Programación Orientada a Objetos en C++

FACULTAD DE CIENCIA Y TECNOLOGÍA - UADER

General

Programación Orientada a Objetos (POO)

- Es un paradigma (modelo de trabajo) de programación que agrupa los datos y los procedimientos para manejarlos en una única entidad: el objeto

Objeto

- Un objeto es una unidad que engloba en sí mismo variables y funciones necesarios para el tratamiento de esos datos.

Cada programa es un objeto
que a su vez está formado por objetos
que se relacionan entre ellos.

Definiciones

Método

- Función perteneciente a determinado objeto.

Atributo

- Variable perteneciente a determinado objeto.

Los objetos se comunican e interrelacionan entre sí a través del acceso a sus atributos y del llamado a sus métodos.

Definiciones

Clase

- Se puede considerar como un patrón para construir objetos.

Interfaz

- Es la parte del objeto que es visible para el resto de los objetos. Es decir, es el conjunto de métodos y atributos que dispone un objeto para comunicarse con él.

- Un objeto es una instancia (un ejemplar) de una clase determinada.
- Las clases tienen partes publicas y partes privadas. A la parte pública se la llama interfaz.

Definiciones

Herencia

- Capacidad de crear nuevas clases basándose en clases previamente definidas de las que se aprovechan ciertos datos y métodos, se desechan otros y se añaden nuevos.

Jerarquía

- Orden de subordinación de un sistema de clases.

Polimorfismo

- Propiedad según la cual un mismo objeto puede considerarse como perteneciente a distintas clases.

Declaración

Sintaxis

```
class  <identificador  de clase> [<lista de clases  base>] {
        <lista de  miembros>
};
```

La lista de clases base se utiliza para derivar clases.

La lista de miembros comprende una lista de funciones y datos.

Ejemplo

```
class punto{
private:
   //Atributos de la clase punto
   float coordX, coordY;
public:
   //Métodos de la clase punto
   void Carga(float x, float y);
   void Lee(float &x, float &y);
};
```

Acceso

Sintaxis

```
class  <identificador  de clase> {
public:
        <lista de  miembros>
private:
       <lista de miembros>
protected:
        <lista de miembros>
};
```

Acceso privado (private)

- Sólo son accesibles por los propios miembros de la clase, pero no desde funciones externas o desde funciones de clases derivadas. Es el acceso por defecto.

Acceso público (public)

- Cualquier miembro público de una clase es accesible desde cualquier otra parte donde sea accesible el propio objeto.

Acceso protegido (protected)

- Respecto a las funciones externas, es equivalente al acceso privado, pero con respecto a las clases derivadas se comporta como el público.

Métodos (Funciones miembro)

La lista de métodos puede ser

- Simplemente una declaración de prototipos
- Pueden ser también definiciones

Cuando se fuera se debe utilizar el operador de ámbito "::"

Las funciones definidas de este modo serán tratadas como *inline* (se inserta código cada vez que son llamadas)

Ejemplo

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

class punto{
private:
   //Atributos de la clase punto
   float coordX, coordY;
public:
   //Métodos de la clase punto
   void Carga(float x, float y);
   void Lee(float &x, float &y);
};
void punto::Carga(float x, float y){
   coordX=x;
   coordY=y;
}
void punto::Lee(float &x, float &y){
   x=coordX;
   y=coordY;
}
```

```
int main(int argc, char *argv[])
{
   punto A;
   A.Carga(10,5);
   float x1,y1;
   A.Lee(x1,y1);
   std::cout << "La coordenada X es:" << x1 <<
std::endl;
   std::cout << "La coordenada Y es:" << y1 <<
std::endl;
   }
```

El puntero this

Para cada objeto declarado de una clase se mantienen todos sus datos, pero todos comparten la misma copia de las funciones de la clase.

- Esto ahorra memoria y hace que los programas ejecutables sean más compactos, pero plantea un problema.

Cada función de una clase puede hacer referencia a los datos de un objeto, modificarlos o leerlos, usando el puntero especial llamado `this` que apunta al mismo objeto.

Ejemplo this

