

CS1112: Programación 2

Unidad 8: Sobrecarga de operadores

Sesión de Teoría - 13

Profesor:

José Antonio Fiestas Iquiria jfiestas@utec.edu.pe

Material elaborado por:

Maria Hilda Bermejo, José Fiestas, Rubén Rivas



Índice:

- Unidad 8: Sobrecarga de operadores
 - Funciones amigas
 - Sobrecarga de operadores
 - Casos particulares de sobrecarga

8.1

Unidad 8: Funciones amigas



Logro de la sesión:

Al terminar esta sesión el alumno se familiariza con el concepto de funciones amigas y de sobrecarga de operadores.

Funciones amigas

- Los atributos privados no son accesibles desde otra clase.
- El modificador **friend** permite dar acceso a los atributos privados a otra clase.
- Reglas
 - La amistad no puede transferirse.
 - La amistad no puede heredarse.
 - La amistad no es simétrica.
- Las funciones amigas son útiles con la sobrecarga de operadores y la creación de cierto tipo de funciones.

Las relaciones de "amistad" entre clases son parecidas a las amistades entre personas:

- **La amistad no puede transferirse**, si A es amigo de B, y B es amigo de C, esto no implica que A sea amigo de C. (La famosa frase: "los amigos de mis amigos son mis amigos" es falsa en C++, y probablemente también en la vida real).
- **La amistad no puede heredarse**. Si A es amigo de B, y C es una clase derivada de B, A no es amigo de C. (Los hijos de mis amigos, no tienen porqué ser amigos míos. De nuevo, el símil es casi perfecto).
- **La amistad no es simétrica**. Si A es amigo de B, B no tiene por qué ser amigo de A. (En la vida real, una situación como esta hará peligrar la amistad de A con B, pero de nuevo me temo que en realidad se trata de una situación muy frecuente, y normalmente A no sabe que B no se considera su amigo).

Funciones Amigas: versión 1

```
#include "tipos.h"
class Rectangulo
{
private:
    TipoEntero m_largo;
    TipoEntero m_ancho;
public:
    Rectangulo() {
        m_largo=10;
        m_ancho=20;
    }
};
```

Rectangulo.h

```
#include <iostream>
#include "Rectangulo.h"
#include "Cuadrado.h"
int main() {
    Rectangulo r;
    Cuadrado c;
    c.mostrar(cout,r);
    return 0;
}
```

```
Largo : 10
Ancho : 20
Lado : 5
```

```
using namespace std;
class Cuadrado {
private:
```

Cuadrado.h

```
    TipoEntero m_lado;
```

```
public:
```

```
    Cuadrado() {
```

```
        m_lado=5;
```

```
    }
```

```
    void mostrar(ostream &os,Rectangulo Rect)
```

```
    {
```

```
        os<<"\nLargo : "<<Rect.m_largo;
```

```
        os<<"\nAncho : "<<Rect.m_ancho;
```

```
        os<<"\nLado : "<<m_lado;
```

```
    }
```

```
};
```

¿Es posible acceder al atributo m_largo?

Tipos.h

```
using TipoEntero = int; //desde c++ 11
```

Funciones Amigas: versión 2

```
#include "tipos.h"
```

```
class Rectangulo
```

```
{
```

```
private:
```

```
    TipoEntero m_largo;
```

```
    TipoEntero m_ancho;
```

```
public:
```

```
    Rectangulo() {
```

```
        m_largo=10;
```

```
        m_ancho=20;
```

```
    }
```

```
    friend class Cuadrado;
```

```
    //Se declara la clase amiga(friend)
```

```
};
```

Rectangulo.h

Tipos.h

```
using TipoEntero = int; //desde c++ 11
```

```
using namespace std;
```

```
class Cuadrado {
```

```
private:
```

```
    TipoEntero m_lado;
```

```
public
```

```
    Cuadrado() {
```

```
        m_lado=5;
```

```
    }
```

```
    void mostrar(ostream &os, Rectangulo Rect)
```

```
    {
```

```
        os<<"\nLargo : "<<Rect.m_largo;
```

```
        os<<"\nAncho : "<<Rect.m_ancho;
```

```
        os<<"\nLado : "<<m_lado;
```

```
    }
```

```
};
```

Cuadrado.h

¿Es posible acceder al atributo m_largo?

```
#include <iostream>
```

```
#include "Rectangulo.h"
```

```
#include "Cuadrado.h"
```

```
int main() {
```

```
    Rectangulo r;
```

```
    Cuadrado c;
```

```
    c.mostrar(cout, r);
```

```
    return 0;
```

```
}
```

Largo : 10

Ancho : 20

Lado : 5

8.2

Unidad 8: Sobrecarga de operadores



Definición

- Los **operadores** son un tipo de tokens que indican al compilador operación que debe hacer para variables u otros.
- La **sobrecarga de operadores** permite redefinir ciertos operadores, como '+' y '-', para usarlos con las clases que hemos definido.
- Se llama **sobrecarga de operadores** cuando reutilizando el mismo operador con un número de usos diferentes, y el compilador decide cómo usar este operador dependiendo sobre qué opera.

