por ejemplo, una parte esencial de la mayoría de las aplicaciones es la presentación de la información en un dispositivo de salida, como la pantalla de un terminal. Debido a que el código para llevar a cabo dicha actividad a menudo es complejo, modificado con frecuencia y muy independiente de los datos reales que se están mostrando, es una buena práctica de programación aislar el comportamiento de presentación en clases separadas de aquellas que guardan los datos mostrados. Por ejemplo, en una abstracción de un juego de cartas crearemos una clase Carta que contendrá los datos la misma y una clase VistaDeCatrtas, para encargarse de dibujar la imagen de una carta en la pantalla. Con frecuencia, los datos de base (por ejemplo la clase Carta, que es una clase entidad) son llamados el *modelo*, mientras que la clase que se exhibe (VistaDeCatras que es una clase de interfaz) es llamada la *vista*.
  Dado que separamos el modelo de la vista, por lo general se simplifica mucho el diseño del modelo. Idealmente, el modelo no debe requerir ni contener ninguna información acerca de la vista. Ello facilita la reutilización del código, ya que el modelo puede usarse en aplicaciones diferentes. No es raro para un modelo único tener más de una vista. Por ejemplo la información financiera podría presentarse tanto en gráficas de barras o gráficas de pastel, como en tablas o figuras, sin cambiar el modelo subyacente.
  En ocasiones es inevitable la interacción entre un modelo y una vista. Por ejemplo, si a las cifras de la tabla financiera que se acaba de describir se les permite cambiar dinámicamente, el programador puede desear que la presentación se actualice en forma instantánea. Entonces es necesario que el modelo avise a la presentación que ha sido cambiado y que ésta se debe actualizar. Algunos programadores se refieren a tal modelo como a un *sujeto* para distinguirlo del modelo que no tiene conocimiento del uso.
- Clases **de control**. Las clases de control representan coordinación, secuencia, transacciones, y control de otros objetos. Los aspectos dinámicos del sistema se modelan con clases de control, debido a que ellas manejan y coordinan las acciones y los flujos de control principales, y delegan trabajo a otros objetos (es decir, objetos de interfaz y de entidad).

Si parece que una clase abarca dos o más de estas categorías, con frecuencia podrá dividirse en dos o más clases. Por ejemplo, el primer diseño de la abstracción del juego de cartas tiene una sola clase, llamada carta. Luego esta se fracciona en la clase de datos y en la clase de vista.

**EJEMPLO: UN JUEGO DE CARTAS**

Usaremos una abstracción de software con una carta de un juego de naipes común. La Carta sabe poco de la intención de su uso y puede incorporarse en cualquier tipo de juegos de cartas. El comportamiento de una carta sería el siguiente:

*Mantener palo y rango*
*Devolver color*

Las responsabilidades de la clase Carta son muy limitadas; básicamente, una carta es tan sólo un manejador de datos que mantiene y devuelve valores de rango y de palo. En particular la clase básica Carta no tiene la capacidad para mostrarse a sí misma.

Se crea una clase de vista llamada VistaDeCarta para manejar las tareas de relacionadas con su visualización, el comportamiento de esta clase sería la siguiente:

*Dibujar carta en la superficie de juego*
*Borrar imagen de carta*
*Mantener estado de bocaarriba o bocaabajo*
*Mantener ubicación en la superficie de juego*
*Moverse a la nueva ubicación en la superficie de juego*

Si separamos la visualización de la carta de los datos de la carta misma, aislamos el dispositivo principal y las dependencias ambientales de las estructuras menos dependientes. Por ejemplo, la capacidad para dibujar la carta dependerá mucho de la interfaz que elijamos, si cambiamos a un sistema diferente, sólo se necesitará cambiar la clase de vista.

