Resumen poo

Conceptos fundamentales

Abstracción: sacar lo esencial lo que define al objeto como tal.

Encapsulamiento: el estado de un objeto esta encapsulado y no puede accederse directamente desde el exterior, para acceder a estos se debe utilizar sus métodos.

Clase: es un patrón para crear un objeto en este se definen sus características (atributos) y sus comportamientos (métodos).

Constructor: es un método de la clase que permite crear un objeto.

Objeto: es una instancia de una clase, posee un estado además de métodos y atributos.

Herencia: capacidad de crear nuevas clases basándose en clases previas, se aprovechan algunos de sus métodos y atributos, se desechan otros y se incluyen nuevos.

Mensaje (métodos): es la forma en que un objeto se comunica con otro.

Composición (agregacion): es cuando una clase forma parte de una clase mayor.

Polimorfismo: es cuando un método actúa de diferente manera para un mismo mensaje.

Programación genérica es un tipo de programación que se centra en generalizar los algoritmos para que funcionen con distintos tipos de datos.

Reutilización de código: son métodos que se utilizan para no escribir código redundante ni empezar de cero cuando ya habíamos codificado algo similar.

Relación entre herencia composición y reutilización de código?

Herencia y composición son formas de reutilización de código.

Archivos

Archivos de texto

1) Acceso secuencial
2) se pueden leer y modificar fácilmente
3) más pesados que los binarios
4) mejor portabilidad que los binarios

Archivos binarios

1) Se puede acceder directamente
2) es más seguro q el de texto ya q no es legible al abrirlo normalmente
3) es más liviano q el de texto
4) es menos portable q el de texto ya q algunos microprocesadores leen los archivos binarios de forma distinta (binario en mac != binario en windows)

Sobrecarga de operadores

Lista de operadores estándar de C++ que es posible sobrecargar:

+		**-**	*****	**/**	**%**	**=**	**<**	**>**	**+=**
***=**		**/=**	**<<**	**>>**	**<<=**	**>>**	**=**	**==**	**!=**
<=		**>=**	**++**	**--**	**%**	**&**	**^**	**!**	**|**
&=	**^=**	**|=**	**&&**	**||**	**%=**	**->**	**->***	**[ ]**	
( )		**~**	**new**	**delete**					

Los siguientes operadores de C++ no pueden sobrecargarse:

. (acceso a miembro) **.*** indirección de acceso a miembro

:: resolución e ámbito **?:** If abreviado

Implementación

La sobrecarga es parte de la clase cuando se implementa como función miembro

Si se implementa como función no miembro será amiga de la clase

- Como función miembro, por ej.:

```
class A
{ public :
    A operator+(A obj ) ;
};
A A::operator+(A obj )
{  //… }
```

- Como función no miembro (generalmente **friend**), por ej.:

```
class A
{public :
   friend A operator+( tipox param1 , A obj2) ;
};
A operator+(tipox param1 , A obj2 )
{  //… }
```

Una función operador no necesita ser una función miembro, pero existen cuatro operadores que sólo pueden ser sobrecargados a través de funciones miembro: "=", "[]", "()" y "->"

La sobrecarga de un operador como función no miembro permitiría definir y realizar operaciones en las que el primer parámetro no pertenece a una clase definida por el programador, por ej: 5+a2

Operador de asignación

La función es miembro de la clase (complejo::, A::)

Es importante no confundir el operador de asignación, que modifica un objeto existente, con el constructor de copia, que crea un nuevo objeto. Para que resulte posible encadenar asignaciones (`a=b=c=d`) la función `operator=` debe finalizar con la sentencia: `return * this ;`

```
Complejo& Complejo::operator=(const Complejo& c)
{    pReal=c.pReal;
     pImag=c.pImag;
     return *this;   }
```

Si se necesita borrar (`delete`) el valor del objeto situado en la parte izquierda del operador de asignación, debe prevenirse la posibilidad de que lo que se pretenda ejecutar sea una autoasignación (`obj1 = obj1`) y evitar que el borrado elimine el objeto a asignar en dichos casos

```
A& A::operator=(const A& obj)
{  if ( this ! = & obj )
   {
      // borrar...
   }
   return *this; }
```

Sobrecarga operadores de inserción y extracción

Estos operadores se sobrecargan como funciones no miembro ya que han de seguir los siguientes prototipos y su primer argumento no puede ser un objeto de la clase que añade o retira del flujo.

- `ostream& operator<< (ostream& o, const clase& obj);`
- `istream& operator>> (istream& i, clase& obj);`

Ejemplo:

```
// sobrecarga para la lectura de un número complejo
istream& operator>>(istream &o, Complejo& c)
{
     o>>c.pReal;
     o>>c.pImag;
     return o;
}
```

Al ser no miembro se declara en el .h como friend

Archivo .h

```
#ifndef R_
#define R_
class R
{ int n , d;
  int Euclides(int, int);
 public :
  R(int, int);
  R& operator ++( ) ;
  R operator ++( int ) ;
  friend ostream& operator<<(ostream& , const R&);
};
#endif
```

Lo importante de esto es darse cuenta que la implementación de lasobrecarga a pesar de estar en R.cpp no es un método de r sino q es amiga por eso no

lleva R:: como los otros métodos q si pertenecen a la clase R

Archivo .cpp (cont.)

```
int R::Euclides(int n, int d)
{    int r;
     while (d>0)
     { r = n % d;
       n = d;
       d = r; }
     return n;    }

ostream& operator<< (ostream& ostr, const R& r)
{ if (r.d == 1)
    ostr << r.n;
  else
    ostr << r.n << "/" << r.d;
  return ostr; }
```

Stl

La Standard Template Library (STL) es una colección de estructuras de datos genéricas y algoritmos escritos en C++.

