PROGRAMACIÓN ORIENTADA A OBJETOS CON C++.

Notas de clase.

Elaborado por:
Carlos Alberto Fernández y Fernández
Instituto de Electrónica y Computación
Universidad Tecnológica de la Mixteca.

Contenido

Características de C++.

Comenzaremos con algunas características de C++ que no tienen que ver directamente con la programación orientada a objetos.

Comentarios en C++.

Los comentarios en C son:

```
/*  comentario en C     */
```

En C++ los comentarios pueden ser además de una sola línea:

```
// este es un comentario en C++
```

Acabando el comentario al final de la línea, lo que quiere decir que el programador no se preocupa por cerrar el comentario.

Flujo de entrada/salida

En C, la salida y entrada estándar estaba dada por printf y scanf principalmente (o funciones similares) para el manejo de los tipos da datos simples y las cadenas. En C++ se proporcionan a través de la librería `iostream.h`, la cual debe ser insertada a través de un `#include`. Las instrucciones son:

- cout Utiliza el flujo salida estándar
  Que se apoya del operador <<, el cual se conoce como *operador de inserción de flujo* "colocar en"

- cin Utiliza el flujo de entrada estándar.
  Que se apoya del operador >>, conocido como *operador de extracción de flujo* "obtener de"

Los operadores de inserción y de extracción de flujo no requieren cadenas de formato (%s, %f), ni especificadores de tipo. C++ reconoce de manera automática que tipos de datos son extraídos o introducidos.

En el caso de el operador de extracción (>>) no se requiere el operador de dirección &.

De tal forma un código de desplegado con printf y scanf de la forma:

```
printf("Número: ");
scanf("%d", &num);
printf("El valor leído es: " %d\n", num);
```

Sería en C++ de la siguiente manera:

```
cout << "Número";
cin >> num;
cout << "El valor leído es: " << num << '\n';
```

Funciones en línea.

Las funciones en línea, se refiere a introducir un calificador *inline* a una función de manera que le sugiera al compilador que genere una copia del código de la función en lugar de la llamada.

Ayuda a reducir el número de llamadas a funciones reduciendo el tiempo de ejecución en algunos casos, pero en contraparte puede aumentar el tamaño del programa.

A diferencia de las macros, las funciones *inline* si incluyen verificación de tipos y son reconocidas por el depurador.

Las funciones *inline* deben usarse sólo para funciones chicas que se usen frecuentemente.

El compilador desecha las solicitudes *inline* para programas que incluyan un ciclo, un *switch* o un *goto*. Tampoco si no tienen *return* (aunque no regresen valores) o si contienen variables de tipo *static*. Y lógicamente no genera una función *inline* para funciones recursivas.

Declaración:

```
inline <declaración de la función>
```

Ejemplo:

```
inline float suma (float a, float b) {
    Return a+b;
}

inline int max( int a, int b) {
    return (a > b) ? a :  b;
}
```

Notas: Las funciones *inline* tienen conflictos con los prototipos, así que deben declararse completas sin prototipo en el archivo .h. Además, si la función hace uso de otra función (cosa que puede anular el inline) en donde se expanda la función debe tener los include correspondientes a esas funciones utilizadas.

Declaraciones de variables.

Mientras que en C, las declaraciones deben ir en la función antes de cualquier línea ejecutable, en C++ pueden ser introducidas en cualquier punto, con la condición de que la declaración esté antes de la utilización de lo declarado.

También puede declararse una variable en la sección de inicialización de la instrucción *for*, pero es incorrecto declarar una variable en la expresión condicional del *while*, *do-while*, *for*, *if* o *switch*.

Ejemplo:

```
#include <iostream.h>
#include <conio.h>

main() {
//   int i=0;
     clrscr();

//   int j=10;
//    cout<<j<<'\n';
     for (int i=1; i<10; i++){
          int j=10;
          cout<<i<<" j: "<<j<<'\n';
    }

    cout<<"\ni al salir del ciclo: "<<i;
     getch();

/*   while((int h=1) <5){    Incorrecto
          cout<<"hola"
          h++;
     }

     switch(int i=10){         Incorrecto
          case 1: cout<<"exito";
     }*/
}
```

El alcance de las variables en C++ es **por bloques**. Una variable es vista a partir de su declaración y hasta la llave } del nivel en que se declaró. Lo cual quiere decir que las instrucciones anteriores a su declaración no pueden hacer uso de la variable ni después de finalizado el bloque.

Operador de resolución de alcance.

Se puede utilizar el operador de resolución de alcance **::** se refiere a una variable (*variable, función, tipo, enumerador u objeto)*, con un alcance de archivo (variable global).

Esto le permite al identificador ser visible aún si el identificador se encuentra oculto.

Ejemplo:

```
float h;

void g(int h) {
     float a;
     int b;

     a=::h;   // a se inicializa con la variable global h

     b=h;// b se inicializa con la variable local h
}
```

Valores por Default.

Las funciones en C++ pueden tener valores por default. Estos valores son los que toman los parámetros en caso de que en una llamada a la función no se encuentren especificados.

Los valores por omisión deben encontrarse en los parámetros que estén más a la derecha. Del mismo modo, en la llamada se deben empezar a omitir los valores de la extrema derecha.

Ejemplo:

```
#include <iostream.h>
#include <conio.h>

int punto(int=5, int=4);

main () {

     cout<<"valor 1: "<<punto()<<'\n';
     cout<<"valor 2: "<<punto(1)<<'\n';
     cout<<"valor 3: "<<punto(1,3)<<'\n';
     getch();
}

int punto( int x, int y){

     if(y!=4)
          return y;
     if(x!=5)
          return x;
     return x+y;
}
```

C++ no permite la llamada omitiendo un valor antes de la extrema derecha de los argumentos:

punto(, 8);

Otro ejemplo de valores o argumentos por default:

```
#include <iostream.h>
#include <conio.h>

int b=1;
int f(int);
int h(int x=f(b));    // argumento default f(::b)

void main () {
     clrscr();
     b=5;
     cout<<b<<endl;
     {
          int b=3;
          cout<<b<<endl;
          cout<<h();        //h(f(::b))
     }
     getch();
}

int h(int z){
     cout<<"Valor recibido: "<<z<<endl;
     return z*z;
}
int f(int y){
     return y;
}
```

Parámetros por referencia.

En C todos los pasos de parámetros son por valor, aunque se pueden enviar parámetros "por referencia" al enviar por valor la dirección de un dato (variable, estructura, objeto), de manera que se pueda accesar directamente el área de memoria del dato del que se recibió su dirección.

C++ introduce parámetros por referencia **reales**. La manera en que se definen es agregando el símbolo & de la misma manera que se coloca el *: después del tipo de dato en el prototipo y en la declaración de la función.

Ejemplo:

```
// Comparando parámetros por valor, por valor con apuntadores
//("referencia")
// y paso por referencia real

#include <iostream.h>
#include <conio.h>

int porValor(int);
void porApuntador(int *);
void porReferencia( int &);

void main() {
    clrscr();

    int x=2;
    cout << "x= " << x << " antes de llamada a porValor \n"
        << "Regresado por la función: "<< porValor(x)<<endl
        << "x= " << x << " después de la llamada a
porValor\n\n";

    int y=3;
    cout << "y= " << y << " antes de llamada a
porApuntador\n";
    porApuntador(&y);
    cout << "y= " << y << " después de la llamada a
porApuntador\n\n";
```

```
    int z=4;
    cout << "z= " << z << " antes de llamada a porReferencia
\n";
    porReferencia(z);
    cout<< "z= " << z << " después de la llamada a
porValor\n\n";

    getch();
    return;
}

int porValor(int valor){
    return valor*=valor;          //parámetro no modificado
}

void porApuntador(int *p){
    *p *= *p;         // parámetro modificado
}

void porReferencia( int &r){
    r *= r;      //parámetro modificado
}
```

Notar que no hay diferencia en el manejo de un parámetro por referencia y uno por valor, lo que puede ocasionar ciertos errores de programación.

Variables de referencia.

También puede declararse una variable por referencia que puede ser utilizada como un seudónimo o alias. Ejemplo:

```
    int max=1000, &sMax=max; //declaro max y sMax es un alias
de max
    sMax++;      //incremento en uno max a través de su alias.
```

Esta declaración no reserva espacio para el dato, pues es como un apuntador pero se maneja como una variable normal.

Supuestamente no se permite reasignar la variable por referencia a otra variable, pero *Borland C++* si lo permite.

Ejemplo:

```
// variable por referencia

#include <iostream.h>
#include <conio.h>

void main() {
     clrscr();

     int x=2, &y=x, z=8;

     cout << "x= "<<x <<endl
          <<"y= "<<y<<endl;

     y=10;
     cout << "x= "<<x <<endl
          <<"y= "<<y<<endl;

     &y=&z;
     cout << "z= "<<z <<endl
          <<"y= "<<y<<endl;

     y++;
     cout << "z= "<<z <<endl
          <<"y= "<<y<<endl;

     getch();
     return;
}
```

Asignación de memoria en C++

En el **ANSI C**, se utilizan *malloc* y *free* para asignar y liberar dinámicamente memoria:

```
float *f;
f = (float *) malloc(sizeof(float));
. . .
free(f);
```

Se debe indicar el tamaño a través de *sizeof* y utilizar una máscara (*cast*) para designar el tipo de dato apropiado.

En **C++**, existen dos operadores para asignación y liberación de memoria dinámica: *new* y *delete*.

```
float *f;
f= new float;
. . .
delete f;
```

El operador *new* crea automáticamente un área de memoria del tamaño adecuado. Si no se pudo asignar le memoria se regresa un apuntador nulo (*NULL* ó 0). Nótese que en C++ se trata de operadores que forman parte del lenguaje, no de funciones de librería.

El operador *delete* libera la memoria asignada previamente por *new*. No se debe tratar de liberar memoria previamente liberada o no asignada con *new*.

Es posible hacer asignaciones de memoria con inicialización:

```
int *max= new int (1000);
```

También es posible crear arreglos dinámicamente:

```
char *cad;
cad= new char [30];
. . .
delete [] cad;
```

Usar *delete* **sin** los corchetes puede no liberar adecuadamente la memoria, sobre todo si son elementos de un tipo definido por el usuario.

Ejemplo 1:

```
#include <iostream.h>

main() {
    int *p,*q;

    p= new int; //asigna memoria

    if(!p) {
        cout<<"No se pudo asignar memoria\n";
        return 0;
    }
    *p=100;
    cout<<endl<< *p<<endl;

    q= new int (123); //asigna memoria
    cout<<endl<< *q<<endl;

    delete p; //libera memoria
//  *p=20;       Uso indebido de p pues ya se libero
//  cout<<endl<< *p<<endl; la memoria
    delete q;
}
```

Ejemplo 2:

```
#include <iostream.h>

void main() {
    float *ap, *p=new float (3) ;
    const int MAX=5;
    ap= new float [MAX]; //asigna memoria
    for(int i=0; i<MAX; i++)
        ap[i]=i * *p;
    for(i=0; i<MAX; i++)
        cout<<ap[i]<<endl;
}
```

Ejemplo 3:

```
#include <iostream.h>

typedef struct {
    int n1,
        n2,
        n3;
}cPrueba;

void main() {
    cPrueba *pr1, *pr2;

    pr1= new cPrueba;
    pr1->n1=11;
    pr1->n2=12;
    pr1->n3=13;

    pr2= new cPrueba(*pr1);
    delete pr1;

   cout<< pr2->n1<<" "<<pr2->n2 <<" "<<pr2->n3<<endl;

    delete pr2;
}
```

Plantillas o "templates"

Cuando las operaciones son idénticas pero requieren de diferentes tipos de datos, podemos usar lo que se conoce como *templates* o plantillas de función.

El termino de plantilla es porque el código sirve como base (o plantilla) a diferentes tipos de datos. C++ genera al compilar el código objeto de las funciones para cada tipo de dato involucrado en las llamadas

Las definiciones de plantilla se escriben con la palabra clave *template*, con una lista de parámetros formales entre < >. Cada parámetro formal lleva la palabra clave *class*.

Cada parámetro formal puede ser usado para sustituir a: tipos de datos básicos, estructurados o definidos por el usuario, tipos de los argumentos, tipo de regreso de la función y para variables dentro de la función.

Ejemplo 1:

```
#include <iostream.h>
#include <conio.h>

template <class T>
T mayor(T x, T y)
{
  return (x > y) ? x : y;
};

void main(){
    int a=10, b=20, c=0;
    float x=44.1, y=22.3, z=0 ;

    clrscr();
    c=mayor(a, b);
    z=mayor(x, y);
    cout<<c<<" "<<z<<endl;

//  z=mayor(x,b); error no hay mayor( float, int)
```

```
//   z=mayor(a, y); ""    ""    "" ""   (int, float)
     getch();
}
```

Consideraciones:

- Cada parámetro formal debe aparecer en la lista de parámetros de la función al menos una vez.
- No puede repetirse en la definición de la plantilla el nombre de un parámetro formal.
- Tener cuidado al manejar mas de un parámetro en los templates.

Ejemplo 2:

```
//prueba de templates o plantillas
#include <iostream.h>
#include <conio.h>

template <class T>
void desplegaArr(T arr[], const int cont, T pr)
{
     for(int i=0; i<cont; i++)
          cout<< arr[i] << " ";
     cout<<endl;
     cout<<pr<<endl;
}

void main() {
     const int contEnt=4, contFlot=5, contCar=10;
     int ent[]={1,2,3,4};
     float flot[]={1.1, 2.2, 3.3, 4.4, 5.5};
     char car[]={"Plantilla"};

     cout<< "Arreglo de enteros:\n";
     desplegaArr(flot, contFlot,(float)3.33);

     cout<< "Arreglo de caracteres:\n";
     desplegaArr(car, contCar, 'Z');

     cout<< "Arreglo de enteros:\n";
     desplegaArr(ent, contEnt, 99);
```

```
    getch();
}
```

Ejemplo 3:

```
#include <iostream.h>
#include <conio.h>

template <class T, class TT>
T mayor(T x, TT y)
{
  return (x > y) ? x : y;
};

void main(){

    int a=10, b=20, c=0;
    float x=44.1, y=22.3, z=0 ;

    clrscr();
    c=mayor(a, b);
    z=mayor(x, y);

    cout<<c<<" "<<z<<endl;
//sin error al aumentar un parámetro formal.
    z=mayor(x,b);
    cout<<z<<endl;
    z=mayor(a,y); //regresa entero pues a es entero (tipo T es
entero para
    cout<<z<<endl;       // este llamado.

    z=mayor(y, a);
    cout<<z<<endl;
    c=mayor(y, a);//regresa flotante pero la asignaci$n lo
corta en entero.
    cout<<c<<endl;
    getch();
}
```

Introducción a la programación orientada a objetos.

Programación no estructurada.

Comúnmente, las personas empiezan a aprender a programar escribiendo programas pequeños y sencillos consistentes en un solo programa principal.

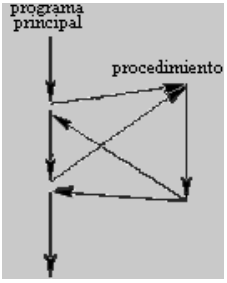
Aquí "programa principal" se refiere a una secuencia de comandos o instrucciones que modifican datos que son a su vez globales en el transcurso de todo el programa.

Programación procedural.

Con la programación procedural se pueden combinar las secuencias de instrucciones repetitivas en un solo lugar.

