Grado en Ingeniería Información

PROGRAMACIÓN II - Sesión 6

Tema 2.

Sobrecarga

Curso 2022-2023

Marta N. Gómez

UNIVERSIDAD NEBRIJA

T2. Sobrecarga

2.1. Concepto de sobrecarga

2.2. Características de la sobrecarga

2.3. Sobrecarga de operadores binarios

2.4. Sobrecarga de operadores unarios

2.5. Sobrecarga de operadores de flujo:

- Flujo de salida (ostream): operador de inserción <<
- Flujo de entrada (istream): operador de extracción >>

ÍNDICE

T2. Sobrecarga

2.1. Concepto de sobrecarga

2.2. Características de la sobrecarga

2.3. Sobrecarga de operadores binarios

2.4. Sobrecarga de operadores unarios

2.5. Sobrecarga de operadores de flujo:

- Flujo de salida (ostream): operador de inserción <<
- Flujo de entrada (istream): operador de extracción >>

ÍNDICE

Sobrecarga del operador unario postfijo

Cuando se quiere tener un comportamiento de un **operador unario postfijo** igual que los operadores básicos en C++, entonces:

1. Hay que utilizar un **objeto temporal** donde se **copia el valor inicial del objeto original** para **devolverlo como resultado.**

2. Después, se **incrementan** o **decrementan** los valores del **objeto original**.

Sintaxis de la forma postfija:

<tipoDato> ***operator*** ***<operUnario>*** ***(int)***;

<tipoDato> ***operator*** ***<operUnario>*** *(<tipDat> <identif>,* ***int)***;

Sobrecarga operador unario postfijo

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

class Rectangulo
{    private :
         int ladox;
         int ladoy;
     public :
         Rectangulo ();
         Rectangulo (int x, int y);

         int getladoX()const;
         int getladoY()const;
         void setLadox(int newLadox);
         void setLadoy(int newLadoy);

         // Forma postfija del operador como en C++: r++
         Rectangulo operator ++(int);
};
```

Sobrecarga operador unario postfijo

```
// Forma postfija del operador como en C++: r++
Rectangulo Rectangulo::operator ++(int) {
    Rectangulo Raux;
    Raux = *this;  // Se copia el objeto
    ladox++;
    ladoy++;

    return Raux;
}
```

Sobrecarga operador unario postfijo

```
// Programa Principal
int main()
{   Rectangulo R1(5, 7), R2;

    cout << "\n\n\tLos lados del rectangulo R1 son: ";
    R1.mostrarLados ();

    R2 = R1++;

    cout << "\n\n\tLos lados del rectangulo R2 son: ";
    R2.mostrarLados ();
    cout << "\n\n\tDatos del rectangulo R1 con post-incremento" << endl;
    R1.mostrarLados ();