Operadores sobrecargables

Los siguientes operadores pueden ser sobrecargados:

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

...Operadores sobrecargables

- Los operadores +, -, * y & son sobrecargables en sus dos versiones, unaria y binaria.
 - Suma binaria +;
 - Más unitario +;
 - Multiplicación *;
 - Indirección *;
 - Referencia &
 - Manejo de bits &.
- C++ ofrece casos de operadores sobrecargados incluso en su Librería Estándar.
 - Los operadores == y != para la clase `type_info`

Excepciones

- Los **operadores** que **NO** pueden ser **sobrecargados**.
 - Selector directo de componente .
 - Operador de indirección de puntero-a-miembro .*
 - Operador de acceso a ámbito ::
 - Condicional ternario ?:
 - Directivas de preprocesado # y # #
 - **sizeof, typeid**
- Los operadores asignación =; elemento de matriz [] ; invocación de función () y selector indirecto de miembro -> pueden ser sobrecargados solamente como funciones-miembro no estáticas y no pueden ser sobrecargados para las enumeraciones.
- Cualquier intento de sobrecargar la versión global de estos operadores produce un error de compilación.

La función operator

- Los operadores C++ pueden considerarse funciones con identificadores un tanto especiales.
- Por ejemplo, cuando tenemos el operador de subíndice de matriz, `x[y]`, donde x es un objeto de la clase X, el compilador lo traduce a la expresión:

`x.operator[](y).`

Es decir, lo interpreta como la invocación de un método de nombre `operator[]`.

Sobrecarga v1

Entero.h

```
#include <iostream>
#include "Tipos.h"
using namespace std;

class Entero {
private:
    TipoEntero dato;
public:
    Entero(){}
    Entero(int _dato) {dato = _dato;}
    int get_dato(){return dato;}
};
```

main.cpp

```
#include <iostream>
#include "Entero.h"
using namespace std;
int main()
{
    Entero a(10);
    Entero b(40);
    Entero c;

    c = a + b;

    return 0;
}
```

No es posible hacer la suma, porque a y b son clases y el operador + solo trabaja con tipos primitivos

Tipos.h

```
using TipoEntero = int; //desde c++ 11
```

Sobrecarga de operador binario +

- Para resolver el problema anterior, se sobrecarga el operador '+' y el operador '<<'

Sobrecarga con operador '+' y '<<'

```
#include <iostream>
#include "Tipos.h"
using namespace std;
class Entero {
private:
    TipoEntero m_dato;
public:
    Entero() { }
    Entero(TipoEntero _dato) { m_dato= _dato; }
    int getData() {return m_dato;}

    friend Entero operator+(Entero& x, Entero& y);
    friend ostream& operator<<(ostream& o, Entero e);
};
```

Entero.h

```
#include "Entero.h"

Entero operator+(Entero &x, Entero &y) {
    return Entero(x.m_dato + y.m_dato);
}

ostream& operator<<(ostream &o, Entero e) {
    o << e.m_dato + 5;
    return o;
}
```

Entero.cpp

```
#include <iostream>
#include "Entero.h"
using namespace std;
int main() {
    Entero a(10);
    Entero b(40);
    Entero c;
    c = a + b;
    cout << "c = " << c.getData() << "\n";
    cout << "c = " << c << "\n";

    return 0;
}
```

main.cpp

Salida:

c = 50
c = 55

Tipos.h

```
using TipoEntero = int; //desde c++ 11
```

Sobrecarga con función

```
#include "Entero.h"
Entero suma(Entero &x, Entero &y)
{
    return Entero( x.m_dato + y.m_dato);
}
```

Entero.cpp

```
#include <iostream>
#include "Tipos.h"
using namespace std;
class Entero {
private:
    TipoEntero m_dato;
public:
    Entero() { }
    Entero(TipoEntero dato) { m_dato= dato; }
    int getData() {return m_dato;}
    friend Entero suma(Entero& x, Entero& y);
};
```

Entero.h

```
#include <iostream>
#include "Entero.h"
using namespace std;
int main() {
    Entero a(10);
    Entero b(20);
    Entero c;
    c = suma(a,b);
    cout << "c = " <<c.getData() << "\n";
    return 0;
}
```

main.cpp

Tipos.h

```
using TipoEntero = int; //desde c++ 11
```

Sobrecarga con función

Entero.h

```
#include <iostream>
using namespace std;
typedef int TipoEntero ;
class Entero {
private:
    TipoEntero dato;
public:
    Entero() { }
    Entero(TipoEntero _dato) { dato= _dato; }
    TipoEntero getData() {return dato;}
    friend Entero suma(Entero& x, Entero& y);
};
```

```
#include "Entero.h"
Entero suma(Entero &x, Entero &y)
{
    return Entero( x.dato + y.dato);
}
```

Entero.cpp

main.cpp

```
#include <iostream>
#include "Entero.h"

using namespace std;

int main() {
    Entero a(10);
    Entero b(20);
    Entero c;

    c = suma(a,b);
    cout << "c = " << c.getData() <<
    "\n";

    return 0;
}
```

Salida:

c = 50

Sobrecarga operador suma(+)