```
void punto::Carga(float x, float y){
    this->coordX=x;
    this->coordY=y;
}
```

Caso this

```
#include <iostream>
using namespace std;
class clase{
public:
    void EresTu(clase &C){
      if (&C==this) cout << "Si soy yo" << endl;
         else cout << "No soy yo" << endl;
    }
};

int main()
{
    clase C1,C2;
    C1.EresTu(C1);
    C1.EresTu(C2);
}
```

Constructores

Son métodos especiales que sirven para inicializar un objeto de una determinada clase al mismo tiempo que se declara.

Son especiales por varios motivos:

- Tienen el mismo nombre que la clase a la que pertenecen.
- No tienen tipo de retorno, y por lo tanto no retornan ningún valor.
- No pueden ser heredados.
- Deben ser públicos, no tendría ningún sentido declarar un constructor como privado, ya que siempre se usan desde el exterior de la clase, ni tampoco como protegido, ya que no puede ser heredados.

Sintaxis constructor

```
class  <identificador  de clase> {
public:
    <identificador de clase>(<lista  de parametros>)[: <lista de constructores>]  {
<codigo del constructor>
}
...
}
```

Ejemplo constructor

Ejemplo constructor

```
class punto{
private:
public:
    //Métodos de la clase punto
    punto(float x, float y);
    void Carga(float x, float y);
    void Lee(float &x, float &y);
};

punto::punto(float x, float y){
    coordX=x;
    coordY=y;
}
```

Si no definimos un constructor el compilador creará uno por defecto, sin parámetros.

Si definimos in constructor que requiere argumentos, es obligatorio suministrarlos.

Será llamado siempre que se declare un objeto de esa clase.

Ejemplo constructor

```
#include <QCoreApplication>
#include <iostream>
#include <math.h>

class punto{
    ...
public:
    //Métodos de la clase punto
    punto(float x, float y);
    void Carga(float x, float y);
    void Lee(float &x, float &y);
};

punto::punto(float x, float y){
    coordX=x;
    coordY=y;
}

...
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    punto A(10,5);
    float x1,y1;
    A.Lee(x1,y1);
    std::cout << "La coordenada X es:" << x1 << std::endl;
    std::cout << "La coordenada Y es:" << y1 << std::endl;
    return a.exec();
}
```

Inicialización de objetos

Se pueden inicializar los datos miembros de los objetos en los constructores invocando los constructores de ellos mismos.

En C++ incluso las variables del tipo básico como int, float o char pueden ser tratados como objetos.

Cada inicializador consiste de un nombre de variable miembro a inicializar seguido de la expresión que se usará para inicializarla entre paréntesis.

Los inicializadores se añadirán a continuación de los parámetros del constructor, antes del cuerpo del constructor y separado del paréntesis por **:**

Es preferible usar la inicialización, siempre que se pueda, en vez de asignación.

Ejemplo inicialización de objetos

```
punto::punto(float x, float y): coordX(x), coordY(y) {}
```

Sobrecarga de constructores

Los constructores son funciones, por lo que pueden definirse varios constructores para cada clase (sobrecarga).

Como en las funciones, no pueden declararse varios constructores con el mismo número y mismo tipo de argumentos.

Ejemplo constructor sobrecargado

```
class punto{
private:
    //Atributos de la clase punto
    float coordX, coordY;
public:
    //Métodos de la clase punto
    punto(float x, float y): coordX(x),coordY(y) {};
    punto():coordX(0),coordY(0) {};
    void Carga(float x, float y);
    void Lee(float &x, float &y);
};
```

Argumentos por defecto

Como a cualquier función pueden asignarse valores por defecto a los argumentos del constructor.

De esta manera se puede reducir la cantidad de constructores necesarios.

Ejemplo constructor sobrecargado