No siempre queda clara la decisión en cuanto a qué clase pertenece un comportamiento. Tras haber separado las tareas manejadoras de datos de las de visualización, resulta fácil decidir que el método color pertenece ala clase Carta y que el método mostrar debe ir en la clase VistaDeCarta, pero, ¿qué pasa con los métodos volver (voltear) y bocaarriba? ¿El estado bocaarriba o bocaabajo de una cata es una propiedad intrínseca de la carta misma o meramente una propiedad que dice cómo se muestra la carta? Se podría defender el punto anterior en ambos sentidos, pero hemos decidido colocar este comportamiento en la clase VistaDeCatra. (Un punto que hay que considerar es si existen usos para la clase Carta que no requieran conocer el estado de bocaarriba. Si es así, entonces no debe asignarse a la abstracción de Carta el mantenimiento de tal información).

En el siguiente paso es refinar nuestras clases, definiendo atributo y métodos, como se muestra a continuación:

**Carta**
**atributos**

palo - *(entero) valor del palo*
rango - *(entero) valor del rango*

**métodos**

FijarPaloYRango - *Fijar palo y rango de carta*
Palo - *devolver palo de carta*
Rango - *devolver rango de carta*
Color - *devolver color de carta*

**VistaDeCarta**

**atributos**

laCarta – *(tipo Carta) el valor de la carta (es un objeto de tipo Carta)*
bocaArriba – *(booleano) estado de bocaarriba o bocaabajo*
ubicx, ubicy – *ubicación en superficie de juego*

**métodos**

Borrar, Dibujar – *borrar o dibujar imagen de carta*
BocaArriba, Volver – *probar o voltear carta*
Incluye (entero, entero) – *probar si el punto está dentro del límite*
MoverA(entero, entero) – *mover carta a nueva ubicación*

El siguiente paso es traducir a código ejecutable el comportamiento y el estado descritos de las clases. Este código puede ser escrito en cualquier lenguaje de programación que soporte POO y que en nuestro caso usaremos el lenguaje C++.

Por ejemplo, la declaración de clases en C++, sería la siguiente:

```
class Carta {
        int palo;
        int rango;
public:
        void FijarPaloYRango(int p, int r) {palo = p, rango = r;}
        int Palo() {return palo;}
        int Rango() {return rango;}
        int Color() {return Palo() % 2;}
};

class VistaDeCarta {
        Carta * laCarta;
        int bocaArriba;
        int ubicX;
        int ubicY;
public:
        Carta * VerCarta();
        void Dibujar();
        void Borrar();
        int BocaArriba();
        void Volver();
        int Incluye(int, int);
        int X();
        int Y();
        void moverA(int, int);
};
```

***ESTRUCTURA GENERAL***

Una clase contiene elementos, llamados miembros, que pueden ser datos, llamados *atributos,* y funciones que manipulan esos datos llamados *métodos*.

Una clase se puede definir con una estructura (struct), una unión (union) o una clase (class). En el tipo class todos los miembros son por defecto privados. Las estructura y uniones son clases en las que todos sus miembros son por defecto public, a no ser que se especifique lo contrario utilizando los especificadores de acceso.

Una clase se define con la palabra reservada class.

La sintaxis de una clase es:

```
class nombre_de_la_clase [: [public | protected | private] ClaseMadre]
{
        [Lista de atributos]
```

```
        [Listado de Métodos]
};
```

Todo lo que está entre [y] es opcional. Como se ve, lo único obligatorio es class y el nombre de la clase.

Donde ClaseMadre indica de qué clase desciende la nuestra y public, protected ó private de que forma se heredarán los miembros. Cuando una clase desciende de otra, significa que hereda sus atributos y sus métodos. Esto quiere decir que, a menos que los redefinamos, sus métodos serán los mismos que los de la clase madre y podrán utilizarse en forma transparente, siempre y cuando no sean *privados* en la clase madre, como veremos más adelante.

### *DECLARACION Y DEFINICION*

La *declaración* lista los miembros de la clase. La definición, también llamada *implementación*, define las funciones de la clase.

