**UNIDAD 5**

## Clases en C++

1.- **Características del C++.**

1.a) Prototipos de funciones.

Cuando usamos el C++, nos exije que declaremos los prototipos de las funciones usadas en el programa, en el caso de las funciones de la librería, es necesario incluir el archivo de cabecera, en el cual se encuentran la declaración de sus prototipos.

En un programa en C++ podemos mezclar archivos de cabecera de "C" y de C++.

```
// Programa # 12, Uso de archivos de cabecera y prototipos.

#include <stdio.h>       /* Mezclando archivos de cabecera */
#include <iostream.h>
float area_cir(int x);   /* Prototipo de area_cir */

void main()
{
    int a;

    cout << "Dame el valor de a: ";
    cin >> a;
    printf("\nArea del círculo = %8.2f\n", area_cir(a));
}


float area_cir(int x)
{
    return 3.14159*x*x;
}
```

OBSERVACIONES.

En el programa anterior tratamos de ejemplificar el uso de un prototipo de función de una función declarada en el mismo archivo, area_cir(), así como la inclusión de 2 archivos de cabecera; <stdio.h>, para la declaración del prototipo de la función printf() y <iostream.h>, para la definición de los objetos, **cout** y **cin**.

1.b) Declaraciones.

En C++ son permitidas las declaraciones de variables dentro de bloques y después de enunciados; Esto le permite al programador declarar una variable lo más cerca de su punto de aplicación.

También es permitido declarar un indice dentro de un lazo.

```
// Programa # 13 Manejo de declaraciones en C++
#include <stdio.h>
#define TAMC 5

main()
{
     int calif[TAMC], suma=0;

     printf("\nIngrese %d calificaciones\n", TAMC);
     for(int k =0; k < TAMC; ++k) {
        printf("Dame calif[%d]: ", k);
        scanf("%d", calif+k);
        suma += calif[k];
     }

     /* ordenamiento burbuja */

     for(int i=0; i<TAMC-1; ++i)
        for(int j=TAMC-1; i < j; --j)
            if(calif[j-1] < calif[j]) {
                    int temp;
                    temp = calif[j-1];
                    calif[j-1] = calif[j];
                    calif[j] = temp;
            }

     printf("\n\nLas calificaciones ordenadas son :\n");
     for(i=0; i < TAMC; ++i)
          printf("\ncalif[%d] = %d", i, calif[i]);

     printf("\n\n%23d suma de calificaciones ", suma);
     float prom;
     prom = (float) suma/TAMC;
     printf("\n%26.2f es el promedio", prom);
}
```

OBSERVACIONES.

En este programa ejemplificamos el uso que hace C++ de las declaraciones, vemos en primer lugar las declaraciones hechas en el lazo **for** de; k, i y j. Una vez

declarada **i** es válida en su ámbito de declaración y no es necesario declararla en el **for** siguiente.

Por otro lado, la variable **temp**, es declarada dentro de un bloque y solo es válida dentro de dicho bloque. Esto sí es válido en "C".

Por último la variable **prom**, es declarada inmediatamente antes de su utilización.

1.c) Operador de Alcance.

El operador de alcance o de resolución de ámbito (**::**) nos permite resolver conflictos de nombre. Puede servir por ejemplo para que en una función determinada donde se declara una variable automática 'x', se pueda también tener acceso a una variable global 'x'.

```
// Programa # 14, uso del operador de alcance (::)
#include <iostream.h>
void cambia_global();

char x = 'D';

void main()
{
    cambia_global();
    cout << "Valor de 'x' Global en main: " << x << endl;
}

void cambia_global()
{
    char x;

    x = 'd';
    cout << "\nValor de 'x' local:  " << x << endl;
    cout << "Valor de 'x' Global: " << ::x << endl;
    ::x = 'A';
}
```

OBSERVACIONES.

En la función cambia_global(), primero mostramos el contenido de la variable local 'x' y luego usando el operador de ámbito (**::**) mostramos el valor de la variable global 'x'. Por último antes de salir de la función cambiamos el valor de la variable global 'x' al caracter 'A', y salimos de la función.

En la función main(), primero llamanos a la función cambia_global() y luego mostramos el valor de la 'x' global, la que ahora debe aparecer con su nuevo valor de: x = 'A'.

1.d) Funciones en línea.

Las macros definidas normalmente en los archivos de cabecera y que son substituidas en el programa por el preprocesador antes de la compilación, nos evitan trabajo, eliminan en cierto grado la posibilidad de errores, además de ahorrar tiempo en la ejecución del programa, evitando el proceso de llamada a una función, sin embargo su comportamiento no es totalmente el de una función, además de no contar con verificación de tipos en el momento de la compilación, lo que puede ocasionar problemas detectables solo en tiempo de ejecución.

C++ nos brinda una solución a este tipo de problema, al ofrecernos además de todos los beneficios de las macro, la verificación de parámetros. A esta solución se le llama **funciones en línea**, se especifican con la palabra **inline**, Ejemplo.

```
inline int func1() { return 0; }
```

Una de las características que deben tener estas funciones es que deben ser cortas, de lo contrario no va a ser aprovechada su definición, además de que el compilador las tratará como una función común y corriente.

