

Informática II

Sobrecarga de operadores en C++

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2024 –

Introducción

Sobrecarga de operadores

Permite que los operadores de C++ pueda trabajar con objetos de clases

Introducción

Sobrecarga de operadores

Permite que los operadores de C++ pueda trabajar con objetos de clases

Por ejemplo:

- ▶ El operador << se utiliza:
 1. como operador a nivel bits de desplazamiento a la izquierda
 2. como operador de inserción de flujo

Introducción

Sobrecarga de operadores

Permite que los operadores de C++ pueda trabajar con objetos de clases

Por ejemplo:

- ▶ El operador << se utiliza:
 1. como operador a nivel bits de desplazamiento a la izquierda
 2. como operador de inserción de flujo
- ▶ El operador >> se utiliza:
 1. como operador a nivel de bits de desplazamiento a la derecha
 2. como operador de extracción de flujo

Introducción

Sobrecarga de operadores

Permite que los operadores de C++ pueda trabajar con objetos de clases

Por ejemplo:

- ▶ El operador << se utiliza:
 1. como operador a nivel bits de desplazamiento a la izquierda
 2. como operador de inserción de flujo
- ▶ El operador >> se utiliza:
 1. como operador a nivel de bits de desplazamiento a la derecha
 2. como operador de extracción de flujo

Esta es una de las características más poderosas de C++

Introducción

- ▶ C++ no permite la creación de nuevos operadores (p.e. `**`).

Introducción

- ▶ C++ no permite la creación de nuevos operadores (p.e. `**`).
- ▶ La sobrecarga de operadores existe en los tipos básicos: `int`, `float`, `double`, etc.

Introducción

- ▶ C++ no permite la creación de nuevos operadores (p.e. **).
- ▶ La sobrecarga de operadores existe en los tipos básicos: `int`, `float`, `double`, etc.

```
double pi = 3.14159;  
cout << pi;  
int num = 10;  
cout << num;
```


Introducción

- ▶ C++ no permite la creación de nuevos operadores (p.e. **).
- ▶ La sobrecarga de operadores existe en los tipos básicos: `int`, `float`, `double`, etc.

```
double pi = 3.14159;  
cout << pi;  
int num = 10;  
cout << num;
```

- ▶ La mayoría de los operadores existentes se pueden sobrecargar para que trabajen con objetos de clases definidos por el usuario:

Introducción

- ▶ C++ no permite la creación de nuevos operadores (p.e. **).
- ▶ La sobrecarga de operadores existe en los tipos básicos: `int`, `float`, `double`, etc.

```
double pi = 3.14159;  
cout << pi;  
int num = 10;  
cout << num;
```

- ▶ La mayoría de los operadores existentes se pueden sobrecargar para que trabajen con objetos de clases definidos por el usuario:

```
Complejo c(2,3);  
cout << c;
```

Introducción

- ▶ C++ no permite la creación de nuevos operadores (p.e. **).
- ▶ La sobrecarga de operadores existe en los tipos básicos: `int`, `float`, `double`, etc.

```
double pi = 3.14159;  
cout << pi;  
int num = 10;  
cout << num;
```

- ▶ La mayoría de los operadores existentes se pueden sobrecargar para que trabajen con objetos de clases definidos por el usuario:

```
Complejo c(2,3);  
cout << c;
```

- ▶ Al menos uno de los operandos debe ser un objeto de la clase sobre la que se quiere sobrecargar el operador.

Introducción

- El operador de asignación (=) puede utilizarse sobre objetos de clases sin sobrecarga explícita. De forma predeterminada se realiza una *asignación de miembros* de datos miembros de la clase. Esto resulta peligroso cuando los datos miembros son punteros (en este caso se debe realizar una sobrecarga explícita)

```
Complejo c1(2,3), c2(3,2);  
Complejo c3 = c1.suma(c2);
```

Introducción

- El operador de asignación (=) puede utilizarse sobre objetos de clases sin sobrecarga explícita. De forma predeterminada se realiza una *asignación de miembros* de datos miembros de la clase. Esto resulta peligroso cuando los datos miembros son punteros (en este caso se debe realizar una sobrecarga explícita)

```
Complejo c1(2,3), c2(3,2);  
Complejo c3 = c1.suma(c2);
```

- El operador de dirección (&) también puede usarse con cualquier objeto de clase sin sobrecarga explícita. Devuelve la dirección del objeto en memoria. También puede sobrecargarse de forma explícita

```
Complejo c(2,3);  
Complejo *complejoPtr = &c;
```

Sobrecarga de operadores

La sobrecarga de operadores se realiza mediante la implementación de funciones –*función operador*– con la particularidad de que el nombre de la función se compone de la palabra reservada **operator** seguida por el símbolo del operador a sobrecargar.