La declaración y la definición de la clase pueden ir juntas o separadas.

En el primer caso se declaran y definen los atributos y las funciones miembros dentro de la clase, en el segundo caso, se declaran los atributos y los prototipos de las funciones miembros dentro de la clase y por separado se definen las funciones. Este último caso es el más recomendado ya que organiza mejor las aplicaciones y es el que vamos a aplicar.

Se puede realizar la declaración y definición en el mismo archivo (.cpp) o realizar la declaración en los archivos de cabecera (.h) y la definición en los archivos de definición (.cpp). Un ejemplo de declaración y definición separadas en el mismo archivo es el siguiente:

```
//archivo ejemplo1.cpp

//Declaración de la clase
class Contador {
        int cnt;
public:
        Contador();
        int incCuenta();    //prototipos
        int getCuenta();
};

// Definición o Implementación de la clase
void Contador :: Contador() {
        cnt=0;       //inicializa
}
int Contador :: incCuenta() {
        cnt++;
        return cnt;
}
int Contador :: getCuenta() {
        return cnt;
}
```

:: operador de resolución de ámbito, identifica la clase a la que pertenece la función.

### *EL CUERPO DE LA CLASE*

El cuerpo de la clase, encerrado entre { y }, es la lista de atributos (variables) y métodos (funciones) que la constituyen. No es obligatorio, pero en general se listan primero los atributos y luego los métodos.

### *INSTANCIAS DE UNA CLASE (OBJETOS)*

Los objetos de una clase son instancias de la misma y se crean en tiempo de ejecución con la estructura definida en la clase.

Como los objetos o instancias son variables se pueden crear en forma estática o dinámica:

Forma estática:

```
nombre_clase  objeto1       //objeto1 es una instancia de la clase tipo
                            nombre_clase
```

Forma dinámica:

```
nombre_clase  *objeto2            //objeto2 es un puntero de tipo
```

```
                                        //nombre_clase
        objeto2 = new nombre_clase      //objeto2 es una instancia de la clase
                                        //tipo nombre_clase
```

Por ejemplo, dada la clase cliente:

```
class Cliente {
      int codigo;
      float ventas;
public:
      int   getCodigo()   { return codigo; }
      float getVentas()   { return ventas; }
};
```

el objeto o instancia de la clase cliente es:

```
//objeto o variable estática
Cliente comprador1; // comprador1 es un objeto, una variable de la clase
                    //y una instancia de la clase cliente.

//objeto o variable dinámica
Cliente *comprador2 = new Cliente;...
delete comprador2;
```

En este ejemplo, comprador1 y comprador2 son variables (instancias u objetos) de tipo clase "Cliente" que se compone de dos campos de datos y dos funciones de acceso. Unicamente las funciones públicas están a disposición de los usuarios de la clase.

**Nota:** recuerde que en C++ una variable creada con new debe ser eliminada con delete.

**Acceso a los miembros**

Desde un objeto se puede acceder a los miembros mediante la siguiente sintaxis

```
objeto.miembro;
```

Por ejemplo, podemos acceder al método setImporte, para cambiar el importe de la deuda de un objeto cliente.

```
comprador.setImporte(100);
```

Si el cliente comprador, por ejemplo, tenía inicialmente un importe de 0, mediante esta sentencia se lo cambiamos a 100.

Desde un objeto llamamos a las funciones miembro para realizar una determinada tarea. Por ejemplo, desde el cliente comprador llamamos a la función getImporte() para mostrar el importe de dicho cliente.

```
comprador.getImporte();
```

La función miembro getImporte() devuelve un número, que guardaremos en una variable adeuda, para luego usar este dato, por ejemplo:

```
float adeuda=comprador.getImporte();
cout << "El importe adeudado es " << adeuda);
```

Como veremos mas adelante, no siempre es posible acceder a los miembros, el acceso dependerá de los controles de acceso a los mismos.

