

# ***Programación Orientada a Objetos***

## **Unidad 4: Sobrecarga de Operadores**

## EJEMPLO: CLASE COMPLEJO

```
class Complejo {  
    float m_preal, m_pimag;  
  
public:  
    Complejo(float pr=0, float pi=0);  
  
    void CargarParteReal(float pr);  
    void CargarParteImag(float pi);  
  
    float VerParteReal() const;  
    float VerParteImag() const;  
  
};
```

# EJEMPLO: CLASE COMPLEJO

```
int main() {  
    Complejo c1(1,3), c2(4,5);  
    Complejo suma(  
        c1.VerParteReal() + c2.VerParteReal(),  
        c1.VerParteImag() + c2.VerParteImag());  
    cout << suma.VerParteReal() << "+";  
    cout << suma.VerParteImag() << "i";  
}
```

Dos alternativas:

```
suma = Sumar(c1, c2); // función global
```

```
suma = c1.Sumar(c2); // método de la clase
```

# EJEMPLO: CLASE COMPLEJO

Suma mediante función global:

```
class Complejo { ... };  
  
Complejo sumar (Complejo c1, Complejo c2) {  
    Complejo suma(  
        c1.VerParteReal()+c2.VerParteReal()  
        c1.VerParteImag()+c2.VerParteImag() );  
    return suma;  
}
```

⚠ No se puede acceder a los atributos privados, a menos que se declare amistad

✅ La suma retorna un tercer complejo, que no es ninguno de los operandos

```
Complejo suma = sumar(c1, c2);
```

# EJEMPLO: CLASE COMPLEJO

Suma como método de la clase Complejo:

```
class Complejo {  
    float m_prealm, m_pimag;  
public:  
    ...  
    Complejo sumar (Complejo c2) const {  
        Complejo suma(  
            this->m_prealm+c2.m_prealm ,  
            this->m_pimag+c2.m_pimag );  
        return suma;  
    }  
};
```

✓ El primer operando es *\*this*, solo el segundo se recibe como argumento

```
Complejo suma = c1.sumar(c2);
```

# **SOBRECARGA DE OPERADORES**

C++ permite especificar el comportamiento de sus operadores estándar al aplicarlos a tipos de datos no fundamentales (clases o estructuras).

```
int a, b, c;  
cin >> a >> b;  
c = a + b;  
cout << c;
```

# SOBRECARGA DE OPERADORES

C++ permite especificar el comportamiento de sus operadores estándar al aplicarlos a tipos de datos no fundamentales (clases o estructuras).


```
int a, b, c;  
cin >> a >> b; // oper. >> aplicado al objeto cin y el tipo int  
c = a + b; // ops. + e = aplicados a un tipo fundamental (int)  
cout << c; // oper. << aplicado al objeto cout y el tipo int
```

```
class Complejo { ... };  
...  
Complejo a, b, c;  
// operadores +, =, << y >> aplicados a una clase arbitraria  
cin >> a >> b;  
c = a + b;  
cout << c;
```

# SOBRECARGA DE OPERADORES EN C++

Los operadores que se pueden sobrecargar son:

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

- El operador de asignación por defecto funciona de forma análoga al constructor de copia por defecto (igualando uno por uno los atributos).  
 entonces va a tener los mismos problemas!
- Los operadores new y delete también tienen un funcionamiento por defecto, el resto no.



# SOBRECARGA DE OPERADORES EN C++

Hay dos formas de sobrecargar operadores:

- Como miembro de una clase (la del primer o único operando):

```
class Complejo {  
public:  
    Complejo operator+ (Complejo c2) const;  
    ...  
};
```

- Como función global:

```
Complejo operator+ (Complejo c1,  
                   Complejo c2);
```

# SOBRECARGA DE OPERADORES EN C++

Sobrecarga mediante función global:

```
class Complejo { ... };  
  
Complejo operator+ (Complejo c1,  
                    Complejo c2)  
{  
    Complejo suma(  
        c1.VerParteReal()+c2.VerParteReal(),  
        c1.VerParteImag()+c2.VerParteImag() );  
    return suma;  
}
```

Se declara y define igual que cualquier otra función, pero con el nombre **operator+**

# SOBRECARGA DE OPERADORES EN C++

Sobrecarga mediante función global:

```
int main() {  
    Complejo c1(1,3), c2(4,5), suma;  
  
    suma = c1+c2; // Es equivalente a:  
                  // suma = operator+(c1, c2);  
  
    cout << suma.VerParteReal() << "+";  
    cout << suma.VerParteImag() << "i";  
}
```

# SOBRECARGA DE OPERADORES EN C++

Sobrecarga mediante método de la clase  
Complejo:

```
class Complejo {  
    float m_preal, m_pimag;  
public:  
    ...  
    Complejo operator+ (Complejo c2) const {  
        Complejo suma(  
            this->m_preal+c2.m_preal ,  
            this->m_pimag+c2.m_pimag );  
        return suma;  
    }  
};
```

Se declara y define igual que cualquier método,  
pero con nombre **operator+**

# SOBRECARGA DE OPERADORES EN C++

Sobrecarga mediante método de la clase  
Complejo:

```
int main() {  
    Complejo c1(1,3), c2(4,5), suma;  
  
    suma = c1+c2; // Es equivalente a:  
                  // suma = c1.operator+(c2);  
  
    cout << suma.VerParteReal() << "+";  
    cout << suma.VerParteImag() << "i";  
}
```