```
class punto{
private:
    //Atributos de la clase punto
    float coordX, coordY;
public:
    //Métodos de la clase punto
    //punto(float x, float y): coordX(x),coordY(y) {};
    //punto():coordX(0),coordY(0) {};
    punto(float x=0.0, float y=0.0): coordX(x),coordY(y) {};
    void Carga(float x, float y);
    void Lee(float &x, float &y);
};
```

Asignación de objetos

La asignación está permitida entre objetos ya existentes (como siempre)

El compilador crea un operador de asignación por defecto.

Este copia los valores de todos los atributos de un objeto a otro.

Ejemplo asignación de objetos

```
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    punto A(10,5),B;
    B=A;
    float x1,y1;
    A.Lee(x1,y1);
    std::cout << "La coordenada X es:" << x1 << std::endl;
    std::cout << "La coordenada Y es:" << y1 << std::endl;
    B.Lee(x1,y1);
    std::cout << "La coordenada X es:" << x1 << std::endl;
    std::cout << "La coordenada Y es:" << y1 << std::endl;
    return a.exec();
}
```

Constructor de copia

Crea un objeto que no existe a partir de otro existente

Sólo tienen un argumento, que es una referencia a un objeto de la misma clase.

El compilador crea un constructor copia por defecto que copia los valores de todos los atributos de un objeto a otro.

Ejemplos uso de constructor de copia (IntroPOO2)

```
//Constructor de copia
   punto A1(20,12);

   punto A2(A1); //caso explícito
   A2.Lee(x1,y1);
   std::cout << "La coordenada X es:" << x1 << std::endl;
   std::cout << "La coordenada Y es:" << y1 << std::endl;

   punto A3=A1; //caso implícito
   A3.Lee(x1,y1);
   std::cout << "La coordenada X es:" << x1 << std::endl;
   std::cout << "La coordenada Y es:" << y1 << std::endl;

   punto A4=14; //caso implícito con objeto diferente
   A4.Lee(x1,y1);
   std::cout << "La coordenada X es:" << x1 << std::endl;
   std::cout << "La coordenada Y es:" << y1 << std::endl;

   punto A5=punto(14,20); //caso implícito con objeto diferente
   A5.Lee(x1,y1);
   std::cout << "La coordenada X es:" << x1 << std::endl;
   std::cout << "La coordenada Y es:" << y1 << std::endl;
```

Copia superficial vs Copia profunda

Tanto la asignación como el constructor copia por defecto hacen una copia superficial.

- Sólo se copian los miembros pero no la memoria a que estos apuntan (para el caso de punteros)
- Por lo tanto al copiarse los valores de los punteros ambos objetos terminan apuntando a la misma posición de memoria.

La copia profunda implica además de hacer una copia de los valores en memoria hacia los bloques donde apunten los punteros miembros del objeto.

Para realizar la copia profunda pueden definirse por el usuario tanto el operador de asignación como el constructor de copia.

Ejemplo copia profunda (IntroPOO3)

```
//constructor de copia con copia profunda
#include <QCoreApplication>
#include <iostream>
#include <math.h>

class punto{
private:
   //Atributos de la clase punto
   float *coord;
public:
   //Métodos de la clase punto
   punto(float x=0.0, float y=0.0);
   punto(punto &P);
   void Carga(float x, float y);
   void Lee(float &x, float &y);
};

punto::punto(float x, float y){
   coord=new float[2];
   coord[0]=x; coord[1]=y;
}

punto::punto(punto &P){
   coord=new float[2];
   coord[0]=P.coord[0];
   coord[1]=P.coord[1];
}
```

```
int main(int argc, char *argv[])
{
   QCoreApplication a(argc, argv);
   punto A(10,5),B;
   B=A;
   float x1,y1;
   A.Lee(x1,y1);
   std::cout << "La coordenada X es:" << x1 << std::endl;
   std::cout << "La coordenada Y es:" << y1 << std::endl;
   B.Lee(x1,y1);
   std::cout << "La coordenada X es:" << x1 << std::endl;
   std::cout << "La coordenada Y es:" << y1 << std::endl;

   //Constructor de copia
   punto A1(20,12);
   punto A2(A1); //caso explícito
   punto A3=A1; //caso implícito
   //punto A4=14; //caso implícito con objeto diferente