Note en la definición que se antepone la palabra **inline**, cuando el compilador encuentra la definición **inline**, no genera código como lo hace con una definición ordinaria de función, en vez de eso, recuerda el código de la función. Cuando es llamada la función **inline** (lo cual se realiza de la misma forma que la llamada a cualquier función), el compilador realiza una verificación de la función, como lo hace con las demás funciones cuando son llamadas, en seguida viene la diferencia con las demás funciones ya que en este caso substituye en el lugar de llamada, el código de la función requerida. Con este proceso, se combina la eficiencia de las macros del preprocesador, con la verificación de errores realizada por el compilador en las funciones ordinarias.

Al usar funciones en línea, éstas deben de ser definidas antes de su llamada, ya que no puede ser declarada como las demás funciones antes de main(), de la forma:

```
inline int func1();
```

Esta declaración no tiene sentido para el compilador, ya que no hay espacio reservado para una función **inline**, ésta es recordada por el compilador y substituido su código cuando es llamada, por lo cual es común definir este tipo de funciones en los archivos de cabecera.

Note que las funciones **inline** van a duplicar el código del programa, ya que cada vez que una función de este tipo es llamada, se substituye su código en el lugar de la llamada. Al usar este tipo de funciones, y más aún cuando se pasan argumentos

debemos pensar en la utilización de una función **inline** cuando el código de la función sea menor que él necesario para la llamada a la función, en este caso, estamos ahorrando espacio.

```
// Programa # 15, Funciones en línea.
#include <stdio.h>

inline float area_cir(int radio) { return 3.14159*radio*radio; }

void main()
{
    int a;
    float b;

    printf("\nDame valor del radio: ");
    scanf("%d", &a);
    b = area_cir(a);
    printf("\nValor del área = %6.2f", b);
}
```

OBSERVACIONES.

En este programa es importante notar 2 aspectos; primero, la función fué definida antes de su llamada y segundo, se le antepuso la palabra **inline**.

1.e) Sobrecarga de funciones.

C++ introduce el concepto de sobrecarga de funciones, lo que significa que podemos llamar a varias funciones utilizando el mismo nombre de función, pero diferentes argumentos, ejemplo:

```
imprime();
imprime(50);
imprime(34.678);
imprime("Hola Verónica");
```

Este tipo de definición de funciones se usa normalmente en las clases, como veremos más adelante, sin embargo es posible sobrecargar funciones ordinarias usando la palabra **overload**, este indicador deberá usarse antes de cualquier declaración de función. **overload** es buena por estilo pero obsoleta en versiones modernas de C++, cuidado al tratar de utilizarla.

```
overload imprime;   // Decimos a C++ que vamos a sobrecargar
```

// esta función.

```
void imprime();
void imprime(int);
void imprime(float);
void imprime(char *);
```

El siguiente programa se muestra de acuerdo a lo aceptado por nuestro compilador (TurboC++).

```
// Programa # 16, Sobrecarga de funciones.
#include <iostream.h>

void imprime();
void imprime(int i);
void imprime(float e);
void imprime(char *a);

void main()
{
    int x = 30;
    float y = 34.567;
    char z[] = "Hola Patricia";

    imprime();
    imprime(x);
    imprime(y);
    imprime(z);
}

void imprime() { cout << "Sobrecarga de funciones\n"; }
void imprime(int i) { cout << i << endl; }
void imprime(float f) { cout << f << endl; }
void imprime(char *s) { cout << s << endl; }
```

OBSERVACIONES.

Para que el compilador distinga entre el uso de una u otra función, cada una deberá tener un conjunto único de argumentos cada vez que la función es sobrecargada, lo cual se puede apreciar en el programa anterior.

Lo bueno de esta forma de operación, es que el programador no necesita recordar un número grande de nombres de funciones, puede usar el mismo nombre para diferentes acciones.

1.f) Parámetros por default.

En C++, el conjunto de parámetros en la llamada a una función, pueden tomar valores por default, de tal manera que la función pueda ser invocada con un número de parámetros menor al definido en la función, es decir, se pueden omitir los parámetros que tengan asignado un valor por default, o bien, se les puede modificar su valor.

Los valores por default, se asignan en la declaración de la función y no en la definición de la función.

Para utilizar valores por default se deben tomar en cuenta las siguientes consideraciones:

1.- Los argumentos por default se deben pasar por valor, no pueden ser pasados por referencia.

2.- Los valores por default pueden ser valores literales o valores constantes (const). No pueden ser valores especificados por una variable.

3.- Todos los argumentos con valores por default deberán estar al final de la línea de parámetros.

Ejemplo:

Si el encabezado de la función fué declarado como:

```
void func2(int x, float pi = 3.14159, char c = 'a');

func2(10);                // Llamada válida
func2(20, 8.56);          // Llamada válida
func2(30, 3.67, 'x');          // Llamada válida
func2(,  , 'y');               // Llamada inválida
```

Ejemplo; un programa completo.

```
// Programa # 17, Parámetros por default.
#include <iostream.h>

#include <conio.h>

void saca_xy(char *cad, int x = -1, int y = -1);

void main()
{
    clrscr();
    saca_xy("Hola Susy", 10, 10);
    saca_xy(" Feliz Cumpleaños");
    saca_xy("Vamos a estudiar C++", 40);
    saca_xy("Si no, Vamonos al cine", 40, 12);
```

```
}


void saca_xy(char *cad, int x, int y)
{
    if(x == -1) x = wherex();
    if(y == -1) y = wherey();
    gotoxy(x, y);
    cout << cad;
}
```

OBSERVACIONES.