# SOBRECARGA Y PUNTERO `this`

Los operadores que modifican al primer operando, suelen retornar al mismo objeto (por referencia) para permitir la aplicación en cadena:

```
class Complejo {  
    ...  
    Complejo& operator=(const Complejo &c) {  
        this->m_prealm = c.m_prealm;  
        this->m_pimag = c.m_pimag;  
        return *this;  
    }  
}  
  
...  
Complejo a, b, c, d(1, 1);  
a = b = c = d; // Es equivalente a:  
                // a.operator=(b.operator=(c.operator=(d) ));
```

# OPERADOR DE ASIGNACIÓN

- C++ genera un operador de asignación por defecto para cualquier clase o estructura
  - copia atributo a atributo.
- Cuando la clase utiliza memoria dinámica, el operador de asignación debe sobrecargarse
  - por el mismo motivo que se debe implementar o prohibir el constructor de copia.

```
class VectorDb1 { double *m_p; ... };  
int main {  
    VectorDb1 v1, v2;  
    ...  
    v1 = v2;
```

! el mismo análisis se puede aplicar al op. de **move**

# OPERADOR DE ASIGNACIÓN

```
class VectorDbl {  
    double *m_p; // vector dinámico  
    int m_n; // tamaño del vector a  
public:  
    VectorDbl() { ... }  
    VectorDbl(const VectorDbl &v2) { ... }  
    // ...otros métodos...  
    VectorDbl& operator=(const VectorDbl &v2) {  
        delete[] this->m_p;  
        ojo! esta es la diferencia con el ctor de copia  
        m_p = new double[v2.m_n];  
        for(int i=0; i<v2.m_n; i++)  
            this->m_p[i] = v2.m_p[i];  
        this->m_n = v2.m_n;  
        return *this;  
    }  
};
```



# LÍMITES DE LA SOBRECARGA DE OPERADORES

- La cantidad de operandos y la precedencia de un operador no pueden alterarse.
- En una sobrecarga, uno de los dos operandos debe ser un tipo definido por el usuario.
- No todos los operadores pueden sobrecargarse (ej. de operadores que todavía no: `?:` `::` `.` `.*`)

# FUNCIÓN GLOBAL VS MÉTODO DE CLASE

- Función miembro:
  - recomendado: operadores asimétricos
  - obligatorios: = ( ) [ ] -> ->\*
- Función global:
  - recomendado: operadores simétricos
  - obligatorio: clases ajenas o tipos fundamentales

```
#include <iostream> // ostream cout;
...
Complejo c1;
cout << c1; // Es equivalente a:
              //      cout.operator<<(c1);
              //      ó
              //      operator<<(cout, c1);
```

# SOBRECARGA DE << Y >> PARA ENTRADA/SALIDA

- cout/cin son instancias de las clases ostream/istream
- Estas clases ya tienen sobrecargas para los tipos de datos fundamentales

```
ostream &operator<<(ostream &o, Complejo c) {  
    o << c.VerParteReal() << "+";  
    << c.VerParteImag() << "i";  
    return o;  
}  
istream &operator>>(istream &i, Complejo &c) {  
    float a; i >> a; c.CargarParteReal(a);  
    float b; i >> b; c.CargarParteImag(b);  
    return i;  
}
```

# CASOS ESPECIALES: PRE Y POST INCREMENTO

- Algunos operadores unarios pueden aplicarse de dos formas diferentes:
  - pre-incremento y post-incremento:

```
int c;  
++c; // pre-incremento  
c++; // post-incremento
```

- Para diferenciarlos se usa (arbitrariamente) un argumento ficticio de tipo int:

```
class Fraccion {  
    ...  
    Fraccion &operator++(); // pre  
    Fraccion operator++(int); // post  
};
```

# CASOS ESPECIALES: PRE Y POST INCREMENTO

```
class Fraccion {
    int m_num, m_den;
public:
    ...
    Fraccion &operator++() { // pre
        m_num += m_den;
        return *this;
    }
    Fraccion operator++(int) { // post
        Fraccion aux = *this;
        m_num += m_den;
        return aux;
    }
    ...
};
```

❓ ¿Por qué uno lleva & y el otro no?

# USO DE const EN MÉTODOS

```
class Alumno {  
    string nombre;  
    float promedio;  
public:  
    ...  
    float VerProm() const {  
        return promedio;  
    }  
};  
...  
int MejorAlumno(const vector<Alumno> &v) {  
    ...  
    if (v[i].VerProm() < v[m].VerProm())  
        ...  
}
```

⚠ si un objeto es const, solo permite invocar métodos const

# USO DE const EN OPERANDOS

```
struct Alumno {  
    string nombre;  
    float promedio;  
    ...  
    bool operator<(const Alumno &o) const {  
        return promedio < o.promedio;  
    }  
};
```

- El `const` del argumento aplica al argumento `o`
- El `const` al final del prototipo aplica al `*this`

```
int MejorAlumno(const vector<Alumno> &v) {  
    ...  
    if ( v[i] < v[m] ) ...
```

# OPERADOR []

```
class Complejo {
    float m_preal, m_pimag;
public:
    float operator[] (int i) const { // p/ver
        retorna el valor (copia)
        if (i==0) return m_preal;
        else      return m_pimag;
    }
    float& operator[] (int i) { // p/modificar
        retorna la variable (referencia)
        if (i==0) return m_preal;
        else      return m_pimag;
    }
};

int main() {
    Complejo c;
    c[0] = 1; c[1] = 5;
    cout << c[0] << "+" << c[1] << "i";
}
```