# CS2013: Programación III

## Teoría: Fundamentos de POO II

José Chávez

UTEC

# Contenido

- Sobrecarga de Operadores
- Herencia
- Polimorfismo

# Contenido

# Sobrecarga de Operadores

Un operador sobrecargada es una función que tiene la palabra `operator` seguida del símbolo del operador. Algunas características/ventajas:

# Sobrecarga de Operadores

Un operador sobrecargada es una función que tiene la palabra `operator` seguida del símbolo del operador. Algunas características/ventajas:

- Puede hacer que nuestro código se pueda escribir y leer mejor.

# Sobrecarga de Operadores

Un operador sobrecargada es una función que tiene la palabra `operator` seguida del símbolo del operador. Algunas características/ventajas:

- Puede hacer que nuestro código se pueda escribir y leer mejor.
- Un operador sobrecargado tiene la misma cantidad de parámetros que los operandos del operador.

# Sobrecarga de Operadores

Un operador sobrecargada es una función que tiene la palabra `operator` seguida del símbolo del operador. Algunas características/ventajas:

- Puede hacer que nuestro código se pueda escribir y leer mejor.
- Un operador sobrecargado tiene la misma cantidad de parámetros que los operandos del operador.
- Cuando un operador sobrecargado es un método, `this` está vinculado al operando de la izquierda.

# Sobrecarga de Operadores

Un operador sobrecargada es una función que tiene la palabra `operator` seguida del símbolo del operador. Algunas características/ventajas:

- Puede hacer que nuestro código se pueda escribir y leer mejor.
- Un operador sobrecargado tiene la misma cantidad de parámetros que los operandos del operador.
- Cuando un operador sobrecargado es un método, `this` está vinculado al operando de la izquierda.
- Una función de operador debe ser un miembro de una clase o tener al menos un parámetro del tipo de la clase.

```
int operator+(int, int);    // ERROR!
```

# Operadores

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

# Operadores de salida

Operador de salida:

```
ostream &operator<<(ostream &os, const Coordenada &item){
    os << "(" << item.x << ",␣";
    os << item.y << ")";
    return os;
}
```

Nota: El operador de salida no debe ser miembro de la clase.

# Operadores de salida: Interfaz

En nuestra interfaz el operador funcionaría de la siguiente manera:

```
int main(){
    Coordenada obj1(1.1, 2.2);
    cout << obj1 << endl;

    return 0;
}
```

```
(1.1, 2.2)
```

# Operadores de entrada

Operador de entrada:

```
istream &operator>>(istream &is, Coordenada &item){
    is >> item.x >> item.y;
    return is;
}
```

Nota: Este operador debería resolver errores de lectura (si existieran).

# Operadores de entrada: Interfaz

En nuestra interfaz el operador de entrada funcionaría de la siguiente manera:

```
int main(){
    Coordenada obj2;
    cin >> obj2;
    cout << obj2 << endl;

    return 0;
}
```

```
5.2 6.1 (5.2 6.1)
```

# Operadores Aritméticos: Sintaxis

Usualmente los operadores aritméticos generan un nuevo valor, por ejemplo para el operador de adición (+) sería:

```
Coor operator+(const Coor &item1, const Coor &item2){
    Coor obj;
    obj.x = item1.x + item2.x;
    obj.y = item1.y + item2.y;
    return obj;
}
```

Para los demás operadores aritméticos se utiliza la misma sintaxis.

# Operadores Aritméticos: Interfaz

En la interfaz el operador funcionaría de la siguiente forma:

```
int main(){
    Coor obj1(1.1, 2.2);
    Coor obj2(3.3, 4.4);

    Coor obj3 = obj1 + obj2;
    cout << obj3 << endl;

    return 0;
}
```

```
(4.4, 6.6)
```

# Operadores Relacionales: Sintaxis

Estos operadores se utilizan principalmente para comparar y ordenar. Por ejemplo para el operador de adición (+) sería:

```
bool operator==(const Coor &item1, const Coor &item2){
    return (item1.x == item2.x) && (item1.y == item2.y);
}
```

Para los demás operadores relacionales como !=, <, > se utiliza la misma sintaxis.

# Operadores Relacionales: Interfaz

En la interfaz el operador funcionaría de la siguiente forma:

```
int main(){
    Coor obj1(1.1, 2.2);
    Coor obj2(3.3, 4.4);
    Coor obj3(1.1, 2.2);

    cout << (obj1 == obj3) << endl;
    cout << (obj1 == obj2) << endl;

    return 0;
}
```

```
1
0
```

# Más operadores

Se pueden realizar sobrecarga a más tipos de operadores:

- Operadores de asignación.
- Operador sub-índice.
- Operador de incremento y decremento.
- Functores

# Contenido

# Herencia

Las clases relacionadas por *Herencia* construyen una jerarquía. En esta jerarquía existen dos tipos de clases:

# Herencia

Las clases relacionadas por *Herencia* construyen una jerarquía. En esta jerarquía existen dos tipos de clases:

- La *Clase Base* o *Super Clase*. Esta clase está en la raíz de la jerarquía, desde donde las demás clases heredan sus miembros.