**La vida de un objeto**

El objeto se destruirá dependiendo de cómo fue creado, si el objeto fue creado en forma estática, este se destruirá al finalizar su ámbito, si el objeto fue creado en forma dinámica se tendrá que eliminar con la orden delete, al final de su uso. Cuando se utiliza asignación dinámica la orden new reserva espacio en memoria para el objeto y delete libera dicha memoria.

***DECLARACION DE MIEMBROS UNA CLASE***

Los datos de una clase se denominan **atributos** y las funciones de una clase se denominan **métodos**.

Los miembros tienen ciertas restricciones en el modo en que se puede manipular los mismos dentro y fuera de la clase, a esto se le llama control de acceso a una clase o visibilidad.

Con estas restricciones se logra la encapsulación que, como vimos en la introducción, consiste en separar los aspectos externos del objeto, a los cuales pueden acceder otros objetos, de los detalles de implementación del mismo, que quedan ocultos para los demás. La encapsulación se basa en la noción de servicios prestados; Una clase proporciona un cierto número de servicios y los usuarios de esta clase no tienen que conocer la forma como se prestan estos servicios. Contrariamente a las cuentas de una asociación caritativa, una clase debe mantenerse opaca.

Hay que distinguir pues en la descripción de la clase dos partes:
la parte pública, accesible por las otras clases;

la parte privada, accesible únicamente por los métodos de la clase.

Se recomienda encarecidamente poner los atributos de una clase en la parte privada, para respetar el concepto de encapsulamiento.

**Modificadores de acceso a miembros de clases**

Una clase puede contener partes *públicas* y partes *privadas*. Por defecto, todos los miembros definidos en la clase son privados. Para hacer las partes de una clase públicas (esto es, accesible desde cualquier parte de su programa) deben declararse después de la palabra reservada **public**. Todas las variables o funciones definidas después de public son accesibles a las restantes funciones del programa. Esencialmente, el resto de su programa accede a un objeto a través de sus funciones y datos públicos. Dado que una característica clave de la POO es la ocultación de datos, debe tener presente que aunque puede tener variables públicas, desde un punto de vista conceptual debe tratar de limitar o eliminar su uso. En su lugar, debe hacer todos los datos privados y controlar el acceso a ellos, a través de funciones públicas. El mecanismo para hacer privados datos o funciones es anteponerle la palabra reservada private.

Por defecto, una clase es privada, aunque es conveniente especificar la visibilidad expresamente, por legibilidad y por compatibilidad con versiones posteriores que pueden recomendar su uso obligatorio.

Existen tres clases de usuarios de una clase:

La propia clase.

Clases hijas (clases derivadas de una clase madre, este concepto se explica en detalle más adelante, pero retenga que las clases Hijas se parecen mucho a la clase Madre.

Clases externas (clases que pueden interactuar o no con la clase de estudio).

Cada clase tiene diferentes privilegios o niveles de acceso. Cada nivel de privilegio de acceso se asocia con una palabra reservada **private**, **public** o **protected.**

*private*

Por defecto todo lo declarado dentro de una clase es privado y sólo puede ser accedido por las funciones miembro declaradas en el interior de la clase o por funciones amigas.