Ventajas

La adopción de STL ofrece varias ventajas:

- Al ser estándar, está (o estará) disponible por todos los compiladores y plataformas. Esto permitirá utilizar la misma librería en todos los proyectos y disminuirá el tiempo de aprendizaje necesario para que los programadores cambien de proyecto.
- El uso de una librería de componentes reutilizables incrementa la productividad ya que los programadores no tienen que escribir sus propios algoritmos. Además, utilizar una librería libre de errores no sólo reduce el tiempo de desarrollo, sino que también incrementa la robustez de la aplicación en la que se utiliza.
- Las aplicaciones pueden escribirse rápidamente ya que se construyen a partir de algoritmos eficientes y los programadores pueden seleccionar el algoritmo más rápido para una situación dada.
- Se incrementa la legibilidad del código, lo que hará que éste sea mucho más fácil de mantener.
- Proporciona su propia gestión de memoria. El almacenamiento de memoria es automático y portátil, así el programador ignorará problemas tales como las limitaciones del modelo de memoria del PC.

-

Stl se divide en tres partes

- Contenedores
- Iteradores
- Algoritmos

1) Contenedores:

Los contenedores se dividen en 3 categorías principales: Contenedores de secuencia, contenedores asociativos y adaptadores de contenedores.

- Contenedores lineales. Almacenan los objetos de forma secuencial permitiendo el acceso a los mismos de forma secuencial y/o aleatoria en función de la naturaleza del contenedor.
- Contenedores asociativos. En este caso cada objeto almacenado en el contenedor tiene asociada una clave. Mediante la clave los objetos se pueden almacenar en el contenedor, o recuperar del mismo, de forma rápida.
- Contenedores adaptados. Permiten cambiar un contenedor en un nuevo contenedor modificando la *interface* (métodos públicos y dato miembro) del primero. En la mayor parte de los casos, el nuevo contenedor únicamente requerirá un subconjunto de las capacidades que proporciona el contenedor original.

2) Iteradores: una generalización del puntero que permite al programador visitar cada elemento de una estructura de datos contenedora.

Tipos:

- ***Input iterators***
- ***Output iterators***
- ***Forward iterators***
- ***Bidirectional iterators***
- ***Random access iterators***

Input iterators

Los iteradores de entrada sólo se pueden mover hacia delante y sólo pueden utilizarse para recuperar valores, no para valores de salida.

Output iterators

Los iteradores de salida, como los de entrada, únicamente se pueden mover hacia delante pero difieren en que sólo se pueden desreferenciar para asignar un valor, no para recuperarlo.

Forward iterators

Los iteradores hacia delante están diseñados para recorrer contenedores cuyos valores puedan ser añadidos y recuperados. Esto relaja algunas de las restricciones de los iteradores de entrada/salida pero mantienen la restricción de que únicamente se pueden mover hacia delante.

Bidirectional iterators

Los iteradores bidireccionales eliminan la restricción de los anteriores en los que sólo se puede avanzar hacia delante.

Random access iterators

Los iteradores de acceso aleatorio eliminan la restricción de que un iterador sólo puede avanzar al siguiente elemento, o retroceder al elemento previo, del contenedor en una única operación.

3) Algoritmos

Funciones genéricas para utilizar con contenedores de stl

Ejemplo de funciones de vector dinámico con punteros dentro de una clase

Vector dinámico

Algo.h

```
class algo
{
private:
   Int *vec; // cambien el int por cualquier clase que quieran crear el vector dinámico

    Int cant_vec; //cantidad de element del vector

public:
    algo();
   Void  agregar_vec (int a); // cambien el int por cualquier clase que quieran crear el
vector dinámico

    Void eliminar_vec (int pos); //acá borro lo q esta en esa posicion (pos) si tienen q
eliminar un dato en particular harían parecido solo q primero buscan la posicion del dato y
después usan esto q yo hice
    ~algo();
};
```

Algo.cpp

```
#include "algo.h"
```

```
algo::algo()
{cant_vec=0; //inicializo cantidad en 0;
}
void algo::agregar_vec(int a)
{
    int *aux=new int[cant_vec+1]; //creo vector auxiliar con uno más de capacidad
    For (int i=0; i<cant_vec; i++) {
        Aux[i]=vec[i];  //copio al nuevo vector lo q habia en el anterior
    }
    Aux [cant_vec]=a; //agrego en la posicion ultima posicion osea siempre va a ser en
cant_vec el nuevo dato
    If (cant_vec>0){
        delete []vec;//esto se hace para limpiar la memoria que ocupaba el antiguo vector
    }
    vec=aux; //asigno al vec el vector nuevo
    cant_vec++; //incremento  cant
}


void algo::eliminar_vec(int pos)
{
    Int *aux=new int [cant_vec-1]; //creo vector auxiliar con uno menos de capacidad
    For (int i=0; i<pos; i++) {
          Aux[i]=vec[i]; //copio al nuevo vector lo q habia en el anterior hasta la
posicion q quiero eliminar
    }
    For (int j=pos; j<cant_vec; j++) {
        Aux[j]=vec [j+1]; // copio al nuevo vector lo q habia en el anterior desde la
posicion siguiente al que deseo eliminar
    }
    vec =aux;//asigno al vec el vector nuevo
    Cant_vec--; //decremento  cant
}
```