   A1.Carga(-3,-2);
   A2.Lee(x1,y1);
   std::cout << "La coordenada X es:" << x1 << std::endl;
   std::cout << "La coordenada Y es:" << y1 << std::endl;

   A3.Lee(x1,y1);
   std::cout << "La coordenada X es:" << x1 << std::endl;
   std::cout << "La coordenada Y es:" << y1 << std::endl;

   return a.exec();
```

Destructores

Son métodos especiales que sirven para eliminar un objeto de determinada clase.

Realizan procesos necesarios cuando un objeto termina su ámbito temporal.

- Liberan la memoria dinámica utilizada por dicho objeto.
- Liberan los recursos usados, como archivos, dispositivo, etc.

Tienen algunas características especiales:

- Tienen el mismo nombre que la clase a la que pertenecen, pero tienen el símbolo ~ adelante.
- No tienen tipo de retorno, por lo tanto no devuelven ningún valor.
- No tienen parámetros.
- No pueden ser heredados.
- Deben ser públicos.
- No pueden ser sobrecargados, lo cual es obvio, ya que al no tener parámetros ni valor de retorno, no hay posibilidad de sobrecarga.

Destructores (cont.)

Cuando se define un destructor para una clase, éste es llamado automáticamente cuando se abandona el ámbito en el que fue definido.

Esto es así salvo cuando el objeto fue creado dinámicamente con el operador ***new***, ya que en ese caso, cuando es necesario eliminar el objeto, hay que hacerlo explícitamente utilizando el operador ***delete***.

Clase cadena de caracteres

Escriba una clase cadena que permita almacenar cadenas de caracteres. Brinde las funciones para cargar el valor desde una cadena C o desde otro objeto de la misma clase.

Debe tener la funcionalidad para concatenar 2 cadenas.

Clase: Cadena
Char * cad

Constructores
Cadena (char *)
Cadena (Cadena c)

Asignar(char *orig)
Concatenar(char *orig2)
Leer(char *dest)

Destructor
~Cadena()

Ejemplos de clase (Cad01)

```
#include <QCoreApplication>
#include <iostream>
#include <cstring>
using namespace std;

class cadena{
private:
    char *cad;
public:
    cadena(); //constructor por defecto
    cadena(const char *c); //constructor desde cadena c
    cadena(const int n); //cadena de n caracteres vacía
    cadena(const cadena &cx); //constructor de copia
    void Asignar(const char *orig);
    void Concatenar(cadena orig2);
    char* Leer(char *dest);
    ~cadena();
};
```

Ejemplo constructores, destructores, métodos. Prog ppal

```
cadena::cadena():cad(NULL) {};

cadena::cadena(const char *c){
   cad=new char[strlen(c)+1];
   strcpy(cad,c);
}
cadena::cadena(const int n){
   cad=new char[n+1];
   cad[0]=0;
}
cadena::cadena(const cadena &cx){
   cad=new char[strlen(cx.cad)+1];
   strcpy(cad,cx.cad);
}
cadena::~cadena(){
   delete[] cad;
}
void cadena::Asignar(const char *orig){
   //Eliminamos lo que tiene la cadena actual
   delete[] cad;
   //asignamos memoria para la nueva y almacenamos
   cad=new char[strlen(orig)+1];
   strcpy(cad,orig);
}
void cadena::Concatenar(cadena orig2){
   char *aux=new char[strlen(cad)+strlen(orig2.cad)+1];
   strcpy(aux,cad);
   strcat(aux,orig2.cad);
   Asignar(aux);
}
char *cadena::Leer(char *dest){
   strcpy(dest,cad);
   return dest;
}
int main(int argc, char *argv[])
{
   QCoreApplication a(argc, argv);
   cadena Cadena1("Cadena de prueba");
   cadena Cadena2(Cadena1);
   cadena *Cadena3;
   cadena Cadena4(Cadena1);

   char c[256];

   //Modificamos cadena1
   Cadena1.Asignar("otra cadena deferente");
   Cadena3=new cadena("Cadena de prueba numero 3");

   Cadena4.Concatenar(Cadena2);
   Cadena4.Concatenar(*Cadena3);

