

# Sobrecarga / Operadores en c++

Informática II - R2004  
2021

# Repasando... ¿Cómo programamos en C++?

```
Class Punto
{
    private:
        float x;
        float y;
    public:
        Punto (float = 0 , float = 0);
        Punto (const Punto &);

        float getX () const;
        float getY () const;

        setX (float);
        setY (float);

        setCoords(float, float);
}
```

Definición de la clase (Punto.h)

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

Punto::Punto (const Punto &a)
{
    x = a.x;
    y = a.y;
}

float Punto::getX ()
{
    return x;
}

float Punto::setX ( float a )
{
    x = a;
}
```

Definición de la clase (Punto.cpp)

# Repasando... ¿Cómo programamos en C++?

```
Class Punto
{
    private:
        float x;
        float y;
    public:
        Punto (float = 0 , float
        Punto (const Punto &);

        float getX () const;
        float getY () const;

        setX (float);
        setY (float);

        setCoords(float, float);
}
```

Definición de la clase (Punto.h)

```
void main ()
{
    using namespace std;

    void Punto P1(1,5),P2(2),P3(P2);

    P1.setX ( P3.getY() );

    P2.setCoords ( P3.getX() , P1.getY() );

    cout << "Ingrese la coordenada Y de P3: " << endl;

    float aux;

    cin >> aux;

    P3.setY (aux);
}
```

Instancio y utilizo los objetos (aplicación.cpp)

# Sobrecarga de operadores - Introducción

De la misma manera que una clase tiene sus métodos, que le indican como tiene que hacer las cosas (como cargar los valores en un vector, como sumar sus componentes, como calcular un módulo, etc.), también podemos modificar el comportamiento de los OPERADORES, cuando se usan con un objeto de esa clase.

De esta manera, podemos indicarle al compilador cómo debe comportarse cuando se quieran sumar (operador +) dos clases de tipo Punto, o cómo se debe hacer para incorporar (con el operador <<, por ejemplo) un objeto de tipo Persona a una lista dinámica de esos objetos.

A la **redefinición** de la **función** de los **operadores** se la llama

**SOBRECARGA DE OPERADORES**



# Sobrecarga de operadores - Introducción (II)

De la misma manera que podemos utilizar métodos para hacer operaciones con los miembros de una clase, también podemos reutilizar los operadores de C/C++ para que hagan lo mismo:

```
Class Punto
{
    private:
        float x;
        float y;
    public:
        Punto (float a=0 , float b=0);
        Punto (const Punto &);

        float getX() const;
        float getY() const;

        setX(float);
        setY(float);

        MultiplicaConstante ( float a )
        {
            x *= a;
            y *= a;
        }
}
```

```
Punto P1(2,3),P2;
```

```
cout << "Coordenadas P1: x = " << P1.getX() << " e y = " << P1.getY() << endl
```

```
P1.MultiplicaConstante(2);
```

```
cout << "Coordenadas P1: x = " << P1.getX() << " e y = " << P1.getY() << endl
```

**En este ejemplo estoy indicándole a la clase que la invocación al método MultiplicaConstante deberá multiplicar el atributo x e y en ese valor constante**

# Sobrecarga de operadores - Introducción (II)

De la misma manera que podemos utilizar métodos para hacer operaciones con los miembros de una clase, también podemos reutilizar los operadores de C/C++ para que hagan lo mismo:

```
Class Punto
{
    private:
        float x;
        float y;
    public:
        Punto (float a=0 , float b=0);
        Punto (const Punto &);

        float getX() const;
        float getY() const;

        setX(float);
        setY(float);

        void operator* ( float a )
        {
            x *= a;
            y *= a;
        }
}
```

```
Punto P1(2,3),P2;
```

```
cout << "Coordenadas P1: x = " << P1.getX() << " e y = " << P1.getY() << endl;
```

```
P1*=2;
```

```
cout << "Coordenadas P1: x = " << P1.getX() << " e y = " << P1.getY() << endl;
```

**Cambiando el nombre del método por la palabra reservada `operator`, e indicando el tipo de operador que quiero modificar, hago que en lugar de tener que invocar al método `MultiplicaConstante` pueda utilizar directamente el operador elegido**

## Y en nuestro ejemplo...