En la primera llamada a la función saca_xy(), definimos la posición de impresión en x = 10, y = 10, (no usamos los parámetros por default), en la siguiente llamada, dejamos que el programa defina tanto renglón como columna, a continuación definimos la columna x = 40, pero dejamos que el programa defina el renglón, por último, definimos tanto renglón como columna; x = 40, y = 12.

1.g) Conversión explicita de tipos.

El nombre de un tipo puede ser utilizado de la misma forma que una función para realizar una conversión de tipos; es una forma alterna de hacer un "cast".

```
// Programa # 18, conversión explicita de tipos.
#include <stdio.h>

void main()
{
    int i = 766;
    float f;

    f = float(2.71828*i);
    printf("Valor = %5.2f", f);
}
```

OBSERVACIONES.

La forma tradicional de haber realizado el "cast" sería:

```
f = (float)2.71828*i;
```

Sin embargo en C++, podemos usar el tipo (float), en este caso, como si se tratara de una función, encerrando entre paréntesis su argumento.

1.h) Funciones con número de argumentos variable.

Algunas veces es conveniente implementar funciones que puedan manejar un número variable de argumentos. Para declarar este tipo de funciones se requiere un proceso ya definido por el lenguaje, un ejemplo clásico de este tipo de funciones es printf, para declarar este tipo de funciones se usa la siguiente sintaxis.

```
// Función con uno o más parámetros.
int fun1(int i, ...);
```

ó

```
// Función con por lo menos 2 parámetros.
int func2(int i, char c, ...);
```

La coma antes de los puntos es opcional.

Veamos un ejemplo completo.

```
// Programa 18A, funciones con número variable de parámetros.

#include <stdarg.h>
#include <stdio.h>

// Declaración de una función con un número variable
// de parámetros.

void func1(int a, ...)
{
    // preparación para acceso de los parámetros.
    va_list parametros;
    va_start(parametros, a);

    // uso de los parámetros.
    for(int i = 0; i < a; i++) {
        int valor = va_arg(parametros, int);
        printf("Parámetro %d = %d\n", i, valor);
    }

    // Se limpia antes de salir
    va_end(parametros);
}
```

```
void main()
{
    func1(4, 50, 60, 70, 80);
}
```

OBSERVACIONES.

C++ usa macros para menejar los parámetros dentro de la función **func1()**, éstos son:

va_list
va_start
va_arg
va_end

Estos macros son declarados en el archivo de cabecera stdarg.h. En Borland C++ una función con un número variable de argumentos requiere por lo menos contar con un parámetro formal, ésto es necesario por las siguientes razones:

1) La función deberá contar con un mecanismo para saber el número de parámetros recibidos en el momento de la llamada.

2) La macro **va_start** espera 2 parámetros; el primero es el nombre de la variable declarada del tipo **va_list**. El segundo deberá ser el nombre de un parámetro formal, de la lista de argumentos de la función.

Se puede usar cualquier forma, para indicar el número de parámetros que van a ser pasados a la función con número variable de argumentos. Por ejemplo la funciómn printf() usa una cadena formateada, donde los argumentos se denotan mediante el caracter [%]. En nuestro ejemplo la función **func1()** recibe el número de parámetros directamente en el parámetro **a**. En la práctica se puede usar cualquier método.

Cuando se usan las macros anteriores, se deberá tener en cuenta los siguientes puntos:

a) Se deberán usar en el orden correcto. Lo contrario puede ser desastrozo.
b) El propósito de la macro **va_end** es el de ajustar el SP (apuntador de la pila) a su valor antes de llamar a la función, es por ello que se coloca al final de ésta, si no se usa, normalmente no causa problemas en Borland C++, sin embargo se recomienda su empleo por razones de portabilidad.
c) La macro **va_arg** no sabe cuando se termina la lista de parámetros, o que tipos de argumentos contiene, ésto es responsabilidad del programador, si se cometen errores en este renglón, el programa va a compilar correctamente, sin embargo no va a trabajar.
d) Existe una promoción de tipos por default, los argumentos del tipo **char** son convertidos a **int** y los de tipo **float** son convertidos a **double**, antes de ser

pasados a la función. No se debe intentar accesar argumentos **char** o **float** usando **va_arg**.

NOTE que el parámetro **a** en el programa contiene el número de argumentos que fueron pasados a la función **func1().**

Veamos otro ejemplo de manejo de funciones con número variable de argumentos, en este caso simulemos una función que trabaja de la misma manera que las de la familia de printf().