    cout << "\n\n\t";
    return 0;
}
```

T2. Sobrecarga

2.1. Concepto de sobrecarga

2.2. Características de la sobrecarga

2.3. Sobrecarga de operadores binarios

2.4. Sobrecarga de operadores unarios

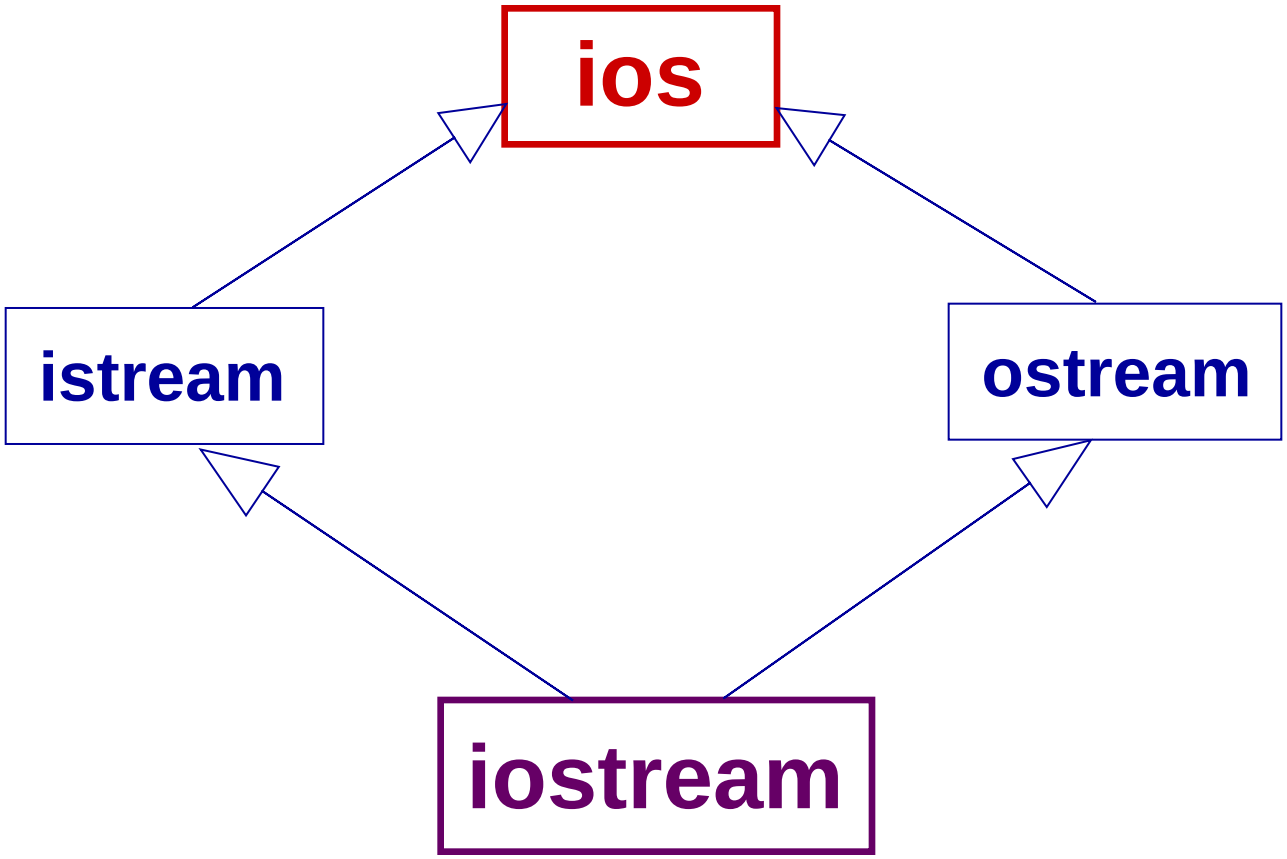
2.5. Sobrecarga de operadores de flujo:

- Flujo de salida (ostream): operador de inserción <<

- Flujo de entrada (istream): operador de extracción >>

ÍNDICE

Jerarquía de clases de E/S Básicas



La clase *ostream* proporciona, entre otras:

- **El operador <<** (**inserta** en el flujo ***cout***).
- **Las funciones para la *salida* con formato.**
- **etc.**

La clase *istream* proporciona, entre otras:

- **El operador >>** (**extrae** del flujo ***cin***).
- **Las funciones para la *entrada* con formato.**
- **etc.**

Sobrecarga del operador de inserción <<

El **operador de inserción <<** está asociado al flujo de salida ***cout*** de la clase **ostream**.

El **operador <<** siempre devolverá **una referencia a ostream** para poder encadenar varios operadores de flujo.

La **sobrecarga de este operador** tiene la siguiente estructura:

Flujo de salida a ***cout***

```
ostream& operator <<(ostream& out, const Tipo& a)
{
        // comportamiento de la sobrecarga

        return  out;
}
```

Objeto que se insertar

Sobrecarga del operador de extracción >>

El **operador de extracción >>** está asociado al flujo de entrada ***cin*** de la clase **istream**.

El **operador >>** siempre devolverá una **referencia a istream** para poder encadenar varios operadores de flujo.

La **sobrecarga de este operador** tiene la siguiente estructura:

Flujo de entrada a ***cin***