Una llamada de procedimiento se utiliza para invocar al procedimiento.

Después de que la secuencia es procesada, el flujo de control procede exactamente después de la posición donde la llamada fue hecha

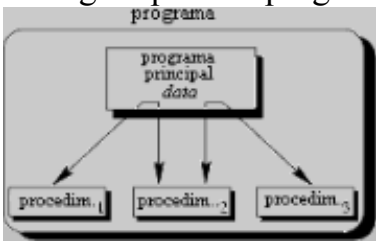


Al introducir parámetros, así como procedimientos de procedimientos (subprocedimientos) los programas ahora pueden ser escritos en forma más estructurada y con menos errores.

Por ejemplo, si un procedimiento ya es correcto, cada vez que es usado produce resultados correctos. Por consecuencia, en caso de errores, se puede reducir la búsqueda a aquellos lugares que todavía no han sido revisados.

De este modo, un programa puede ser visto como una secuencia de llamadas a procedimientos . El programa principal es responsable de pasar los datos a las llamadas individuales, los datos son procesados por los procedimientos y, una vez que el programa ha terminado, los datos resultantes son presentados.

Así, el flujo de datos puede ser ilustrado como una gráfica jerárquica, un árbol, como se muestra en la figura para un programa sin subprocedimientos.



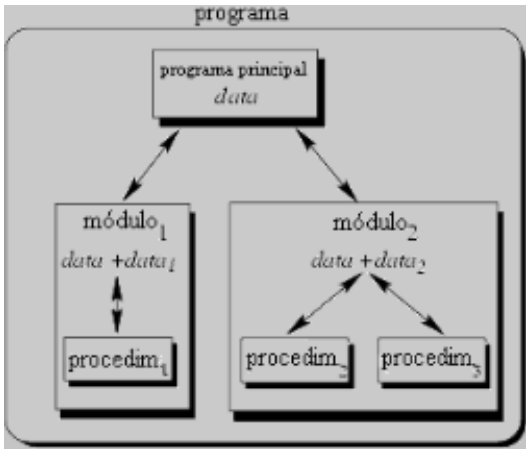
Programación modular.

En la programación modular, los procedimientos con una funcionalidad común son agrupados en módulos separados.

Un programa por consiguiente, ya no consiste solamente de una sección. Ahora está dividido en varias secciones más pequeñas que interactúan a través de llamadas a procedimientos y que integran el programa en su totalidad.

Cada módulo puede contener sus propios datos. Esto permite que cada módulo maneje un estado interno que es modificado por las llamadas a procedimientos de ese módulo.

Sin embargo, solamente hay un estado por módulo y cada módulo existe cuando más una vez en todo el programa.



Datos y Operaciones separados

La separación de datos y operaciones conduce usualmente a una estructura basada en las operaciones en lugar de en los datos : *Los Módulos agrupan las operaciones comunes en forma conjunta.*

Al programar entonces se usan estas operaciones proveyéndoles explícitamente los datos sobre los cuáles deben operar.

La estructura de módulo resultante está por lo tanto orientada a las operaciones más que sobre los datos. Se podría decir que las operaciones definidas especifican los datos que serán usados.

En la programación orientada a objetos, la estructura se organiza por los datos.

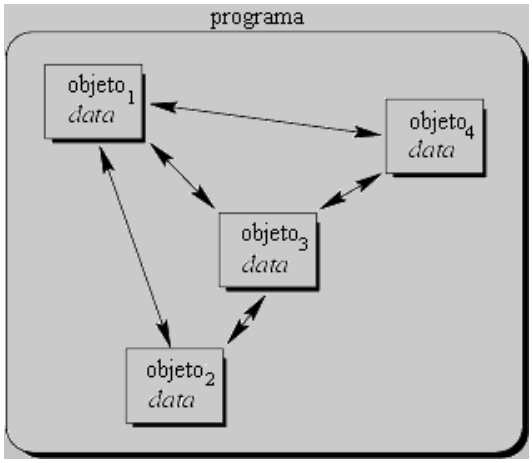
Se escogen las representaciones de datos que mejor se ajusten a tus requerimientos. Por consecuencia, los programas se estructuran por los datos más que por las operaciones.

Así, esto es exactamente del otro modo : Los datos especifican las operaciones válidas. Ahora, los módulos agrupan representaciones de datos en forma conjunta.

Programación orientada a objetos.

La programación orientada a objetos resuelve algunos de los problemas que se acaban de mencionar. De alguna forma se podría decir que obligan a prestar atención a los datos.

En contraste con las otras técnicas, ahora tenemos una telaraña de objetos interactuantes, cada uno de los cuáles manteniendo su propio estado.



Por ejemplo, en la programación orientada a objetos deberíamos tener tantos objetos de pila como sea necesario. En lugar de llamar un procedimiento al que le debemos proveer el manejador de la pila correcto, mandaríamos un mensaje directamente al objeto pila en cuestión.

En términos generales, cada objeto implementa su propio módulo, permitiendo por ejemplo que coexistan muchas pilas.

Cada objeto es responsable de inicializarse y destruirse en forma correcta. Por consiguiente, ya no existe la necesidad de llamar explícitamente al procedimiento de creación o de terminación.

¿No es ésta solamente una manera más elegante de técnica de programación modular ?

Tipos de Datos Abstractos

Algunos autores describen la programación orientada a objetos como programación de tipos de datos abstractos y sus relaciones. Los tipos de datos abstractos son como un concepto básico de orientación a objetos.

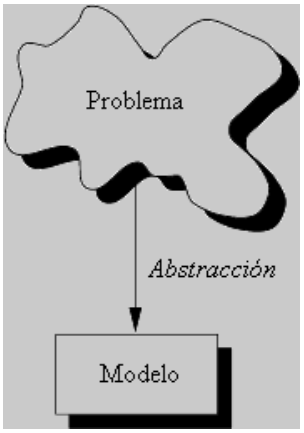
Los Problemas

La primera cosa con la que uno se enfrenta cuando se escriben programas es el problema.

Típicamente, uno se enfrenta a problemas "de la vida real" y nos queremos facilitar la existencia por medio de un programa para dichos problemas.

Sin embargo, los problemas de la vida real son nebulosos y la primera cosa que se tiene que hacer es tratar de entender el problema para separar los detalles esenciales de los no esenciales :

Tratando de obtener tu propia perspectiva abstracta, o modelo, del problema. Este proceso de modelado se llama abstracción y se ilustra en la Figura:



El modelo define una perspectiva abstracta del problema. Esto implica que el modelo se enfoca solamente en aspectos relacionados con el problema y que uno trata de definir propiedades del problema. Estas propiedades incluyen

- los datos que son afectados
- las operaciones que son identificadas

por el problema.

Para resumir, la abstracción es la estructuración de un problema nebuloso en entidades bien definidas por medio de la definición de sus datos y operaciones. Consecuentemente, estas entidades combinan datos y operaciones. No están desacoplados unos de otras.

Tipos de Datos Abstractos y Orientación a Objetos

Los TDAs permiten la creación de instancias con propiedades bien definidas y comportamiento bien definido. En orientación a objetos, nos referimos a los TDAs como clases. Por lo tanto, una clase define las propiedades de objetos en un ambiente orientado a objetos.

Los TDAs definen la funcionalidad al poner especial énfasis en los datos involucrados, su estructura, operaciones, así como en axiomas y precondiciones. Consecuentemente, la programación orientada a objetos es "programación con

TDAs" : al combinar la funcionalidad de distintos TDAs para resolver un problema. Por lo tanto, instancias de TDAs son creadas dinámicamente, usadas y destruídas.

Conceptos de básicos de objetos.

La programación tradicional separa los datos de las funciones, mientras que la programación orientada a objetos define un conjunto de objetos donde se combina de forma modular los datos con las funciones.

Aspectos principales:

1) **Objetos**.
- El objeto es la entidad básica del modelo orientado a objetos.
- El objeto integra una estructura de datos (atributos) y un comportamiento (operaciones).
- Se distinguen entre sí por medio de su propia identidad, aunque internamente los valores de sus atributos sean iguales.

2) **Clasificación**.
- Las clases describen posibles objetos, con una estructura y comportamiento común.
- Los objetos que contienen los mismos atributos y operaciones pertenecen a la misma clase.
- La estructura de clases integra las operaciones con los atributos a los cuales se aplican.

3) **Instanciación**.
- El proceso de crear objetos que pertenecen a una clase se denomina instanciación.
- Pueden ser instanciados un número indefinido de objetos de cierta clase.

4) **Generalización**.

- En una jerarquía de clases, se comparten atributos y operaciones entre clases basados en la generalización de clases.
- La jerarquía de generalización se construye mediante la herencia.
- Las clases más generales se conocen como superclases.
- Las clases más especializadas se conocen como subclases. La herencia puede ser simple o múltiple.

5) **Abstracción**.

- La abstracción se concentra en lo primordial de una entidad y no en sus propiedades secundarias.
- Además en lo que el objeto hace y no en cómo lo hace.
- Se da énfasis a cuales son los objetos y no cómo son usados.

6) **Encapsulación**.

- Encapsulación o encapsulamiento es la separación de las propiedades externas de un objeto de los detalles de implementación internos del objeto.
- Al separar la interfaz del objeto de su implementación, se limita la complejidad al mostrarse sólo la información relevante.
- Disminuye el impacto a cambios en la implementación, ya que los cambios a las propiedades internas del objeto no afectan su interacción externa.

7) **Modularidad**.

- El encapsulamiento de los objetos trae como consecuencia una gran modularidad.
- Cada módulo se concentra en una sola clase de objetos.
- Los módulos tienden a ser pequeños y concisos.
- La modularidad facilita encontrar y corregir problemas.
- La complejidad del sistema se reduce facilitando su mantenimiento.

8) **Extensibilidad**.

- La extensibilidad permite hacer cambios en el sistema sin afectar lo que ya existe.
- Nuevas clases pueden ser definidas sin tener que cambiar la interfaz del resto del sistema.
- La definición de los objetos existentes puede ser extendida sin necesidad de cambios más allá del propio objeto.

9) **Polimorfismo**.

- El polimorfismo es la característica de definir las mismas operaciones con diferente comportamiento en diferentes clases.
- Se permite llamar una operación sin preocuparse de cuál implementación es requerida en que clase, siendo responsabilidad de la jerarquía de clases y no del programador.

10) **Reusabilidad de código**.

- La orientación a objetos apoya el reuso de código en el sistema.
- Los componentes orientados a objetos se pueden utilizar para estructurar librerías resuables.
- El reuso reduce el tamaño del sistema durante la creación y ejecución.
- Al corresponder varios objetos a una misma clase, se guardan los atributos y operaciones una sola vez por clase, y no por cad objeto.
- La herencia es uno de los factores más importantes contribuyendo al incremento en el reuso de código dentro de un proyecto.

Lenguajes de programación orientada a objetos.

Comentario: Los años de los lenguajes son relativos, y más bien se refieren a los documentos de los cuales se tomo la información original.

Simula I fue originalmente diseñado para problemas de simulación y fue el primer lenguaje en el cual los datos y procedimientos estaban unificados en una sola entidad. Su sucesor **Simula** , derivó definiciones formales a los conceptos de objetos y clase.

Simula sirvió de base a una generación de lenguajes de programación orientados a objetos. Es el caso de **C++**, **Eiffel** y **Beta**.

Ada (1983), se derivan de conceptos similares, e incorporan el concepto de jerarquía de herencia. **CLU** -*clusters*- también incorpora herencia.

Smalltalk es descendiente directo de **Simula**, generaliza el concepto de objeto como única entidad manipulada en los programas. Existen tres versiones principales: **Smalltalk-72**, introdujo el paso de mensajes para permitir la comunicación entre objetos. **Smalltalk-76** que introdujo herencia. **Smalltalk-80** se inspira en **Lisp**.

Lisp contribuyó de forma importante a la evolución de la programación orientada a objetos.

Flavors maneja herencia múltiple apoyada con facilidades para la combinación de métodos heredados.

CLOS , es el estándar del sistema de objetos de **Common Lisp**.

Los programas de programación orientada a objetos pierden eficiencia ante los lenguajes imperativos, pues al ser interpretado estos en la arquitectura von Neumann resulta en un excesivo manejo dinámico de la memoria por la constante creación de objetos, así como una fuerte carga por la división en múltiples operaciones (métodos) y su ocupación. Sin embargo se gana mucho en comprensión de código y modelado de los problemas.

Características de los principales LPOO.

	Ada 95	**C++**	**Eiffel**	**Java**	**Smalltalk**
Paquetes	Sí	No	No	Sí	No
Herencia	Simple	Múltiple	Múltiple	Simple	Simple
Control de tipos	Fuerte	Fuerte	Fuerte	Fuerte	Sin tipos
Enlace	Dinámico	Dinámico	Dinámico	Dinámico	Dinámico
Concurrencia	Sí	No	No	Sí	No
Recolección de basura	No	No	Sí	Sí	Sí
Afirmaciones	No	No	Sí	No	No
Persistencia	No	No	No	No	No

Abstracción de datos: Clases y objetos.

Clases.

Se mencionaba anteriormente que la base de la programación orientada a objetos es la abstracción de los datos o los TDAs. La abstracción de los datos se da realmente a través de las clases y objetos.

Def. Clase. Se puede decir que una clase es la implementación real de un TDA, proporcionando entonces la estructura de datos necesaria y sus operaciones. Los datos son llamados **atributos** y las operaciones se conocen como **métodos**. [6]

La unión de los atributos y los métodos dan forma al comportamiento (*comportamiento común*) de un grupo de objetos. La clase es entonces como la definición de un esquema dentro del cual encajan un conjunto de objetos.

Ejemplos de clases: automóvil, persona, libro, revista, reloj, silla,...

Objetos e instancias.

Una de las características más importantes de los lenguajes orientados a objetos es la instanciación. Esta es la capacidad que tienen los nuevos tipos de datos, para nuestro caso en particular las clases de ser "instanciadas" en cualquier momento.

El instanciar una clase produce un objeto o instancia de la clase requerida.

Def. Objeto. Un objeto es una instancia de una clase. Puede ser identificado en forma única por su nombre y define un estado, el cuál es representado por los valores de sus atributos en un momento en particular. [6]

El estado de un objeto cambia de acuerdo a los métodos que le son aplicados. Nos referimos a esta posible secuencia de cambios de estado como el comportamiento del objeto :

Def. Comportamiento. El comportamiento de un objeto es definido por un conjunto de métodos que le pueden ser aplicados.[6]

Instanciación.

Los objetos pueden ser creados de la misma forma que una estructura de datos:

1. **Estáticamente**. En tiempo de compilación se le asigna un área de memoria.
2. **Dinámicamente**. Se le asigna un área de memoria en tiempo de ejecución y su existencia es temporal. Es necesario liberar espacio cuando el objeto ya no es útil, para esto puede ser que el lenguaje proporcione mecanismos de recolección de basura.

Clases en C++.

Una clase entonces, permite encapsular la información a través de atributos y métodos que utilizan la información, ocultando la información y la implementación del comportamiento de las clases.

La definición de una clase define nuevos TDAs y la definición en C++ consite de la palabra reservada *class*, seguida del nombre de la clase y finalmente el cuerpo de la clase encerrado entre llaves y finalizando con ;.

El cuerpo de la clase contiene la declaración de los atributos de la clase (variables) y la declaración de los métodos (funciones). Tanto los atributos como los métodos pertenecen exclusivamente a la clase y sólo pueden ser usados a través de un objeto de esa clase.