```
class Punto {
```

```
...
```

```
bool esMayor ( const Punto & );  
bool esMenor ( const Punto & );
```



```
class Punto {
```

```
...
```

```
bool operator > ( const Punto & );  
bool operator < ( const Punto & );
```

```
int main ()  
{
```

```
    Punto A, B;
```

```
...
```

```
if ( A > B ) {
```

```
    ...
```

```
}|
```

```
}
```

# Sobrecarga de operadores - declaración

De acuerdo con lo que dijimos, para sobrecargar un operador debo DECLARAR UN MÉTODO con el nombre OPERATOR, seguido por el operador que quiero sobrecargar. A pesar de que la declaración sea distinta, no deja de ser un método de una clase, y por lo tanto recibe argumentos y devuelve un valor de retorno, de un determinado tipo de datos:

**<tipo\_retornado>** **operator** **<OPERADOR>** ( **<argumentos>** )





# ¿Qué operadores puedo sobrecargar?





A continuación un listado de los operadores que puedo sobrecargar:

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

¿Qué operadores **NO** puedo sobrecargar?

.	.*	::	?:	sizeof
---	----	----	----	--------

# Sobrecarga de operadores como MIEMBRO de la clase

<code>P1 += 2;</code>		<code>Punto Punto::operator += ( float a );</code>
<code>P1 + P2;</code>		<code>Punto Punto::operator + ( const Punto &amp;a ) const;</code>
<code>P3 = P1 + P2;</code>		<code>Punto Punto::operator = ( const Punto &amp;a );</code>
<code>P4 = P1 + P2 + P3;</code>		<code>Punto Punto::operator + ( const Punto &amp;a ) const;</code>
<code>if ( P1 == P2 )</code>		<code>bool Punto::operator == ( const Punto &amp;a ) const;</code>

¿Qué cantidad y tipo de parámetros  
recibe cada instrucción?

**Se puede modificar la función del operador pero NO  
su gramática (la cantidad de argumentos que recibe)**

# Ejemplo...

Agregar a nuestra clase punto la sobrecarga de los operadores necesaria para poder hacer las siguientes operaciones:

```
Punto A, B, C;
```

```
if ( A > B ) { ... }
```

```
if ( A < B ) { ... }
```

```
C = A + B;
```

```
B = A * 2;
```



# Puntero this - introducción

Ya hemos visto el diseño y la utilización de clases como la siguiente:

Punto P1 , P2, P3;

P1.setX(4);

P2.setY(9);

P3 = P1 + P2;



¿Cómo sabe el compilador que la invocación a la función (método) setX debe modificar la variable X perteneciente a P1, y no la de P2 o la de P3?

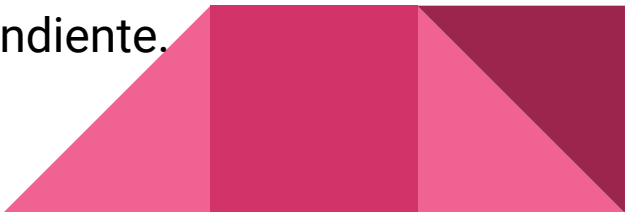
¿Hay una función distinta por cada objeto instanciado?

Esto haría que se deba sobrecargar la memoria con muchísimas funciones, y se haga en tiempo de ejecución (el compilador no puede saber de antemano cuantos objetos se van a instanciar en el programa)

¿Qué otra forma puedo pensar para que el compilador sepa en cada llamado a setX, setY, operator + y operator = que variables son las que debe modificar según que objeto lo esté llamando?

# Puntero this

## ¿Cómo saben las funciones miembro cuál objeto deben manipular?


- Todo objeto tiene acceso a su dirección a través de un puntero llamado **this** (que no forma parte de él).
  - **this** contiene en todo momento la dirección del objeto concreto que se está ejecutando. Es un puntero constante a él (no podemos cambiar su valor).
  - Cada vez que se invoca a una función miembro (*no static*), el compilador lo pasa como un argumento implícito.
  - Se puede decir que **\*this** es un alias del objeto correspondiente.
- 

# Puntero this

## *En otras palabras...*

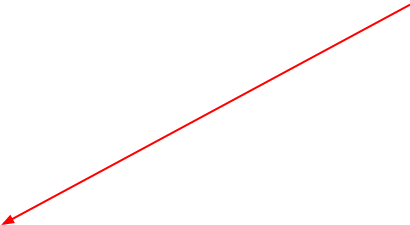
Cuando se invoca a una función miembro de un objeto, el compilador asigna (*automáticamente*) la dirección del objeto en cuestión al puntero **this**, **y por medio de él la llama** (*de forma transparente para el programador*).

*Cada vez que una función miembro accede a un dato miembro, se está utilizando también de forma implícita el puntero **this**.*



# Puntero this - uso trivial (solo a modo de ejemplo)

