Capitulo I

Programación Orientada a Objetos

La programación orientada a objetos, ha tomado las mejores ideas de la programación estructurada y los ha combinado con varios conceptos nuevos y potentes que incitan a contemplar las tareas de programación desde un nuevo punto de vista. La programación orientada a objetos, permite descomponer mas fácilmente un problema en subgrupos de partes relacionadas del problema. Entonces, utilizando el lenguaje se pueden traducir estos subgrupos a unidades autocontenidas llamadas objetos.

El término Programación Orientada a Objetos (POO), hoy en día ampliamente utilizado, es difícil de definir, ya que no es un concepto nuevo, sino que ha sido el desarrollo de técnicas de programación desde principios de la década de los setenta, aunque sea en la década de los noventa cuando ha aumentado su difusión, uso y popularidad. No obstante, se puede definir POO como una técnica o estilo de programación que utiliza objetos como bloque esencial de construcción.

Un objeto es una unidad que contiene datos y las funciones que operan sobre esos datos. A los elementos de un objeto se les conoce como miembros; las funciones que operan sobre los objetos se denominan métodos y los datos se denominan miembros datos.

1.1 ORIGENES DE LA PROGRAMACION ORIENTADA A OBJETOS.

1.ª Etapa. Lenguajes Ensambladores.
La unidad de programación es la instrucción, compuesta de un operador y los operandos. El nivel de abstracción que se aplica es muy bajo.

2.ª Etapa. Lenguajes de Programación: Fortran, Algol, Cobol.
Los objetos y operaciones del mundo real se podían modelar mediante datos y estructuras de control separadamente. En esta etapa el diseño del software se enfoca sobre la representación del detalle procedimental y en función del lenguaje elegido. Conceptos como: refinamiento progresivo, modularidad procedimientos y programación estructurada son conceptos básicos que se utilizan en esta etapa. Existe mayor abstracción de datos.

3.ª Etapa.
Se introducen en esta etapa los conceptos de abstracción y ocultación de la información.

4.ª Etapa.
A partir de los años setenta se trabaja sobre una nueva clase de lenguajes de simulación y sobre la construcción de prototipos tales como Simula-70 y basado en parte de éste, el Smalltalk. En estos lenguajes, la abstracción de datos tiene una gran importancia y los problemas del mundo real se representan mediante objetos de datos a los cuales se les añade el correspondiente conjunto de operaciones asociados a ellos. Términos como Abstracción de datos, objeto, encapsulación entre otros, son conceptos básicos sobre la que se fundamenta la POO.

1.2 CONCEPTOS DE LA PROGRAMACION ORIENTADA A OBJETOS.

La POO representa una metodología de programación que se basa en las siguientes características:

1) Los diseñadores definen nuevas clases (o tipos) de objetos.
2) Los objetos poseen una serie de operaciones asociadas a ellos.
3) Las operaciones tienden a ser genéricas, es decir, operan sobre múltiples tipos de datos.
4) Las clases o tipos de objetos comparten componentes comunes mediante mecanismos de herencia.

Objeto: Una estructura de datos y conjunto de procedimientos que operan sobre dicha estructura. Una definición más completa de objeto es: una entidad de programa que consiste en datos y todos aquellos procedimientos que pueden manipular aquellos datos; el acceso a los datos de un objeto es solamente a través de estos procedimientos, únicamente estos procedimientos pueden manipular, referenciar y/o modificar estos datos.

Para poder describir todos los objetos de un programa, conviene agrupar éstos en clases.
Clase: Podemos considerar una clase como una colección de objetos que poseen características y operaciones comunes. Una clase contiene toda la información necesaria para crear nuevos objetos.

Encapsulación: Es una técnica que permite localizar y ocultar los detalles de un objeto. La encapsulación previene que un objeto sea manipulado por operaciones distintas de las definidas. La encapsulación es como una caja negra que esconde los datos y solamente permite acceder a ellos de forma controlada.

Las principales razones técnicas para la utilización de la encapsulación son:

1) Mantener a salvo los detalles de representación, si solamente nos interesa el comportamiento del objeto.
2) Modificar y ajustar la representación a mejores soluciones algorítmicas o a nuevas tecnologías de software.

Abstracción: En el sentido mas general, una abstracción es una representación concisa de una idea o de un objeto complicado. En un sentido mas especifico, la abstracción localiza y oculta los detalles de un modelo o diseño para generar y manipular objetos.

Una abstracción tiene un significado más general que la encapsulación, pudiendo hablar de abstracción de datos en lugar de encapsulación de datos.

Como resumen de los 3 conceptos expuestos anteriormente podemos decir que:

1) Los objetos son encapsulaciones de abstracciones en la POO.
2) La unidad de encapsulación en la POO es el objeto.

Una clase es un tipo: Un objeto es una instancia de ese tipo. Además, la clase es un concepto estático: una clase es un elemento reconocible en el texto del programa.

Un objeto es un concepto puramente dinámico, el cual pertenece, no al texto del programa, sino a la memoria de la computadora, donde los objetos ocupan un espacio en tiempo de ejecución una vez que haya sido creado.

La programación orientada a objetos, ha tomado las mejores ideas de la programación estructurada y los ha combinado con varios conceptos nuevos y potentes que incitan a contemplar las tareas de programación desde un nuevo punto de vista. La programación orientada a objetos, permite descomponer mas fácilmente un problema en subgrupos de partes relacionadas del problema. Entonces, utilizando el lenguaje se pueden traducir estos subgrupos a unidades autocontenidas llamadas objetos.

Objetos: Un objeto es una entidad lógica que contiene datos y un código que manipula estos datos; el enlazado de código y de datos, de esta manera suele denominarse encapsulación.
Cuando se define un objeto, se esta creando implícitamente un nuevo tipo de datos.
Polimorfismo: Significa que un nombre se puede utilizar para especificar una clase genérica de acciones.

Herencia: La herencia es un proceso mediante el cual un objeto puede adquirir las propiedades de otro objeto.

1.3 PRESENTACION DE LAS CLASES Y LOS OBJETOS

Objeto: Un objeto es una entidad abstracta que tiene las características de un objeto real.
Los objetos se crean y eliminan durante la ejecución del programa, además interactúan con otros objetos. Los objetos son construcciones de programación que se obtienen a partir de

entidades llamadas clases. La definición de una clase se conoce como *instanciación* de clases.

Para crear un objeto, es preciso definir primero su forma general utilizando la palabra reservada **class**. Una class es parecida a una estructura, es un tipo definido por el usuario que determina las estructuras de datos y las operaciones asociadas con este tipo.
Las clases son como plantillas o modelos que describen como se construyen ciertos tipos de objetos, cada vez que se construye un objeto de una clase se crea una instancia de esa clase, por consiguiente; los objetos son instancias de clases.

Una clase es una colección de objetos similares y un objeto es una instancia de una definición de una clase; una clase puede tener muchas instancias y cada una es un objeto independiente.
Una clase es simplemente un modelo que se utiliza para describir uno o mas objetos del mismo tipo.