Sintaxis:

```
class <nombre_clase> {
    <cuerpo de la clase>
};
```

Ejemplo:

```
class cEjemplo1 {
    int x;
 float y;
 void fun(int a, float b) {
    x=a;
    y=b;
 }
};
```

Miembros de una clase.

Una clase está formada por un conjunto de miembros que pueden ser datos, funciones, clases anidadas, enumeraciones, tipos de dato, etc. . Por el momento nos vamos a centrar en los datos y las funciones (atributos y métodos).

Es importante señalar que un miembro no puede ser declarado más de una vez.[1] Tampoco es posible añadir miembros después de la declaración de la clase.

Ejemplo:

```
class cEjemplo2{
    int i;
    int i;   //error
    int j;
    int func(int, int);
}
```

[1] Aunque existe el concepto de sobrecarga que se verá más adelante

Atributos miembro.

Todos los atributos que forman parte de una clase deben ser declarados dentro de la misma.

Métodos miembro.

Los métodos al igual que los atributos, deber ser definidos en la clase, pero el cuerpo de la función puede ir dentro o fuera de la clase. Si un método se declara completo dentro de la clase, se considera como *inline*.

La declaración dentro de la clase no cambia con respecto a la declaración de una función, salvo que se hace dentro de la clase. Recordemos el ejemplo inicial, pero ahora modificado con el cuerpo del méotdo fuera del cuerpo de la clase.

Ejemplo:

```
//código en ejemplo3.h
class cEjemplo3 {
    public:
    int x;
    float y;
    int funX(int a) {
        x=a;
        return x;
    }
    float funY(float);
};
```

Podemos ver que en la definición de la clase se incluye un método en línea y un prototipo de otro método.

Para definir un método miembro de una clase fuera de la misma, se debe escribir antes del nombre del método la clase con la que el método esta asociado. Para esto se ocupa el operador de resolución de alcance **::** .

Continuación del ejemplo:

```
float cEjemplo3::funY(float b){
     y=b;
     return y;
}
```

Reiteramos que al declarar los métodos fuera de la clase **no** puede mencionarse la declaración de un método que no esté contemplado dentro de la clase, pues entonces cualquiera podría ganar acceso a la clase con sólo declarar una función adicional.

Ejemplo:

```
     //error en declaración de un método
     class x{
          public:
          int a;
          f();
     };

     int x::g() { //error sólo se puede con f()
          return a*=3.1234;
     }
```

La declaración de una función miembro es considerada dentro del ámbito de su clase. Lo cual significa que puede usar nombres de miembros de la clase directamente sin usar el operador de acceso de miembro de la clase.

Recordar que por convención en la programación orientada a objetos las funciones son llamadas métodos y la invocación o llamada se conoce como mensaje.

Un vistazo al acceso a miembros.

Otra de las ventajas de la POO es la posibilidad de encapsular datos, ocultándolos de otros objetos si es necesario. Para esto existen principalmente dos calificadores que definen a los datos como públicos o privados.

Miembros públicos. Se utiliza cuando queremos dar a usuarios de una clase *)* el acceso a miembros de esa clase, los miembros deben ser declarados públicos.
Sintaxis:

```
public:
    <definición de miembros>
```

Miembros privados. Si queremos ocultar ciertos miembros de una clase de los usuarios de la misma, debemos declarar a los miembros como privados. De esta forma nadie más que los miembros de la clase pueden usar a los miembros privados. Por omisión los miembros se consideran privados. En una estructura se consideran públicos por omisión.
Sintaxis:

```
private:
    <definición de miembros>
```

Normalmente, se trata de que los atributos de la clase sean privados; así como los métodos que no sean necesarios externamente o que puedan conducir a un estado inconsistente del objeto.[2]

En el caso de los atributos, estos al ser privados deberían de contar con métodos de modificación y de consulta pudiendo incluir alguna validación.

Es una buena costumbre de programación accesar a los atributos solamente a través de las funciones de modificación, sobre todo si es necesario algún tipo de verificación sobre el valor del atributo.

Ejemplo:

```
//código en ejemplo3.h
class cFecha {
        private:
    int dia;
int mes;
int an;
```

[2] Un estado inconsistente sería ocasionado por una modificación indebida de los datos, por ejemplo una modificación sin validación.

```
    public:
    char setDia(int);   //poner día
    int getDia();    //devuelve día
    char setMes(int);
    int getMes();
    char setAn(int);
    int getAnd();
};
```

Objetos de clase.

Ya se ha visto como definir una clase, declarando sus atributos y sus operaciones, mismas que pueden ir dentro de la definición de la clase (inline) o fuera. Ahora vamos a ver como es posible crear objetos o instancias de esa clase.

Hay que recordar que una de las características de los objetos es que cada uno guarda un estado particular de acuerdo al valor de sus atributos[3].

Lo más importante de los LOO es precisamente el objeto, el cual es una identidad lógica que contiene datos y código que manipula esos datos.

En C++, un objeto es una variable de un tipo definido por el usuario.[5] .

Un ejemplo completo:

```
#include <iostream.h>
#include <conio.h>

class cEjemplo3 {
public:
    int i;
    int j;
};
```

[3] A diferencia de la programación modular, donde cada módulo tiene un solo estado. Aunque está característica se puede lograr en la programación modular.

```
void main() {
     cEjemplo3 e3;
     cEjemplo1 e1;

     e1.i=10;
     e1.j=20;

     e2.i=100;
     e2.j=20;

     cout<<e1.i<<endl;
     cout<<e2.i<<endl;
     getch();
}
```

Otro ejemplo, una cola:

```
//cola definida en un arreglo
#include <iostream.h>
#include <conio.h>

class cCola{
     private:
       int q[10];
       int sloc, rloc;

     public:
       void ini() {  //funci¢n en l¡nea
         sloc=rloc=-1;
       }
       char set(int);
       int get();
};

char cCola::set(int val){
     if(sloc>=10){
         cout<<"la cola est  llena";
         return 0;
     }
     sloc++;
```

```
    q[sloc]=val;
    return 1;
}
int cCola::get(){
    if(rloc==sloc)
        cout<<"la cola esta vac¡a";
    else {
        rloc++;
        return q[rloc];
    }
}

void main(){
    cCola a,b, *pCola= new cCola;  //¢ *pCola=NULL y despu,s
asignarle
    clrscr();
    a.ini();
    b.ini();
    pCola->ini();
    a.set(1);
    b.set(2);
    pCola->set(3);
    a.set(11);
    b.set(22);
    pCola->set(33);
    cout<<a.get()<<endl;
    cout<<a.get()<<endl;
    cout<<b.get()<<endl;
    cout<<b.get()<<endl;
    cout<<pCola->get()<<endl;
    cout<<pCola->get()<<endl;
    getch();
   delete pCola;
}
```

Nota: Otra forma de manejar los archivos .h, además en el proyecto se puede incluir el correspondiente obj para no proporcionar el código.

#ifndef PILA_H
#define PILA_H
<definición de la clase>
#endif

Alcance de Clase.

El nombre de un miembro de una clase es local a la clase. Las funciones no miembros se definen en un alcance de archivo.

Dentro de la clase los miembros pueden ser accesados directamente por todos los métodos miembros. Fuera del alcance la clase, los miembros de la clase se pueden utilizar seguidos del **operador de selección de miembro** de punto **.** ó del **operador de selección de miembro** de flecha → , posteriormente al nombre de un objeto de clase.

Ejemplo:

```
class cMiClase{
    public:

      Int otraFuncion();
}

void main () {
    cMiClase cM;

    cM.otraFuncion();

    }
```

Sobrecarga de operaciones.

En C++ es posible tener el mismo nombre para una operación con la condición de que tenga parámetros diferentes. La diferencia debe de ser al menos en el tipo de datos.

Si se tienen dos o más operaciones con el mismo nombre y diferentes parámetros se dice que dichas operaciones están sobrecargadas.

El compilador sabe que operación ejecutar a través de la firma de la operación, que es una combinación del nombre de la operación y el número y tipo de los parámetros.

El tipo de regreso de la operación puede ser igual o diferente.

La sobrecarga de operaciones sirve para hacer un código más legible y modular. La idea es utilizar el mismo nombre para operaciones relacionadas. Si no tienen nada que ver entonces es mejor utilizar un nombre distinto.

Ejemplo:

```
class cMiClase{
     int x;
     public:
      void modifica() {
          x++;
     }
      void modifica(int y){
          x=y*y;
     }
}
```

Ejemplo 2:

```
//fuera de POO
#include <iostream.h>

int cuadrado(int i){
     return i*i;
}
double cuadrado(double d){
     return d*d;
}

void main() {
     cout<<"10 elevado al cuadrado:  "<<cuadrado(10)<<endl;
     cout<<"10.5 elevado al cuadrado: "<<cuadrado(10.5)<<endl;
}
```

Constructores y destructores.

Con el manejo de los tipos de datos primitivos, el compilador se encarga de reservar la memoria y de liberarla cuando estos datos salen de su ámbito.

En la programación orientada a objetos, se trata de proporcionar mecanismos similares. Cuando un objeto es creado es llamada un método conocido como constructor, y al salir se llama a otro conocido como destructor. Si no se proporcionan estos métodos se asume la acción más simple.

Constructor.

Un constructor es un método con el mismo nombre de la clase. Este método no puede tener un tipo de dato y si puede permitir la homonimia o sobrecarga.

Ejemplo:

```
class cCola{
 private:
    int q[100];
    int sloc, rloc;
 public:
    cCola( );    //constructor
    void put(int);
    int get( );
};

//implementación del constructor
cCola::cCola ( ) {
    sloc=rloc=0;
    cout<<"Cola inicializada \n";
}
```

Un constructor si puede ser llamado desde un método de la clase.

Destructor.

La contraparte del constructor es el destructor. Este se ejecuta momentos antes de que el objeto sea destruido, ya sea porque salen de su ámbito o por medio de una instrucción *delete*. El uso más común para un destructor es liberar la memoria asignada dinámicamente, aunque puede ser utilizado para otras operaciones de finalización, como cerrar archivos, una conexión a red, etc.

El destructor tiene al igual que el constructor el nombre de la clase pero con una tilde como prefijo (~).

El destructor tampoco regresa valores ni tiene parámetros.

Ejemplo:

```
class cCola{
 private:
    int q[100];
    int sloc, rloc;
 public:
    cCola( );    //constructor
    ~cCola();  //destructor
    void put(int);
    int get( );
};

cCola::~cCola( ){
    cout<<"cola destruida\n";
}
```

Ejemplo completo de cCola con constructor y destructor:

```
//cola definida en un arreglo
//incluye constructores y destructores de ejemplo
#include <iostream.h>
#include <string.h>
#include <conio.h>
#include <stdio.h>

class cCola{
```

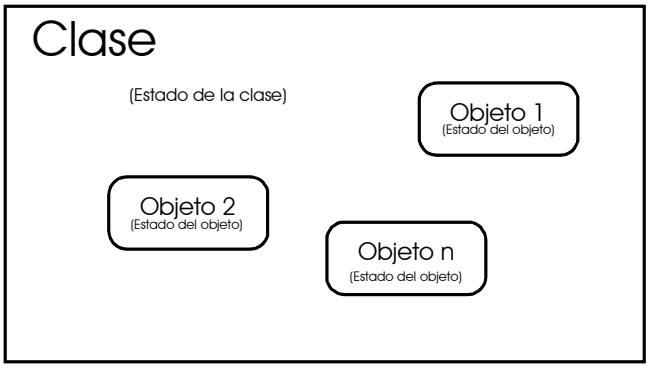
```
    private:
      int q[10];
      int sloc, rloc;
      char *nom;

    public:
      cCola(char *cad=NULL) {  //funcion en linea
        if(cad){   //cadena!=NULL
            nom=new char[strlen(cad)+1];
            strcpy(nom, cad);
        }else
            nom=NULL;
        sloc=rloc=-1;
      }
      ~cCola( ) {
            if(nom){  //nom!=NULL
                cout<<"Cola : "<<nom<<" destruida\n";
                delete [] nom;
            }
      }

      char set(int);
      int get();
};

char cCola::set(int val){
    if(sloc>=10){
        cout<<"la cola esta llena";
        return 0;
    }
    sloc++;
    q[sloc]=val;
    return 1;
}
int cCola::get(){
    if(rloc==sloc)
        cout<<"la cola esta vacia";
    else {
        rloc++;
        return q[rloc];
```

```
    }
}

void main(){
    cCola a("Cola a"),b("Cola b"),
        *pCola= new cCola("Cola dinamica pCola");

    clrscr();
    a.set(1);
    b.set(2);
    pCola->set(3);
    a.set(11);
    b.set(22);
    pCola->set(33);
    cout<<a.get()<<endl;
    cout<<a.get()<<endl;
    cout<<b.get()<<endl;
    cout<<b.get()<<endl;
    cout<<pCola->get()<<endl;
    cout<<pCola->get()<<endl;
    getch();

    delete pCola;
}
```

Miembros estáticos.

Cada objeto tiene su propio estado, pero a veces es necesario tener valores por clase y no por objeto. En esos casos es necesario tener atributos estáticos que sean compartidos por todos los objetos de la clase.

Existe solo una copia de un miembro estático y no forma parte de los objetos de la clase.



Ejemplo:

```
class cObjeto{
 private:
    char nombre[10];
    static int numObjetos;
 public:
    cObjeto(char *cadena=NULL);
    ~cObjeto();
};

cObjeto::cObjeto(char *cadena){
    if(cadena!=NULL)
        strcpy(nombre, cadena);
    else
        nombre=NULL;
    numObjetos++;
}
```

```
cObjeto::~cObjeto(){
    numObjetos--;
}
```

Un miembro estático es accesible desde cualquier objeto de la clase o mediante el operador de resolución de alcance binario (::) y el nombre de la clase, dado que el miembro estático **existe** aunque no haya instancias de la clase.

Sin embargo, el acceso sigue restringido bajo las reglas de acceso a miembros:

- Si se quiere accesar a un miembro estático que es privado deberá hacerse mediante un método público.
- Si no existe ninguna instancia de la clase entonces deberá ser por medio de un método público y estático.

Además, un método estático solo puede tener acceso a miembros estáticos.

Los atributos estáticos deben de ser inicializados al igual que los atributos constantes, fuera de la declaración de la clase. Por ejemplo:

```
int cClase::atributo=0;      int const cClase::ATRCONST=50;
```

Ejemplo :

```
//prueba de miembros estáticos
#include <iostream.h>
#include <stdio.h>
#include <conio.h>
#include <string.h>

class cPersona{
private:
    static int nPersonas;
    static const int MAX;
    char *nombre;
public:
    cPersona(char *c=NULL){
        if(c!=NULL){
```

```
            nombre= new char[strlen(c)+1];
            strcpy(nombre, c);
        }else
            nombre=NULL;
        cout<<"Persona: "<<nombre<<endl;
        nPersonas++;
    }
    ~cPersona(){
        cout<<"eliminando persona : "<<nombre<<endl;
        if(nombre)
            delete []nombre;
        nPersonas--;
    }

    static int getMax(){
        return MAX;
    }
    static int getnPersonas(){
        return nPersonas;
    }
};

int cPersona::nPersonas=0;
const int cPersona::MAX=10;

void main() {
    clrscr();
    cout<<"Máximo de personas: "<<cPersona::getMax()<<endl;
    cout<<"Número de personas:
"<<cPersona::getnPersonas()<<endl;

    cPersona per1;
    cout<<"Máximo de personas: "<<cPersona::getMax()<<endl;
    cout<<"Número de personas:
"<<cPersona::getnPersonas()<<endl;

    cPersona per2("persona 2");
    cout<<"Máximo de personas: "<<per2.getMax()<<endl;
    cout<<"Número de personas: "<<per2.getnPersonas()<<endl;
}
```

Objetos constantes.