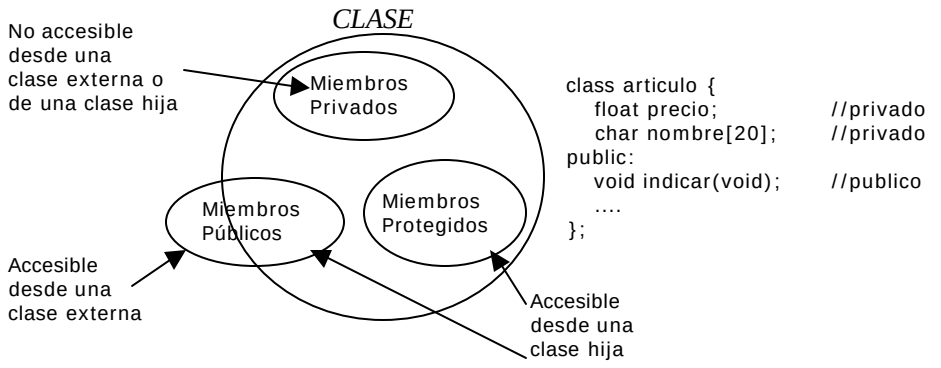
*public*

Los miembros que se declaran en la región pública pueden ser accedidos por cualquier función miembro y no miembro de esa clase. Estos son los únicos miembros que tienen permiso para ser accedidos por los usuarios de la clase. Se suele decir que los miembros públicos conforman la interfaz de la clase con el usuario.

*protected*

Los miembros que se declaran en la región protegida pueden ser accedidos por funciones miembros declaradas dentro de la clase, por funciones amigas o por funciones miembro de clases derivadas de esta clase.

**Visibilidad de una clase**




## Separación de la interfaz

¿Cuándo debe utilizarse qué? O en otras palabras: ¿cuáles son los diferentes casos de utilización de los mecanismos de protección? Nos proponemos distinguir principalmente dos casos:

Los miembros que pertenecen a la *interfaz* de la clase: debe ser public;

Los miembros que pertenecen al *cuerpo* de la clase: deben ser protegidos. Esta protección es diferente según los casos; en general, la protección más fuerte es aconsejable porque es fácil desproteger un atributo, y es mucho más difícil hacerlo inaccesible si ya se utiliza.

La interfaz de la que hablamos es pues la interfaz conceptual de la clase, es decir los atributos y las signaturas de los métodos (tipo devuelto + nombre + parámetros), directamente utilizados desde el exterior porque corresponden a un servicio prestado.

El cuerpo de la clase es la implementación de dicho servicio.

Es decir, la interfaz de la clase es el *qué* -qué hace la clase-, mientras que su cuerpo es el *cómo* -cómo lo hace.

Por esto conviene hacer pública la interfaz y proteger el cuerpo.

Ahora, veamos un ejemplo en pseudocódigo sobre cómo crear un objeto reloj que demuestre cuáles serían sus funciones públicas y sus funciones privadas:

**Función inicial o Constructor:**
Reloj negro, hora inicial 12:00am;
**Funciones Públicas:**
Apagar
Encender
Poner despertador;
**Funciones Privadas:**
Mecanismo interno de control
Mecanismo interno de baterías
Mecanismo de manecillas

Al utilizar uno de estos relojes nos importa su operación no su mecanismo interno, por eso existen funciones públicas o privadas. Las funciones públicas son la interfaz que usaremos. El constructor inicializa al objeto en un estado inicial estable para su operación.

Un ejemplo más, esta vez con una computadora:

**Función inicial o Constructor**:
Computadora portátil compaq, sistema operativo windows98, encendida
**Funciones Públicas:**
Apagado
Teclado
Pantalla
Impresora
Bocinas
**Funciones Privadas:**
Caché del sistema
Procesador
Dispositivo de Almacenamiento
Motherboard

Obviamente si se abre (físicamente) la computadora se tendrá acceso a todo, sucede lo mismo si se abre el código fuente del archivo de una clase, se puede modificar todo, pero debe quedar bien claro que la razón por la cual se hace todo lo anterior es para organizar mejor el código, no para impedir el acceso a nadie a ciertas cosas, todo es para mejorar la operación general de una clase ocultando la información que no es necesario que conozcamos y exponiendo la que sí.

### *ATRIBUTOS DE UNA CLASE*

Todas las variables se declaran dentro del cuerpo de la clase o dentro de un método. Las variables declaradas dentro de un método son locales a él; las variables declaradas en el cuerpo de la clase se dice que son *miembros* de ella y son accesibles por todos los métodos de la clase.