Así, por ejemplo sea una clase ventana, un tipo de dato, que contenga los miembros dato:

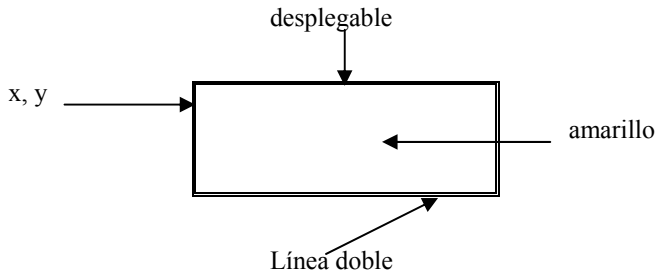
posx, posy
tipo_ventana
tipo_borde
color_ventana

y unas funciones miembro:

mover_horizontal
mover_vertical

Un objeto de la clase ventana, es una ventana concreta (una instancia de la clase) cuyos datos tienen por valores:

posx	x
posy	y
tipo_ventana	desplegable
tipo_borde	linea doble
color_ventana	amarillo



1.3.1 DEFINICIÓN DE UNA CLASE.

Una *clase* es la evolución natural de una estructura, la existencia de clases es la característica más significativa que convierte a C++ en un lenguaje orientado a objetos. Las clases son estructuras que contienen no sólo declaraciones de datos, sino también declaraciones de funciones. Las funciones se conocen como funciones miembro, e indican qué tipos de cosas puede hacer una clase. La palabra reservada ***class*** introduce una declaración de clase.

1.3.2 IDENTIFICADORES DE CLASE.

La longitud máxima para un identificador de clase es 32 caracteres. Una convensión que se adopta en todas las clases de borland es utilizar nombres que comiencen con una letra mayúscula para denotar clases y estructuras globales.

1.3.2.1 CUERPO DE UNA CLASE.

La forma general de la declaración de una clase es:

```
class Nombre_de_la_clase{
        datos y funciones privados
                .
                .
                .
 public:
datos y funciones publicas
                .
                .
                .
}lista de objetos;
```

Una clase puede contener tanto partes publicas como partes privadas, por defecto, todos los elementos que se definen en la clase son privados; esto significa que no pueden acceder a ellas ninguna función que no sea miembro de la clase.

```
 Class Counter{

        long count;                    //  variable privada , variable miembro de la clase
```

```
Public:

void SetValue(long);            //  Funciones públicas, funciones miembro de la clase
        long GetValue();

};
```

La variable long count, no está disponible o no se puede usar por otras funciones que no están declaradas en la clase, por lo que tratar de hacer esto es erróneo:

```
void main()
{

count = 3.111;

}
```

Una clase puede tener tantas variables como necesite. Estas pueden ser de cualquier tipo, incluyendo otras clases, apuntadores a objetos de clases e incluso apuntadores a objetos dinámicamente asignados.

Las funciones miembro SetValue(long) y GetValue(). Solo están declaradas dentro de la clase, la definición de estas funciones sería así:

```
void Counter::SetValue(long value)
{ count = value;  }

long Counter::GetValue()
{
        return count;
}
```

1.3.3 USO DE UNA CLASE.

Ya que se ha definido la clase, se debe definir un objeto con ella. Las variables de una clase se definen de igual manera que como se definen las variables de tipo estructura.
Para el ejemplo de la clase anterior, si quiere declarar un objeto Gente de tipo Counter, lo podría hacer asi:

```
Class Counter{
        .
        .
public:
        .
        .
}Gente;
```

O la declaración la podría hacer de la siguiente forma:

Counter Gente;

En algunos lenguajes orientados a objetos, Smalltalk en particular, la definición de una variable de clase se denomina *instanciación de la clase*.
Una instanciación es simplemente una instancia de una clase en la forma de una variable específica.

Las variables instanciadas a partir de clases son objetos.

El objeto Gente se podría usar así en un programa:

```
void main()
{
        Counter  Gente;                 //    Declaración de un objeto
        Gente.SetValue(1000);           //     Invocación a función miembro de Counter
        long value = GetValue();        //     Invocación a función miembro de Counter
}
```

La iniciación se tiene que hacer a través de sus funciones miembro, por lo que hacer lo siguiente sería un error.

```
void main()
{

        Counter  Gente;
        Gente = 1000;       //   error, la variable no esta disponible en la función main()
        long value = GetValue();
}
```

El código anterior no hace mucho, pero ilustra 2 aspectos importantes:
La declaración de un objeto dentro de una función y la invocación de funciones miembro de un objeto.

En otro ejemplo, ésta clase define un tipo llamado cola, que se utiliza para crear un objeto de tipo cola.

```
# include <iostream.h>

class cola{
        int c[100];
        int posfin, posprin;

public:
        void iniciar(void);
        void ponent(int i);
```

```
        int quitaent(void);

};
```

Cuando llega el momento de codificar realmente una función que es miembro de una clase, es preciso decir al compilador a que clase pertenece la función, calificando el nombre de la función con el nombre de la clase del cual es miembro. p.e.

```
void cola ::  ponent(int i)
{
        if(posfin>=100)
          {
             cout<<"la cola esta llena ";
        return;
           }
        posfin++;
        c[posfin] = i;
}
```

El :: se llama operador de resolución de ámbito; indica al compilador que la función ponent(int i) pertenece a la clase cola, o dicho de otra manera, ponent(int i) está dentro del ámbito de cola.

Para llamar a una función miembro desde una parte del programa que no sea parte de la clase, se debe utilizar el nombre del objeto y el operador punto. p.e.

```
Cola a, b;        //   se crean 2 objetos tipo cola.

a.iniciar();       //  llama a la función iniciar para el objeto a.
```

Consideremos el siguiente ejemplo, de un programa en C++, aunque en una aplicación real la declaración de las clases debe estar contenida en un archivo de cabecera.

```
# include <iostream.h>

class cola{
        int c[100];
        int posfin, posprin;
public:
        void iniciar(void);
        void ponent(int i);
        int quitaent(void);
};


main(void)
```

```
{
        cola a, b;
        a.iniciar();
        b.iniciar();
        a.ponent(15);
        b.ponent(39);
        a.ponent(55);
        b.ponent(19);
        cout<<a.quitaent() << " ";
        cout<<b.quitaent() << " ";
        cout<<a.quitaent() << " ";
        cout<<b.quitaent() << " ";
        return 0;
}

void cola::iniciar()
{
        posprin=posfin=0;
}

void cola::ponent(int i)
{
        if(posfin==0)
          {
              cout<<"la cola esta llena ";
        return;
           }
        posfin++;
        c[posfin] = i;

}
int cola::quitaent(void)
{
        if(posfin==posprin)
        {
          cout<<"la cola está vacía";
        return 0;
        }
        posprin++;
        return c[posprin];
}
```

1.4 CONTROL DE ACCESO A UNA CLASE.

La tarea de una clase consiste en ocultar la mayor cantidad de información posible. Por lo tanto es necesario imponer ciertas restricciones a la forma en que se puede manipular una clase. Existen 3 tipos de usuario de una clase:

1.- La clase misma.
2.- Usuarios genéricos.
3.- Clases derivadas.

Cada tipo de usuarios tiene privilegios de acceso asociados a una palabra clave:

1.- Private.
2.- Public.
3.- Protected.

Ejemplo:

```
class  controlAcceso{
        int a;
public:
        int b;
        int fi(int a);
protected:
        int c;
        float C1(float t);
};
```

Cualquier declaración que aparezca antes de cualquiera de las tres palabras clave, por default es private; así, int a; es private.

1.4.1 MIEMBROS DE LA CLASE PRIVATE.

Los miembros de la clase private tienen el mas estricto control de acceso. Solo la clase misma puede tener acceso a un miembro private. En este ejemplo nadie puede usar la clase ya que todo es private.

```
Class Privada{

        long valor;
```

```
        void F1();
        void F2();

};

void main()
{

        privada objeto1;            // Se crea objeto1 de clase privada.
        long L = &objeto.valor;     // acceso no valido por ser private.
        objeto1.F1();               // acceso no valido por ser private.
        objeto1.F2();               // acceso no valido por ser private.

}
```

Para poder tener acceso necesitaría que las funciones miembro fueran declaradas en la sección public.