Es posible tener objetos de tipo constante, los cuales no podrán ser modificados en ningún momento.[4] Tratar de modificar un objeto constante se detecta como un error en tiempo de compilación.

Sintaxis:

```
const <clase> <lista de objetos>; const cHora h1(9,30,20);
```

Para estos objetos, algunos compiladores llegan a ser tan rígidos en el cumplimiento de la instrucción, que no permiten que se hagan llamadas a métodos sobre esos objetos. En el caso de C++ de Borland, el compilador únicamente manda una advertencia y permite que se ejecute, pero advierte que debe ser considerado como un error.

Sin embargo, es posible que se quiera consultar al objeto mediante llamadas a métodos *get*, para esto lo correcto es declarar métodos con la palabra reservada *const*, para permitirles actuar libremente sobre los objetos sin modificarlo. La sintaxis es añadir después de la lista de parámetros la palabra reservada *const* en la declaración y en su definición.

Sintaxis:

Declaración.

```
<tipo> <nombre> (<parámetros>) const;
```

Definición del método fuera de la declaración de la clase.

```
<tipo> <clase> :: <nombre> (<parámetros>) const {
    <código>
}
```

[4] Ayuda a cumplir el principio del mínimo privilegio, donde se debe restringir al máximo el acceso a los datos cuando este acceso estaría de sobra. [1]

Definición del método dentro de la declaración de la clase.

```
<tipo> <nombre> (<parámetros>) const {
    <código>
}
```

El compilador de Borland permite usar los métodos constantes de manera indiferente para objetos constantes y no constantes siempre y cuando no modifiquen al objeto; sin embargo, algunos compiladores restringen el uso de métodos constantes a objetos constantes. Para solucionarlo es posible sobrecargar el método con la única diferencia de la palabra *const*, aunque el resto de la firma del método sea la misma.

Los constructores no necesitan la declaración *const*, puesto que deben poder modificar al objeto.

Ejemplo:

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

class cArr{
    private:
      int a[10];
    public:
      cArr(int x=0) {
        for( int i=0; i<10; i++){
            if (x==0)
                x=random(10);
            a[i]=x;
        }
      }
      char set(int, int);
      int get(int) const ;
      int get(int);
};
```

```
char cArr::set(int pos, int val ){
     if(pos>=0 && pos<10){
          a[pos]=val;
          return 1;
     }
     return 0;
}
int cArr::get(int pos) const {
     if(pos>=0 && pos<10)
          return a[pos];
//    a[9]=0;  error en un m,todo constante
}

int cArr::get(int pos) {  //no es necesario sobrecargar
     if(pos>=0 && pos<10)   // si el m,todo no modifica
          return a[pos];
}

void main(){
     const cArr a(5),b;
     cArr c;

     randomize();
     clrscr();
     a.set(0,1);  //error llamar a un m,todo no const
     b.set(0,2);   // para un objeto constante
     c.set(0,3);
     a.set(1,11);
     b.set(1,22);
     c.set(1,33);
     cout<<a.get(0)<<endl;
     cout<<a.get(1)<<endl;
     cout<<b.get(0)<<endl;
     cout<<b.get(1)<<endl;
     cout<<c.get(0)<<endl;
     cout<<c.get(1)<<endl;
     getch();
}
```

Objetos compuestos.

Algunas veces una clase no puede modelar adecuadamente una entidad basándose únicamente en tipos de datos simples. Los LPOO permiten a una clase contener objetos. Un objeto forma parte directamente de la clase en la que se encuentra declarado.

El objeto compuesto es una especie de relación, pero con una asociación más fuerte con los objetos relacionados. A la noción de objeto compuesto se le conoce también como objeto complejo o agregado.

Rumbaugh en [10] define a la agregación como "una forma fuerte de asociación, en la cual el objeto agregado está formado por componentes. Los componentes forman parte del agregado. El agregado, es un objeto extendido que se trata como una unidad en muchas operaciones, aun cuando conste físicamente de varios objetos menores."

Ejemplo: Un automóvil se puede considerar ensamblado o agregado, donde el motor y la carrocería serian sus componentes.

El concepto de agregación puede ser relativo a la conceptualización que se tenga de los objetos que se quieran modelar.

Hay que tener en cuenta que pasa con los objetos que son parte del objeto compuesto cuando éste último se destruye. En general hay dos opciones:

1. Cuando el objeto agregado se destruye, los objetos que lo componen no tienen necesariamente que ser destruidos.

2. Cuando el agregado es destruido también sus componentes se destruyen.

Por el momento vamos a considerar la segunda opción, por ser más fácil de implementar y es la acción natural de los objetos que se encuentran embebidos como un atributo más una clase.

Ejemplo:

```
class Nombre {
private:
  char paterno[20],
  materno[20],
  nom[15];
public:
    set(char *, char*, char *);
    ...
};

class Persona {
private:
    int edad;
    Nombre nombrePersona;
....
};
```

Al crear un objeto compuesto, cada uno de sus componentes es creado con sus respectivos constructores. Para inicializar esos objetos componentes tenemos dos opciones:

1. En el constructor del objeto compuesto llamar a los métodos set correspondientes a la modificación de los atributos de los objetos componentes.

2. Pasar en el constructor del objeto compuesto los argumentos a los constructores de los objetos componentes.

   Sintaxis:

```
<clase>::<constructor>(<lista de argumentos>) : <objeto
componente 1>(<lista de argumentos sin el tipo>),...
```

donde la lista de argumentos del objeto compuesto debe incluir a los argumentos de los objetos componentes, para que puedan ser pasados en la creación del objeto.

Ejemplo:

```
#include <stdio.h>
#include <string.h>
#include <iostream.h>

class cNombre {
    char *nombre,
         *paterno,
         *materno;
    public:
         cNombre(char *n, char *p, char*m){
              nombre=new char[strlen(n)+1];
              paterno=new char[strlen(p)+1];
              materno=new char[strlen(m)+1];
              strcpy(nombre, n);
```

```
            strcpy(paterno, p);
            strcpy(materno, m);
        }
        ~cNombre(){
            delete []nombre, paterno, materno;
            cout<<"destructor de cNombre"<<endl;
        }
};

class cPersona{
    cNombre miNombre;
    int edad;
    public:
    cPersona(char *n, char *p, char*m): miNombre(n, p, m){
        edad=0;
    }
};

void main() {
    cPersona *per1;
    per1= new cPersona("uno", "dos", "tres");
    cPersona per2("carlos alberto", "fernandez y",
"fernandez");
    delete per1;
}
```

Un objeto que es parte de otro objeto, puede a su vez ser un objeto compuesto. De esta forma podemos tener múltiples niveles . Un objeto puede ser un agregado recursivo, es decir, tener un objeto de su misma clase. Ejemplo: Directorio de archivos.

Asociaciones entre clases.

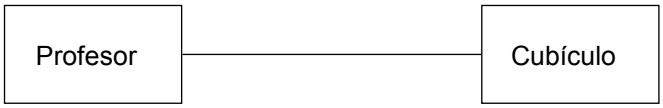
Una clase puede estar relacionada con otra clase, o en la práctica un objeto con otro objeto.

En el modelado de objetos a la relación entre clases se le conoce como asociación; mientras que a la relación entre objetos se le conoce como instancia de una asociación.

Ejemplo:

Una clase estudiante está relacionada con una clase Universidad.

Una relación es una conexión física o conceptual entre objetos. Las relaciones se consideran de naturaleza bidireccional; es decir, ambos lados de la asociación tienen acceso a clase del otro lado de la asociación. Sin embargo, algunas veces únicamente es necesario una relación en una dirección (unidireccional).



Asociación

Comúnmente las asociaciones se representan en los LPOO como apuntadores. Donde un apuntador a una clase B en una clase A indicaría la asociación que tiene A con B; aunque no así la asociación de B con A.

Para una asociación bidireccional es necesario al menos un par de apuntadores, uno en cada clase. Para una asociación unidireccional basta un solo apuntador en la clase que mantiene la referencia.

Ejemplo: un programa que guarda una relación bidireccional entre clases A y B.

```
class A{
    //lista de atributos

    B   *pB;
};

class B{
    //lista de atributos
    A   *pA;
};
```

En el ejemplo anterior se presenta una relación bidireccional, por lo que cada clase tiene su respectivo apuntador a la clase contraria de la relación. Además, deben proporcionarse métodos de acceso a la clase relacionada por medio del apuntador.

En el caso de las relaciones se asumirá que cada objeto puede seguir existiendo de manera independiente, a menos que haya sido creado por el objeto de la clase relacionada, en cuyo caso deberá ser eliminado por el destructor del objeto que la creo. Es decir:

Si el objeto **A** crea al objeto **B**, es responsabilidad de **A** eliminar a la clase **B** antes de que **A** sea eliminada. En caso contrario, si **B** es independiente de la clase **A**, **A** debería enviar un mensaje al objeto **B** para que asigne *NULL* al apuntador de **B** o para que tome una medida pertinente, de manera que no quede apuntando a una dirección inválida.

Es importante señalar que las medidas que se tomen pueden variar de acuerdo a las necesidades de la aplicación, pero bajo **ningún** motivo se deben dejar accesos a áreas de memoria no permitidas o dejar objetos "volando", sin que nadie haga referencia a ellos.

Mencionamos a continuación estructuras clásicas que pueden ser vistas como una relación:

1. Ejemplo de relación **unidireccional**: lista ligada.

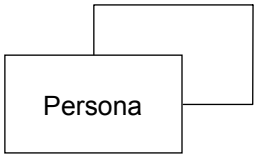
2. Ejemplo de relación **bidireccional**: lista doblemente ligada.

Asociaciones reflexivas.

Cuando se hablo de agregación, se hablo de una clase que pudiera tener un objeto de la misma clase; lo que es llamado agregado recursivo. En los términos de asociaciones existe un concepto similar, pero se conose como asociación reflexiva.

Si una clase mantiene una asociación consigo misma se dice que es una asociación reflexiva.

Ejemplo: Persona puede tener relaciones entre si, si lo que nos interesa es representar a las personas que guardan una relación entre sí, por ejemplo si son parientes. Es decir, un objeto mantiene una relación con otro objeto de la misma clase.



Asociación reflexiva

Multiplicidad de una asociación.

La multiplicidad de una asociación especifica cuantas instancias de una clase se pueden relacionar a **una sola instancia** de otra clase.

Se debe determinar la multiplicidad para cada clase en una asociación.

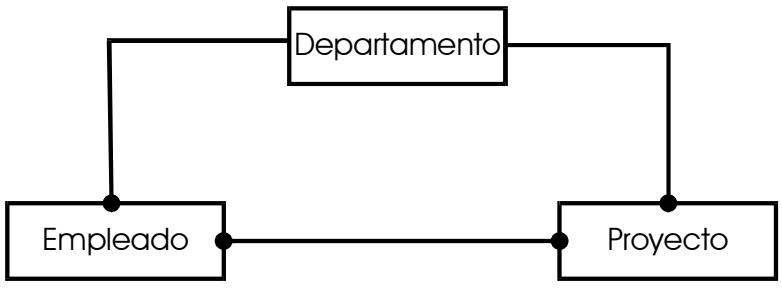
Tipos de asociaciones según su multiplicidad

- **"uno a uno"**: donde dos objetos se relacionan de forma exclusiva, uno con el otro.
  Ejemplo:
  Un alumno tiene una boleta de calificaciones.

- **"uno a muchos"**: donde uno de los objetos puede estar relacionado con muchos otros objetos.
  Ejemplo:
  Un usuario de la biblioteca puede tener muchos libros prestados.

- **"muchos a muchos"**: donde cada objeto de cada clase puede estar relacionado con muchos otros objetos.
  Ejemplo:
  Un libro puede tener varios autores, mientras que un autor puede tener varios libros.

Ejemplo de diagrama con diversas multiplicidades:



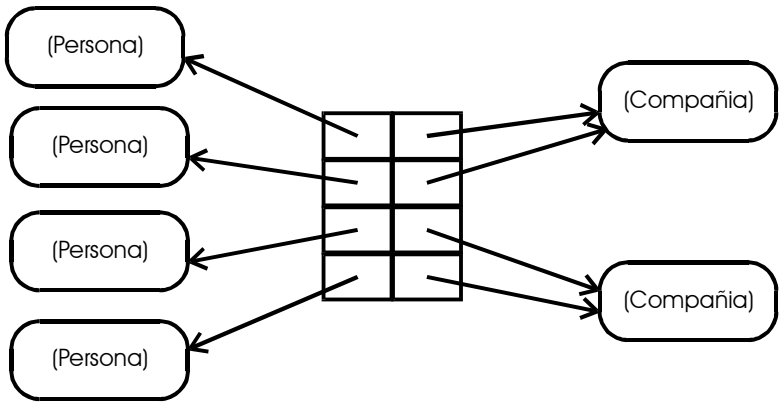
La forma de implementar en C++ este tipo de relaciones puede variar, pero la más común es por medio de apuntadores a objetos. Suponiendo que tenemos relaciones bidireccionales:

- **"uno a uno"**. Un apuntador de cada lado de la relación, como se ha visto anteriormente.

- **"uno a muchos"**. Un apuntador de un lado y un arreglo de apuntadores a objetos definido dinámica o estáticamente.