```
istream& operator >>(istream& in, Tipo& a)
{
        // comportamiento de la sobrecarga

        return  in;
}
```

Objeto que se extraer

Sobrecarga operadores << y >>

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

class Rectangulo
{    private :
         int ladox;
         int ladoy;
     public :
         Rectangulo ();
         Rectangulo (int x, int y);

         int getladoX()const;
         int getladoY()const;
         void setLadox(int newLadox);
         void setLadoy(int newLadoy);
};

ostream & operator << (ostream &os, Rectangulo const &p);

istream &operator >> (istream &is, Rectangulo &p);
```

Sobrecarga operadores << y >>

```
// Programa Principal
int main()
{   Rectangulo R1(5, 7), R2;

    cout << "\n\n\tIntroduzca los lados del rectangulo R2 ";
    cin >> R2;

    cout << "\n\n\tLas medidas de R1 son: " << R1 << " mientras que las de R2 son: " << R2;

    cout << "\n\n\t";
    return 0;
}

ostream &operator << (ostream &os, Rectangulo const &p) {
    os << " " << p.getladoX() << " y " << p.getladoY();

    return os;
}


istream &operator >> (istream &is, Rectangulo &p) {
    int X, Y;
    is >> X >> Y;
    p.setLadox(X);
    p.setLadoy(Y);

    return is;
}
```

UNIVERSIDAD NEBRIJA

Clases y Funciones *friend:*

- **Características de las relaciones de amistad**
- Funciones friend (Amigas)
- Clases friend (Amigas)

ÍNDICE

Uno de los principios de la POO es el ***ENCAPSULAMIENTO***:

- Almacenando de forma ***privada*** los **datos** y las **implementaciones internas** de un objeto.
- La única información que **cada objeto** debe tener **sobre otros objetos** es su **comportamiento**.
- Favorece la construcción de aplicaciones basadas en la **funcionalidad que cada objeto**:

El ***qué*** frente al ***cómo***

Relaciones de **confianza** o tipo ***friend***:

- Permiten el **acceso directo** a los **atributos** y **métodos privados de una clase**.
- Dan **flexibilidad** para hacer algunas operaciones.

Normas de las relaciones ***friend***:

1. **La amistad no puede transferirse:** *los amigos de mis amigos no tienen por qué ser mis amigos.*

   Son relaciones punto a punto e independientes

   - **Si hay** una relación ***friend*** entre las clases **X** e **Y**.
   - **Si hay** una relación ***friend*** entre las clases **X** y **Z**.

   No significa que exista una relación *friend* entre las clases Y y Z.

Normas de las relaciones ***friend***:

2. **La amistad no se hereda:** *los hijos de mis amigos no tienen por qué ser mis amigos.*

   - **Si existe** una relación ***friend*** entre las clases **X** e **Y**.

   No significa que exista una relación *friend* entre las clases derivadas de X (clases hijas) y la clase Y.

Normas de las relaciones ***friend***:

3. **La amistad no es simétrica o bidireccional por defecto.**

   - **Si existe** una relación ***friend*** entre las clases **X** e **Y**.

   No significa que la clase Y tenga acceso a la parte privada de la clase X.

   Por tanto, se deben implementar las relaciones *friend* que se necesiten.

Características de las relaciones de amistad

Relaciones *friend* se puede establecer entre:

- **Una clase y una función**
- **Dos clases**

UNIVERSIDAD NEBRIJA

Clases y Funciones *friend:*

- Características de las relaciones de amistad
- **Funciones friend (Amigas)**
- Clases friend (Amigas)

ÍNDICE

Funciones *friend*:

Pueden ser:

- **Funciones generales de la aplicación.**
- **Funciones miembro de otra clase.**

Una **función** puede ser declarada ***friend*** de **una** o de **varias clases**.

Función *friend* de una clase:

- **La clase autoriza a acceder a toda su parte privada** a través de los **operadores**: Punto (**.**) Flecha (**->**)
- La **clase** debe indicar esta relación en su **declaración,** en **cualquier sección** de la **declaración de la clase** (zona pública o privada), con la palabra reservada ***friend***.
- ***Sintaxis:***

friend tipDevuelto nomFuncAmiga (parámetros);

Ejemplo:

```
class Cualquiera   {
   friend  void  fAmiga  (Cualquiera &Una);
   private:
         int secreto;
};

// Función general que modifica el valor del atributo
// privado de la clase Cualquiera
void fAmiga(Cualquiera Una)   {
   Una.secreto++;
}
```