1.4.2 MIEMBROS DE LA CLASE PUBLIC:

Para utilizar un objeto de una clase, usted debe tener acceso a datos miembro, a funciones miembro o a ambos. Para hacer que algunos datos o funciones sean accesibles, se declaran en la sección public.

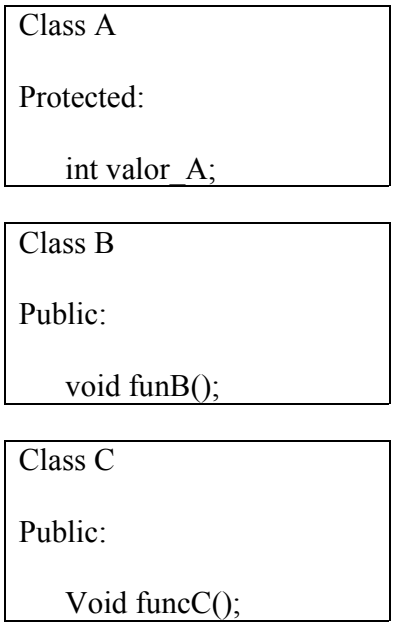
```
class Ej_public{

public:
        int variable;
        void función1();

};

void Ej_public::función1(){}
void main()
{

        Ej_public Objeto2;
        int i = Objeto2.variable;
        Objeto2.función1();

}
```

Cualquier cosa que se declara en la sección public, hace posible el acceso ilimitado a cualquier persona.

1.4.3 MIEMBROS DE LA CLASE PROTECTED.

Cuando se define una clase que se utiliza subsiguientemente como clase de base para otras clases, se puede hacer que los miembros estén accesibles solo para funciones de las clases derivadas mediante el uso de la palabra clave protected.

Considere una jerarquía de objetos como se ilustra a continuación:



La jerarquía de clase se puede expresar con código así:

```
class A{
Protected:
        int valor_A;
};

class B{
public:
        void funB();
};

class C{
public:
        void funC();
};
```

La propiedad de ser protected se extiende indefinidamente hacia abajo en un árbol de herencia, en tanto que se declare que las clases derivadas tengan clases de base public. Por ejemplo el código siguiente es aceptable.

```
void funB()
{
        valor_A = 0;
}

void funC()
{
        valor_A = 1000;
}
```

1.5 APUNTADORES COMO MIEMBROS DE DATOS.

Los miembros de datos pueden ser también apuntadores. Los apuntadores no se pueden inicializar dentro de la declaración de una clase. Si un miembro de datos apunta a un tipo de clase, el apuntador no se iniciliza, ni las construcciones son llamadas de manera automática. Ejemplo.

```
class Segunda{

        int id;

public:

        Primera* Object;
        Segunda();
        Int getnom() { return id; }

};
```

Aquí, el miembro se declara para apuntar a un elemento de otra clase llamada primera. En la construcción de la clase segunda, el espacio de almacenamiento se asigna al objeto apuntador, pero el apuntador se deja sin inicializar.

1.5.1 APUNTADORES A MIEMBROS DE DATOS DE CLASES.

Las clases no son objetos, pero aveces puede utilizarlas como si lo fueran. Un ejemplo es la declaración de un apuntador a un miembro de clase.

```
class Ejemplo{

public:

        int valor;
        int identificador;

};

void SetValue(Ejemplo& Objeto)
{

        int Ejemplo::*ip = & Ejemplo::valor;
        Objeto.*ip = 3;

}

void main()
{

        Ejemplo Objeto1;
        Ejemplo Objeto2;

        SetValue(Objeto1);
        SetValue(Objeto2);

}
```

La función SetValue() tiene la declaración inusual:

```
        Int Ejemplo::*ip = & Ejemplo::valor;
```

Esta instrucción declara la variable ip que apunta a un valor de miembro de datos int en un objeto de clase Ejemplo sin indicar un objeto especifico.

1.6 CONSTRUCTORES.

Un constructor es una función especial que es miembro de esa clase y que tiene el mismo nombre de la clase.

Es muy frecuente que una cierta parte de un objeto necesite una iniciación antes de que pueda ser utilizada; como el requisito de iniciación es tan frecuente C++ permite que los objetos se den a sí mismos valores iniciales cuando se crean. Esta iniciación automáticamente se lleva a cabo mediante el uso de una función de construcción o constructor.
Por ejemplo este es el aspecto que tiene la clase cola cuando se modifica para utilizar las iniciaciones:

```
# include <iostream.h>

class cola{
        int c[100];
        int posfin, posprin;

public:
        cola(void);             // este es el constructor de la clase cola
        void ponent(int i);
        int quitaent(void);

};
```

Obsérvese que no se especifica un tipo de dato proporcionado para el constructor cola(). En C++, los constructores pueden proporcionar valores.
La función cola() se codifica de la siguiente manera:

```
cola::cola(void)
{
        posfin=posprin=0;
        cout<<"la cola ya tiene valores iniciales \n";
}
```

La función de construcción de un objeto se invoca cuando se crea el objeto. Esto significa que se invoca cuando se ejecuta la declaración del objeto. Además, para los objetos locales, el constructor se invoca cada vez que se llega a la declaración del objeto.
Como lo dice el nombre, un constructor es una función que se utiliza para construir un objeto de una clase dada; esto puede implicar la presencia de diferentes escenarios.

1.- Creación de objetos con iniciación definida.
2.- Creación de objetos con iniciación especifica.
3.- Creación de objetos copiando otro objeto.

Cada uno de estos procesos implica un tipo diferente de constructor. Un constructor tiene el nombre de la clase a la que pertenece.

Un sub objeto es un objeto de clase que se declara dentro de otra clase. Cuando se tiene una instancia en una clase, su constructor debe crear un objeto de esa clase. Si la clase tiene sub objetos declarados en ella, el constructor tiene que invocar los constructores de estos objetos. Considere el ejemplo siguiente.

```
class counter{
        int value;
public:
        Counter() { value = 0; }
};

class Example{
        int value;
public:
        Counter cars;
        Example() { value = 0; }
};

void main()
{
        Example e;
}
```

Cuando se crea el objeto e en main(), se llama al constructor de la clase Example; en este caso, la función Example::Example() antes de ejecutar su cuerpo, invoca al constructor Counter::Counter() del sub objeto cars. Cuando se completa este constructor, se ejecuta el cuerpo de Example::Example().

1.6.1 CONSTRUCTORES PRIVATE.

Obsérvese que el constructor anterior aparece en la sección public de la clase; este no es un requisito, pero normalmente es el caso.

Un constructor private, impediría que los usuarios genéricos crearan objetos a partir de esa clase y forzarán el cumplimiento de una de las condiciones siguientes antes de que se pueda crear un objeto.

1.- Un miembro estático de la clase invoca al constructor.
2.- Una clase friend de esa clase invoca al constructor.
3.- Un objeto existente de la clase tiene una función miembro que crea nuevos objetos invocando al constructor.