Por ejemplo, la función operador **operator+** se utiliza para sobrecargar el operador de suma (+).

Sobrecarga de operadores

La sobrecarga de operadores se realiza mediante la implementación de funciones –*función operador*– con la particularidad de que el nombre de la función se compone de la palabra reservada **operator** seguida por el símbolo del operador a sobrecargar.

Por ejemplo, la función operador **operator+** se utiliza para sobrecargar el operador de suma (+).

¿Cómo funciona?

Sobrecarga de operadores

La sobrecarga de operadores se realiza mediante la implementación de funciones –*función operador*– con la particularidad de que el nombre de la función se compone de la palabra reservada `operator` seguida por el símbolo del operador a sobrecargar.

Por ejemplo, la función operador `operator+` se utiliza para sobrecargar el operador de suma (+).

¿Cómo funciona?

- En la clase `Complejo` se declaran los prototipos:

```
Complejo operator+(Complejo c); // Complejo suma(Complejo c);  
Complejo operator-(Complejo c); // Complejo resta(Complejo c);
```


Sobrecarga de operadores

La sobrecarga de operadores se realiza mediante la implementación de funciones –*función operador*– con la particularidad de que el nombre de la función se compone de la palabra reservada **operator** seguida por el símbolo del operador a sobrecargar.

Por ejemplo, la función operador **operator+** se utiliza para sobrecargar el operador de suma (+).

¿Cómo funciona?

- En la clase **Complejo** se declaran los prototipos:

```
Complejo operator+(Complejo c); // Complejo suma(Complejo c);  
Complejo operator-(Complejo c); // Complejo resta(Complejo c);
```

- Cuando el compilador encuentra:

```
Complejo c1(2,3), c2(3,2);  
Complejo c3 = c1 + c2; // <-- Sobrecarga
```

Sobrecarga de operadores

La sobrecarga de operadores se realiza mediante la implementación de funciones *función operador* con la particularidad de que el nombre de la función se compone de la palabra reservada `operator` seguida por el símbolo del operador a sobrecargar.

Por ejemplo, la función operador `operator+` se utiliza para sobrecargar el operador de suma (+).

¿Cómo funciona?

- ▶ En la clase `Complejo` se declaran los prototipos:

```
Complejo operator+(Complejo c); // Complejo suma(Complejo c);  
Complejo operator-(Complejo c); // Complejo resta(Complejo c);
```

- ▶ Cuando el compilador encuentra:

```
Complejo c1(2,3), c2(3,2);  
Complejo c3 = c1 + c2; // <-- Sobrecarga
```

- ▶ Se genera una llamada a la función operador, o sea:

```
c3 = c1.operator+(c2);
```

Sobrecarga de operadores

```
Complejo Complejo::suma(Complejo op2) // c3 = c1.suma(c2)
{
    double real = parteReal + op2.parteReal;
    double imag = parteImaginaria + op2.parteImaginaria;

    Complejo resultado(real, imag);
    return resultado;
}
```

Sobrecarga de operadores

```
Complejo Complejo::suma(Complejo op2) // c3 = c1.suma(c2)
{
    double real = parteReal + op2.parteReal;
    double imag = parteImaginaria + op2.parteImaginaria;

    Complejo resultado(real, imag);
    return resultado;
}
```

```
Complejo Complejo::operator+(Complejo op2) // c3 = c1 + c2;
{
    double real = parteReal + op2.parteReal;
    double imag = parteImaginaria + op2.parteImaginaria;

    Complejo resultado(real, imag);
    return resultado;
}
```

Sobrecarga de operadores

Sobrecargar un operador de asignación y suma para permitir sentencias como:

```
objeto2 = objeto2 + objeto1; // p.e.: c2 = c2 + c1;
```

no implica que el operador += también resulte sobrecargado para poder hacer

```
objeto2 += objeto1; // p.e.: c2 += c1;
```

Esto se logra únicamente sobrecargando de forma explícita el operador += para la clase dada.

Sobrecarga de operadores

Sobrecargar un operador de asignación y suma para permitir sentencias como:

```
objeto2 = objeto2 + objeto1; // p.e.: c2 = c2 + c1;
```

no implica que el operador += también resulte sobrecargado para poder hacer

```
objeto2 += objeto1; // p.e.: c2 += c1;
```

Esto se logra únicamente sobrecargando de forma explícita el operador += para la clase dada.