- Si tenemos la clase CComplejo. Redefinimos el operador suma.

```
class CComplejo
{
    public:
        // Permite sumar un CComplejo con un double
        CComplejo operator + (double sumando);

        // Permite sumar un CComplejo con un array
        CComplejo operator + (const double sumando[]);

        // Permite sumar dos CComplejo entre sí.
        CComplejo operator + (const CComplejo &sumando);
};
```

se usa **const** para
no cambiar los
valores

se usa **&** para
evitar que se
hagan copias

Sobrecarga operador suma(+)

- Después de implementar, podemos hacer:

```
CComplejo * CComplejo;  
CComplejo b;
```

```
// Aprovechando la primera sobrecarga  
b = a + 1.0;
```

```
// Aprovechando la segunda sobrecarga  
TipoDoble c[] = {1.0, 2.0};  
b = a + c;
```

```
// Aprovechando la tercera sobrecarga  
b = a + b;
```

Cuando el compilador lee `a + 1.0`, lo interpreta como `a.operator + (1.0)`, es decir, la llamada al operador suma al que se le pasa como parámetro un `double`.

¿Qué pasa si ponemos `1.0 + a`? Debería ser lo mismo, pero al compilador le da un pequeño problema. Intenta llamar al método `operator +` de `1.0`, que no existe, puesto que `1.0` ni es una clase ni tiene métodos.

Para solucionar este problema tenemos la sobrecarga de operadores globales.

Sobrecarga de operadores suma globales

- Un **operador global** es una función global que no pertenece a ninguna clase y que nos indica cómo operar con uno o dos parámetros (depende del tipo de operador).
- Para poder poner los operandos al revés, debemos:

```
CComplejo operator +(TipoDoble sumando1, const CComplejo  
&sumando2);
```

```
CComplejo operator +(TipoDoble sumando1[], const  
CComplejo &sumando2);
```

Estas funciones le indican al compilador cómo debe sumar los dos parámetros y qué devuelve

...Sobrecarga de operadores suma globales

```
b = 1.0 + a;  
// c era un array de double  
b = c + a;
```

Con ellas definidas e implementadas, ya podemos hacer

Esta sobrecarga es especialmente útil cuando tratamos con una clase ya hecha y que no podemos modificar. Por ejemplo, **cout** es de la clase **ostream** y no podemos modificarla, sin embargo nos sería de utilidad sobrecargar el operador **<<** de **ostream** de forma que pueda escribir nuestros números complejos.

```
// a es un CComplejo  
cout << a << endl;
```

La siguiente llamada nos dará error mientras no redefinamos el operador **<<** de **ostream**.

...Sobrecarga de operadores suma globales

Con la sobrecarga de operadores globales podemos definir la función

```
ostream &operator << (ostream &salida, const CComplejo &valor);
```

Con esta función definida, el complejo se escribirá en pantalla como indique dicha función. Esta función deberá escribir la parte real e imaginaria del complejo en algún formato, utilizando algo como

```
salida << valor.getX() << " + " << valor.getY() << "j";
```

El operador devuelve un ostream, que será un return cout. De esta forma se pueden encadenar las llamadas a cout de la forma habitual.

```
cout << a << " + " << b << endl;
```


...Sobrecarga de operadores suma globales

Primero se evalúa operator << (cout, a), que escribe a en pantalla y devuelve un cout, con lo que la expresión anterior quedaría, después de evaluar esto

```
cout << " + " << b << endl;
```

y así consecutivamente.

Hay que tener en cuenta que estos operadores globales no son de la clase, así que sólo pueden acceder a métodos y atributos públicos de la misma.

Sobrecarga de operadores: operador cast

- En C++, tenemos dos tipos básicos distintos, podemos pasar de uno a otro haciendo un cast (siempre que sean compatibles).

Por ejemplo

```
TipoEntero num;  
double dato;  
num = (int)dato;
```

- El cast consiste en poner delante, entre paréntesis, el tipo que queramos que sea. En algunos casos, como en el de este ejemplo, el cast se hace automáticamente y no es necesario ponerlo. Puede que de un "warning" en el compilador avisando de que perderemos los decimales.
- En principio, con las clases no se puede hacer cast a otros tipos, pero es posible declarar operadores que lo hagan. La sintaxis sería:

```
class CComplejo  
{  
    public:  
        // Permite hacer un cast de CComplejo a  
double  
    operator double ();  
}
```

...Sobrecarga de operadores: operador cast

- En principio, con las clases no se puede hacer cast a otros tipos, pero es posible declarar operadores que lo hagan. La sintaxis sería:

```
class CComplejo{  
    public:  
        // Permite hacer un cast de CComplejo a  
double  
        operator TipoDoble ();  
};
```

- Con este podemos hacer cast de nuestra clase a un double . Es nuestro problema decidir cómo se hace ese cast. En el código de ejemplo que hay más abajo se ha definido como la operación módulo del número complejo.

```
TipoDoble a;  
CComplejo b;  
a = (TipoDoble)b;
```