El siguiente programa, escribe ciertos valores en un archivo temporal, para lo cual se vale de la funcion funvar(), y posteriormente estos valores son leidos y mostrados desde main() usando variables diferentes.

```
// Programa 18B, Ejemplo de uso de número variable de argumentos.

#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>

int funvar(FILE *pt, char *fmt, ...);

void main()
{
    FILE *fp;
    int x, ia, inum = 30;
    float fb, fnum = 90.5;
    char scad[4], cad[4] = "abc";

    fp = tmpfile();
    if(!fp) {
        printf("Error en creación de tmpfile()");
        exit(1);
    }
    x = funvar(fp, "%d %f %s", inum, fnum, cad);
    printf("El valor regresado por funvar() es = %d\n", x);
    rewind(fp);
    fscanf(fp, "%d%f%s", &ia, &fb, scad);
    printf("%d %f %s\n", ia, fb, scad);
    fclose(fp);
}


int funvar(FILE *pt, char *fmt, ...)
{
    va_list argptr;
    int cn;
```

```
    va_start(argptr, fmt);
    cn = vfprintf(pt, fmt, argptr);
    va_end(argptr);
    return cn;
}
```

OBSERVACIONES.

Note que en la función funvar() se estan pasando tanto parámetros fijos como parámetros variables.

La función vfprintf() es similar a fprintf() excepto en la forma de manejar los argumentos. Su sintaxis es la siguiente.

Sintasis:

```
#include <stdio.h>
int vfprintf(FILE *stream, const char *format, va_list arglist);
```

Donde: **stream** Es un puntero a FILE
**arglist**. Es un apuntador a una lista de argumentos.
**format** Es un apuntador a una cadena.

V. regresado El número de bytes escritos en el archivo.

La salida del programa es la siguiente:

```
El valor regresado por funvar() es = 16
30 90.500000 abc
```

Donde el valor de 16 es el número de bytes escritos al archivo temporal.

## 2.- **Introducción a clases.**

La **clase** es un nuevo tipo de dato definido por el usuario, el cual nos brinda las ventajas de una estructura y además tiene la habilidad de controlar el acceso a sus miembros.

* Los arreglos son una colección de datos del mismo tipo.
* Las estructuras nos permiten agrupar datos de diferente tipo.
* Las **clases** de C++ nos permiten implementar un tipo de dato, asociandolo con operadores y funciones miembro.

La **clase** es la base de la programación orientada a objetos en C++.

## 3.- **Modelación del mundo real con Clases.**

La **clase** en C++ nos proporciona una manera natural de construir modelos computacionales de sistemas del mundo real. Por ejemplo para la modelación de vehículos, debemos de tomar en cuenta tanto su descripción física (número de ejes, potencia, peso, etc.) y su comportamiento (aceleración, velocidad, consumo de combustible, etc.). Entonces una clase **Carro** deberá encapsular los parámetros físicos (datos) y su comportamiento (funciones) de una manera general. Usando herencia es posible derivar vehículos más específicos; clase VW, clase Datsun, etc, sumando nuevos tipos de datos y funciones, y modificando (sobre-escribiendo) algunas de las funciones de la clase base, por lo tanto los códigos escritos para la clase base pueden ser re-usados.

## 4.- **Construyendo clases:** Un ejemplo.

### 4.a) Datos miembro.

En un ambiente Gráfico, el punto de partida deberá ser una **clase** que modele el pixel sobre la pantalla, usando para ello las coordenadas cartesianas. Una primera aproximación puede ser un tipo **struct** llamado punto, el cual maneje juntas las coordenadas X y Y, como datos miembro.

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

Una vez definida una clase, se cuenta como un nuevo tipo de dato, él cual es manejado por C++ de la misma forma que sus propios tipos.
Una vez declarado el nuevo tipo, podemos declarar variables de dicho tipo.

* En C la declaración se haría como sigue:

```
struct punto origen, centro, pos_act;
```

* En C++, se hace de la siguiente forma:

```
punto origen, centro, pos_act;
```

Una variable del tipo (semejante a origen) es una de varias instancias del tipo punto (instancia y objeto en C++ se manejan de manera indistinta), cuando se asignan valores, se asignan a la instancia no a la clase punto.

Al definir este nuevo tipo **punto**, es importante considerar el hecho de que; se puede manejar de manera independiente el punto X o el punto Y, sin embargo, hay muchas situaciones en que es conveniente, manejarlos juntos, en este caso una instancia del tipo punto nos ayuda mucho.

Al enviar pixeles a la pantalla, es conveniente definir si van a estar apagados o encendidos, para ello debemos agregar un miembro adicional a la estructura tipo punto. Para mayor claridad, podemos pensar en un tipo booleano, él cual puede ser definido mediante una enumeración.

```
enum boolean {false, true};

struct punto {
          int x;
          int y;
          boolean visible;
     };
```

Recuerde que los valores de los tipos enumerados empiezan en CERO, por lo que tendremos:

```
false = 0;
true  = 1;
```

4.b) Funciones miembro.

Una clase puede contener tanto datos miembro como funciones miembro.
Una **función miembro**, es una función declarada dentro de la definición de clase y estrechamente ligada a la clase (las funciones miembro son conocidas como métodos en algunos lenguajes, como Turbo PASCAL y smalltalk).

* Los datos miembro, es lo que la clase conoce.
* Las funciones miembro es lo que la clase hace.

Para sumar una función miembro a la clase punto, hay dos formas de hacerlo:

1) Definir la función dentro de la clase.
2) Declararla dentro de la clase y definirla fuera de la clase.

Método 1.

Al definir una función dentro de la clase, se maneja la función como **inline** por default.

Brevemente; las funciones **inline** son funciones cortas, las cuales se tratan como se trata a una función, verificando tipos, la excepción es que no son llamadas como cualquier función, su código es substituido en el lugar de llamada, son equivalentes a los macro del preprocesador de "C", con la ventaja, de verificación de tipos.