```
class A{
    ...
    B *pB;
};

class B{
    A *p[5];
//ó
A **p;
}
```

Otra forma es manejar una clase que agrupe a pares de direcciones en un objeto independiente de la clase. Por ejemplo una lista.



- **"muchos a muchos"**. Normalmente se utiliza un objeto u objetos independientes que mantiene las relaciones entre los objetos, de manera similar a la gráfica anterior.

Ejemplo:

Se muestra un código simplificado para manejo de asociaciones.

Clase cLibro

```
class cPersona;

class cLibro {
public:
    char nombre[10];
    cPersona *pPersona;

    cLibro();
    ~cLibro();
};
---
#include <stdio.h>
#include "cPersona.h"
#include "cLibro.h"

cLibro::cLibro(){
    nombre[0]='\0';
    pPersona=NULL;
}

cLibro::~cLibro(){
    if(pPersona!=NULL)
    for(int i=0; i<5; i++)
        if (pPersona->pLibrosPres[i]==this)
            pPersona->pLibrosPres[i]=NULL;

}
```

Clase cPersona

```
class cLibro;

class cPersona {
public:
    cLibro *pLibrosPres[5];

    cPersona();
    ~cPersona();
};
```

```
#include <stdio.h>
#include "clibro.h"
#include "cpersona.h"


cPersona::cPersona(){
    int i;

    for(i=0; i<5; i++)
        pLibrosPres[i]=NULL;
}

cPersona::~cPersona(){
        int i;

    for(i=0; i<5; i++)
        if(pLibrosPres[i]!=NULL)
//          if(pLibrosPres[i]->pPersona!=NULL)
              pLibrosPres[i]->pPersona=NULL;  //¢
(*pLibrosPres[i]).pPersona=NULL;


}
```

Constructor de Copia

Es útil agregar a todas las clases un constructor de copia que reciba como parámetro un objeto de la clase y copie sus datos al nuevo objeto.

C++ proporciona un constructor de copia por omisión, sin embargo es una copia a nivel de miembro y puede no realizar una copia exacta de lo que queremos. Por ejemplo en casos de apuntadores a memoria dinámica, se tendría una copia de la dirección y no de la información referenciada.

Sintaxis:

```
<nombre clase>(const <nombre clase> &<objeto>);
```

Ejemplo:

```
//ejemplo de constructor de copia
#include <iostream.h>
#include <conio.h>
#include <time.h>
#include <stdlib.h>

class cArr{
    private:
      int a[10];
    public:
      cArr(int x=0) {
        for( int i=0; i<10; i++){
            if (x==0)
                x=random(10);
            a[i]=x;
        }
      }
      cArr(const cArr &copia){   //constructor de copia
        for( int i=0; i<10; i++)
            a[i]=copia.a[i];
      }
```

```
      char set(int, int);
      int get(int) const ;
      int get(int);
};

char cArr::set(int pos, int val ){
    if(pos>=0 && pos<10){
        a[pos]=val;
        return 1;
    }
    return 0;
}
int cArr::get(int pos) const {
    if(pos>=0 && pos<10)
        return a[pos];
//   a[9]=0;  error en un m,todo constante
}

int cArr::get(int pos) {  //no es necesario sobrecargar
    if(pos>=0 && pos<10)   // si el m,todo no modifica
        return a[pos];
}

void main(){
    cArr a(5);
    randomize();
    clrscr();
    a.set(0,1);
    a.set(1,11);
    cout<<a.get(0)<<endl;
    cout<<a.get(1)<<endl;

    cArr d(a);
    cout<<d.get(0)<<endl;
    cout<<d.get(1)<<endl;

    getch();
}
```

Comentario: Eliminar este ultimo (cAlumno y cLibro) al colocar este tema en el orden adecuado

Sobrecarga de operadores.

C++ **no** permite la creación de nuevos operadores, pero si permite en cambio sobrecargar los operadores existente para que se utilicen con los objetos. De esta forma se les da a los operadores un nuevo significado de acuerdo al objeto sobre el cual se aplique.

Para sobrecargar un operador, se define un método que es invocado cuando el operador es aplicado sobre ciertos tipos de datos.

Para utilizar un operador con objetos, es necesario que el operador este sobrecargado, aunque existen dos excepciones:

- El **operador de asignación =**, puede ser utilizado sin sobrecargarse explícitamente, pues el comportamiento por omisión es una copia a nivel de miembro de los miembros de la clase. Sin embargo no debe de usarse si la clase cuenta con miembros a los que se les asigne memoria de manera dinámica.

- El **operador de dirección &**, esta sobrecargado por omisión para devolver la dirección de un objeto de cualquier clase.

Algunas restricciones:

1. Operadores que no pueden ser sobrecargados:

```
.        .*       ::       ?:       sizeof
```

2. La precedencia de un operador no puede ser modificada. Deben usarse los paréntesis para obligar un nuevo orden de evaluación.

3. La asociatividad de un operador no puede ser modificada.

4. No se puede modificar el número de operandos de un operador. Los operadores siguen siendo unarios o binarios.

5. No es posible crear nuevos operadores.

6. No puede modificarse el comportamiento de un operador sobre tipos de datos definidos por el lenguaje.

La sintaxis para definir un método con un operador difiere de la definición normal de un método, pues debe indicarse el operador seguido de la palabra reservada *operator* :

```
<tipo> operator <operador> (<argumentos>) ;
```

ó

```
<tipo> operator <operador> (<argumentos>) {
    <cuerpo del método>
}
```

Para la definición fuera de la clase:

```
<tipo> <clase>::operator <operador> (<argumentos>) {
    <cuerpo del método>
}
```

Ejemplo:

```
//programa de ejemplo de sobrecarga de operadores.

class punto {
    float x, y;
public:
    punto(float xx=0, float yy=0){
        x=xx;
        y=yy;
    }
    float magnitud();
    punto operator =(punto);
    punto operator +(punto);
    punto operator -();
    punto operator *(float);
    punto operator *=(float);
    punto operator ++();
    int operator >(punto);
    int operator <=(punto);
};


punto punto::operator =(punto a){ //copia o asignaci¢n
    x=a.x;
    y=a.y;
    return *this;
}

punto punto::operator +(punto p){
    return punto(x+p.x, y+p.y);
}

punto punto::operator -(){
    return punto(-x, -y);
}

punto punto::operator *(float f){
    punto temp;
```

```
    temp=punto(x*f, y*f);
    return temp;
}

punto punto::operator ++(){
    x++;
    y++;
    return *this;
}

int punto::operator >(punto p){
    return (x>p.x && y>p.y) ? 1 : 0;
}

int punto::operator <=(punto p){
    return (x<=p.x && y<=p.y) ? 1 : 0;
}

void main(){
    punto a(1,1);
    punto b;
    punto c;

    b++;
    ++b;
    c=b;
    c=a+b;
    c=-a;
    c=a*.5;

}
```

Ejercicio *: Crear una clase* ***cString*** *para el manejo de cadenas. Tendrá dos atributos: apuntador a carácter y un entero tam, para almacenar el tamaño de la cadena. Sobrecargar operadores = (asignación) e (==) igualdad. Usar un programa de prueba.*
La estructura será la siguiente:

```
class cString{
   char *s;
   int tam;
public:
   cString(char *=NULL);
   cString(const cString &copia);  //constructor de copia
   ~cString();
//sobrecarga de constructor de asignación
   const cString &operator =(const cString &);
   //igualdad
   int operator ==(const cString &) const ;
};
```

Véase que es posible asignar una cadena " " sin sobrecargar el operador de asignación, o comparar un objeto cString con una cadena. Esto se logra gracias a que se provee de un constructor que convierte una cadena a un objeto cString. De esta manera, este constructor de conversión es llamado automáticamente, creando un objeto temporal para ser comparado con el otro objeto. No es posible que la cadena (o apuntador a *char*) vaya del lado izquierdo, pues se estaría llamando a la funcionalidad del operador para un apuntador a char.

Ejemplo: código de CString:

```
//Sobrecarga de operadores. Implementaci¢n de una clase
cString
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
#include <string.h>

class cString{
    //operadores de inserci¢n y extracci¢n de flujo
    friend ostream &operator << (ostream &, const cString &);
    friend istream &operator >> (istream &, cString &);
private:
    char *s;
    int tam;
public:
    cString(char * =NULL); {
        if(c==NULL){
            s=NULL;
            tam=0;
        }
        else {
            tam=strlen(c);
            s= new char[tam+1];
            strcpy(s, c);
        }
    }
    cString(const cString &copia){
        s=NULL;
        tam=0;
        *this=copia;//¿se vale o no?
    }
    ~cString(){
        if(s!=NULL)
            delete []s;
    }

    //sobrecarga de constructor de asignaci¢n
    const cString &operator =(const cString &);
```

```
    //igualdad
    int operator ==(const cString &) const ;

    //concatenaci¢n
    cString operator +(const cString &);

    //concatenaci¢n y asignaci¢n
    const cString &operator +=(const cString &);

    cString &copia (const cString &);

    //sobrecarga de los corchetes
    char operator[] (int);
};

    //operadores de inserci¢n y extracci¢n de flujo
    ostream &operator << (ostream &salida, const cString
&cad){
        salida<<cad.s;
        return salida;  //permite concatenaci¢n
    }

    istream &operator >> (istream &entrada, cString &cad){
        char tmp[100];
        entrada >> tmp;
        cad=tmp; //usa operador de asignación de String y
const. de conversión
        return entrada;  //permite concatenaci¢n
    }

    cString::cString(char *c){
        if(c==NULL){
            s=NULL;
            tam=0;
        } else {
            tam=strlen(c);
            s= new char[tam+1];
            strcpy(s, c);
        }
    }
```

```
const cString &cString::operator =(const cString &c){
    if(this!= &c) {      //verifica no asignarse a si mismo
        if(s!=NULL)
            delete []s;
        tam=c.tam;
        s= new char[tam+1];
        strcpy(s, c.s);
    }
    return *this; //permite concatenaci¢n de asignaciones
}

int cString::operator ==(const cString &c)const {
        return strcmp(s, c.s)==0;
}

//operador de suma regresa una copia de la suma obtenida
//en un objeto local.
cString cString::operator +(const cString &c){
    cString tmp(*this);
    tmp+=c;
    return tmp;
}

const cString &cString::operator +=(const cString &c){
    char *str=s, *ctmp= new char [c.tam+1];
    strcpy(ctmp, c.s);
    tam+=c.tam;
    s= new char[tam+1];
    strcpy(s, str);
    strcat(s, ctmp);
    delete []str;
    delete []ctmp;
    return *this;
}

cString &cString::copia (const cString &c){
    if(this!= &c) {      //verifica no asignarse a si mismo
        if(s!=NULL)
            delete []s;
```

```
        tam=c.tam;
        s= new char[tam+1];
        strcpy(s, c.s);
    }
    return *this; //permite concatenaci¢n de asignaciones
}

char cString::operator[] (int i){
    if(i>0 && i<tam)
        return s[i];
    return 0;
}

void main(){
    clrscr();
    cString a("AAA");
    cString b("Prueba de cadena");
    cString c(b);
/*es un error hacer una asignaci¢n sin liberar memoria.
    ese es el principal peligro de usar el operador
sobrecargado por default de asignaci¢n*/
    a=b;
    b.copia("H o l a");
    b=c+c;
    b="nueva";
    c+=c;
    cString d("nueva cadena");
    d+="Hola";
    cString e;
    e=d+"Adios";
    d="coche";
    int x=0;
    x=d=="coche"; //Lo contrario no es válido "coche"==d
    char ch;
    ch=d[7];
    cin>>e>>d;
    cout<<e<<d;
    printf("fin");
}
```

Funciones amigas (friends).

En Objetos existe la amistad. Aunque a algunos la consideran como una intrusión a la encapsulación o a la privacidad de los datos:

> *"... la amistad corrompe el ocultamiento de información y debilita el valor del enfoque de diseño orientado a objetos"*[Deitel, 1995].

Un amigo de una clase es una función u otra clase que no es miembro de la clase, pero que tiene permiso de usar los miembros públicos y privados de la clase.

El ámbito de una función amiga no es el de la clase, y los amigos no son llamados con los operadores de acceso de miembros.

Sintaxis para una función amiga:

```
class c1{
     friend void miAmiga(int);
...
public:
     ...
};
```

Sintaxis para una clase amiga:

```
class c1{
     friend c2;
     ...
public:
     ...
};
```

Las funciones o clases amigas no son privadas ni públicas (o protegidas), pueden ser colocadas en cualquier parte de la definición de la clase, pero se acostumbra que sea al principio.

Como la amistad entre personas, esta es **concedida** y no tomada. Si la clase B quiere ser amigo de la clase A, la clase A debe declarar que la clase B es su amigo.

La amistad **no es simétrica ni transitiva**: si la clase A es un amigo de la clase B, y la clase B es un amigo de la clase C, no implica:

1. Que la clase B sea un amigo de la clase A.

2. Que la clase C sea un amigo de la clase B.

3. Que la clase A sea un amigo de la clase C.

Ejemplo 1:

```
//Ejemplo de funcion amiga con acceso a miembros privados
#include <iostream.h>
class claseX{
    friend void setX(claseX &, int);  //declaraci¢n friend
public:
    claseX(){
        x=0;
    }
    void print() const {
        cout<<x<<endl;
    }
private:
    int x;
};

void setX(claseX &c, int val){
```

```
    c.x=val; //es legal el acceso a miebros privados por
amistad.
}

void main(){
    claseX pr;

    cout<<"pr.x después de instanciaci¢n : ";
    pr.print();
    cout<<"pr.x después de la llamada a la funci¢n amiga setX
: ";
    setX(pr, 10);
    pr.print();
}
```

Ejemplo 2:

```
//ejemplo 2 de funciones amigas
#include <iostream.h>
#include <conio.h>

class Linea;

class Recuadro {
    friend int mismoColor(Linea, Recuadro);
private:
    int color; //color del recuadro
    int xsup, ysup; //esquina superior izquierda
    int xinf, yinf; //esquina inferior derecha
public:
    void ponColor(int);
    void definirRecuadro(int, int, int, int);
    void mostrarRecuadro(void);
};

class Linea{
    friend int mismoColor(Linea, Recuadro);
private:
```

```
    int color;
    int xInicial, yInicial;
    int lon;
public:
    void ponColor(int);
    void definirLinea(int, int, int);
    void mostrarLinea();
};

int mismoColor(Linea l, Recuadro r){
    if(l.color==r.color)
        return 1;
    return 0;
}

//métodos de la clase Recuadro
void Recuadro::ponColor(int c) {
    color=c;
}

void Recuadro::definirRecuadro(int x1, int y1, int x2, int y2)
{
    xsup=x1;
    ysup=y1;
    xinf=x2;
    yinf=y2;
}

void Recuadro::mostrarRecuadro(){
    int i;
    textcolor(color);

    gotoxy(xsup, ysup);
    for(i=xsup; i<=xinf; i++)
        cprintf("-");

    gotoxy(xsup, yinf-1);
    for(i=xsup; i<=xinf; i++)
        cprintf("-");
```

```
    gotoxy(xsup, ysup);
    for(i=ysup; i<=yinf; i++){
        cprintf("|");
        gotoxy(xsup, i);
    }

    gotoxy(xinf, ysup);
    for(i=ysup; i<=yinf; i++){
        cprintf("|");
        gotoxy(xinf, i);
    }
}

//métodos de la clase Linea
void Linea::ponColor(int c) {
    color=c;
}

void Linea::definirLinea(int x, int y, int l) {

    xInicial=x;
    yInicial=y;
    lon=l;
}

void Linea::mostrarLinea(){
    int i;
    textcolor(color);

    gotoxy(xInicial, yInicial);
    for(i=0; i<lon; i++)
        cprintf("-");
}
```

```
void main(){
    Recuadro r;
    Linea l;

    clrscr();
    r.definirRecuadro(10, 10, 15, 15);
    r.ponColor(3);
    r.mostrarRecuadro();

    l.definirLinea(2, 2, 10);
    l.ponColor(4);
    l.mostrarLinea();

    if(!mismoColor(l, r))
        cout<<"No tienen el mismo color";
    getch();

    //se ponen en el mismo color
    l.ponColor(3);
    l.mostrarLinea();

    if(mismoColor(l, r))
        cout<<"Tienen el mismo color";
    getch();
}
```

Herencia.

Introducción.

La herencia es un mecanismo potente de abstracción que permite compartir similitudes entre clases manteniendo al mismo tiempo sus diferencias.

Es una forma de reutilización de código, tomando clases previamente creadas y formando a partir de ellas nuevas clases, heredándoles sus atributos y métodos. Las nuevas clases pueden ser modificadas agregándoles nuevas características.

En C++ la clase de la cual se toman sus características se conoce como clase base; mientras que la clase que ha sido creada a partir de la clase base se conoce como clase derivada. Existen otros términos para estas clases:

Clase base	Clase derivada
Superclase	Subclase
Clase padre	Clase hija

Una clase derivada es potencialmente una clase base, en caso de ser necesario.

Cada objeto de una clase derivada también es un objeto de la clase base. En cambio, un objeto de la clase base no es un objeto de la clase derivada.

La implementación de herencia a varios niveles forma un árbol jerárquico similar al de un árbol genealógico. Este es conocida como jerarquía de herencia.

Generalización. Una clase base o superclase se dice que es más general que la clase derivada o subclase.

Especialización. Una clase derivada es por naturaleza una clase más especializada que su clase base.

Implementación en C++.

La herencia en C++ es implementada permitiendo a una clase incorporar a otra clase dentro de su declaración.

Sintaxis general:

```
class claseNueva: <acceso>  claseBase {
    //cuerpo clase nueva
};
```

Ejemplo:

Una clase vehículo que describe a todos aquellos objetos vehículos que viajan en carreteras. Puede describirse a partir del número de ruedas y de pasajeros.

De la definición de vehículos podemos definir objetos más específicos (especializados). Por ejemplo la clase camión.