   //Mostrar resultados
   cout << "Cadena 1:" << Cadena1.Leer(c) << endl;
   cout << "Cadena 2:" << Cadena2.Leer(c) << endl;
   cout << "Cadena 3:" << Cadena3->Leer(c) << endl;
   cout << "Cadena 4:" << Cadena4.Leer(c) << endl;
   return a.exec();
}
```

Jerarquía de clases

1 - HERENCIA

Clases base y clases derivadas

Cada nueva clase obtenida mediante herencia se conoce como clase derivada.

Las clases a partir de las cuales se deriva, clases base.

Cada clase derivada puede usarse como clase base para obtener una nueva clase derivada.

Cada clase derivada puede serlo de una o más clases base (herencia múltiple).

Puede crearse una jerarquía de clases tan compleja como sea necesario.

Derivación de clases

Sintaxis

```
class  <clase_derivada> : [public|private] <base1> [,[public|private] <base2>] {};
```

Para cada clase base podemos definir dos tipos de acceso:

- public: los miembros heredados de la clase base conservan el tipo de acceso con que fueron declarados en ella.
- private: todos los miembros heredados de la clase base pasan a ser miembros privados en la clase derivada.

Si no se especifica ninguno de los dos, por defecto se asume que es private.

Ejemplos de herencia de clases (persona)

```
#include <QCoreApplication>
#include <iostream>
using namespace std;
class Persona{
public:
    Persona(char *n, int e);
    char* LeerNombre(char *n) const;
    int LeerEdad() const;
    void CambiarNombre(const char *n);
    void CambiarEdad(int e);
protected:
    char nombre[40];
    int edad;
};

class Empleado:public Persona{
public:
    Empleado(char *n,int e,float s);
    float LeerSalario() const;
    void CambiaSalario(const float s);
protected:
    float salarioMensual;
};
```

El acceso ***protected*** nos permite que los datos sean inaccesibles desde el exterior de las clases, pero a la vez, permite que sean accesibles desde las clases derivadas.

Ejemplo herencia métodos y programa ppal (persona)

```
Persona::Persona(char *n, int e){
   strcpy(nombre,n);
   edad=e;
}

char* Persona::LeerNombre(char *n) const{
   strcpy(n,this->nombre);
   return n;
}

int Persona::LeerEdad() const {
   return edad;
}

void Persona::CambiarNombre(const char *n){
   strcpy(nombre,n);
}

void Persona::CambiarEdad(int e){
   edad=e;
}

Empleado::Empleado(char *n,int e,float s):Persona(n,e){
   salarioMensual=s;}


float Empleado::LeerSalario()const {
   return salarioMensual;
}

void Empleado::CambiaSalario(const float s){
   salarioMensual=s;
}

int main(int argc, char *argv[])
{
   QCoreApplication a(argc, argv);
   char c[256];
   Empleado E1("Juan Perez",25,3500);
   cout << "Nombre:" << E1.LeerNombre(c) << endl;
   cout << "Edad:" << E1.LeerEdad() << endl;
   cout << "Salario:" << E1.LeerSalario() << endl;
   float ns;
   ns=E1.LeerSalario()*1.18;
   E1.CambiaSalario(ns);
   cout << "Salario:" << E1.LeerSalario() << endl;

   return a.exec();
}
```

Constructores y clases derivadas

Cuando se crea un objeto de una clase derivada, primero se invoca al constructor de la clase o clases base y a continuación al constructor de la clase derivada.

Si la clase base es a su vez una clase derivada, el proceso se repite recursivamente.

Si no hemos definido los constructores de las clases, se usan los constructores por defecto que crea el compilador.

Ejemplo herencia constructores (Herencia0)

```
#include <QCoreApplication>
#include <iostream>
using namespace std;

class ClaseA{
public:
   ClaseA():datoA(10){
      cout << "Constructor de ClaseA" << endl;
   }
   int LeerA() const {
      return datoA;
   }
protected:
   int datoA;
};

class ClaseB:public ClaseA{
public:
   ClaseB():datoB(20){
      cout << "Constructor de ClaseB" << endl;
   }
   int LeerB() const {  return datoB;  }
protected:
   int datoB;
};
```