Lo anterior las hace más rápidas en su ejecución, sin embargo este beneficio puede disminuir proporcionalmente al aumento de tamaño de la función, y entre mayor sea la función y más llamadas se hagan a ella, va a crecer el tamaño del programa, y el uso de memoria.

Para ejemplificar lo anteriormente expuesto, definamos dos funciones miembro, getx(), la cual nos regresa el valor de X, y gety(), la cual regresa el valor de Y.

```
enum boolean {false, true};

struct punto {
          int x;
          int y;
          boolean visible;
          int getx() { return x; }
          int gety() { return y; }
     };
```

De acuerdo a la sintaxis de "C", en este caso particular, las funciones definidas dentro de la estructura no tienen parámetros y regresan un entero, el cuerpo de ambas funciones se encuentra encerrado entre las llaves **{ }** y contienen las sentencias de definición de la función, en este caso una sola sentencia el return x.

Método 2.

En este método simplemente se declara la función miembro dentro de la estructura, usando la forma estandar de declaración de "C" implementando la definición fuera de la estructura.

```
enum boolean {false, true};

struct punto {
          int x;
          int y;
          boolean visible;
          int getx();
          int gety();
```

```
};


int punto::getx()
{
     return x;
}

int punto::gety()
{
     return y;
}
```

Las funciones miembro definidas fuera de la declaración de la clase pueden también ser implementadas como **inline** (si se cumplen ciertas condiciones), para hacerlo es necesario incluir la palabra **inline** y definir la función dentro del mismo archivo.

Note en la definición de las funciones el uso del operador de alcance (::) en donde se define la función haciendo referencia a la estructura **punto** como (punto::getx() y punto::gety()). El nombre de la clase es necesario para decirle al compilador a cual clase pertenece una determinada función (en este caso getx() y gety()), ya que pueden existir otras versiones de dichas funciones.

Cuando la definición de la función se hace dentro de la definición de la clase, es claro que no es necesario hacer ésto.

La sentencia **punto::** adelante del nombre de la función (**getx()** y **gety()**) tiene también el propósito de extender la influencia, de la estructura **punto**, de tal forma que la **x** en el return x; es una referencia a la **x**, miembro de la estructura **punto**.

Los cuerpos de las funciones en realidad estan dentro del alcance de **punto**, despreciando su localización física.

Un punto interesante en la implementación del método es que hemos definido funciones miembro ligadas a una estructura, en este caso **punto**, como son funciones miembro, pueden accesar a todas las variables de la estructura, en este caso cada función accesa una variable de la estructura punto y regresa su valor respectivo.

4.c) Llamando una función miembro.

Ahora las funciones miembro representan operaciones sobre objetos de su clase, así cuando llamamos a getx(), debemos indicar a que objeto de la clase, en este caso **punto** pertenece, entonces, de la misma forma que llamamos a un dato miembro, debemos llamar a una función miembro.

Si consideramos la declaración: punto origen;

Donde origen es una instancia de la clase **punto**.

Llamada a dato miembro.

```
origen.x
```

Llamada a función miembro.

```
origen.getx();
origen.gety();
```

El operador (**.**) es el selector tanto de datos miembro como de funciones miembro.

De la misma forma si se tiene: punto origen, *ptr;
ptr = &origen;

Para llamada a dato miembro.

```
ptr -> x
```

Para llamada a función miembro.

```
ptr -> getx();
ptr -> gety();
```

Ejemplo, usando el método 1.

```
// Programa # 19, definición de clases
#include <stdio.h>
enum boolean {false, true};

struct punto {
    int x;
    int y;
    boolean visible;
    int getx() { return x; }
    int gety() { return y; }
    };

void main()
{
    char pv;
```

```
    punto origen, cord_act;

    origen.x = 0;
    origen.y = 0;
    origen.visible = false;

    printf("\nDame la coordenada actual X: ");
    scanf("%d", &cord_act.x);
    printf("\nDame la coordenada actual Y: ");
    scanf("%d", &cord_act.y);
    fflush(stdin);
    printf("\nPunto visible ? S/N: ");
    pv = getchar();
     if((pv == 'S') || (pv == 's')) cord_act.visible = true;
     else cord_act.visible = false;

     printf("\nValor del origen: (x, y) = ");
     printf("(%d, %d)\n", origen.getx(), origen.gety());
     printf("El punto es: ");
     if(origen.visible == true) puts("visible");
     else puts("oculto");

     printf("\nValor de coordenadas actuales: (x, y) = ");
     printf("(%d, %d)\n", cord_act.getx(), cord_act.gety());
     printf("El punto es: ");
     if(cord_act.visible == true) puts("visible");
     else puts("oculto");
}
```

OBSERVACIONES.

La corrida del programa nos resulta:

```
Dame la coordenada actual X: 56
Dame la coordenada actual Y: 65
Punto visible ? S/N: S
Valor del origen: (x, y) = (0, 0)
El punto es: oculto

Valor de coordenadas actuales: (x, y) = (56, 65)
El punto es: visible
```

La instancia **origen** fué inicializada con valores fijos y a la instancia **cord_act** se le asignaron sus valores desde el teclado.