Clases y Funciones *friend*

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

class Rectangulo
{   private :
        int ladox;
        int ladoy;
    public :
        Rectangulo ();
        Rectangulo (int x, int y);

        int getladoX()const;
        int getladoY()const;
        void setLadox(int newLadox);
        void setLadoy(int newLadoy);

        friend ostream &operator << (ostream &os, Rectangulo const &p);
};
```

```
// Programa Principal
int main()
{   Rectangulo R1(5, 7), R2;

    cout << "\n\n\tLos lados de R1 son: " << R1;

    cout << "\n\n\t";
    return 0;
}

ostream &operator << (ostream &os, Rectangulo const &p) {
    os << " " << p.ladox << " y " << p.ladoy;

    return os;
}
```

¿Rompiendo la encapsulación?

- Las **funciones *friend*** dan **mayor flexibilidad** a la programación orienta a objetos.
- Se **preserva la seguridad** y **la protección** que proporcionan las clases.
- **Ninguna función** puede **autodeclararse amiga** de **otra**.

UNIVERSIDAD NEBRIJA

Clases y Funciones *friend:*

- Características de las relaciones de amistad
- Funciones friend (Amigas)
- **Clases friend (Amigas)**

ÍNDICE

Clase *friend* de otra clase

- La **clase que autoriza el acceso** a sus datos privados **lo debe indicar** en su **declaración**.
- **Una clase tiene permiso para acceder a toda la parte privada de la otra clase, pero no al contrario.**
- **Se utilizan los operadores:**

  Punto (**.**) Flecha (**->**)

- **Sintaxis:**

  friend class nomClaseAmiga;

Clases y Funciones *friend*

```
class CPunto {
        friend class CSegmento;
        private:
            int x;
            int y;
         public:
            CPunto ():x(0), y(0) {}
            CPunto (int a, int b):x(a), y(b) {}
            void verPunto () const;
};

class CSegmento {
        private:
            CPunto P1, P2;
        public:
            CSegmento ();
            void valorSegmento (const CPunto &A1, const CPunto &A2)
                        {   P1.x=A1.x; P2.x=A2.x; P1.y=A1.y; P2.y=A2.y;  }
            void verSegmento () const;
};
```

```
//--------CLASE Punto
void CPunto::verPunto () const {
        cout << "x: " << x << " y: " << y << endl;
}

//--------CLASE Segmento
CSegmento::CSegmento () {
        P1.x = 0;     P1.y = 0;
        P2.x = 0;     P2.y = 0;
}

void CSegmento::valorSegmento (const CPunto &A1, const CPunto &A2)
{
        P1.x=A1.x;  P1.y=A1.y
        P2.x=A2.x;  P2.y=A2.y;
}
```

Ejercicio:

Definir la clase Complejo con dos atributos reales, **parte real** y **parte imaginaria**, que permita declarar números complejos (objetos de la clase Complejo) como z1 = (a, b) y z2 = (c, d), donde a y b son la parte real y b y d la parte imaginaria de los números z1 y z2, respectivamente.

Declarar e implementar los **constructores** y **métodos/funciones get/set** necesarios para acceder a sus variables.

Finalmente, definir e implementar operadores sobrecargados para realizar las siguientes **operaciones binarias y unarias miembros de la clase Complejo**:

- Suma: $z1 + z2 = (a + c, b + d)$
- Diferencia: $z1 - z2 = (a - c, b - d)$
- Producto: $z1\ z2 = (ac - bd, ad + bc)$
- División: $\frac{Z1}{Z2} = \left(\frac{ac + bd}{c^2 + b^2}, \frac{bc - ad}{c^2 + b^2}\right)$

Ejercicio:

- Pre Incremento: $++z1 = (a+1, b+1)$
- Pre Decremento: $--z1 = (a-1, b-1)$
- Post Incremento como en C++: $z1++ = (a+1, b+1)$
- Post Decremento como en C++: $z1-- = (a-1, b-1)$
- Operador de extracción >>
- Operador de inserción <<

Repetir los operadores anteriores (incremento, decremento, << y >>) como funciones generales y declaradas como operadores ***friend*** dentro de la clase Complejo.