...Sobrecarga de operadores: operador cast

- En el operador cast se pone **operator** seguido del **tipo** al que se quiere hacer el cast.
- No se pone el tipo del valor devuelto, puesto que ya está claro.
- Si ponemos operator double, hay que devolver un double .
- En el **operator cast** no se pone parámetro, puesto que el parámetro recibido será una instancia de la clase.
- **¿Cómo hacemos un cast al revés?**. Es decir, ¿Cómo podemos convertir un double a CComplejo?. Se debe hacer un constructor que admite un double como parámetro.

```
class CComplejo
{
    public:
        CComplejo (double valor);
};

CComplejo a(5.0);

CComplejo a;
a = (CComplejo)5.0;
```

Este constructor sirve para poder crear:

Para hacer un cast de double a CComplejo

Clase complejo

A continuación tenemos el código de la CComplejo

Ejemplo: Clase complejo

CComplejo.h

```
#include <iostream>
#include "Tipos.h"

/**
 * Un número complejo.
 * Sobrecarga el operador + para poder sumar otras clases CComplejo, arrays de dos
 * doubles y doubles sueltos.
 */
using namespace std;

class CComplejo {
private:
    /** Parte real del complejo */
    TipoDoble m_x;

    /** Parte imaginaria del complejo */
    TipoDoble m_y;
public:
    /** Constructor por defecto. Rellena los atributos x e y a 0.0 */
    CComplejo ();

    /** Constructor con array de doubles. Mete el primer double del array en
     * x y el segundo en y */
    CComplejo (const TipoDoble origen[]);

    /** Constructor copia. Copia los atributos del CComplejo recibido en los internos
     * de la clase. */
    CComplejo (const CComplejo &origen);
```

Ejemplo: Clase complejo

CComplejo.h

```
/** Constructor con un double. Lo pone en la parte real x */
CComplejo (TipoDoble valor);
CComplejo (TipoDoble valor1, TipoDoble valor2);

/** Operador igual para array de doubles. Mete el primer double del array
 * en x y el segundo en y */
CComplejo &operator = (const TipoDoble origen[]);

/** Operador igual para otro CComplejo. Copia los atributos del
 * CComplejo recibido. */
CComplejo &operator = (const CComplejo &origen);

/** Operador suma para un double. Suma el double recibido en x */
CComplejo operator + (TipoDoble sum) const;

/** Operador suma para un array de doubles. Suma el primer double del array en
 * x y el segundo en y */
CComplejo operator + (const TipoDoble sum[]);

/** Operador suma para otro CComplejo. Suma los atributos del CComplejo recibido
 * con los internos */
CComplejo operator + (const CComplejo &sum) const;

/** cast a double. Devuelve el módulo */
operator double ();
```

Ejemplo: Clase complejo

CComplejo.h

```
/** Devuelve X */
TipoDoble getX() const { return m_x; }

/** Devuelve Y */
TipoDoble getY() const { return m_y; }

/** Devuelve X */
TipoDoble setX(TipoDoble valor) { m_x = valor; }

/** Devuelve Y */
TipoDoble setY(TipoDoble valor) { m_y = valor; }
};
/*****
* FUNCIONES EXTERNAS A LA CLASE
*****/

/** Sobrecarga del operador suma global para sumar un double con un CComplejo */
CComplejo operator + (TipoDoble k, const CComplejo &origen);

/** Sobrecarga del operador suma global para sumar un array de doubles y un CComplejo */
CComplejo operator + (const TipoDoble sum1[], const CComplejo &sum2);

/** Sobrecarga del operador << global para que cout "sepa" escribir un CComplejo */
ostream &operator << (ostream &salida, const CComplejo &origen);

/** Sobrecarga del operador << global para que cout "sepa" escribir un array de doubles */
ostream &operator << (ostream &salida, const TipoDoble origen[] );
```


Ejemplo: Clase complejo

CComplejo.cpp

```
#include "Complejo.h"
#include <math.h>

/*****
 * METODOS DE LA CLASE
 *****/

/***** CONSTRUCTORES *****/

/* Constructor por defecto. Pone Los atributos x e y a 0.0 */
CComplejo::CComplejo () {
    m_x=0.0;
    m_y=0.0;
}

/* Constructor por defecto. Pone Los atributos x e y a 0.0 */
CComplejo::CComplejo (TipoDoble valor) {
    m_x=valor;
    m_y=0.0;
}

/* Constructor por defecto. Pone Los atributos x e y a 0.0 */
CComplejo::CComplejo (TipoDoble valor1, TipoDoble valor2) {
    m_x=valor1;
    m_y=valor2;
}
```

Ejemplo: Clase complejo

CComplejo.cpp

```
/* Constructor con un array de doubles. Pone el primer elemento del array en x y el
 * segundo en y */
CComplejo::CComplejo (const TipoDoble origen[]) {
    *this = origen;
}

/** Constructor copia. Copia los atributos de origen en los internos de la clase */
CComplejo::CComplejo (const CComplejo &origen) {
    m_x = origen.m_x;
    m_y = origen.m_y;
}

/***** OPERADORES SUMA EN LA CLASE *****/

/** operador suma para un double. Lo suma a la parte real y devuelve CComplejo para
 * poder concatenar sumas */
CComplejo CComplejo::operator + (TipoDoble sum) const {
    CComplejo aux(*this);
    aux.m_x = m_x + sum;
    return aux;
}
```