El programa nos muestra el uso de una clase (struct) la cual cuenta con datos y métodos que nos permiten el manejo de dichos datos.

Un punto importante a considerar, en el manejo del tipo **struct** en C++, es que podemos tener acceso libremente tanto a los datos miembro, como a las funciones miembro. Lo que significa que son **públicos**.

El método 2 se escribiría:

```
// Programa # 19A, definición de clases
#include <stdio.h>
enum boolean {false, true};
struct punto {
     int x;
     int y;
     boolean visible;
     int getx();
     int gety();
     };

void main()
{
     char pv;
     punto origen, cord_act;

     origen.x = 0;
     origen.y = 0;
     origen.visible = false;

     printf("\nDame la coordenada actual X: ");
     scanf("%d", &cord_act.x);
     printf("\nDame la coordenada actual Y: ");
     scanf("%d", &cord_act.y);
     fflush(stdin);
     printf("\nPunto visible ? S/N: ");
     pv = getchar();
     if((pv == 'S') || (pv == 's')) cord_act.visible = true;
     else cord_act.visible = false;

     printf("\nValor del origen: (x, y) = ");
     printf("(%d, %d)\n", origen.getx(), origen.gety());
     printf("El punto es: ");
     if(origen.visible == true) puts("visible");
     else puts("oculto");

     printf("\nValor de coordenadas actuales: (x, y) = ");
     printf("(%d, %d)\n", cord_act.getx(), cord_act.gety());
     printf("El punto es: ");
     if(cord_act.visible == true) puts("visible");
     else puts("oculto");
```

```
}

int punto::getx()
{
     return x;
}

int punto::gety()
{
     return y;
}
```

En este caso las funciones fueron definidas fuera de la declaración de la clase. Esta forma de definir las funciones miembro, como las funciones globales, después de main() no se usa, aunque si es posible. Las funciones miembro se definen después de la declaración de la clase o en una archivo llamado de implementación.

4.d) Constructores y Destructores.

Hay dos tipos de funciones miembro que juegan un papel importante dentro del C++, éstos son: constructores y destructores, para resaltar su importancia vamos a hacer un pequeño análisis de ellos:

Un problema común en los lenguajes tradicionales es la inicialización, antes de usar un dato estructurado, normalmente se le deberá asignar memoria e inicializarlo, considere el hecho de inicializar la estructura **punto**, anteriormente definida.

```
struct punto {
          int x;
          int y;
          boolean visible;
          };
```

Una forma común de inicialización es la siguiente:

punto origen;

```
origen.x = 0;
origen.y = 0;
origen.visible = false;
```

El trabajo de inicializar al objeto **origen**, es tedioso y largo, se requieren de varias sentencias para inicializar el objeto, y si son varios objetos los que hay que inicializar, el trabajo crece.

Un proceso natural es la construcción de una función de inicialización de tal manera que generalice las sentencias de inicialización de tal forma que pueda manejar cualquier objeto a **punto** que sea pasado como un argumento.

```
void ini_punto(punto *a, int b, int c)
{
     a -> x = b;
     a -> y = c;
     a -> visible = false;
}
```

Esta función define un puntero al objeto **punto**, el cual lo usa para asignar los valores también pasados como argumento, a sus miembros. Se supone que la función fué correctamente diseñada para servir como inicialización a la estructura **punto**. Note que debe especificar el tipo de clase y el objeto en particular sobre el cual deberá actuar. Note que la función ini_punto() no es una función miembro, y lo que se requiere es una función miembro que inicialice cualquier objeto del tipo punto. C++ nos proporciona un tipo de función miembro llamada **constructor**.

Un **constructor** nos especifica la forma en que un nuevo objeto de la clase tipo deberá ser inicializado, pudiendo por ejemplo incluir código para asignación de memoria e inicialización (asignación de valores a miembros, conversión de un tipo a otro y cualquier cosa que pueda ser útil es la inicialización de un objeto).

Los **constructores** pueden ser llamados implicitamente o explicitamente, el compilador de C++ llama automaticamente el constructor mientras se esta definiendo un nuevo objeto de una clase.

Los **destructores**, como su nombre lo indica, destruyen un objeto de una clase creada con anterioridad mediante un **constructor**, el proceso de destrucción de un objeto significa; la liberación de la memoria, y el limpiado de valores. De la misma forma que con los **constructores**, éstos pueden ser llamados explicita (usando el operador delete) o implicitamente (cuando un objeto sale de su ámbito). Si no se define un destructor para una clase, C++ genera uno por default.

La siguiente versión de **punto**, agrega un constructor.

```
struct punto {
          int x;
          int y;
          boolean visible;
          punto(int nx, int ny);
          int getx() { return x; }
          int gety() { return y; }
          };

punto::punto(int nx, int ny)
{
     x = nx;
     y = ny;
```

```
        visible = false;
    };
```

En este caso la definición del constructor fué realizada fuera de la declaración de la clase, pueden también definirse dentro de la clase, como cualquier función miembro, en cuyo caso se manejará como una función **inline**, recuerde que también se puede definir fuera de la clase y usar la palabra **inline** antes de la función para que el compilador la trate como tal, sin embargo, tenga cuidado con la cantidad de código escrito dentro de la función. En el caso del constructor, el código mostrado no es proporcional al código generado.