```
//ejemplo 01 de herencia
#include <iostream.h>

class cVehiculo{
    int ruedas;
    int pasajeros;
public:
    void setRuedas(int);
    int  getRuedas();
    void setPasajeros(int);
    int  getPasajeros();
};

void cVehiculo::setRuedas(int num){
    ruedas=num;
}

int cVehiculo::getRuedas(){
```

```
    return ruedas;
}

void cVehiculo::setPasajeros(int num){
    pasajeros=num;
}

int cVehiculo::getPasajeros(){
    return pasajeros;
}

//clase cCamion con herencia  de cVehículo
class cCamion: public cVehiculo {
    int carga;
public:
    void setCarga(int);
    int getCarga();
    void muestra();
};

void cCamion::setCarga(int num){
    carga=num;
}
int cCamion::getCarga(){
    return carga;
}

void cCamion::muestra(){
    cout<<"Ruedas: "<< getRuedas()<<endl;
    cout<<"Pasajeros: "<< getPasajeros()<<endl;
    cout<<"Capacidad de carga: "<<getCarga()<<endl;
}

void main(){
    cCamion ford;
    ford.setRuedas(6);
    ford.setPasajeros(3);
    ford.setCarga(3200);
    ford.muestra();
}
```

Control de Acceso a miembros.

Existen tres palabras reservadas para el control de acceso: *public, private y protected*. Estas sirven para proteger los miembros de la clase en diferentes formas.

El control de acceso, como ya se vio anteriormente, se aplica a los métodos, atributos, constantes y tipos anidados que son miembros de la clase.

Resumen de tipos de acceso:

Tipo de acceso	Descripción
private	Un miembro **privado** únicamente puede ser utilizado por los métodos miembro y funciones amigas de la clase donde fue declarado.
protected	Un miembro **protegido** puede ser utilizado únicamente por los métodos miembro y funciones amigas de la clase donde fue declarado o por los métodos miembro y funciones amigas de las clases derivadas. El acceso protegido es como un nivel intermedio entre el acceso privado y público.
public	Un miembro **público** puede ser utilizado por cualquier método. Una estructura es considerada por C++ como una clase que tiene todos sus miembros públicos.

Ejemplo:

```
//acceso01
//ejemplo de control de acceso

class s{
    char *f1();
    int a;
 protected:
    int b;
    int f2();
```

```
 private:
    int c;
    int f3();
 public:
    int d, f;
    char *f4(int);
 };

 void main(){
    s obj;
    obj.f1(); //error
    obj.a=1; //error
    obj.f2();//error
    obj.b=2; //error
    obj.c=3; //error
    obj.f3(); //error
    obj.d=5;
    obj.f4(obj.f);
}
```

Control de acceso en herencia.

Hasta ahora se han usado la herencia con un solo tipo de acceso, usando el especificador *public*, los miembros públicos de la clase base son miembros públicos de la clase derivada, los miembros protegidos permanecen protegidos para la clase derivada.

Ejemplo:

```
//acceso02
//ejemplo de control de acceso en herencia

class cBase{
    int a;
 protected:
    int b;
 public:
    int c;
```

```
};

class cDerivada: public cBase {
    void g();
};

void cDerivada::g(){
    a=0; //error, es privado
    b=1; //correcto, es protegido
    c=2; //correcto, es público
}
```

Para accesar a los miembros de una clase base desde una clase derivada, se pueden ajustar los permisos por medio de un calificador *public*, *private* o *protected*.

Si una clase base es declarada como **pública** de una clase derivada, los miembros públicos y protegidos son accesibles desde la clase derivada, no así los miembros privados.

Si una clase base es declarada como **privada** de otra clase derivada, los miembros públicos y protegidos de la clase base serán miembros privados de la clase derivada. Los miembros privados de la clase base permanecen inaccesibles.

Si se omite el calificador de acceso de una clase base, se asume por omisión que el calificador es *public* en el caso de una estructura y *private* en el caso de una clase.

Ejemplo de sintaxis:

```
class base {
 ...
};

class d1: private base {
```

```
...
};

class d2: base {
...
};

class d3: public base {
...
};
```

Es recomendable declarar explícitamente la palabra reservada private al tomar una clase base como privada para evitar confusiones:

```
class x{
 public:
        F();
};

class y: x {   //privado por omisión
...
};

void g( y *p){
    p→f();  //error
}
```

Si una clase base es declarada como **protegida** de una clase derivada, los miembros públicos y protegidos de la clase base, se convieten en miembros protegidos de la clase derivada.

Ejemplo:

```
//acceso por herencia
//acceso03
#include <iostream.h>

class X{
protected:
    int i;
    int j;
public:
    void preg_ij();
    void pon_ij();
};

void X::preg_ij() {
    cout<< "Escriba dos n£meros: ";
    cin>>i>>j;
}

void X::pon_ij() {
    cout<<i<<' '<<j<<endl;
}



//en Y, i y j de X siguen siendo miembros protegidos
//Si se llegara a cambiar este acceso a private i y j se
heredan como
// miembros privados de Y, además de los métodos públicos
class Y: public X{
    int k;
public:
    int preg_k();
    void hacer_k();
};

int Y:: preg_k(){
    return k;
}
```

```
void Y::hacer_k() {
    k=i*j;
}

// Z tiene acceso a i y j de X, pero no a k de Y
// porque es private por omisión
// Si Y heredara de x como private, i y j serían privados en
Y,
// por lo que no podrían ser accesados desde Z
class Z: public Y {
public:
    void f();
};


// Si Y heredara a X con private, este método ya no
funcionaría
// no se podría acceder a i ni a j.
void Z::f() {
    i=2;
    j=3;
}


// si Y hereda de x como private, no es posible accesar a los
métodos
//públicos desde objetos de Y ni de Z.
void main() {
    Y var;
    Z var2;

    var.preg_ij();
    var.pon_ij();

    var.hacer_k();
    cout<<var.preg_k()<<endl;

    var2.f();
    var2.pon_ij();
}
```

Manejo de objetos de la clase base como objetos de una clase derivada y viceversa.

Un objeto de una clase derivada pública, puede ser manejado como un objeto de su clase base. Sin embargo, un objeto de la clase base no es posible tratarlo de forma automática como un objeto de clase derivada.

La opción que se puede utilizar, es enmascarar un objeto de una clase base a un apuntador de clase derivada. El problema es que no debe ser desreferenciado (accesado) así, primero se tiene que hacer que ¿?

Si se realiza la conversión explícita de un apuntador de clase base - que apunta a un objeto de clase base- a un apuntador de clase derivada y posteriormente, se hace referencia a miembros de la clase derivada, es un error pues esos miembros no existen en el objeto de la clase base.

Ejemplo :

```
// POINT.H
// clase Point
#ifndef POINT_H
#define POINT_H

class Point {
   friend ostream &operator<<(ostream &, const Point &);
public:
   Point(float = 0, float = 0);
   void setPoint(float, float);
   float getX() const { return x; }
   float getY() const { return y; }

protected:
   float x, y;
};

#endif
```

```
// POINT.CPP

#include <iostream.h>
#include "point.h"

Point::Point(float a, float b)
{
   x = a;
   y = b;
}

void Point::setPoint(float a, float b)
{
   x = a;
   y = b;
}

ostream &operator<<(ostream &output, const Point &p)
{
   output << '[' << p.x << ", " << p.y << ']';

   return output;
}

// CIRCLE.H
// clase Circle
#ifndef CIRCLE_H
#define CIRCLE_H

#include <iostream.h>
#include <iomanip.h>
#include "point.h"

class Circle : public Point {  // Circle hereda de Point
   friend ostream &operator<<(ostream &, const Circle &);
public:
   Circle(float r = 0.0, float x = 0, float y = 0);

   void setRadius(float);
   float getRadius() const;
```

```
   float area() const;
protected:
   float radius;
};

#endif
// CIRCLE.CPP
#include "circle.h"

Circle::Circle(float r, float a, float b)
   : Point(a, b)   // llama al constructor de la clase base
{ radius = r; }

void Circle::setRadius(float r) { radius = r; }

float Circle::getRadius() const { return radius; }

float Circle::area() const
{ return 3.14159 * radius * radius; }

// salida en el formato:
// Center = [x, y]; Radius = #.##
ostream &operator<<(ostream &output, const Circle &c)
{
   output << "Center = [" << c.x << ", " << c.y
          << "]; Radius = " << setiosflags(ios::showpoint)
      << setprecision(2) << c.radius;

   return output;
}
```

```
//Prueba.cpp
// Probando apuntadores a clase base a apuntadores a clase
derivada
#include <iostream.h>
#include <iomanip.h>
#include "point.h"
#include "circle.h"

main()
{
    Point *pointPtr, p(3.5, 5.3);
    Circle *circlePtr, c(2.7, 1.2, 8.9);

    cout << "Point p: " << p << "\nCircle c: " << c << endl;

    // Maneja a un Circle como un Circle
    pointPtr = &c;   // asigna la direccion de Circle a
pointPtr
    circlePtr = (Circle *) pointPtr;  // mascara de base a
derivada
    cout << "\nArea of c (via circlePtr): "
    << circlePtr->area() << endl;

    // Es riesgoso manejar un Point como un Circle
    // getRadius() regresa basura
    pointPtr = &p;   // asigna direccion de Point a pointPtr
    circlePtr = (Circle *) pointPtr;  // mascara de base a
derivada
    cout << "\nRadio de objeto apuntado por circlePtr: "
    << circlePtr->getRadius() << endl;

    return 0;
}
```

Constructores de clase base.

El constructor de la clase base puede ser llamado desde la clase derivada, para inicializar los atributos heredados. La sintaxis es igual que el inicializador de objetos componentes.

Los constructores y operadores de asignación de la clase base **no** son heredados por las clases derivadas. Pero pueden ser llamados por los de la clase derivada.

Un constructor de la clase derivada llama primero al constructor de la clase base. Si se omite el constructor de la clase derivada, el constructor por omisión de la clase derivada llamará al constructor de la clase base.

Los destructores son llamados en orden inverso a las llamadas del constructor: un destructor de una clase derivada será llamado antes que el de su clase base.

Ejemplo:

```
// POINT2.H
#ifndef POINT2_H
#define POINT2_H
class Point {
public:
   Point(float = 0.0, float = 0.0);
   ~Point();
protected:
   float x, y;
};

#endif


// POINT2.CPP
#include <iostream.h>
#include "point2.h"
```

```
Point::Point(float a, float b)
{
   x = a;
   y = b;

   cout << "Constructor Point: "
    << '[' << x << ", " << y << ']' << endl;
}

Point::~Point()
{
   cout << "Destructor Point:  "
        << '[' << x << ", " << y << ']' << endl;
}

// CIRCLE2.H
#ifndef CIRCLE3_H
#define CIRCLE3_H

#include "point2.h"
#include <iomanip.h>

class Circle : public Point {
public:
   Circle(float r = 0.0, float x = 0, float y = 0);
   ~Circle();
private:
   float radius;
};

#endif
// CIRCLE2.CPP
#include "circle2.h"


Circle::Circle(float r, float a, float b)
   : Point(a, b)   // llamada al constructor de clase base
{
```

```
   radius = r;

   cout << "Constructor Circle: radio es "
        << radius << " [" << a << ", " << b << ']' << endl;
}

Circle::~Circle()
{
   cout << "Destructor Circle:   radio es "
        << radius << " [" << x << ", " << y << ']' << endl;
}

// Main.CPP
#include <iostream.h>
#include "point2.h"
#include "circle2.h"

main()
{
   // Muestra llamada a constructor y destructor de Point
   {
      Point p(1.1, 2.2);
   }

   cout << endl;
   Circle circle1(4.5, 7.2, 2.9);
   cout << endl;
   Circle circle2(10, 5, 5);
   cout << endl;
   return 0;
}
```

Redefinición de métodos

Algunas veces, los métodos heredados ya no cumplen completamente la función que quisiéramos que realicen. Es posible en C++ redefinir un método de la clase base en la clase derivada. Cuando se hace referencia al nombre del método, se ejecuta le versión de la clase en donde fue redefinida. Es posible sin embargo, utilizar el método de la clase base por medio del operador de resolución de alcance.

De hecho se sugiere redefinir métodos que no vayan a ser empleados en la clase derivada, inclusive sin código para inhibir cualquier acción que no nos interese.

La redefinición de métodos **no** es una sobrecarga porque se define exactamente con la misma firma.

Ejemplo :

```
// EMPLOY.H
#ifndef EMPLOY_H
#define EMPLOY_H

class Employee {
public:
   Employee(const char*, const char*);
   void print() const;
   ~Employee();
private:
   char *firstName;
   char *lastName;
};

#endif
// EMPLOY.CPP
```

```
#include <string.h>
#include <iostream.h>
#include <assert.h>
#include "employ.h"

Employee::Employee(const char *first, const char *last)
{
   firstName = new char[ strlen(first) + 1 ];
   assert(firstName != 0);     strcpy(firstName, first);

   lastName = new char[ strlen(last) + 1 ];
   assert(lastName != 0);
   strcpy(lastName, last);
}

void Employee::print() const
{ cout << firstName << ' ' << lastName; }

Employee::~Employee()
{
   delete [] firstName;
   delete [] lastName;
}

// HOURLY.H
#ifndef HOURLY_H
#define HOURLY_H

#include "employ.h"

class HourlyWorker : public Employee {
public:
   HourlyWorker(const char*, const char*, float, float);
   float getPay() const;
   void print() const;
private:
   float wage;
   float hours;
};
```

```
#endif
// HOURLY_B.CPP
#include <iostream.h>
#include <iomanip.h>
#include "hourly.h"

HourlyWorker::HourlyWorker(const char *first, const char
*last, float initHours, float initWage) : Employee(first,
last)
{
   hours = initHours;
   wage = initWage;
}

float HourlyWorker::getPay() const { return wage * hours; }

void HourlyWorker::print() const
{
   cout << "HourlyWorker::print()\n\n";

   Employee::print();   // llama a función de clase base

   cout << " es un trabajador por hora con sueldo de"
        << " $" << setiosflags(ios::showpoint)
        << setprecision(2) << getPay() << endl;
}
// main.CPP
#include <iostream.h>
#include "hourly.h"

main()
{
   HourlyWorker h("Bob", "Smith", 40.0, 7.50);
   h.print();
   return 0;
}
```

Herencia Múltiple.

Una clase puede heredar miembros de dos o más clases; lo que se conoce como herencia múltiple.

Herencia múltiple es entonces, la capacidad de una clase derivada de heredar miembros de varias clases base.

Sintaxis:

```
    class <nombre clase derivada> : <clase base 1> , <clase
base 2>, ...<clase base n> {
        ...
    };
```

Ejemplo:

```
class A{
public:
    int i;
    int a();
};

class B{
public:
    int j;
    int b();
};

class C{
public:
    int k;
    int c();
};

class D: public A, public B, public C {
public:
    int l;
    int d();
```

```
};

void main() {
    D var1;

    var1.a();
    var1.b();
    var1.c();
    var1.d();
}
```

Otro ejemplo :

```
// BASE1.H
#ifndef BASE1_H
#define BASE1_H

class Base1 {
public:
   Base1(int x) { value = x; }
   int getData() const { return value; }
protected:          int value;    };

#endif

// BASE2.H
#ifndef BASE2_H
#define BASE2_H

class Base2 {
public:
   Base2(char c) { letter = c; }
   char getData() const { return letter; }
protected:
   char letter;
};
#endif

// DERIVED.H
```

```
#ifndef DERIVED_H
#define DERIVED_H

#include "base1.h"
#include "base2.h"

// herencia múltiple
class Derived : public Base1, public Base2 {
   friend ostream &operator<<(ostream &, const Derived &);
public:
   Derived(int, char, float);
   float getReal() const;
private:
   float real;
};
#endif

// DERIVED.CPP
#include <iostream.h>
#include "derived.h"

Derived::Derived(int i, char c, float f): Base1(i), Base2(c)
{ real = f; }

float Derived::getReal() const { return real; }

ostream &operator<<(ostream &output, const Derived &d)
{
   output << "     Entero: " << d.value
          << "\n   Caracter: " << d.letter
          << "\nNúmero real: " << d.real;

   return output;
}
```

```
// main.CPP
#include <iostream.h>
#include <conio.h>
#include "base1.h"
#include "base2.h"
#include "derived.h"

main(){
    Base1 b1(10), *base1Ptr;      Base2 b2('Z'), *base2Ptr;
    Derived d(7, 'A', 3.5);       clrscr();
   cout << "Objeto b1 contiene entero "
    << b1.getData()
    << "\nObjeto b2 contiene caracter "
    << b2.getData()
    << "\nObjeto d contiene:\n" << d;
   cout << "\n\nmiembros de clase derivada pueden ser"
    << " accesados individualmente:\n"
    << "    Entero: " << d.Base1::getData()
    << "\n  Caracter: " << d.Base2::getData()
    << "\Número real: " << d.getReal() << "\n\n";
    // ¿es un error?:  cout<<d.getData();
   cout << "Miembros derivados pueden ser tratados como "
    << "objetos de su clase base:\n";

   base1Ptr = &d;
   cout << "base1Ptr->getData() "
    << base1Ptr->getData();

   base2Ptr = &d;
   cout << "\nbase2Ptr->getData() "
    << base2Ptr->getData() << endl;
   return 0;
}
```

En el ejemplo anterior se tiene un problema de ambigüedad al heredar dos métodos con el mismo nombre de clases diferentes. Se resuelve poniendo antes del nombre del miembro el nombre de la clase: *objeto.<clase::>miembro.* El nombre del objeto es necesario pues no se esta haciendo referencia a un miembro estático.

Ambigüedades.

En el ejemplo anterior se vio un caso de ambigüedad al heredar de clases distintas un miembro con el mismo nombre. Normalmente de deben tratar de evitar esos casos, pues vuelven confuso nuestro jerarquía de herencia.

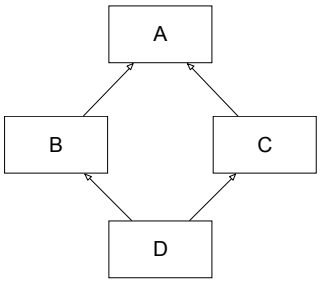
Existen otro casos donde es posible que se de la ambigüedad.

Ejemplo:

```
//ejemplo de ambigüedad en la herencia
class B{
public:
    int b;
};

class D: public B, public B { //error
};

void f( D *p) {
    p->b=0;  //ambiguo
}

void main(){
    D obj;

    f(&obj);
}
```

El ejemplo anterior tiene un error en la definición de herencia múltiple, ya que no es posible heredar más de una vez a una misma clase de manera directa.

Sin embargo, si es posible heredar las características de una clase más de una vez indirectamente:




```
//ejemplo de ambigüedad en la herencia
#include <iostream.h>

class A{
public:
    int next;
};

class B: public A{

};

class C: public A{

};
class D: public B, public C {
    int g();
};

int D::g(){
    //next=0;  Error: asignación ambigua
    return B::next == C::next;
}
class E: public D{
 public:
    int x;
    int getx(){
        return x;
    }
};
```

```
void main(){
     D obj;
     E obje;

     obj.B::next=10;
     obj.C::next=20;
//    obj.A::next=11; no se vale
//   obj.next=22;
     cout<<"next de B: "<<obj.B::next<<endl;
     cout<<"next de C: "<<obj.C::next<<endl;

     obje.x=0;
     obje.B::next=11;
     obje.C::next=22;
     cout<<"obje next de B: "<<obje.B::next<<endl;
     cout<<"obje next de C: "<<obje.C::next<<endl;
}
```

Este programa hace que las instancias de la clase D tengan objetos de clase base duplicados y provoca los accesos ambiguos. Este problema se resuelve con herencia virtual.

Herencia de clase base virtual: Si se especifica a una clase base como virtual, solamente un objeto de la clase base existirá en la clase derivada.

En este caso las clases B y c deben declarar a la clase A como clase base virtual:

```
class B: virtual public A {...}

class C: virtual public A {...}
```

Constructores.

Si los constructores de la clase base no tienen argumentos, no es necesario crear un constructor de la clase derivada. Pero si hay constructores con argumentos, es posible que sea necesario llamarlos desde el constructor de la clase derivada.

Para ejecutar los constructores de las clases base, pasando los argumentos, es necesario especificarlos después de la declaración de la función de construcción de la clase derivada, separados por coma.

Sintaxis general:

```
<Constructor clase derivada>(<argumentos> : <base1>
(<argumentos>), <base2> (<argumentos>), ... , <basen>
(<argumentos>) {
    ...
}
```

donde como en la herencia simple, el nombre base corresponde al nombre de la clase, o en este caso, clases base.

Funciones virtuales y polimorfismo.

La capacidad de polimorfismo permite crear programas con mayores posibilidad de expansiones futuras, aún para procesar en cierta forma objetos de clases que no han sido creadas o están en desarrollo [Deite, 1995].

El polimorfismo es implementado en C++ a través de clases derivadas y funciones virtuales.

Una función virtual es un método miembro declarado como *virtual* en una clase base y siendo este método redefinido en una o más clases derivadas.

Las funciones virtuales son muy especiales, debido a que cuando una función es accesada por un apuntador a una clase base, y este esta haciendo referencia a un objeto de una clase derivada, el programa determina en tiempo de ejecución a que función llamar, de acuerdo al tipo de objeto al que se apunta. Esto se conoce como ligadura tardía y el compilador de C++ incluye en el código máquina el manejo de ese tipo de asociación de métodos.

La utilidad se da cuando se tiene un método en una clase base, y este es declarado virtual. De esta forma, cada clase derivada puede tener su propia implementación del método si es que así lo requiere la clase; y si un apuntador a clase base hace referencia a cualquiera de los objetos de clases derivadas, se determina dinámicamente cual de todos los métodos debe ejecutar.

La sintaxis en C++ implica declarar al método de la clase base con la palabra reservada *virtual*, redefiniendo ese método en cada una de las clases derivadas.

Al declarar un método como virtual, este método se conserva asi a través de toda la jerarquía de herencia, del punto en que se declaro hacia abajo. Aunque de este modo no es necesario volver a usar la palabra virtual en ninguno de los métodos inferiores del mismo nombre, es posible hacerlo de forma explícita para que el programa sea más claro.

Es importante señalar que las funciones virtuales que sean redefinidas en clases derivadas, deben tener además de la misma firma que la función virtual base, el mismo tipo de retorno.

Sintaxis:

```
class base {
    Virtual <tipo> <método> (<parámetros);
};
```

Ejemplo:

```
 //ejemplo 1 funciones virtuales
#include <iostream.h>

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

class primera: public base {
public:
    void quien(){
        cout<<"primera\n";
    }
};

class segunda: public base {
public:
    void quien(){
        cout<<"segunda\n";
    }
};

class tercera: public base {
};
```

```
class cuarta: public base {
public:
/*   int quien(){   No se vale con un tipo de dato diferente
         cout<<"cuarta\n";
         return 1;
     }*/
};


void main() {
     base objBase, *pBase;
     primera obj1;
     segunda obj2;
     tercera obj3;
     cuarta  obj4;

     pBase=&objBase;
     pBase->quien();

     pBase=&obj1;
     pBase->quien();

     pBase=&obj2;
     pBase->quien();

     pBase=&obj3;
     pBase->quien();

     pBase=&obj4;
     pBase->quien();


}
```

Hay que hacer notar que las funciones virtuales puede seguirse usando sin apuntadores, mediante un objeto de la clase.

De esta forma, el método a ejecutar se determina de manera estática; es decir, en tiempo de compilación (ligadura estática). Obviamente el método a ejecutar es aquel definido en la clase del objeto o el heredado de su clase base.

Si se declara en una clase derivada un método con otro tipo de dato como retorno, el compilador manda un error, ya que esto no es permitido. Si se declara un método con el mismo nombre pero diferentes parámetros, la función virtual queda desactivada de ese punto hacia abajo en la jerarquía de herencia.

Clase abstracta y clase concreta.

Existen clases que son útiles para representar una estructura en particular, pero que no van a tener la necesidad de generar objetos directamente a partir de esa clase, estas se conocen como clases abtractas, o mejor dicho como clases base abstractas, puesto que sirven para definir una estructura jerarquica.

La clase base abstracta entonces, tiene como objetivo proporcionar una clase base que ayude al modelado de la jerarquía de herencia, aunque esta sea muy general y no sea práctico tener instancias de esa clase.

Por lo tanto, de una clase abstracta no se pueden definir objetos, mientras que en clases a partir de las cuales se puedan instanciar objetos se conocen como clases concretas.

En C++, una clase se hace abstracta al declarar **al menos uno** de los métodos virtuales como puro. Un método o función virtual pura es una que en su declaración tiene el inicializador de =0 :

```
virtual <tipo> <nombre>(<parámetros>) =0;   //virtual pura
```

Es importante tener en cuenta que una clase sigue siendo abstracta hasta que no se implemente la función virtual pura, en una de las clases derivadas. Si no se

hace la implementación, la función se hereda como virtual pura y por lo tanto la clase sigue siendo considerada como abstracta.

Aunque no se pueden tener objetos de clases abstractas, si se pueden tener apuntadores a objetos de esas clases, permitiendo una manipulación de objetos de las clases derivadas mediante los apuntadores a la clase abstracta.

Polimorfismo.

> El polimorfismo se define como la capacidad de objetos de clases diferentes, relacionados mediante herencia, a responder de forma distinta a una misma llamada de un método. [Deitel, 1995]

En C++, el polimorfismo se implementa con las funciones virtuales. Al hacer una solicitud de un método, a través de un apuntador a clase base para usar un método virtual, C++ determina el método que corresponda al objeto de la clase a la que pertenece, y no el método de la clase base.

Tener en cuenta que no es lo mismo que simplemente redefinir un método de clase base en una clase derivada, pues como se vio anteriormente, si se tiene a un apuntador de clase base y a través de el se hace la llamada a un método, se ejecuta el método de la clase base independientemente del objeto referenciado por el apuntador. Este no es un comportamiento polimórfico.

Destructores virtuales.

Cuando se aplica la instrucción *delete* a un apuntador de clase base, será ejecutado el destructor de la clase base sobre el objeto, independientemente de la clase a la que pertenezca. La solución es declarar al destructor de la clase base como virtual. De esta forma al borrar a un objeto se ejecutará el destructor de la clase a la que pertenezca el objeto referenciado, a pesar de que los destructores no tengan el mismo nombre.

Un constructor no puede ser declarado como virtual.

Ejemplos de funciones virtuales y polimorfismo:

Programa de cálculo de salario.

```
// EMPLEADO.H
// Abstract base class Employee
#ifndef EMPLEADO_H
#define EMPLEADO_H

class Employee {
public:
   Employee(const char *, const char *);
   ~Employee();
   const char *getFirstName() const;
   const char *getLastName() const;

   virtual float earnings() const = 0; // virtual pura
   virtual void print() const = 0;     // virtual pura
private:
   char *firstName;
   char *lastName;
};

#endif

// EMPLEADO.CPP
#include <iostream.h>
#include <string.h>
#include <assert.h>
#include "empleado.h"

Employee::Employee(const char *first, const char *last)
{
   firstName = new char[ strlen(first) + 1 ];
   assert(firstName != 0);
   strcpy(firstName, first);

   lastName = new char[ strlen(last) + 1 ];
   assert(lastName != 0);
   strcpy(lastName, last);
```

```
}

Employee::~Employee()
{
   delete [] firstName;
   delete [] lastName;
}

const char *Employee::getFirstName() const
{

   return firstName;
}

const char *Employee::getLastName() const
{
      return lastName;
}

// JEFE.H
// Clase drivada de empleado
#ifndef JEFE_H
#define JEFE_H
#include "empleado.h"

class Boss : public Employee {
public:
   Boss(const char *, const char *, float = 0.0);
   void setWeeklySalary(float);
   virtual float earnings() const;
   virtual void print() const;
private:
   float weeklySalary;
};

#endif

// JEFE.CPP
#include <iostream.h>
#include "jefe.h"
```

```
Boss::Boss(const char *first, const char *last, float s)
   : Employee(first, last)
{ weeklySalary = s > 0 ? s : 0; }

void Boss::setWeeklySalary(float s)
{ weeklySalary = s > 0 ? s : 0; }

float Boss::earnings() const { return weeklySalary; }

void Boss::print() const
{
   cout << "\n             Jefe: " << getFirstName()
    << ' ' << getLastName();
}

// COMIS.H
// Trabajador por comisión derivado de Empleado
#ifndef COMIS_H
#define COMIS_H
#include "empleado.h"

class CommissionWorker : public Employee {
public:
   CommissionWorker(const char *, const char *,
                    float = 0.0, float = 0.0, int = 0);
   void setSalary(float);
   void setCommission(float);
   void setQuantity(int);
   virtual float earnings() const;
   virtual void print() const;
private:
   float salary;     // salario base por semana
   float commission; // comisión por cada venta
   int quantity;     // cantidad de elementos vendidos por
semana
};

#endif
```

```
// COMIS.CPP
#include <iostream.h>
#include "comis.h"

CommissionWorker::CommissionWorker(const char *first,
   const char *last, float s, float c, int q)
   : Employee(first, last)
{
   salary = s > 0 ? s : 0;
   commission = c > 0 ? c : 0;
   quantity = q > 0 ? q : 0;
}

void CommissionWorker::setSalary(float s)
{ salary = s > 0 ? s : 0; }

void CommissionWorker::setCommission(float c)
{ commission = c > 0 ? c : 0; }

void CommissionWorker::setQuantity(int q)
{ quantity = q > 0 ? q : 0; }

float CommissionWorker::earnings() const
{ return salary + commission * quantity; }

void CommissionWorker::print() const
{
   cout << "\nTrabajador  por comision: " << getFirstName()
    << ' ' << getLastName();
}
```