Es recomendable sobrecargar operadores relacionados utilizando otros operadores previamente sobrecargados. Por ejemplo:

- ▶ implementar la sobrecarga del operador += a partir del operador sobrecargado +
- ▶ implementar la sobrecarga del operador != a partir del operador sobrecargado ==

Sobrecarga de operadores

Operadores que pueden sobrecargarse

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

Sobrecarga de operadores

Operadores que pueden sobrecargarse

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

Operadores que no puede sobrecargarse

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

Sobrecarga de operadores

Operadores que pueden sobrecargarse

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

Operadores que no puede sobrecargarse

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

La sobrecarga de operadores no permite modificar

- ▶ la precedencia y asociatividad de operadores
- ▶ la cantidad de operandos de un operador
- ▶ la manera que el operador funciona con objetos de tipos predefinidos

Sobrecarga de operadores

Implementación de funciones operador

Las funciones operador pueden ser

1. Funciones miembros, o
2. funciones no miembros (amigas de la clase por cuestiones de rendimiento).

Sobrecarga de operadores

Implementación de funciones operador

Las funciones operador pueden ser

1. Funciones miembros, o
2. funciones no miembros (amigas de la clase por cuestiones de rendimiento).

- Si el operando más a la izquierda (o el único operando) debe ser un objeto de la clase (o una referencia a un objeto de la clase) → función miembro

Sobrecarga de operadores

Implementación de funciones operador

Las funciones operador pueden ser

1. Funciones miembros, o
2. funciones no miembros (amigas de la clase por cuestiones de rendimiento).

- ▶ Si el operando más a la izquierda (o el único operando) debe ser un objeto de la clase (o una referencia a un objeto de la clase) → función miembro
- ▶ Si el operando izquierdo tiene que ser un objeto de una clase diferente o un tipo predefinido → función no miembro

Sobrecarga de operadores

Implementación de funciones operador

Las funciones operador pueden ser

1. Funciones miembros, o
2. funciones no miembros (amigas de la clase por cuestiones de rendimiento).

- ▶ Si el operando más a la izquierda (o el único operando) debe ser un objeto de la clase (o una referencia a un objeto de la clase) → función miembro
- ▶ Si el operando izquierdo tiene que ser un objeto de una clase diferente o un tipo predefinido → función no miembro

Si la función operador debe tener acceso a los miembros privados (o protegidos) de la clase, entonces debe declararse como amiga de dicha clase.

Sobrecarga de operadores binarios

Se puede sobrecargar un operador binario para una clase como:

1. una función miembro con un único argumento, o
2. una función no miembro con dos argumentos (uno de los cuales debe ser un objeto o referencia de la clase).

Sobrecarga de operadores binarios

Se puede sobrecargar un operador binario para una clase como:

1. una función miembro con un único argumento, o
2. una función no miembro con dos argumentos (uno de los cuales debe ser un objeto o referencia de la clase).

Ejemplo: El operador binario `+=` de una clase **Cadena** para concatenar cadenas

Sobrecarga de operadores binarios

Se puede sobrecargar un operador binario para una clase como:

1. una función miembro con un único argumento, o
2. una función no miembro con dos argumentos (uno de los cuales debe ser un objeto o referencia de la clase).

Ejemplo: El operador binario `+=` de una clase `Cadena` para concatenar cadenas

```
class Cadena {  
    public:  
        const Cadena &operator+=(const Cadena & );  
        . . .  
};
```

Sobrecarga de operadores binarios

Se puede sobrecargar un operador binario para una clase como:

1. una función miembro con un único argumento, o
2. una función no miembro con dos argumentos (uno de los cuales debe ser un objeto o referencia de la clase).

Ejemplo: El operador binario `+=` de una clase `Cadena` para concatenar cadenas

```
class Cadena {  
    public:  
        const Cadena &operator+=(const Cadena & );  
        . . .  
};
```

```
class Cadena {  
    friend const Cadena &operator+=(Cadena & , const Cadena & );  
    . . .  
};
```

Sobrecarga de operadores binarios

```
// Caso 1: Operador como función miembro
const Cadena &operator+=(const Cadena & );

// Caso 2: Operador como función NO miembro
friend const Cadena &operator+=(Cadena & , const Cadena & );
```

Sobrecarga de operadores binarios

```
// Caso 1: Operador como función miembro
const Cadena &operator+=(const Cadena & );

// Caso 2: Operador como función NO miembro
friend const Cadena &operator+=(Cadena & , const Cadena & );
```

1. Como función miembro

```
Cadena y, z;
y += z;
```

el operador += se traduce en una llamada a la función miembro `operator+=`, o sea

```
y.operator+=(z);
```