1.6.2 CONSTRUCTORES CON ARGUMENTOS.

La función básica de un constructor consiste en inicializar un objeto antes de usarlo.

```
Counter(long);

Counter::Counter(long value)
{
        count = value;
}

void main()
{
        Counter object(5);
}
```

Observe los paréntesis después del nombre de la variable, que hacen que la definición del objeto se asemeje a una llamada a función. La definición del objeto es en realidad una llamada a función con argumentos.
Suponga que desea crear una clase counter que sea lo suficientemente flexible para aceptar cualquier tipo de inicialización, utilizando elementos float, long, int, cadena o incluso ningún argumento. Estas son las construcciones que se deben declarar.

```
class Counter{

public:
        Counter(int = 0);
        Counter(long);
        Counter(double);
        Counter(char *);
};

//          declaración de constructores.

Counter::Counter(long val_inic)
{
        count = val_inic;
}

Counter::Counter(double val_inic)
{
        count = val_inic;
}

Counter::Counter(char* val_inic)
{       count = atol(val_inic);     }
```

```
//        uso de los constructores.

void main()
{

        Counter Object("5");        //  Utilizando constructor char*
        Counter Object1(5);         //  Utilizando constructor int
        Counter Object2(5L);        //  Utilizando constructor long
        Counter Object3(5.0);       //  Utilizando constructor double
        Counter Object4();          //  Utilizando constructor por omisión

}
```

El compilador puede determinar automáticamente a que constructor llamar en cada caso examinando los argumentos.

1.6.3 CONSTRUCTORES PARA COPIAR OBJETOS.

Cuando se crea un objeto, a menudo no se desea inicializar ningún valor de manera específica; simplemente se desea que un objeto "sea como otro". Esto implica hacer una copia de un objeto preexistente, lo cual requiere un tipo especial de construcción, llamada en general: constructor de copia. Ejemplo.

```
class Counter{
        .
        .
        .
public:

        Counter(Counter&);
        .
        .
        .
};



Counter::Counter(Counter &referencia)
{

        count = referencia.count;

}
```

```
void main()
{

        Counter Object(5);              //  Constructor entero
        Counter Object1 = Object;       //  Constructor de copia

}
```

1.7 DESTRUCTORES.

Los destructores entran en la misma categoría que los constructores. Se utilizan para realizar ciertas operaciones que son necesarias cuando ya no se utiliza un objeto como es la liberación de memoria.

Existen algunas diferencias importantes entre los constructores y los destructores:

1.- Los destructores pueden ser virtuales, los constructores NO.
2.- A los destructores no se les puede mandar argumentos.
3.- Sólo se puede declarar un destructor para una clase dada.

El destructor se nombra como la clase pero este va precedido de un tilde (~).

Se podría escribir una clase que se encargue de manejar todas las gráficas generadas por un programa de la siguiente manera:

```
class Graphics{

public:
        Graphics();
        ~Graphics();
        void DrawCircle(int x, int y, int radio);
        void DrawDot(int x, int y);
                .
                .
};
```

El destructor se utiliza para cerrar el dispositivo gráfico y rechazar cualquier espacio de memoria asignado al objeto.

Por ejemplo vea la clase cola con su constructor y destructor (en el ejemplo de la clase cola no es necesario un destructor, pero en este caso se pone para ejemplificar su uso).

```
# include <iostream.h>

class cola{
        int c[100];
        int posfin, posprin;

public:
        cola(void);             // este es el constructor de la clase cola
        ~cola(void);            // este es el destructor de la clase cola
        void ponent(int i);
        int quitaent(void);

};
//   Función de Construcción
cola::cola(void)
{
        posfin=0;
        posprin=0;
}


// Función de destrucción.
cola::~cola(void)
{
        cout<<"la cola ha sido destruida \n";
}
```

Veamos como funcionan los constructores y destructores en la nueva versión del programa que crea una cola.

```
# include <iostream.h>

class cola{
        int c[100];
        int posfin, posprin;
public:
        cola(void);             // este es el constructor de la clase cola
        ~cola(void);            // este es el destructor de la clase cola
        void ponent(int i);
        int quitaent(void);
};
```

```
main(void)
{
        cola a, b;
        a.ponent(15);
        b.ponent(39);
        a.ponent(55);
        b.ponent(19);
        cout<<a.quitaent() << " ";
        cout<<b.quitaent() << " ";
        cout<<a.quitaent() << " ";
        cout<<b.quitaent() << " ";
        return 0;
}


//   Función de Construcción

cola::cola(void)
{
        posfin=0;
        posprin=0;
        cout<<"La cola ya tiene valores iniciales \n"
}


// Función de destrucción.

cola::~cola(void)
{
        cout<<"La cola ha sido destruida \n";
}


void cola::ponent(int i)
{
        if(posfin>=100)
           {
              cout<<"la cola esta llena ";
        return;
           }
        posfin++;
        c[posfin] = i;

}
```

```
int cola::quitaent(void)
{
        if(posfin==posprin)
        {
          cout<<"la cola está vacía";
        return 0;
        }

        posprin++;
        return c[posprin];
}
```

Este programa da como resultado lo siguiente:

```
La cola ya tiene valores iniciales.
La cola ya tiene valores iniciales.
15      39
55      19
La cola ha sido destruida.
La cola ha sido destruida.
```

1.8 CLASES AMIGAS (Palabra reservada *friend*)

A veces se necesitan 2 clases que son tan conceptualmente cercanas que usted desearía que una de ellas tuviera acceso irrestricto a los miembros de la otra. Considere la implantación de una lista asociada: necesita una clase que represente nodos individuales, y una que se encargue de la lista misma. El acceso a los miembros de la lista es a través del manejador de la misma, pero el manejador debe tener acceso absoluto a los miembros de la clase o una función. Una clase que no ha sido declarada aún se puede definir como friend de esta manera:

```
class Node{
        friend class ObjectList;
        int value;
        Node* Predecesor;
        Node* Sucesor;

public:
        void value(int i){ value = i; }
```

```
};
class ObjectList{

        Node* head;
        Node* tail;
        Node* current;
public:
        void InsertNode(Node * ) {}
        void DeleteNode(Node *) {}
        int CurrentObject(Node* node) { return node->value;}
};
```

1.8.1 FUNCIONES AMIGAS.

Es posible que una función de una clase que no sea un miembro tenga acceso a las partes privadas de esa clase, declarando que se trata de un *friend* (amigo) de esa clase. Por ejemplo amg() se declara como *friend* de la class C1

```
Class C1{
        .
        .
        .
public:
        friend void amg(void);
        .
        .
        .
};
```

Como se puede ver, la palabra reservada *friend* precede a toda la declaración de la función, que es lo que se hace en general.

La razón por la cual se permite en C++ las funciones *friend* es la de resolver situaciones en las cuales dos clases deban compartir una misma función, para así aumentar la eficiencia. Para ver un ejemplo, consideremos un programa que defina dos clases llamadas linea y recuadro. La clase linea contiene todos los datos y código necesarios para dibujar una línea horizontal discontinua de cualquier longitud, empezando en la coordenada X, Y que se indique y utilizando un color especificado, La clase recuadro contiene todo el código y los datos necesarios para dibujar un recuadro en las coordenadas especificadas para la esquina superior izquierda y para la esquina inferior derecha, y con el color que se indique. Las dos clases tienen la misma función mismo_color() para determinar si una línea y un recuadro están pintados del mismo color. Las clases se declaran según se muestra a continuación:

```
class linea;

class recuadro{
        int color;
        int xsup, ysup;
        int xinf, yinf;
public:
        friend int mismo_color(linea l, recuadro b);
        void pon_color(int c);
        void definir_recuadro(int x1, int y1, int x2, int y2);
        void mostrar_recuadro(void);
};

class linea{
        int color;
        int xinicial, yinicial;
        int longitud;
public:
        friend int mismo_color(linea l, recuadro b);
        void pon_color(int c);
        void definir_linea(int x, int y, int l);
        void mostrar_linea();
};
```