# Herencia

Las clases relacionadas por *Herencia* construyen una jerarquía. En esta jerarquía existen dos tipos de clases:

- La *Clase Base* o *Super Clase*. Esta clase está en la raíz de la jerarquía, desde donde las demás clases heredan sus miembros.
- La *Clase derivada* o *Sub Clase*. Estas son las que heredan los miembros de la Clase Base.

# Herencia

Veamos como funciona la Herencia con un ejemplo:

```
class A {
public:
    void f() { cout << "A::f()" << endl; }
    void g() { cout << "A::g()" << endl; }
};
class B : public A {
public:
    void f() { cout << "B::f()" << endl; }
};
```

# Herencia

Veamos como funciona la Herencia con un ejemplo:

```
int main() {
    A a;
    B b;
    a.f(); a.g();
    b.f(); b.g();
    return 0;
}
```

```
A::f()
A::g()
B::f()
A::g()
```

# Redefiniendo Métodos

La clase base puede indicar que métodos se deben redefinir, para esto debe utilizar el modificador `virtual`.

# Redefiniendo Métodos

La clase base puede indicar que métodos se deben redefinir, para esto debe utilizar el modificador `virtual`.Pero, ¿Porque redefiniríamos un método?

# Redefiniendo Métodos

La clase base puede indicar que métodos se deben redefinir, para esto debe utilizar el modificador `virtual`.Pero, ¿Porque redefiniríamos un método?

- Esto garantiza que se llame a la función adecuada para un determinado objeto. Independiente de la referencia/puntero que se utilice en la llamada a la función.

# Reglas para Redefinir Métodos

¿Cuales son las reglas al definir y utilizar una función virtual?

# Reglas para Redefinir Métodos

¿Cuales son las reglas al definir y utilizar una función virtual?

- Se puede invocar a las funciones virtuales con punteros a la clase base.

# Reglas para Redefinir Métodos

¿Cuales son las reglas al definir y utilizar una función virtual?

- Se puede invocar a las funciones virtuales con punteros a la clase base.
- El puntero de la clase base debe apuntar a objetos de clases derivadas.

# Reglas para Redefinir Métodos

¿Cuales son las reglas al definir y utilizar una función virtual?

- Se puede invocar a las funciones virtuales con punteros a la clase base.
- El puntero de la clase base debe apuntar a objetos de clases derivadas.
- Uno puede re-definir (o no) una función virtual con la clase derivada.

# Reglas para Redefinir Métodos

¿Cuales son las reglas al definir y utilizar una función virtual?

- Se puede invocar a las funciones virtuales con punteros a la clase base.
- El puntero de la clase base debe apuntar a objetos de clases derivadas.
- Uno puede re-definir (o no) una función virtual con la clase derivada.
- Una clase no puede tener un constructor virtual, pero si un destructor virtual.

# Reglas para Redefinir Métodos

¿Cuales son las reglas al definir y utilizar una función virtual?

- Se puede invocar a las funciones virtuales con punteros a la clase base.
- El puntero de la clase base debe apuntar a objetos de clases derivadas.
- Uno puede re-definir (o no) una función virtual con la clase derivada.
- Una clase no puede tener un constructor virtual, pero si un destructor virtual.
- Este tipo de funciones no pueden ser declaradas como `static`.

# Redefiniendo métodos

Veamos como podemos redefinir un método con un ejemplo:

```
class A {
public:
    virtual void f() { cout << "A::f()" << endl; }
    void g() { cout << "A::g()" << endl; }
};
class B : public A {
public:
    void f() { cout << "B::f()" << endl; }
    void g() { cout << "B::g()" << endl; }
};
```

# Redefiniendo métodos

Veamos como podemos redefinir un método con un ejemplo:

```
int main() {
    A *a;
    B b;
    a = &b; // 'a' tiene acceso unicamente a los
            // miembros publicos de la clase base
    a->f(); a->g();

    return 0;
}
```

```
B::f()
A::g()
```

# Contenido

# ¿Es es el Polimorfismo?

Por definición, el *Polimorfismo* es la capacidad de adoptar múltiples formas.