```
// PIEZA.H
// Trabajador por pieza derivado de Empleado
#ifndef PIECE1_H
#define PIECE1_H
#include "empleado.h"

class PieceWorker : public Employee {
public:
   PieceWorker(const char *, const char *,
               float = 0.0, int = 0);
   void setWage(float);
   void setQuantity(int);
   virtual float earnings() const;
   virtual void print() const;
private:
   float wagePerPiece; // pago por cada pieza
   int quantity;       // piezas por semana
};

#endif

// PIEZA.CPP
#include <iostream.h>
#include "pieza.h"

// Constructor for class PieceWorker
PieceWorker::PieceWorker(const char *first,
             const char *last, float w, int q)
   : Employee(first, last)
{
   wagePerPiece = w > 0 ? w : 0;
   quantity = q > 0 ? q : 0;
}

void PieceWorker::setWage(float w)
{ wagePerPiece = w > 0 ? w : 0; }

void PieceWorker::setQuantity(int q)
{ quantity = q > 0 ? q : 0; }
```

```
float PieceWorker::earnings() const
{ return quantity * wagePerPiece; }

void PieceWorker::print() const
{
   cout << "\n     Tabajador por pieza: " << getFirstName()
    << ' ' << getLastName();
}

// HORA.H
// Trabajador por hora derivado de Empleado
#ifndef HOURLY1_H
#define HOURLY1_H
#include "empleado.h"

class HourlyWorker : public Employee {
public:
   HourlyWorker(const char *, const char *,
                float = 0.0, float = 0.0);
   void setWage(float);
   void setHours(float);
   virtual float earnings() const;
   virtual void print() const;
private:
   float wage;   // salario por hora
   float hours;  // horas trabajadas en la semana
};

#endif

// HORA.CPP
#include <iostream.h>
#include "hora.h"

HourlyWorker::HourlyWorker(const char *first, const char
*last,
                           float w, float h)
   : Employee(first, last)
{
   wage = w > 0 ? w : 0;
```

```
   hours = h >= 0 && h < 168 ? h : 0;
}

void HourlyWorker::setWage(float w) { wage = w > 0 ? w : 0; }

void HourlyWorker::setHours(float h)
{ hours = h >= 0 && h < 168 ? h : 0; }

float HourlyWorker::earnings() const { return wage * hours; }

void HourlyWorker::print() const
{
   cout << "\n    Trabajador por hora: " << getFirstName()
    << ' ' << getLastName();
}



// main.CPP
#include <iostream.h>
#include <iomanip.h>
#include "empleado.h"
#include "jefe.h"
#include "comis.h"
#include "pieza.h"
#include "hora.h"
main()
{
   // formato de salida
   cout << setiosflags(ios::showpoint) << setprecision(2);

   Employee *ptr;  // apuntador a clase base

   Boss b("John", "Smith", 800.00);
   ptr = &b;  // apuntador de clase base apuntando a objeto de
clase derivada
   ptr->print();                          // ligado dinámico
   cout << " ganado $" << ptr->earnings(); // ligado dinámico
   b.print();                             // ligado estático
   cout << " ganado $" << b.earnings();    // ligado estático
```

```
   CommissionWorker c("Sue", "Jones", 200.0, 3.0, 150);
   ptr = &c;
   ptr->print();
   cout << " ganado $" << ptr->earnings();
   c.print();
   cout << " ganado $" << c.earnings();

   PieceWorker p("Bob", "Lewis", 2.5, 200);
   ptr = &p;
   ptr->print();
   cout << " ganado $" << ptr->earnings();
   p.print();
   cout << " ganado $" << p.earnings();

   HourlyWorker h("Karen", "Precio", 13.75, 40);
   ptr = &h;
   ptr->print();
   cout << " ganado $" << ptr->earnings();
   h.print();
   cout << " ganado $" << h.earnings();

   cout << endl;
   return 0;
}
```

Programa de figuras geométricas con una interfaz abstracta Shape (Forma)

```
// Figura.H
#ifndef figura_H
#define figura_H

class Shape {
public:
   virtual float area() const { return 0.0; }
   virtual float volume() const { return 0.0; }
   virtual void printShapeName() const = 0; // virtual pura
};

#endif

// Punto.H
#ifndef Punto_H
#define Punto_H
#include <iostream.h>
#include "figura.h"

class Point : public Shape {
   friend ostream &operator<<(ostream &, const Point &);
public:
   Point(float = 0, float = 0);
   void setPoint(float, float);
   float getX() const { return x; }
   float getY() const { return y; }
   virtual void printShapeName() const { cout << "Punto: "; }
private:
   float x, y;
};

#endif
```

```
// Punto.CPP
#include <iostream.h>
#include "punto.h"
Point::Point(float a, float b)
{
   x = a;
   y = b;
}

void Point::setPoint(float a, float b)
{
   x = a;
   y = b;
}

ostream &operator<<(ostream &output, const Point &p)
{
   output << '[' << p.x << ", " << p.y << ']';
   return output;
}

// Circulo.H
#ifndef Circulo_H
#define Circulo_H
#include "punto.h"

class Circle : public Point {
   friend ostream &operator<<(ostream &, const Circle &);
public:
   Circle(float r = 0.0, float x = 0.0, float y = 0.0);

   void setRadius(float);
   float getRadius() const;
   virtual float area() const;
   virtual void printShapeName() const { cout << "Circulo: ";
}
private:
   float radius;
};
#endif
```

```
// Circulo.CPP
#include <iostream.h>
#include <iomanip.h>
#include "circulo.h"

Circle::Circle(float r, float a, float b)
   : Point(a, b)
 { radius = r > 0 ? r : 0; }

void Circle::setRadius(float r) { radius = r > 0 ? r : 0; }

float Circle::getRadius() const { return radius; }

float Circle::area() const { return 3.14159 * radius * radius;
}

ostream &operator<<(ostream &output, const Circle &c)
{
   output << '[' << c.getX() << ", " << c.getY()
          << "]; Radio=" << setiosflags(ios::showpoint)
          << setprecision(2) << c.radius;

   return output;
}

// Cilindro.H
#ifndef Cilindro_H
#define Cilindro_H
#include "circulo.h"

class Cylinder : public Circle {
   friend ostream &operator<<(ostream &, const Cylinder &);
public:
   Cylinder(float h = 0.0, float r = 0.0,
         float x = 0.0, float y = 0.0);

   void setHeight(float);
   virtual float area() const;
   virtual float volume() const;
```

```
   virtual void printShapeName() const { cout << "Cilindro: ";
}
private:
   float height;   // altura del cilindro
};

#endif

// Cilindro.CPP
#include <iostream.h>
#include <iomanip.h>
#include "cilindro.h"

Cylinder::Cylinder(float h, float r, float x, float y)
   : Circle(r, x, y)
{ height = h > 0 ? h : 0; }

void Cylinder::setHeight(float h)
   { height = h > 0 ? h : 0; }

float Cylinder::area() const
{
      return 2 * Circle::area() +
          2 * 3.14159 * Circle::getRadius() * height;
}

float Cylinder::volume() const
{
   float r = Circle::getRadius();
   return 3.14159 * r * r * height;
}

ostream &operator<<(ostream &output, const Cylinder& c)
{
   output << '[' << c.getX() << ", " << c.getY()
          << "]; Radio=" << setiosflags(ios::showpoint)
          << setprecision(2) << c.getRadius()
          << "; Altura=" << c.height;
   return output;
}
```

```
// main.CPP
//include...
main(){
   Point point(7, 11);
   Circle circle(3.5, 22, 8);
   Cylinder cylinder(10, 3.3, 10, 10);

   point.printShapeName();     // ligado estático
   cout << point << endl;

   circle.printShapeName();
   cout << circle << endl;

   cylinder.printShapeName();
   cout << cylinder << "\n\n";
   cout << setiosflags(ios::showpoint) << setprecision(2);
   Shape *ptr;                 // apuntador de clase base

   // apuntador de clase base referenciando objeto de clase
derivada
   ptr = &point;
   ptr->printShapeName();      // ligado dinámico
   cout << "x = " << point.getX() << "; y = " << point.getY()
    << "\nArea = " << ptr->area()
    << "\nVolumen = " << ptr->volume() << "\n\n";

   ptr = &circle;
   ptr->printShapeName();
   cout << "x = " << circle.getX() << "; y =" << circle.getY()
    << "\nArea = " << ptr->area()
    << "\nVolumen = " << ptr->volume() << "\n\n";

   ptr = &cylinder;
   ptr->printShapeName();      // dynamic binding
   cout << "x = " << cylinder.getX() << "; y = " <<
cylinder.getY()
    << "\nArea = " << ptr->area()
    << "\nVolumen = " << ptr->volume() << endl;
   return 0;
}
```

Plantillas de clase.

El concepto de plantillas es aplicable también a la programación orientada a objetos en C++a través de plantillas de clase. Estas favorecen la reutilización de software, permitiendo que se generen objetos específicos para un tipo a partir de clases genéricas. Las plantillas de clase también son llamadas clases parametrizadas.

El uso de plantillas de clase no es diferente al uso de plantillas en operaciones no orientadas a objetos:

```
template < class T >
```

Veamos el ejemplo clásico de un programa de pila aprovechando el uso de plantillas.

Ejemplo:

```
// stack.h
// Clase de plantilla Pila
#ifndef TSTACK1_H
#define TSTACK1_H

#include <iostream.h>

template< class T >
class Stack {
public:
    Stack( int = 10 );
    ~Stack() { delete [] stackPtr; }
    char push( const T& );
    char pop( T& );
private:
    int size;
    int top;
    T *stackPtr;
```

```
    char isEmpty() const { return top == -1; }
    char isFull() const { return top == size - 1; }
};


template< class T >
Stack< T >::Stack( int s )
{
    size = s > 0 ? s : 10;
    top = -1;
    stackPtr = new T[ size ];
}

template< class T >
char Stack< T >::push( const T &pushValue )
{
    if ( !isFull() ) {
        stackPtr[ ++top ] = pushValue;
        return 1;
    }
    return 0;
}

template< class T >
char Stack< T >::pop( T &popValue )
{
    if ( !isEmpty() ) {
        popValue = stackPtr[ top-- ];
        return 1;
    }
    return 0;
}

#endif
```

```
// Ejemplo uso de plantillas de clase
#include <iostream.h>
#include "stack.h"
int main()
{
    Stack< double > doubleStack( 5 );
    double f = 1.1;
    cout << "Insertando elementos en doubleStack \n";

    while ( doubleStack.push( f ) ) {
        cout << f << ' ';
        f += 1.1;
    }

    cout << "\nLa pila está llena. No se puede insertar el
elemento " << f
         << "\n\nSacando elementos de doubleStack\n";

    while ( doubleStack.pop( f ) )
        cout << f << ' ';

    cout << "\nLa pila está vacía. No se pueden eliminar más
elementos\n";

    Stack< int > intStack;
    int i = 1;
    cout << "\nInsertando elementos en intStack\n";

    while ( intStack.push( i ) ) {
        cout << i << ' ';
        ++i;
    }

    cout << "\nLa pila está llena.  " << i
         << "\n\nSacando elementos de intStack\n";

    while ( intStack.pop( i ) )
        cout << i << ' ';
```

```
    cout << "\nLa pila está vacía. No se pueden eliminar más
elementos \n";
    return 0;
}
```

Las plantillas de clase ayudan a la reutilización de código, al permitir varias versiones de clases para un tipo de dato a partir de clases genéricas. A estas clases específicas se les conoce como clases de plantilla.

Con respecto a la herencia en combinación con el uso de plantillas, se deben tener en cuenta las siguientes situaciones:

- Una plantilla de clase se puede derivar de una clase de plantilla.
- Una plantilla de clase se puede derivar de una clase que no sea plantilla.
- Una clase de plantilla se puede derivar de una plantilla de clase.
- Una clase que no sea de plantilla se puede derivar de una plantilla de clase.

En cuanto a los miembros estáticos, cada clase de plantilla que se crea a partir de una plantilla de clases mantiene sus propias copias de los miembros estáticos.

Manejo de Excepciones.

Introducción.

Siempre se ha considerado importante el manejo de los errores en un programa, pero no fue hasta que surgió el concepto de manejo de excepciones que se dio una estructura más formal para hacerlo.

El término de excepción viene de la posibilidad de detectar eventos que no forman parte del curso normal del programa, pero que de todas formas ocurren.

Un evento "*excepcional*" puede se por una falla en la conexión a red, un archivo que no puede encontrarse, o un acceso indebido en memoria. La intención de una excepción es responder de manera dinámica a los errores, sin que afecte gravemente la ejecución de un programa, o que al menos se controle la situación posterior al error.

¿Cuál es la ventaja con respecto al manejo común de errores?

Normalmente, cada programador agrega su propio código de manejo de errores y queda revuelto con el código del programa. El manejo de excepciones indica claramente en que parte se encuentra el manejo de los errores, separándolo del código normal.

Además, es posible recibir y tratar muchos de los errores de ejecución y tratarlos correctamente, como podría ser una división entre cero.

Se recomienda el manejo de errores para aquellas situaciones en las cuales el programa necesita ayuda para recuperarse.

Excepciones en C++.

El manejo de excepciones en C++, involucra los siguientes elementos sintácticos:

- ***try***. El bloque definido por la instrucción *try*, especifica el código que potencialmente podría generar un error que deba ser manejado por la excepción:

```
try
{
    // instrucciones donde las excepciones
    // pueden ser generadas
}
```

- ***throw***: Esta instrucción seguida por una expresión de un cierto tipo, genera una excepción del tipo de la expresión. Esta instrucción debería ser ejecutada dentro de algún bloque *try*, de manera directa o indirecta:

```
throw "Se genera una excepción de tipo char *";
```

- ***catch:*** La instrucción *catch* va seguida de un bloque *try*. *Catch* define un segmento de código para tratar una excepción (de un tipo) lanzada:

```
catch (char *mensaje)
{
    // instrucciones donde la excepción
    // thrown char *
    // será procesada
}
```

Bibliografía

De consulta y de referencia para este documento.

1. Deitel, P.J.; Deitel H.M. **COMO PROGRAMAR EN C/C++**. Prentice Hall. 2ª edición. México. 1995.
2. Schildt, Herbert. **APLIQUE TURBO C++.** Osbrone/ Mc Graw-Hill. 1ª edición. 1991.
3. Cox, Brad j.; Novobilski, Andrew J. **PROGRAMACIÓN ORIENTADA A OBJETOS. Un enfoque evolutivo**. Addison Wesley/Díaz de Santos. 2ª edición. 1993.
4. Pappas, chris H.; Murray, William H. **MANUAL DE BORLAND C++.** Mc Graw-Hill. 3ª edición. 1993.
5. Fundación Arturo Rosenblueth. **PROGRAMACIÓN ORIENTADA A OBJETOS EN C++.** México.
6. Müller, Peter. **INTRODUCCIÓN A LA PROGRAMACIÓN ORIENTADA A OBJETOS EMPLEANDO C++.** Globewide Network Academy (GNA). 1997. http://uu-gna.mit.edu:8001/uu-gna/text/cc/Tutorial//tutorial.html
7. Weitzenfeld, Alfredp. **PARADIGMA ORIENTADO A OBJETOS.** División Académica de Computación. Depto. Académico de Computación. ITAM. México. 1994.
8. Muller, Pierre-Alain. **MODELADO DE OBJETOS CON UML**. Edit. Gestión 2000. España. 1997.
9. Deitel, P.J.; Deitel H.M. **COMO PROGRAMAR EN C++**. Prentice Hall. 2ª edición. México. 1999.