Ejemplo: Clase complejo

CComplejo.cpp

```
/** operador suma para array de doubles. Devuelve CComplejo para poder concatenar sumas */
CComplejo CComplejo::operator + (const TipoDoble sum[]) {
    //CComplejo aux;
    //aux.m_x = m_x + sum[0];
    //aux.m_y = m_y + sum[1];
    //return aux;
    return CComplejo(m_x + sum[0], m_y + sum[1]);
}

/** operador suma para otro CComplejo. Devuelve CComplejo para poder concatenar sumas */
CComplejo CComplejo::operator + (const CComplejo &sum) const {
    return CComplejo(m_x + sum.m_x, m_y + sum.m_y);
}
```

Ejemplo: Clase complejo

CComplejo.cpp

```
/****** OPERADORES DE ASIGNACION EN LA CLASE *****/
```

```
/** operador = para arrays de doubles. Devuelve CComplejo para poder concatenar  
* operaciones de = estilo a = b = c; */
```

```
CComplejo &CComplejo::operator = (const TipoDoble origen[]) {  
    m_x = origen[0];  
    m_y = origen[1];  
    return *this;  
}
```

```
/** operador = para otro CComplejo. Devuelve CComplejo para poder concatenar  
* operaciones de =, estilo a = b = c; */
```

```
CComplejo &CComplejo::operator = (const CComplejo &origen) {  
    m_x = origen.m_x;  
    m_y = origen.m_y;  
    return *this;  
}
```

```
/****** OPERADORES DE CAST EN LA CLASE *****/
```

```
/** operador cast a double. Devuelve el módulo del complejo */
```

```
CComplejo::operator TipoDoble () {  
    return sqrt (m_x*m_x + m_y*m_y);  
}
```

Ejemplo: Clase complejo

CComplejo.cpp

```

/*****
* FUNCIONES EXTERNAS A LA CLASE
*****/

/***** OPERADOR << PARA EL COUT *****/

/* operador global << para escribir CComplejo en pantalla. */
ostream &operator << (ostream &os, const CComplejo &origen) {
    // Se pone signo positivo por defecto
    TipoChar signo='+';

    // Si la y es negativa, no se pone signo, ya que al escribir la y ya sale un signo
    // negativo.
    if (origen.getY() < 0.0)
        signo = 0;

    // Se escriben los campos separados por el signo
    os << origen.getX() << signo << origen.getY() << "j";
}

```

Ejemplo: Clase complejo

CComplejo.cpp

```
/* operador global << para escribir array de doubles en pantalla. */
ostream &operator << (ostream &salida, const TipoDoble origen[]) {
    // Se pone signo positivo por defecto
    TipoChar signo='+';

    // Si la y es negativa, no se pone signo, ya que al escribir la y ya sale un signo
    // negativo.
    if (origen[1] < 0.0)
        signo = 0;

    // Se escriben los campos separados por el signo
    cout << origen[0] << signo << origen[1] << "j";
}

/***** OPERADORES SUMA EXTERNOS *****/

/* operador global + para sumar array de doubles con CComplejo. * Devuelve un CComplejo para poder encadenar sumas
a+b+c+d */
CComplejo operator + (const TipoDoble sum1[], const CComplejo &sum2) {
    return CComplejo(sum2 + sum1);
}

/* operador global + para sumar un double con un complejo. Devuelve CComplejo
* para poder encadenar sumas */
CComplejo operator + (TipoDoble sum, const CComplejo &origen) {
    return CComplejo(origen + sum);
}
```

Ejemplo: Clase complejo

Tipos.h

```
using TipoDoble = double;  
using TipoChar = char;
```

main.cpp

```
#include <iostream>  
#include "Complejo.h"  
using namespace std;  
  
int main() {  
    TipoDoble c1[] = {1.0, -1.0}; // Un array de doubles.  
    TipoDoble c2[] = {-1.0, 1.0}; // Otro array de doubles.  
    CComplejo complejoA(c1) ; // complejoA, copia del array c1.  
    CComplejo complejoB;      // complejoB, por defecto.  
  
    // Se realizan varios tipos de sumas, escribiendo en pantalla los resultados.  
  
    // CComplejo operator + (double sum);  
    // 1-j + 1 da 2-j  
    cout << complejoA << " + " << 1 << " = ";  
    cout << complejoA + 1.0 << endl;  
  
    // CComplejo operator + (double sum[]);  
    // 1-j + 1-j da 2-2j  
    cout << complejoA << " + " << c1 << " = ";  
    cout << complejoA + c1 << endl;
```

Ejemplo: Clase complejo

main.cpp

```
// CComplejo operator + (CComplejo sum);
// 1-j + 0+0j da 1-j
cout << complejoA << " + " << complejoB << " = ";
cout << complejoA + complejoB << endl;

// CComplejo operator + (double k, CComplejo &origen);
// 1 + 1-j + 1-j + -1+j + 0+0j da 2-j
cout << 1 << " + " << complejoA << " + " << c1 << " + " << c2 << " + " << complejoB \
    << " = ";
cout << 1.0 + complejoA + c1 + c2 + complejoB << endl;

// CComplejo operator + (double sum1[], CComplejo &sum2);
// -1+j + 1-j da 0+0j
cout << c2 << " + " << complejoA << " = ";
cout << c2 + complejoA << endl;

// CComplejo operator double
// El módulo de 1-j es 1.4142 (raiz de 2)
cout << "|" << complejoA << "| = ";
cout << (TipoDoble)complejoA << endl;

// cast de double a CComplejo
// Debe dar 2.3+0j
cout << (CComplejo)2.3 << endl;

return 0;
}
```