# ¿Es es el Polimorfismo?

Por definición, el *Polimorfismo* es la capacidad de adoptar múltiples formas.En POO, el Polimorfismo es la capacidad de llamar a distintos métodos con un mismo nombre.

# ¿Es es el Polimorfismo?

Por definición, el *Polimorfismo* es la capacidad de adoptar múltiples formas.En POO, el Polimorfismo es la capacidad de llamar a distintos métodos con un mismo nombre.

En otras palabras, el Polimorfismo permite que un objeto de una clase base se comporte como un objeto de cualquiera de sus clases derivadas. Esto dependerá de la forma en que se llame a los métodos.

# Enlaces estáticos y dinámicos

Antes de utilizar Polimorfismo es necesario diferenciar entre un *Enlace Estático* y un *Enlace Dinámico*:

# Enlaces estáticos y dinámicos

Antes de utilizar Polimorfismo es necesario diferenciar entre un *Enlace Estático* y un *Enlace Dinámico*:

- El enlace estático sucede en tiempo de compilación y el enlace dinámico en tiempo de ejecución.

# Enlaces estáticos y dinámicos

Antes de utilizar Polimorfismo es necesario diferenciar entre un *Enlace Estático* y un *Enlace Dinámico*:

- El enlace estático sucede en tiempo de compilación y el enlace dinámico en tiempo de ejecución.
- El enlace estático se logra usando las llamadas a funciones, sobrecarga de funciones y sobrecarga de operadores. Mientras que el enlace dinámico se logra usando las funciones virtuales.

# Enlaces estáticos y dinámicos

Antes de utilizar Polimorfismo es necesario diferenciar entre un *Enlace Estático* y un *Enlace Dinámico*:

- En el enlace estático, toda la información útil para manejar una función está disponible antes de la ejecución, esto conlleva a una ejecución más rápida del programa.

# Enlaces estáticos y dinámicos

Antes de utilizar Polimorfismo es necesario diferenciar entre un *Enlace Estático* y un *Enlace Dinámico*:

- En el enlace estático, toda la información útil para manejar una función está disponible antes de la ejecución, esto conlleva a una ejecución más rápida del programa.
- La ventaja del enlace dinámico es su flexibilidad, debido a que con una sola función puede manejar distintos tipos de objetos. Esto reduce el tamaño del código.

# Ejemplo de un enlace estático

Un ejemplo de enlace estático utilizando sobrecarga de funciones:

```
class A{
public:
    int f(int x) { return x;}
    int f(int x, int y) { return x + y;}
};
```

# Ejemplo de un enlace estático

En la interfaz tenemos lo siguiente:

```
int main(){
    A a;
    cout << "f:" << a.f(2) << endl;
    cout << "f:" << a.f(2,3) << endl;

    return 0;
}
```

```
f:2
f:5
```

# Ejemplo de un enlace dinámico

Un ejemplo de enlace dinámico utilizando sobrecarga de funciones:

```
class B{
public:
    virtual void f() { cout << "B::f()";}
};
class D: public B{
    public:
    void f() { cout << "D::f()";}
};
```

# Ejemplo de un enlace dinámico

En la interfaz tenemos lo siguiente:

```
int main(){
    B b;
    D d;
    B *bPtr = &b;
    bPtr->f();
    bPtr = &d;
    bPtr->f();

    return 0;
}
```

```
f:2
f:5
```

# Polimorfismo

Así como existe el enlace estático y dinámico. También existe dos tipos de polimorfismo:

# Polimorfismo

Así como existe el enlace estático y dinámico. También existe dos tipos de polimorfismo:

- Polimorfismo Estático. Si la clase tiene distintas funciones con el mismo nombre, pero con distintos parámetros.

# Polimorfismo

Así como existe el enlace estático y dinámico. También existe dos tipos de polimorfismo:

- Polimorfismo Estático. Si la clase tiene distintas funciones con el mismo nombre, pero con distintos parámetros.
- Polimorfismo Dinámico. Si las mismas funciones (nombre, parámetros y tipos) están definidas en la clase base y clase derivada.

# Polimorfismo Dinámico

Veamos un ejemplo de polimorfismo dinámico:

```
class Forma{
public:
    virtual void dibujar() {cout << "Forma" << endl;}
};
class Circulo: public Forma{
public:
    void dibujar() { cout << "Circulo" << endl; }
};
class Triangulo: public Forma{
public:
    void dibujar() { cout << "Triangulo" << endl; }
};
```

# Polimorfismo Dinámico

Luego en nuestra interfaz:

```
int main(){
    Forma *f;
    Circulo c;
    Triangulo t;

    f = &c;
    f->dibujar(); // Resultado: Circulo
    f = &t;
    f->dibujar(); // Resultado: Triangulo

    return 0;
}
```