Note que el nombre del constructor es el mismo nombre que el de la clase, y además puede tener argumentos, como cualquier otra función. El cuerpo del constructor es igual al de cualquier función, así mismo, puede llamar a cualquier función de su clase o accesar cualquier dato, también de su clase. Un constructor, NUNCA regresará valores, siempre es del tipo **void**.

Ahora podemos declarar un objeto e inicializarlo al mismo tiempo.

```
punto origen(1, 1);
```

Esta declaración invoca al constructor de la clase **punto**. Además recuerde que C++, puede manejar valores de default en sus funciones, por lo que, en un **constructor** podemos incluir valores de default.

```
punto::punto(int nx = 0, int ny = 0, boolean nv = false)
{
    x = nx;
    y = ny;
    visible = nv;
};
```

NOTA: Recuerde que los valores de default se ponen en la declaración de la función o en su definición si ésta se hace dentro de la clase (struct).

Si usamos la declaración: punto origen(4);

Se deberá inicializar con x = 4, y = 0.

Veamos un ejemplo completo.

```
// Programa # 20, definición de clases y constructor
#include <stdio.h>
enum boolean {false, true};
struct punto {
    int x;
    int y;
    boolean visible;
```

```
     punto(int nx = 0, int ny = 0);
     int getx() { return x; }
     int gety() { return y; }
     };

punto::punto(int nx, int ny)
{
     x = nx;
     y = ny;
     visible = false;
}



void main()
{
     char pv;
     punto origen(2), cord_act;

     printf("\nDame la coordenada actual X: ");
     scanf("%d", &cord_act.x);
     printf("\nDame la coordenada actual Y: ");
     scanf("%d", &cord_act.y);
     fflush(stdin);
     printf("\nPunto visible ? S/N: ");
     pv = getchar();
     if((pv == 'S') || (pv == 's')) cord_act.visible = true;
     else cord_act.visible = false;

     printf("\nValor del origen: (x, y) = ");
     printf("(%d, %d)\n", origen.getx(), origen.gety());
     printf("El punto es: ");
     if(origen.visible == true) puts("visible");
     else puts("oculto");

     printf("\nValor de coordenadas actuales: (x, y) = ");
     printf("(%d, %d)\n", cord_act.getx(), cord_act.gety());
     printf("El punto es: ");
     if(cord_act.visible == true) puts("visible");
     else puts("oculto");
}
```

OBSERVACIONES.

En si, el programa es idéntico a los anteriores, con la salvedad que estamos definiendo un constructor, por lo cual se elimina, la inicialización primitiva usada en las versiones anteriores. Cuando se declaran parámetros por default, se hace en la

declaración del constructor; si éste se define dentro de la clase, alli mismo se indican los parámetros por default, si no, los parámetros por default son colocados en la declaración del constructor, en su definición NO se pueden indicar.

4.e) Código y datos juntos.

Una de las características más importantes de la Programación Orientada a Objetos, es el manejo dentro del objeto de código y datos, en el diseño del programa, ambas partes se interrelacionan; Los datos definen el flujo de código y el código manipula la forma y valores de los datos.

Cuando código y datos son manejados como entidades separadas, existe siempre el riesgo de llamar a la función adecuada con los datos erroneos o viceversa, es trabajo del programador, mantener una relación adecuada entre ambas entidades.

Escribiendo las declaraciones de código y datos juntos, C++ ayuda a mantener la relación que hay entre ellos. Tipicamente para definir el valor de un dato miembro, se deberá hacer un llamado a una función miembro de la clase a la cual pertenecen.

4.f) Control de acceso a miembros.

Mientras la estructura ampliada de C++ nos permite mantener datos y funciones juntos, no los encapsula, como debería de ser, ya que el acceso a los datos y a las funciones dentro de una estructura es público por default, lo que significa que cualquier sentencia dentro del mismo ámbito puede leer o cambiar los valores de los datos dentro de la clase estructura, como se ha venido mencionando, ésto no es deseable, y nos puede traer serios problemas. Una buena práctica en diseño usando C++, es el ocultamiento de datos u ocultamiento de información, lo cual se consigue haciendo los datos miembro privados o protegidos con lo que se requiere una autorización para accesarlos.

La REGLA general es hacer todos los datos privados de tal manera que solo puedan ser accesados a través de funciones declaradas como públicas.

Existen pocas situaciones donde será necesario manejar datos miembro como públicos en vez de datos miembro privados o protegidos, de la misma forma, aquellas funciones involucradas solamente en operaciones internas pueden ser declaradas como privadas o protegidas en vez de públicas.

Existen tres palabras reservadas por medio de las cuales podemos ejercer control sobre estos aspectos en los miembros de las estructuras o clases. Para poder manejar un miembro de una determinada manera, deberemos usar la palabra (public, private o protected) seguida de (**:**), antes de la declaración del miembro que desemos se vea afectado.

**private** Los miembros que siguen a private, pueden ser accesados solamente por funciones miembro declaradas dentro de la misma clase.

**protected** Los miembros que siguen a protected, pueden ser accesados por funciones miembro dentro de la misma clase y por funciones miembro de clases que son derivadas de esta clase.

**public** Los miembros que siguen a public, pueden ser accesados desde cualquier parte, dentro del ámbito de la definición de la clase.