Por otra parte, además de los atributos de la propia clase, se puede acceder a todos los atributos de la clase de la cual desciende. Por ejemplo, cualquier clase que descienda de la clase **Polygon** hereda los atributos **npoints, xpoints** e **ypoints**. Esto lo veremos con mayor detalle cuando estudiemos herencia.

Finalmente, los atributos miembros de la clase pueden ser *atributos de clase* o *atributos de instancia;* se dice que son atributos *de clase* si se usa la palabra clave static: en ese caso la variable es única para todas las instancias (objetos) de la clase (ocupa un único lugar en memoria). Si no se usa static, el sistema crea un lugar nuevo para esa variable con cada instancia (o sea que es independiente para cada objeto).

Los atributos pueden ser:

tipos básicos. (no son clases)
objetos de otras clases (clases propias, de terceros, etc.)
estructuras de datos.

Para definir cada atributo se tiene que: definir primero el tipo y luego el nombre del atributo. Por ejemplo:

```
int n_entero;
float n_real;
char *p;
```

Un dato miembro de una clase no puede ser inicializado. Esto es, no sería correcto realizar la siguiente declaración como miembro de una clase:

```
int n_entero = 0;
```

La lista de miembros de una clase puede comprender cualquier tipo válido en C++. Puede contener:

Tipos primarios:

```
class primario {
        int a;              // entero
        char b ;            //carácter
        float c ;           //real
        double d ;          //real doble precisión
};
```

Punteros a cualquier tipo válido:

```
class estructura {
        int *ptr1;
};
```

Objetos, punteros a objetos y referencias a objetos. Para declarar un objeto de una clase como miembro de otra clase, es necesario que aquella haya sido previamente definida, por ejemplo:

```
class clase1 {
        int val;
        ...
};

class clase2 {
        clase1 objeto;                  //objeto de la clase1
        clase1 *ptr_objeto;       //puntero al objeto de la clase1
        ...
};
```

Un objeto de una clase no puede ser miembro de ella misma, pero si puede serlo un puntero a dicho objeto, por ejemplo:

```
class clase {
        clase *ptr_objeto;  //puntero al objeto de la clase
        ...
};
```

**Miembros estáticos de una clase**

Le hemos explicado anteriormente que cada objeto poseía sus propios atributos. Es posible que todos los objetos de una misma clase tengan atributos en común: son los atributos de clase, introducidos por la palabra clave static. Estos atributos son legibles y modificables por todos los objetos de una misma clase. La modificación de un atributo static es tenida en cuenta inmediatamente por los otros objetos, porque lo *comparten.*

Un miembro de una clase se puede declarar estático (static). Para un miembro dato, la designación static significa que existe sólo una instancia de ese miembro. Un miembro dato estático es compartido por todos los objetos de una clase y existe incluso si ningún objeto de esta clase existe siendo su valor común a la clase completa.

A un miembro dato static se le asigna una zona fija de almacenamiento en tiempo de compilación, al igual que una variable global, pero el identificador de la variable está dentro del ámbito utilizando solamente el operador de resolución con el nombre de la clase.

Ejemplo:

```
Include “iostream.h”
class Alumno {
        static int conteo_legajo = 0;
        int legajo;
public:
        int Conteo_Legajo () { return conteo_Legajo; }
```

```
                int Legajo () { return legajo; }
                void Inicializa () {
                        conteo_legajo = conteo_legajo + 1;
                        legajo = conteo_legajo;
                }
        };

        void main () {
                Alumno a1, a2, a3;
                al.Inicializa ();
                a2.Inicializa ();
                a3.Inicializa ();
                cout << “Alumno l: ” << al.Legajo() << ” ” << al.Conteo_Legajo();
                cout << “Alumno 2: ” << a2.Legajo() << ” ” << a2.Conteo_Legajo();
                cout << “Alumno 3: ” << a3.Legajo() << ” ” << a3.Conteo_Legajo();
        }
```

dará como resultado:

```
        Alumno l: 1 3
        Alumno 2: 2 3
        Alumno 3: 3 3
```

La variable conteo_alumno lleva el valor del último legajo asignado, sirve como un valor de autoincremento de la clase que se va asignando al legajo de cada objeto.