La función mismo_color(), que no es miembro de ninguna de ellas pero es friend de ambas, proporciona un valor verdadero si tanto el objeto línea como el objeto recuadro, que son sus argumentos, se dibujan del mismo color; en caso contrario, proporciona un valor nulo. La función mismo_color se muestra a continuación:

```
Int mismo_color(linea l, recuadro b)
{
        if(l.color == b.color) return 1;
        return 0;
}
```

También puede declarar una función no miembro como *friend* antes que el identificador de la función esté en el campo de acción. Por ejemplo.

```
class Node{

        friend int GetObject(Node*);
        int value;
        Node* Predecesor;
        Node* Sucesor;
```

```
public:
        void value(int i){ value = i; }
};

int GetObject(Node* n)
{
        return n->value;
}
```

1.8.2 PROPIEDADES DE LA PALABRA RESERVADA **friend.**

Las funciones y clases declaradas *friend* para otras clases gozan de privilegios especiales. Si la función FUN0() es un elemento *friend* de la clase B y la clase B se deriva de la clase A. FUN0(), tiene acceso también a los miembros de datos de las clases A, B, y C:

```
class A{
        friend class FRIEND;
        int a1;
protected:
        int a2;
public:
        int a3;
};


class B{
        int b1;
protected:
        friend class FRIEND;
        int b2;
public:
        int a3;
};
```

La clase friend puede tener acceso a todos los miembros de datos de C.

```
class C{
        int c1;
protected:
        int c2;
public:
        friend class FRIEND;
        int c3;
};
```

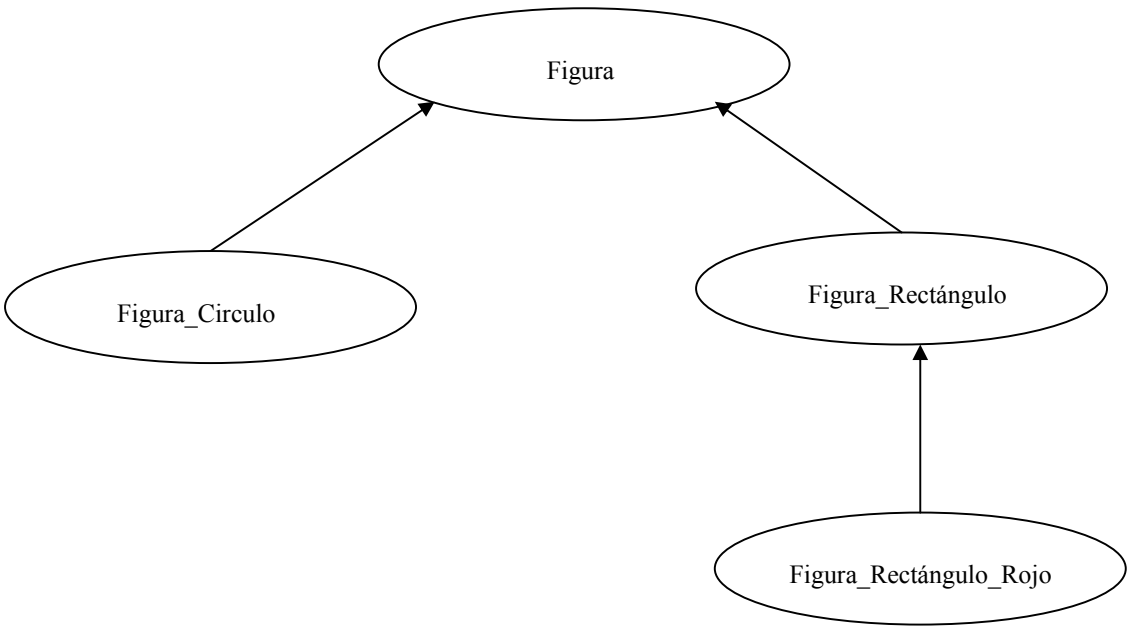
1.9 HERENCIA

La herencia es uno de los rasgos fundamentales de un lenguaje de programación orientado a objetos. En C++, la herencia se basa en permitir que una clase contenga a otra clase en su declaración; supongamos una clase Figura:

```
Class Figura{
        .
        .
public:
        .
        .
};
```

Una clase derivada Figura_Circulo se declara así.

```
Class Figura_Circulo:public Figura
{
public:
        .
private:
double x_centro, y_centr;
double radio;
};
```



La declaración general de la herencia es la que se muestra a continuación:

```
Class Nombre_de_la_clase_nueva : acceso clase_heredada{

        .
        .
        .
};
```

Aquí *acceso* es opcional, sin embargo, si está presente tiene que ser *public*, private o protected.
El uso de *public* significa que todos los elementos *public* del antecesor también serán *public* para la clase que lo hereda.

El siguiente ejemplo muestra 2 clases donde la segunda de ellas hereda las propiedades de la primera.

```
class Box{

public:

        int width, height;
        void SetWidth(int w) { width = w; }
        void SetHeight(int h) { height = h; }

};


class ColoredBox:public Box{

public:

        int color;
        void Setcolor(int c) { color = c; }

};
```

La clase Box recibe el nombre de clase base de la clase ColoredBox. Que a su vez recibe el nombre de clase derivada. La clase Colored Box se declara solo con una función, pero también hereda 2 funciones y 2 variables de su clase base. Así, se puede crear el código siguiente:

```
ColoredBox  Cb;                         // se crea una instancia de ColoredBox
void main()
{

        Cb.Setcolor(5);             //  función miembro de ColoredBox.
        Cb.SetWidth(30);            //  función heredada.
        Cb.setHeight(50);           //  función heredada.

}
```

Observe como las funciones heredadas se utilizan exactamente como si fueran miembro.

1.9.1 LIMITACIONES DE LA HERENCIA.

Cómo y cuándo se deriva una clase de otra es puramente decisión del programador. Esto puede parecer obvio, pero es una limitación. El diseñador de un programa debe decidir al momento de la compilación quien hereda qué, de quién, cómo y cuándo se lleva a cabo la herencia.

1.9.2 QUE NO SE PUEDE HEREDAR.

Tal y como en la vida real, en C++ no todo se puede transmitir a través de la herencia. Esto se puede considerar en un principio como una desventaja o limitación, pero en realidad solo algunos casos especiales inconsistentes por definición con la herencia:

1.- Constructores.
2.- Destructores.
3.- Nuevos operadores definidos por el usuario.
4.- Relaciones *friend*.

El constructor de una clase de base no puede ser invocado de manera explícita en una clase derivada como otras funciones heredadas. Considere el código siguiente:

```
class Parent{
        int value;

public:
```

Continua…

```
        Parent(){ value = 0; }
        Parent(int v){ value = 0; }

};
class Child:public Parent{
        int total;
public:
        Child(int t)  { total = t; }
        void  SetTotal(int t);
};

void Child::SetTotal(int t)
{
        Parent::Parent(i);              // Esto no se puede hacer, ya que el constructor de la
                                        // clase no es heredado como otras funciones.
        Total = t;
}
```

De manera análoga, los destructores están diseñados para ser invocados automáticamente cuando un objeto sale del campo de acción.

La relación *friend* no es heredada. Esto es similar a la vida real; los amigos de sus padres no son automáticamente amigos suyos.