Salida:

$$1 - 1j + 1 = 2 - 1j$$

$$1 - 1j + 1 - 1j = 2 - 2j$$

$$1 - 1j + 0 + 0j = 1 - 1j$$

$$1 + 1 - 1j + 1 - 1j + -1 + 1j + 0 + 0j = 2 - 1j$$

$$-1 + 1j + 1 - 1j = 0 + 0j$$

$$|1 - 1j| = 1.41421$$

$$2.3 + 0j$$

Sobrecarga de operadores: operador igual

- C++ por defecto tiene el operador igual definido para clases del mismo tipo

```
CComplejo a;  
CComplejo b;  
  
a = b;
```

Sin necesidad de redefinir nada, podemos hacer **a=b**

Este igual por defecto lo único que hace es copiar el contenido del uno sobre el otro, como si fueran bytes, sin saber qué atributos está copiando ni qué significan.

Para clases sencillas, que sólo tienen atributos que no son punteros, esto es más que suficiente.

Sobrecarga de operadores: operador igual

```
CClass
{
    public:
        // Se crea un array de tres enteros
        CClass () {
            codigo= new int[5];
        }
        // Se libera el array.
        ~CClass () {
            delete [] codigo;
        }
    protected:
        TipoEntero *codigo;
};
```

- Sí **atributo** es un **puntero**, debemos tener cuidado.

- En el ejemplo la CClass tiene un atributo que es un puntero.

- El constructor de la clase, se hace new del puntero para que tenga algo y en el destructor se hace el delete correspondiente.

Sobrecarga de operadores: operador igual

```
// a tiene ahora un array de 5 enteros
en su interior
CClass *a = new CClass();

// b tiene ahora otro array de 5 enteros
en su interior
CClass *b = new CClass();

/* Se copia el Atributo de b sobre el de
a, es decir, ahora a->Atributo apunta al
mismo sitio que b->Atributo */
*a = *b;

// eliminamos b.
delete b;
```

- Cuando hacemos `a=b`, con el operador igual por defecto de C++, se hace que el puntero Atributo de `a` apunte al mismo sitio que el de `b`. El array original de `a->Atributo` lo hemos perdido, sigue ocupando memoria y no tenemos ningún puntero a él para liberarlo..
- Cuando hacemos `delete b`, el destructor de `b` se encarga de liberar su array. Sin embargo, al puntero `a->Atributo` nadie le avisa de esta liberación, se queda apuntando a una zona de memoria que ya no es válida

Sobrecarga de operadores: operador igual

```
class CClase
{
    public:
        CClase &operator = (const CClase
&original)
        {
            TipoEntero i;
            /* Damos por supuesto que ambos arrays
existen y son de tamaño 5 */
            for (i=0;i<5;i++)
                codigo[i] = original.codigo[i];

            return *this;
        }
};
```

- Para solucionar esto, es definiendo un operador = que haga una copia real del array, liberando previamente el nuestro o copiando encima los datos.

Sobrecarga de operadores: operador subíndice []

- Este operador sirve para señalar subíndices de matrices simples y multidimensionales; de ahí su nombre, operador **subíndice** o de **selección de miembro de matriz**.

La expresión:

`<exp1>[exp2]`



`arrX[3] == *(arrX + 3)`

Se define como: `*((exp1) + (exp2))` donde `exp1` es un puntero y `exp2` es un entero o viceversa.

Por ejemplo, `arrX[3]` se define como: `*(arrX + 3)` o `*(3 + arrX)`, donde `arrX` es un puntero al primer elemento de la matriz. `(arrX + 3)` es un puntero al cuarto elemento, y `*(arrX + 3)` es el valor del cuarto elemento de la matriz.

Sobrecarga de operadores: operador subíndice []

```
class Vector {  
    public: TipoEntero x, y;  
};  
  
class mVector {  
    public:  
    Vector* mVptr;  
    mVector(TipoEntero n = 1) {  
        mVptr = new Vector[n];  
    }  
    virtual ~mVector() {           // destructor  
        delete [] mVptr;  
    }  
};
```

- La clase Vector tiene solo dos miembros.
- La clase mVector que contiene una **matriz** (es una matriz de vectores)
- La clase mVector tiene un solo miembro; un puntero-a-Vector mVptr
- El constructor del objeto tipo mVector, crea una matriz de objetos tipo Vector del tamaño indicado en el argumento (1 por defecto) y la señala con el puntero mVptr

Sobrecarga de operadores: operador subíndice []

- Siguiendo el paradigma de la POO, esta clase deberá contener los datos (la matriz) y los algoritmos (métodos) para manejarla. Deseamos utilizar los objetos de tipo `mVector` como auténticas matrices, por lo que deberíamos poder utilizarlos con álgebra de matrices C++.