## *CLASE PRACTICA*

Ejercicios con entidades básicas, ponerse de acuerdo como usar los main

Se sugiere que las clases de datos no tengan interfaz de usuario y que la clase de interfaz en lo posible sea una clase, por ejemplo, aplicación, así en el main solo se crea un objeto de este tipo y se llama al método que comienza la ejecución.

**Por ejemplo**

4to enunciado del primer práctico para el "contracurso" 1k7, año 2004.

**Clase Figura**
**Atributos : lado del cuadrado, radio del círculo, área del cuadrado, área del círculo.**
**Métodos : Inicialización (constructor), carga de los atributos (lado y radio), calcular el área del cuadrado, mostrar el área del cuadrado, calcular el área del círculo, mostrar el área del círculo.**

```
Class Figura {
   private:
       float lado;
       float radio;
public:
       Figura() {lado = 0; radio = 0; area = 0;}
       void SetLado(float lado)  { this.lado = lado; }
       void SetRadio(float radio)  { this.radio = radio; }
float GetLado() { return  lado;}
float GetRadio() { return radio;}
float GetAreaCuadrado() { return (lado * lado);}
       float GetAreaCirculo()  { return ( 3.14 * radio * radio); }
};


class Aplicacion    //se usa en lugar del main
{
     public:
       void Comenzar()
       {
              float n1, n2;
              Figura fig;

              cout << “Ingrese el primer valor : ”;     cin >>n1;
              cout << “Ingrese el segundo valor :”;   cin >>n2;
              if (n1 < n2)
              {
                     fig.SetLado(n1);
                     fig.SetRadio(n2);
              }
              else
              {
                     fig.SetLado(n2);
                     fig.SetRadio(n1);
              }
              cout << “El área del Cuadrado con lado ” << fig.GetLado()
              <<” es : ”<< fig.GetAreaCuadrado();
              cout << “El área del Círculo con radio” << fig.GetRadio()
              <<” es : ”<< fig.GetAreaCirculo();
       }
};

void main()
{
```

```
                Aplicacion prog;

                Prog.Comenzar();
        }
```

***/ /*  **Mínimo y Máximo en una secuencia de números**

Otro ejemplo. Ahora se trata de procesar una secuencia de números enteros. Vienen en cualquier orden; el número 999 es fin de datos. Se pide determinar y mostrar el mínimo y máximo de esa secuencia.

El concepto que estamos aplicando es el de valores mínimo y máximo de una serie de números. Por eso, un nombre adecuado a la clase que implementará ese concepto puede ser MiniMax. Cual puede ser la estrategia para determinar estos extremos ? Puede haber varias, exponemos una.

En el caso del mínimo, necesitamos una variable que almacene el número mas pequeño de la secuencia. Dentro del ciclo, cada vez que aparece uno menor al que el que ya teníamos, lo reemplazamos. Pero cual es el número de partida para la comparación con el que acabamos de leer ? Un buen criterio es almacenar como mínimo la primera lectura. Si aparece otro menor, lo reemplazaremos.

Un criterio semejante vale para la determinación del máximo.

Bueno, tenemos un ciclo de lectura. El ciclo debe mantenerse mientras no sea fin de datos (numero = 999). Esto significa que tenemos una primera lectura antes del ciclo, en momentos de inicio del proceso, o sea en tiempo de actuar el constructor. Es en ese momento que guardamos el primer valor leído como minimo y maximo de partida. En conclusión, el comportamiento del objeto MiniMax debe comprender:

- Su construcción (MiniMax()). Primera lectura, almacenamiento.
- Su procesamiento dentro del ciclo, verificando existencia de nuevo mínimo o máximo. El ciclo lo dejamos en el main, (decisión arbitraria, podría ser parte del comportamiento de MiniMax )
- Exhibición del objeto. Llamamos toString() a esta función miembro. (Así en Java)