1.9.3 HERENCIA MULTIPLE.

Una clase puede heredar los atributos de dos o más clases. Para lograr esto, se utiliza una lista de herencia separada mediante comas en la lista de clases base de la clase derivada. La forma General es:

```
Class Nombre_clase_derivada : lista de clases base  {
        .
        .
        .
};
```

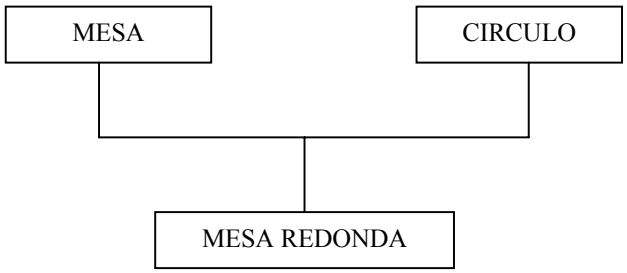
Por ejemplo en este programa Z hereda tanto a X como a Y.

```
# include <iostream.h>

class X{

protected:
```

```
        int a;

public:
        void hacer_a(int i);
};

class Y{

protected:
        int b;
public:
        void hacer_b(int i);
};

// Z hereda tanto a X como a Y

class Z : public X, public Y {

public:
        hacer_ab(void);
};


void X::hacer_a(int i)
{
        a = i;
}

void Y::hacer_b(int i)
{
        b = i;
}

int Z::hacer_ab(void)
{
        return a*b;
}


main(void)
{
        Z var;
        var.hacer_a(10);
        var.hacer_b(25);
        cout << var.hacer_ab();
        return 0;
```

```
}
```

En este ejemplo, Z tiene acceso a las partes public y protected tanto de X como de Y. Remarcando, una clase puede tener muchos padres y heredar propiedades de cada una de sus clases base. Considere crear una clase MesaRedonda, que no solo tenga las propiedades de las mesas, sino también la característica geométrica de ser redonda.



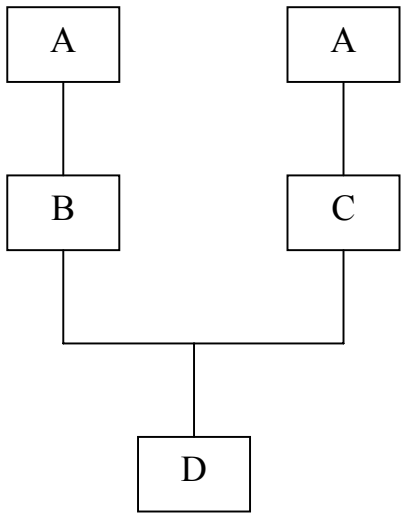

```
#include <stdio.h>

class Circle{

        float radio;
public:
        Circle(float r){ radio = r; }
        Float Area(){ return radio*radio*3.1416; }
};

class Mesa{
        float height;
public:
        Mesa(float h) { height = h; }
        float  Height() { return height; }
};

class MesaRedonda:public Mesa, public Circle{
        int color;
public:
        MesaRedonda(float h, float r, int c);
        int Color() { return color; }
};

MesaRedonda::MesaRedonda(float h, float r, int c): Circle(r),Mesa(h)
{
        color = c;
}
```

```
void main()
{
        MesaRedonda Mesa1(15.0, 3.0, 5);
        printf("\n Las propiedades de la Mesa son:");
        printf("\n Altura = %f ", Mesa.Height());
        printf("\n Area = %f ", Mesa.Area());
        printf("\n Color = %d ", Mesa.Color());

}
```

La función main() invoca las tres funciones miembro MesaRedonda::Height(), MesaRedonda::Area() y MesaRedonda::Color(). Todo sin indicar cuales son funciones heredadas y cuales no.

1.9.4 USO DE CLASES DE BASE VIRTUAL.

Las clases de base virtual se utilizan sólo en el contexto de la herencia múltiple. Dada la complejidad de relaciones que pueden surgir en un árbol de herencia construido en torno a la herencia múltiple, existen situaciones en las que el programador necesita tener cierto nivel de control sobre la forma en que se heredan las clases de base. Considere el árbol de herencia de la siguiente figura.



La clase D tiene a A como clase de base. El problema es que hay dos clases A diferentes que aparecen como clases de base de D, cada una con datos propios.

Esto se ejemplifica con el siguiente código.

```
class A{
public:
        int value;
};

class B : public A{};
class C : public A{};
class D : public B, public C {
public:
        int value() {return value; }
};
```

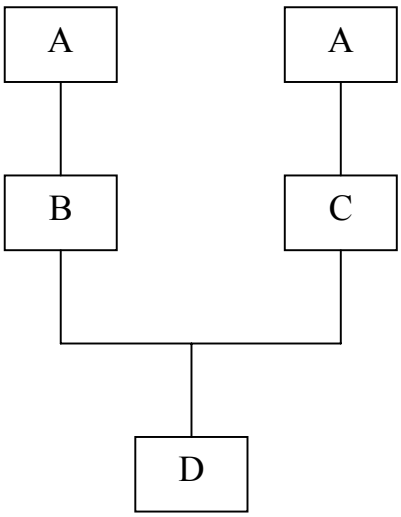
El valor miembro de acceso a la instrucción en D es ambiguo. Borland C++ genera error.

Tener múltiples copias de la misma clase de base en árbol de herencia no sólo es confuso, sino que puede ser un desperdicio de espacio de almacenamiento.

Declarar una base *virtual* resuelve el problema. Obliga al compilador a admitir sólo una copia de la clase de base dada en la declaración de una clase derivada. Por lo tanto el ejemplo anterior lo podemos corregir de la siguiente manera:

```
class B : public virtual A{};
class C : public virtual A{};
class D : public B, public C {
public:
        int value() {return value; }
};
```

y el árbol de herencia quedaría como realmente lo deseamos.



1.10 SOBRECARGA.

Una de las maneras que tiene el C++ de llegar al polimorfismo es a través de la sobrecarga de funciones. En C++ , dos o mas funciones pueden compartir un nombre, siempre y cuando en la declaración, sus parámetros sean diferentes.

Sobrecarga se refiere a la práctica de cargar una función con más de un significado. Básicamente, el término expresa que se cargan uno o más identificadores de función sobre un identificador previo.

1.10.1 PORQUE USAR LA SOBRECARGA.

La sobrecarga no es un concepto nuevo en los lenguajes de programación, por ejemplo el operador = está sobrecargado en muchos lenguajes de alto nivel y se utilizan en instrucciones de asignación y en expresiones condicionales como:

```
a = b;

if( a = b)
```

La sobrecarga otorga flexibilidad, permite a las personas utilizar código con menos esfuerzo, ya que extiende operaciones que son conceptualmente similares en naturaleza.

1.10.2 SOBRECARGA DE FUNCIONES.

Las funciones sobrecargadas se distinguen por el número y tipo de sus argumentos.

El tipo de retorno no se utiliza para distinguir funciones sobrecargadas, por lo tanto las funciones:

```
void Muestra(int q);
long Muestra(int q);
```

No son distinguibles y producen un error del compilador.

Cualquier nombre de función puede ser sobrecargada en C++, pero la sobrecarga solo funciona dentro de un campo de acción dado.
Cuando se declara en una clase más de una función miembro con el mismo nombre, se dice que el nombre de la función está sobrecargado en esa clase, y su campo de acción será en el ámbito de esa clase.

```
class Ejemplo{

        int value;

public:

        void value(int v) { value = v; }
        int value() { return value; }

};

void main()
{

        Ejemplo Ee;
        Ee.value(3);
        Int i = Ee.value();

}
```

Este código demuestra que la clase Ejemplo tiene 2 funciones sobrecargadas: una función para escribir y una para leer una variable.

Las funciones sobrecargadas necesitan diferir en una u otra o en las dos formas siguientes:

1.- Las funciones deben contener un número de argumentos diferente.
2.- Cuando menos uno de los argumentos debe ser diferente.