Como un ejemplo redefinimos la estructura **punto** de tal forma que sus datos miembros sean privados y sus funciones miembro sean publicas. También va a ser necesario hacer algunos cambios adicionales para que pueda trabajar como las versiones anteriores.

```
// Programa # 21, definición de clases
#include <stdio.h>
enum boolean {false, true};

struct punto {
private:
     int x;
     int y;
     boolean visible;
public:
     punto(int nx = 0, int ny = 0, boolean nv = false);
     int getx() { return x; }
     int gety() { return y; }
     boolean getvisible() { return visible; }
     };


punto::punto(int nx, int ny, boolean nv)
{
     x = nx;
     y = ny;
     visible = nv;
}


void main()
{
     char pv;
     int a, b;
     boolean c;
     punto origen(2);
```

```
      printf("\nDame la coordenada actual X: ");
      scanf("%d", &a);
      printf("\nDame la coordenada actual Y: ");
      scanf("%d", &b);
      fflush(stdin);
      printf("\nPunto visible ? S/N: ");
      pv = getchar();
      if((pv == 'S') || (pv == 's')) c = true;
      else c = false;

      printf("\nValor del origen: (x, y) = ");
      printf("(%d, %d)\n", origen.getx(), origen.gety());
      printf("El punto es: ");
      if(origen.getvisible() == true) puts("visible");
      else puts("oculto");

      punto cord_act(a, b, c);

      printf("\nValor de coordenadas actuales: (x, y) = ");
      printf("(%d, %d)\n", cord_act.getx(), cord_act.gety());
      printf("El punto es: ");
      if(cord_act.getvisible() == true) puts("visible");
      else puts("oculto");
}
```

OBSERVACIONES.

En esta versión ya no es posible accesar los datos miembro directamente por lo tanto debemos recurrir a variables auxiliares que nos capturen los valores dados del teclado, y por medio del constructor almacenar en la estructura dichos valores. NOTE que el objeto **cord_act** es definido antes de su uso e inicializado con los valores capturados del teclado.

Tuvimos también que agregar una función miembro para mostrar el estado del pixel, y modificar ligeramente la función **constructor**.

Un aspecto interesante, es que va creciendo la sección de declaraciones.

4.g) La **struct** es public por default.

La clase **struct** es public por default, entonces como vimos en el programa anterior es necesario usar **private** para especificar la parte privada y a continuación la palabra **public** para la sección de la estructura a la cual se va a tener libre acceso.

En C++, por definición el contenido de una clase debe ser privado a menos que se indique lo contrario. El C++ cuenta con la palabra **class**, la cual nos brinda esta característica. Resumiendo; mientras para la **struct**, sus miembros son por deffault públicos, para **class** sus miembros por default son privados.

Veamos la definición anterior de **struct** ahora con **class**.

```
class punto {
     int x;
     int y;
     boolean visible;
public:
     punto(int nx = 0, int ny = 0, boolean nv = false);
     int getx() { return x; }
     int gety() { return y; }
     boolean getvisible() { return visible; }
     };
```

NOTA: Esta declaración puede ser substituida en el programa anterior y va a correr normalmente.

Ahora no es necesario el modificador **private**, ya que por default lo son sus miembros. Las funciones miembro deberán ser declaradas **public** de tal manera que puedan se usadas fuera de la clase para el manejo de los datos privados.

Es posible, una vez declarada una sección **public**, volver a declarar la siguiente sección **private** y repetir de nuevo **public**.

4.h) Meditando acerca de Objetos.

Podemos pensar de un Objeto como una entidad con un **estado** interno y **operaciones** externas. Las "operaciones externas" son las funciones miembro. Las funciones que ejecutan los mensajes en un lenguaje Orientado a Objetos se llaman **métodos**. El mensaje, es la función de llamada. El concepto de estado significa que un objeto recuerda cosas acerca de él mismo, cuando no se esta usando. Por ejemplo una función ordinaria (sin ninguna variable **static**), pierde su estado, ya que siempre inicia en el mismo punto cuando se usa. Puesto que un Objeto tiene un estado, es posible definir una función que inicie en diferente punto cada vez que se le llama.

Por ejemplo:

```
// Programa # 21B, Un objeto recuerda su estado.
#include <iostream.h>
enum estado_actual {cero, uno, dos, tres, cuatro, cinco };


class estado {
    int a;
public:
    void inicio() { a = cero; }
    void siguiente();
```

```
    };

void estado::siguiente()
{
    switch(a) {
        case cero   : a = uno;    break;
        case uno    : a = dos;    break;
        case dos    : a = tres;   break;
        case tres   : a = cuatro; break;
        case cuatro : a = cinco;  break;
        case cinco  : a = cero;   break;
        default     : a = cero;
    }
    cout << "a = " << a << endl;
}


void main()
{
    estado  n;

    n.inicio();
    for(int i = 0; i < 7; i++)
        n.siguiente();
}
```

OBSERVACIONES.

Si vemos la corrida esta nos resulta:

a = 1
a = 2
a = 3
a = 4
a = 5
a = 0
a = 1

Podemos constatar lo que dijimos anteriormente, en cada llamada a la función miembro siguiente(), ésta recuerda su estado anterior, lo usa y lo modifica.