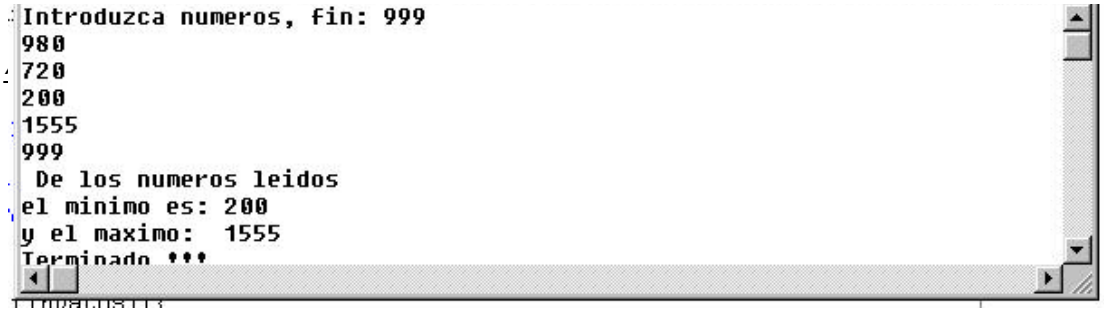
```
# include <iostream.h>
int numero;             //  Global
class MiniMax{
private:
        int minimo, maximo;
public:
        MiniMax();              // Constructor
        void procNum();         // Procesa el número
        void toString();        // Visualizando el objeto
};

MiniMax::MiniMax(){       // Constructor
        cout << "Introduzca numeros, fin: 999 \n";
        cin >> numero;
        minimo= numero;
        maximo= numero;
}

void MiniMax::procNum(){
                if(numero < minimo) minimo = numero;
                else maximo = numero;
}

void MiniMax::toString(){
        cout << " De los numeros leidos \n";
        cout << "el minimo es: "<< minimo << "\n";
        cout << "y el maximo:  "<< maximo << " \n";
        cout << "Terminado !!!\n";
}

void main(){
        MiniMax num;            // Construimos el objeto num de la clase MiniMax
        while(numero!=999){          // Mientras no sea fin de datos
```




```
                num.procNum();          // Controlamos, contabilizamos
                cin >> numero;          // Leo siguiente caracter
        };
        num.toString();                 // Visualizamos el objeto
};
```

**// Pares, Impares en una secuencia de numeros**

```
# include <iostream.h>
int numero;
class Paridad{
private:
        int pares, impares;
public:
        Paridad();              // Constructor
        void procNum();                 // Procesa el número
        void toString();        // Visualizando el objeto
};

Paridad::Paridad(){     // Constructor
        pares=impares=0;
        cout << "Introduzca numeros, fin: 999 \n";
        cin >> numero;
        }

void Paridad::procNum(){
                int resto = numero % 2;
                if(resto) impares++; // hay resto, es impar ...
                else pares++;
}

void Paridad::toString(){
        cout << " De los numeros leidos \n";
        cout << "Tenemos, pares: "<< pares << " \n";
        cout << " e impares    : "<< impares << " \n";
        cout << "Terminado !!!\n";
}

void main(){
        Paridad num;                    // Construimos el objeto num de la clase Paridad
        while(numero!=999){                     // Mientras no sea fin de datos
                num.procNum();          // Controlamos, contabilizamos
                cin >> numero;;         // Leo siguiente numero
        };
        num.toString();                 // Visualizamos el objeto
};
```


(Inactive G:\BORLAND\BC45\EJEMPLOS\AED2004\UNIDAD1\PARIDAD\PARIDAD.EXE)
Introduzca numeros, fin: 999
123 444 555 654 987 654 999
De los numeros leidos
Tenemos, pares: 3
e impares : 3
Terminado !!!