Considere el siguiente programa en el cual la función al_cuadrado se sobrecarga 3 veces.

```
# include<iostream.h>

int al_cuadrado(int i);
double al_cuadrado(double d);
long al_cuadrado(long l);

main(void)
{
        cout <<  al_cuadrado(10)    <<" \n";
        cout <<  al_cuadrado(1.25)  <<" \n";
        cout <<  al_cuadrado(9L)    <<" \n";

        return 0;
}

int al_cuadrado(int i)
{
cout<<" función al_cuadrado con parámetro entero"

        return i*i;
}

double al_cuadrado(double d)
{
cout<<" función al_cuadrado con parámetro doble"

        return d*d;
}


long al_cuadrado(long l)
{
cout<<" función al_cuadrado con parámetro largo"

        return l*l;
}
```

La ventaja de sobrecargar las funciones es que permite acceder a conjuntos de funciones que están relacionadas utilizando un solo nombre. En el programa anterior se crean 3 funciones similares que se llaman, al_cuadrado(), y cada una de las cuales regresa el cuadrado de su argumento; en cierto sentido, la sobrecarga de funciones permite crear un nombre genérico para alguna operación, y el compilador resuelve que función es la adecuada para llevar a cabo la operación.

Así, al_cuadrado(), representa la acción general que se realiza; el programador solo necesita recordar la acción general que se lleva a cabo, por lo tanto al aplicar el polimorfismo se han reducido a una las 3 cosas que había que recordar. Aunque este ejemplo es bastante trivial,

si expande el concepto se puede ver que el polimorfismo puede ayudarnos a entender programas muy complejos.

Para sobrecargar la función de construcción de una clase, solo hay que declarar las diferentes formas que tiene que adoptar y hay que definir su acción con respecto a esas formas. Por ejemplo el programa siguiente declara una clase llamada temporizador, que se comporta como un temporizador descendente. Cuando se crea un objeto del tipo temporizador, se le da un valor inicial de la hora. Cuando se invoca a la función ejecutar(), el temporizador cuenta hasta llegar a cero, y hace sonar el timbre. En ese ejemplo, se ha sobrecargado el constructor para especificar la hora como un entero, como una cadena, o como dos enteros que especifican los minutos y los segundos.

```
# include <iostream.h>
# include <stdlib.h>
# include <time.h>

class temporizador{
        int segundos;
public:
                        //  Se especifican los segundos como una cadena.
        temporizador(char *t) {segundos = atoi(t);}
                        //  Se especifican los segundos como un entero
        temporizador(int t) {segundos = t;}
                        //  Se especifica la hora en minutos y segundos
        temporizador(int min, int seg) {segundos =  min* 60 + seg; }
        void ejecutar(void);
};


void temporizador::ejecutar(void)
{
        clock_t t1, t2;
        t1 = t2 = clock()/CLK_TCK;
        while(segundos)  {
                if(t1/CLK_TCK+1 <= (t2=clock()) /CLK_TCK){
                  segundos --;
                  t1 = t2;
                }
        }
        cout << "\a";   // toca el timbre
}


main(void)
```

```
{
        temporizador a(10), b("20"), c(1, 10);
        a.ejecutar();    // cuenta 10 segundos
        b.ejecutar();    // cuenta 20 segundos
        c.ejecutar();    //cuenta 1 minuto, 10 segundos

        return 0;
}
```

Como se puede ver, cuando a, b, y c se crean dentro de main(), se les dan valores iniciales utilizando los tres métodos diferentes que admiten las funciones de construcción sobrecargadas.

1.10.3 CONSTRUCTORES SOBRECARGADOS.

Uno de los usos más comunes de la sobrecarga de funciones es con los constructores. La razón es que cuando se instancía una clase, se deben conservar las cosas lo más flexibles que sea posible; de modo que los usuarios pueden realizar diferentes clases de instancias.

Considere una clase ventana desplegable en una interfaz gráfica de usuario.

```
PopupWindow Window;                 // Genera una ventana con parámetros por
                                    // omisión.
PopupWindow Window_1(x, y);         // Genera una ventana con coordenadas
                                    //  específicas.
PopupWindow Window_2(x, y, width, Height);      // Genera una ventana con
                                                // dimensiones controladas.
PopupWindow Window_3 = Window_2;    // Genera una ventana igual a la
                                    // anterior.
```

La implantación de esta clase podría parecerse al siguiente código.

```
class PopupWindow{

        Int x, y, Width, Height;
public:
        PopupWindow();
        PopupWindow(int, int);
        PopupWindow(int, int, int, int);
        PopupWindow(PopupWindow&);

};
```

```
PopupWindow:: PopupWindow()
{
        x = y = 100;
        Widht = Heigth = 100;
}


PopupWindow:: PopupWindow(int px, int py)
{
        x = px;
        y = py;
        Widht = Heigth = 100;
}

PopupWindow:: PopupWindow(int px,int py, int w, ,int h)
{
        x = px;
        y = py;
        Widht = w;
        Heigth = h;
}


PopupWindow:: PopupWindow(PopupWindow& pw)
{
        x = pw.x;
        y = pw.y;
        Widht = pw.Widht;
        Heigth = pw.Height;
}
```

La clase utiliza cuatro funciones sobrecargadas que realizan el trabajo.

Con constructores sobrecargados, usted puede permitir que el usuario especifique qué variables han de ser inicializadas de manera explícita y cuales deben asumir valores definidos.

1.10.4 SOBRECARGA DE OPERADORES.

Otra forma en que se logra el polimorfismo en C++ es mediante la sobrecarga de operadores. En general se puede sobrecargar cualquiera de los operadores de C++ definiendo lo que significa con respecto a una cierta clase.

Aunque los operadores están asociados comúnmente con operaciones matemáticas o lógicas, simplemente son una notación alternativa para una llamada a una función

La sobrecarga de operadores se utiliza en otros lenguajes de programación, pero sin un nombre especial. En algunos lenguajes como Pascal, es posible hacer lo siguiente:

```
StructureA := structureB + structureC;
```

Lo que da lugar a una adición byte por byte de las estructuras b y c que se copiarán en la estructura a. Esta sintaxis implica que el operador de adición está sobrecargado para estructuras, aunque con ciertas reglas de apego a tipos. En el ejemplo anterior se utiliza también un operador de asignación sobrecargado, ya que la instrucción y no una asignación escalar definida, fue la que activó una operación de copia de estructuras.

1.10.4.1 OPERADORES COMO LLAMADAS A FUNCION.

Hay dos formas en las que se pueden implantar operadores para objetos de clase: como funciones miembro y como amigos. Un operador unario aplicado a un objeto es equivalen te a una llamada a una función. Dados un objeto W y un operador unario @ , la expresión @W es equivalente a las llamadas a funciones:

```
W.operator@()          //  uso de un operador con función miembro.
operator@(W)           //  uso de un operador con función friend.
```

Un operador binario aplicado a los objetos es equivalente también a una llamada a función. Dados los objetos X y Y, y un operador @, la expresión X @ Y es equivalente a las llamadas a funciónes:

```
X.opertor@(Y)          // uso de un operador con función miembro.
Operator@(X, Y)        // uso de un operador con función friend.
```

El código anterior demuestra que los operadores pueden invocar dos funciones diferentes; una función que es miembro y otra que es amigo.