Utilizando una analogía, si por ejemplo **m** es una **matriz de enteros**, sabemos que el lenguaje nos permite utilizar las expresiones siguientes:

```
m[i];           // Acceso a elemento con el operador subíndice  
int x = m[i];   // Asignación a un miembro de la clase int  
m[i] = m[j];    // Asignación a miembro  
m[i] = 3 * m[j]; // Producto por un escalar  
m[i] = m[j] * m[k]; // Producto entre miembros
```

Sobrecarga de operadores: operador subíndice []

- Preparamos el diseño de la clase `mVector` de forma que que pueda mimetizarse el comportamiento anterior con sus objetos. Es decir, deben permitirse las siguientes expresiones:

```
mVector m1;  
m1[i]; // acceso a elemento con el operador subíndice  
Vector v1 = m1[i]; // asignación a un miembro de la clase Vector  
m1[i] = m1[j]; // asignación a miembro  
m1[i] = 3 * m1[j]; // producto por un escalar  
m1[i] = m1[j] * m1[k]; // producto entre miembros
```


Sobrecarga de operadores: operador subíndice []

```
const size_t sV = sizeof(Vector);  
Vector& operator[](int i) {  
    return *( mVptr + (i * sV));  
}
```

El operador subíndice [], recibe un int y retorna la referencia.

```
Vector& operator[](int i) {  
    return *( mVptr + i );  
}
```

mVptr es un puntero a Vector, por lo que su algebra lleva implícito el tamaño de los objetos Vector(eliminamos a sV)

```
Vector& operator[](int i) {  
    return mVptr[i];  
}
```

La expresión anterior equivale a:

```
mVector mV1(5);  
mV1[2];
```

Como resultado, podemos utilizar expresiones del tipo:

Sobrecarga de operadores: operador de asignación

```
Vector& operator= (const Vector& v) {  
// función operator=  
  x = v.x; y = v.y;  
  return *this;  
}
```

Para utilizar el operador de asignación = con los objetos devueltos por el selector de miembro [], debemos sobrecargarlo para los objetos tipo Vector.

```
Vector v1;  
v1 = mV1[0];
```

Su implementación, nos permite utilizar expresiones del tipo

Sobrecarga de operadores: producto por un escalar

```
m[i] = m[j] * 3; // producto por un escalar (por la derecha)
m[i] = 3 * m[j]; // producto por un escalar (por la izquierda)
```

Esto significa que debemos definir el producto en ambos sentidos

```
Vector operator* (int i) {
    // producto por un escalar (por la derecha)
    Vector vr;
    vr.x = x * i;
    vr.y = y * i;
    return vr;
}
```

Para el primero podemos definir una función miembro que acepte un argumento tipo int (además del correspondiente puntero this). Este método sería:

```
mV1[4] = mV1[0] * 5;
```

Su implementación, nos permite utilizar expresiones del tipo

Sobrecarga de operadores: producto por un escalar

```
Vector operator* (int i, Vector v) {  
    Vector vr;  
    vr.x = v.x * i;  
    vr.y = v.y * i;  
    return vr;  
}
```

- El **producto** por la **izquierda** debemos definirlo como una **función-operador externa**. Se trata de una función independiente (no pertenece a una clase) que acepta dos argumentos, un int y un Vector.

- La declaramos friend de la clase Vector para que pueda tener acceso a sus miembros (aunque en este caso no es necesario porque todos son públicos). Su diseño es muy parecido al anterior, aunque en este caso no existe puntero implícito this y debemos referenciar el objeto Vector directamente:

```
mV1[2] = 5 * mV1[0];
```

Su implementación, nos permite utilizar expresiones del tipo

Sobrecarga de operadores: producto por un escalar

```
#include "Tipos.h"
using namespace std;
```

CVector.h

```
class Vector {
public:
    TipoEntero m_x, m_y;
    Vector& operator= (const Vector& v) { // asignación V = V
        m_x = v.m_x;    m_y = v.m_y;    return *this;
    }
    Vector operator* (TipoEntero i) {      // Producto V * int
        Vector vr;
        vr.m_x = m_x * i;
        vr.m_y = m_y * i;
        return vr;
    }
    void showV(ostream &os);
    friend Vector operator* (TipoEntero, Vector); // Producto int * V
};

void Vector::showV(ostream &os) {
    os << "X = " << m_x << "; Y = " << m_y << endl;
}
```

Sobrecarga de operadores: producto por un escalar

```
#include "Tipos.h"
using namespace std;
```

```
class Vector {
public:
    TipoEntero m_x, m_y;

    Vector() {}
    Vector(TipoEntero x, TipoEntero y) {
        m_x = x;
        m_y = y;
    }
    Vector& operator= (const Vector& v) { // asignación V = V
        m_x = v.m_x;    m_y = v.m_y;    return *this;
    }
    Vector operator* (TipoEntero i) {      // Producto V * int
        return Vector(m_x*i, m_y*i);
    }
    void showV(ostream &os);
    friend Vector operator* (TipoEntero, Vector); // Producto int * V
};

void Vector::showV(ostream &os) {
    os << "X = " << m_x << "; Y = " << m_y << endl;
}
```