```
int main(int argc, char *argv[])
{
   QCoreApplication a(argc, argv);
   ClaseB objeto;

   cout << "a= " << objeto.LeerA() <<
         " b= " << objeto.LeerB() << endl;

   return a.exec();
}
```

Inicialización de clases base en constructores

Para inicializar las clases base usando parámetros desde el constructor de una clase derivada se utiliza el mismo método que para con los datos miembro.

Las llamadas a los constructores deben escribirse antes de las inicializaciones de los parámetros.

Sintaxis

```
<clase_derivada>(<lista_de_parametros>) :
<clase_base>(<lista_de_parametros>)  {}
```

Ejemplo herencia constructores (Herencia1)

```
#include <QCoreApplication>
#include <iostream>
using namespace std;

class ClaseA{
public:
   ClaseA(int a):datoA(a){
      cout << "Constructor de ClaseA" << endl;
   }
   int LeerA() const {
      return datoA;
   }
protected:
   int datoA;
};

class ClaseB:public ClaseA{
public:
   ClaseB(int a, int b):ClaseA(a),datoB(b){
      cout << "Constructor de ClaseB" << endl;
   }
   int LeerB() const {   return datoB;   }
protected:
   int datoB;
};
```

```
int main(int argc, char *argv[])
{
   QCoreApplication a(argc, argv);
   ClaseB objeto(23,18);

   cout << "a= " << objeto.LeerA() <<
         " b= " << objeto.LeerB() << endl;

   return a.exec();
}
```

Inicialización de objetos miembros de clases

Cuando una clase tiene miembros objetos de otras clases, esos miembros pueden inicializarse se procede del mismo modo que con cualquier dato miembro

Se añade el nombre del objeto junto con sus parámetros a la lista de inicializaciones del constructor.

Ejemplo jerarquía de contención (Contencion0)

```
#include <QCoreApplication>
#include <iostream>
using namespace std;

class ClaseA{
public:
   ClaseA(int a):datoA(a){
      cout << "Constructor de ClaseA" << endl;
   }
   int LeerA() const {  return datoA;  }
protected:
   int datoA;
};

class ClaseB{
public:
   ClaseB(int a, int b):ca(a),datoB(b){
      cout << "Constructor de ClaseB" << endl;
   }
   int LeerB() const { return datoB;}
   int LeerA() const { return ca.LeerA();}
protected:
   int datoB;
   ClaseA ca;
};
```

```
int main(int argc, char *argv[])
{
   QCoreApplication a(argc, argv);
   ClaseB objeto(45,98);

   cout << "a= " << objeto.LeerA() <<
         " b= " << objeto.LeerB() << endl;

   return a.exec();
}
```

Destructores de clases derivadas

Cuando se destruye un objeto de una clase derivada, primero se invoca al destructor de la clase derivada

Si existen objetos miembro a continuación se invoca a sus destructores.

Finalmente, se llama al destructor de la clase o clases base.

Si la clase base es a su vez una clase derivada, el proceso se repite recursivamente

Ejemplo destructores en herencia (Herencia2)

```
#include <QCoreApplication>
#include <iostream>
using namespace std;

class ClaseA{
public:
   ClaseA():datoA(10){
      cout << "Constructor de ClaseA" << endl;
   }
   ~ClaseA() {cout << "Destructor de claseA" << endl;}
   int LeerA() const { return datoA; }
protected:
   int datoA;
};

class ClaseB:public ClaseA{
public:
   ClaseB():datoB(20){
      cout << "Constructor de ClaseB" << endl;
   }
   ~ClaseB() {cout << "Destructor de claseB" << endl;}
   int LeerB() const { return datoB; }
protected:
   int datoB;
};

void mostrar(){
   ClaseB objeto;

   cout << "a= " << objeto.LeerA() <<
         " b= " << objeto.LeerB() << endl;
}

int main(int argc, char *argv[])
{
   QCoreApplication a(argc, argv);

   mostrar();

   return a.exec();
}
```