1.10.4.2 OPERADORES SOBRECARGADOS COMO FUNCIONES MIEMBRO.

Las funciones que implantan operadores son un tanto inusuales. Para comenzar, sus nombres deben comenzar con la cadena *operator*, seguida de los caracteres que representan el operador que se implanta. Por ejemplo la función miembro para implantar el operador de

adición tendría que llamarse *operator+* . La segunda restricción se aplica al número de argumentos que pueden tomar estas funciones. Las funciones miembro que implantan operadores unarios no deben tomar argumentos, en tanto que las que implantan operadores binarios pueden tomar solo un argumento.

El siguiente código muestra operadores sobrecargados implantados como funciones miembro.

```
class Counter{

public:
        int value;
        Counter(int i) { value = i; }
        Counter operator!();                    //      operador unario.
        Counter operator+(Counter & c);         //      operador binario.

};
Counter Counter::operator!()
{
        return Counter( ! value );
}


Counter Counter::operator+( Counter & c)
{
        return Counter( value + c.value);
}



void main()
{

        Counter c1(3), c2(5);           //      se crean 2 objetos tipo counter.
        c1 =  ! c1;                     //      se aplica el operador unario
        c1 = c1 + c2;                   //      se aplica el operador binario

}
```

El uso de los operadores en la función main() es completamente intuitivo, y no requiere que el usuario conozca los detalles de la implantación de clases para averiguar cual será el recultado de las operaciones.

La restricción en el número de argumentos (solamente uno), limita las posibilidades; pero esto lo podemos resolver sobrecargando el mismo operador tantas veces como sea necesario, vea un ejemplo de como se sobrecarga el operador de adición tres veces.

```
class M{

public:
        int value;
        M(int i) { value = i; }
        M operator+(M&  m);
        M operator+(int i);
        M operator+(double d);
};

M M::operator+(M&  m)
{
        return M(value + m.value);
}


M M::operator+(int i)
{
        return M(value + i);
}

M M::operator+(double d)
{
        returnM(value + d);
}


void main()
{
        M m1(3), m2(10);        //      se crean dos objetos de clase M.
        m1 = m1 + m2;           //      uso M::operator+(M&)
        m1 = m2 + 200;          //      uso M::operator+(int)
        m1 = m2 + 3.14159;      //      uso M::operator+(double)
}
```

1.10.4.3 OPERADORES SOBRECARGADOS COMO FUNCIONES FRIEND.

En el siguiente ejemplo el operador + se declara como función friend y el operador = como función miembro de la clase X, ya que el operador = solo puede sobrecargarse como función miembro, lo que implica que el operador = global no puede ser sobrecargado. Los diseñadores del lenguaje decidieron que permitir que las funciones friend cambiaran el significado del operador de asignación causaría mas problemas de los que resolvería.

```
class  X{
```

```
        friend X operator+(X&, X&);
public:
        int value;
        X(int i) { value = i; }
        X& operator = (X&);
};


X& X::operator = (X& b)
{
        value = b.value;
        return *this;
}


X operator+(X& a, X& b)
{
        return X(a.value + b.value);
}

void main()
{

        X g(2), h(5), i(3);
        G = h +h +i;

}
```

En general, los operadores friend sobrecargados se comportan de manera muy similar a las funciones miembro.

1.11 POLIMORFISMO

El origen del término polimorfismo es simple: proviene de las palabras griegas *poly* (muchos) y *morphos* (forma) multiforme. El polimorfismo describe la capacidad del código C++ de comportarse de diferentes maneras dependiendo de situaciones que se presenten al momento de la ejecución.

El concepto de polimorfismo es crucial para la programación orientada a objetos. En su concepción relativa a C++, el término polimorfismo se utiliza para describir el proceso mediante el cual se puede acceder a diferentes implementaciones de una función utilizando el mismo nombre. Por esta razón el polimorfismo se define a veces mediante la frase “una interface métodos múltiples”. Esto significa que en general se puede acceder a toda una

clase de operaciones de la misma manera, aunque las acciones concretas que estén asociadas a cada una de las operaciones pueda ser diferente.

En C++, el polimorfismo se admite tanto en el momento de la ejecución como en el momento de la compilación. La sobrecarga de operadores y de funciones es un ejemplo de polimorfismo en el momento de la compilación. Sin embargo, aunque la sobrecarga de operadores y de funciones son muy potentes, no pueden llevar a cabo todas las tareas que requiere un verdadero lenguaje orientado a objetos. Por tanto, C++ permite también el polimorfismo en el momento de la ejecución mediante el uso de clases derivadas y de funciones virtuales.

1.11.1 FUNCIONES VIRTUALES.

El polimorfismo en el momento de la ejecución se consigue mediante el uso de tipos derivados y funciones virtuales. En pocas palabras, una función virtual es una función que se declara como virtual en una clase base y que se define en una o más clases derivadas.

Lo que hace especiales a las funciones *virtual* es que cuando se accede a una de ellas utilizando un puntero de clase base señala a un objeto de clase derivada, C++ determina qué función debe llamar en el momento de la ejecución, basándose en el tipo del objeto al cual apunta. Por tanto, si apunta a diferentes objetos, se ejecutan versiones diferentes de la función *virtual*.

Como ejemplo examine el siguiente código:

```
#include<iostream.h>

class Base {
public:
        virtual void quien()  { cout << "Base \n"; }
};

class primera_deriv : public Base {
public:
        void quien() { cout << " Primera derivación \n"; }
};


class seguna_deriv : public Base {
public:
        void quien() { cout << " Segunda derivación \n"; }
};
```

```
main(void)
{
        Base obj_base;
        Base *p;
        Primera_deriv   obj_primera;
        Segunda_deriv  obj_segunda;
        p = &obj_base;
        p->quien();
        p = &obj_primera;
        p->quien();
        p = &obj_segunda;
        p->quien();
        return 0;
}
```

El programa produce la siguiente salida.

Base
Primera derivación
Segunda derivación.

La clave de la utilización de funciones virtual para lograr el polimorfismo en el momento de la ejecución es que se debe acceder a esas funciones mediante el uso de un puntero declarado como puntero de la clase base.

Parte de la clave para aplicar con éxito el polimorfismo consiste en comprender que la base y las clase derivadas forman una jerarquía, que va desde la mayor generalización a la menor. Por tanto la clase base cuando se utiliza correctamente, proporciona todos los elementos que puede utilizar directamente una clase derivada, mas aquellas funciones que la clase derivada debe implementar por sí misma. Sin embargo dado que la forma de la interface está determinada por la clase base todas las clases derivadas van a compartir esa interface .

El código siguiente utiliza la clase figura para derivar dos clase concretas llamadas cuadro y triángulo.

```
#include<iostream.h>

class Figura{
protected:
        double x, y:
```

```
public:
        void pon_dim(double I, double j) { x = I; y = j; }
        virtual void mostrar_area() { cout << "función no implementada \n";}
};


class triángulo : public Figura {
public:
        void mostrar_area() { cout<<"Triangulo de altura   " << x   <<" y base "
                                        << y  << " y área " x * y * 0.5; }
};

class cuadrado : public Figura {
public:
        void mostrar_area() { cout << "Cuadrado de lado " << x  <<" por " <<  y
                                        << "área  = " << x * y <<"\n"; }
};


main (void )
{
        Figura *p;
        triangulo trian;
        cuadrado cuad;
        p = &trian;
        p->pon_dim(10.0, 5.0);
        p->mostrar_area();
        p = &cuad;
        p->pon_dim(10.0, 5.0);
        p->mostrar_area();
        return 0;
}
```

Como se puede ver al examinar este programa, la interface de cuadrado y de triángulo es la misma, aunque cada uno de ellos proporcione sus propios métodos para calcular el área de cada uno de sus objetos.