CMatrizVector.h

Tipos.h

```
using TipoEntero = int;
```

Sobrecarga de operadores: producto por un escalar

main.cpp

```
#include <iostream>
#include "CVector.h"
#include "CMatrizVector.h"

int main() {
    mVector mV1(5);
    mV1[0].m_x = 2; mV1[0].m_y = 3;
    mV1.showmem(cout,0);

    Vector v1;
    v1 = mV1[0];
    v1.showV(cout);
    mV1[4] = mV1[0] * 5;
    mV1.showmem(cout, 4);

    mV1[2] = 5 * mV1[0];
    mV1.showmem(cout,2);

    return 0;
}
```

Resultado

X = 2; Y = 3

X = 2; Y = 3

X = 10; Y = 15

X = 10; Y = 15

Ejemplo 1: sobrecarga sort

```
#include "Alumno.h"
```

Alumno.h

```
class Alumno {  
public:  
    TipoEntero m_edad;  
    TipoCadena m_nombre;  
  
    Alumno(TipoEntero edad, TipoCadena nombre) :  
        m_edad(edad), m_nombre(nombre)  
    { }  
    bool operator <(const Alumno & alu) const {  
        return m_edad < alu.m_edad;  
    }  
};
```

```
#include "Alumno.h"
```

AlumnoComparador.h

```
class AlumnoComparator {  
public:  
    // Compara 2 alumnos usando el nombre  
    bool operator()(const Alumno & alu1, const Alumno & alu2)  
    {  
        if(alu1.m_nombre == alu2.m_nombre)  
            return alu1 < alu2;  
        return alu1.m_nombre < alu2.m_nombre;  
    }  
};
```

```
using TipoEntero = int;  
using TipoCadena = string;
```

Alumno.h

Ejemplo 1: sobrecarga sort

```
cout<<"****LISTA INICIAL ****"<<endl;
mostrar(cout,listaAlumnos);

cout<<"****ORDENADO POR EDAD ****"<<endl;
//Ordenando usando el operador <
listaAlumnos.sort();
mostrar(cout,listaAlumnos);

cout<<"****ORDENADO POR NOMBRE ****"<<endl;
// Ordenando usando comparador
listaAlumnos.sort(AlumnoComparator());
mostrar(cout, listaAlumnos);

return 0;
}
```

main.cpp

Resultado:

****LISTA INICIAL ****

22 :: Hugo

3 :: Paco

43 :: Luis

30 :: Rufo

12 :: Coyote

****ORDENADO POR EDAD ****

3 :: Paco

12 :: Coyote

22 :: Hugo

30 :: Rufo

43 :: Luis

****ORDENADO POR NOMBRE ****

12 :: Coyote

22 :: Hugo

43 :: Luis

3 :: Paco

30 :: Rufo

Ejemplo 1: sobrecarga sort

```
#include "Tipos.h"
using namespace std;
```

CPunto.h

```
class Punto{
public:
    TipoEntero x, y;
    Punto( int x = 0, int y = 0 ): x( x ), y( y ) { }
    bool sortfunc( const Punto& lhs, const Punto& rhs ) const
    {
        return (lhs.x == rhs.x)
            ? (lhs.y < rhs.y)
            : (lhs.x < rhs.x);
    }
};

// comparador para la función sort ()
bool operator < ( const Punto& lhs, const Punto& rhs ) {
    return lhs.sortfunc( lhs, rhs );
}

// mostrar los puntos
ostream& operator << ( ostream& os, const Punto& punto ) {
    return os << "(" << punto.x << "," << punto.y << ")";
}
```

Ejemplo 1: sobrecarga sort

CPunto.h

```
// ...Lista de puntos
ostream& operator << ( ostream& os, const vector <Punto> & puntos ) {
    for (unsigned n = 0;;)
    {
        os << puntos[ n ];
        if (++n >= puntos.size()) break;
        os << ", ";
    }
    return os;
}
```

Ejemplo 1: sobrecarga sort

main.cpp

```
#include <algorithm>
#include <iostream>
#include <vector>
#include "CPunto.h"
using namespace std;

int main()
{
    vector <Punto> v;

    v.push_back( Punto( 10, -7 ) );
    v.push_back( Punto( -2, 14 ) );
    v.push_back( Punto( 3, 5 ) );
    v.push_back( Punto( 3, 0 ) );
    v.push_back( Punto( 3, 78 ) );
    v.push_back( Punto( 24, 42 ) );

    cout << "v = " << v << endl;

    cout << "ordenando v..." << flush;
    sort( v.begin(), v.end() );
    cout << "Listo.\n";

    cout << "v = " << v << endl;

    return 0;
}
```

Tipos.h

```
using TipoEntero = int; //desde c++ 11
```

Resultado

v = (10,-7), (-2,14), (3,5), (3,0), (3,78), (24,42)
ordenando v...Listo.
v = (-2,14), (3,0), (3,5), (3,78), (10,-7), (24,42)

Resumen:

- Funciones amigas
- Sobrecarga de operadores

¡Nos vemos en la
siguiente clase!