```
class D {  
    int i, j, k;  
public:  
    D () {this->i = this->j = this->k = 0};  
    void mostrar (void){  
        cout<< this->i<<" "<<this->j<<" "<<this->k;  
    }  
}
```



es lícito el uso del puntero **this** cuando se accede a los datos miembro, **aunque es innecesario y está mal visto.**



# Puntero this como valor de retorno

Muchas veces me encuentro en la necesidad de devolver un objeto que tenga los valores del objeto donde estoy, o tal vez un alias del objeto. Si por ejemplo anido operaciones:

`P1 = ++P2;`

Puede pensarse como que P1 invoca al método `operator=`, que recibe como argumento el valor de retorno del método `operator++` invocado por P2. O sea:

`P1.operator= ( P2.operator++ );`

Por lo que si pensamos el método `operator++`, el mismo deberá modificar P2 y luego devolver el objeto modificado (devolverse a si mismo luego de modificarse. Esta referencia a si mismo puede hacerse mediante el puntero `this`.



# Puntero this - retorno

```
class Punto{
private:
    int x,y;
public:
    Punto(int a=0,int b=0){
        x=a; y=b;
    }
    Punto& operator++(void) {
        x++; y++;
        return *this;
    }

    ...

};
```

Usar el puntero **this** como retorno permite anidar métodos sobre el objeto modificado

```
int main( ) {
    Punto A(3,4),B(2,1),C;
    C = ++A;

    if( ((++C).getX() > 5) )
        ...
}
```

Cada llamado a **operator++** devuelve una referencia al objeto desde donde se lo llamó, y por lo tanto puede volver a llamarse a un método de ese objeto.

En el ejemplo vemos como se puede llamar al método `getX()` desde el RETORNO de `operator++`.

# Pre incremento y post incremento

A partir del ejemplo anterior: ¿Cómo diferenciamos el pre incremento (++a) del post incremento (a++)?

```
class Punto{
```

```
    ...  
    Punto& operator++(void) {  
        //pre incremento: summo y luego devuelvo:  
        x++; y++;  
        return *this;  
    }
```

```
    Punto operator++(int a) {  
        //post incremento: devuelvo el objeto sin modificar:  
        Punto aux(x,y);  
        x++;y++;  
        return aux;  
    }
```

```
};
```

```
void main ( void )  
{
```

```
    Punto A(2,3),B,C;
```

```
    C = ++A;
```

```
    if ( (A++).getX() == 3 )
```

```
{
```

```
}
```

```
}
```

¿Por qué el pre incremento devuelve Punto & y el post incremento devuelve Punto?

# Ejemplo...

¿Cómo haríamos la sobrecarga del operador =?

¿Cómo hacerla de manera que permita hacer lo siguiente?:

**Punto A, B, C;**

**A = B = C;**

Con esto en mente, pensar la sobrecarga del operador +=, de manera que permita lo siguiente:

**A+=B+=C;**

**if ( A+=B > C ) { ... }**



# Sobrecarga de operadores globales

Se puede definir una función global (que no sea miembro de una clase) que redefina un operador.

La principal diferencia de esta forma de sobrecargar operadores es que tengo que incluir TODOS los argumentos en la definición de la función:

Por ejemplo:

```
P3 = P1 + P2;
```

Se definiría (en forma local):

```
Punto Punto::operator + (const Punto &a) const  
{  
    return Punto C (x + a.x , y + a.y);  
}
```

Se definiría (en forma global):

```
Punto operator + ( const Punto &a , const Punto &b )  
{  
    return Punto C (a.x + b.x , a.y + b.y);  
}
```

# Y... ¿para qué me sirve?

```
cout << "Las coordenadas de P1: " << P1.getX() << ";" << P1.getY() << endl;  
cout << "Las coordenadas de P1: " << P1 << endl;
```

**cout** es un objeto de tipo ostream (class ostream)

**cin** es un objeto de tipo istream (class istream)

¿Cómo hago para redefinir un operador en una clase que ya está definida?

## LO REDEFINO COMO UN OPERADOR GLOBAL

```
ostream & operator << ( ostream & out , Punto & a )  
{  
    out << "(" << a.x << " ; " << a.y << " )";  
    return out;  
}
```

# Clases y funciones *friend*

Los miembros privados de una clase no son accesibles para funciones y clases exteriores a dicha clase. La palabra reservada **friend** me permite declarar relaciones de amistad entre clases o entre clases y funciones.

```
class ClaseA {  
    //declaro función amiga de la clase  
    friend void función1 (ClaseA);  
private:  
    int a_;  
public:  
    ClaseA (int i=0) {a_=i;}  
    void mostrar( ) {cout << a_ << endl;}  
};
```

función global. No es exclusiva de la clase. observar que el calificador friend no aparece en la implementación de la función

```
void función1 (ClaseA z) {  
    cout << z.a_ << endl;  
}  
int main (void)  
{  
    .....  
    ClaseA objeto1;  
    función1 (objeto1);  
    objeto1. mostrar ( );  
    .....  
}
```

# Ejemplo...

Hacer las funciones necesarias para poder hacer las siguientes operaciones:

```
Punto A(2,3), B;
```

```
B = 2 * A;
```

```
cout << "El punto B vale: " << B << endl;
```