Sobrecarga de operadores binarios

```
// Caso 1: Operador como función miembro
const Cadena &operator+=(const Cadena & );

// Caso 2: Operador como función NO miembro
friend const Cadena &operator+=(Cadena & , const Cadena & );
```

1. Como función miembro

```
Cadena y, z;
y += z;
```

el operador += se traduce en una llamada a la función miembro `operator+=`, o sea

```
y.operator+=(z);
```

2. Como función NO miembro

```
Cadena y, z;
y += z;
```

el operador += se traduce en una llamada a la función miembro `operator+=`, o sea

```
operator+=(y, z);
```


Sobrecarga de operadores unarios

Se puede sobrecargar un operador unario para una clase como:

1. una función miembro sin argumentos
2. una función no miembro con un argumento (objeto o referencia a objeto de la clase).

Sobrecarga de operadores unarios

Se puede sobrecargar un operador unario para una clase como:

1. una función miembro sin argumentos
2. una función no miembro con un argumento (objeto o referencia a objeto de la clase).

Ejemplo: El operador unario ! de una clase **Cadena** que evalúa si la cadena está vacía

Sobrecarga de operadores unarios

Se puede sobrecargar un operador unario para una clase como:

1. una función miembro sin argumentos
2. una función no miembro con un argumento (objeto o referencia a objeto de la clase).

Ejemplo: El operador unario ! de una clase **Cadena** que evalúa si la cadena está vacía

```
class Cadena {  
    public:  
        bool operator!() const;  
        . . .  
};
```

Sobrecarga de operadores unarios

Se puede sobrecargar un operador unario para una clase como:

1. una función miembro sin argumentos
2. una función no miembro con un argumento (objeto o referencia a objeto de la clase).

Ejemplo: El operador unario `!` de una clase `Cadena` que evalúa si la cadena está vacía

```
class Cadena {  
    public:  
        bool operator!() const;  
        . . .  
};
```

```
class Cadena {  
    friend bool operator!(const Cadena & );  
    . . .  
};
```

Sobrecarga de operadores unarios

Sobrecarga del operador unario ! como función miembro sin argumentos

```
class Cadena {  
    public:  
        bool operator!() const;  
        . . .  
};
```

Sobrecarga de operadores unarios

Sobrecarga del operador unario `!` como función miembro sin argumentos

```
class Cadena {  
    public:  
        bool operator!() const;  
        . . .  
};
```

- ▶ Si `s` es un objeto o una referencia de la clase `Cadena`, cuando el compilador encuentra la expresión `!s`, se genera una llamada a
`s.operator!();`
- ▶ El operando `s` es un objeto de la clase sobre el cual se invoca a la función miembro operador (`operator!`) de la clase `Cadena`

Sobrecarga de operadores unarios

Sobrecarga del operador unario ! como función amiga no miembro con un argumento

Sobrecarga de operadores unarios

Sobrecarga del operador unario ! como función amiga no miembro con un argumento

Alternativas:

1. Con un argumento objeto de la clase: se realiza una copia del objeto y la función no puede modificar el objeto original
2. Con un argumento de referencia al objeto de la clase: no se realiza una copia y el objeto puede ser modificado en la función operador

```
class Cadena {  
    friend bool operator!(const Cadena & );  
    . . .  
};
```

Sobrecarga de operadores unarios

Sobrecarga del operador unario ! como función amiga no miembro con un argumento

Alternativas:

1. Con un argumento objeto de la clase: se realiza una copia del objeto y la función no puede modificar el objeto original
2. Con un argumento de referencia al objeto de la clase: no se realiza una copia y el objeto puede ser modificado en la función operador

```
class Cadena {  
    friend bool operator!(const Cadena & );  
    . . .  
};
```

- Si **s** es un objeto o una referencia de la clase **Cadena**, cuando el compilador encuentra la expresión **!s**, se genera una llamada a

`operator!(s);`

Ejemplo: clase Arreglo (básica)

```
1  class Arreglo {
2      public:
3          Arreglo(int = 10); // constructor predeterminado
4          ~Arreglo(); // destructor
5
6          int obtenerTamano() const;
7          void imprimir() const;
8
9
10
11
12
13
14
15
16
17
18     private:
19         int tamano; // Tamaño del arreglo
20         int *ptr; // Puntero al primer elemento del arreglo
21 };
```

Ejemplo: clase Arreglo (básica)

```
1 class Arreglo {
2     public:
3         Arreglo(int = 10); // constructor predeterminado
4         ~Arreglo(); // destructor
5
6         int obtenerTamano() const;
7         void imprimir() const;
8
9         // Sobrecarga de operadores
10        const Arreglo &operator=(const Arreglo & ); // asigna arreglos
11        bool operator==(const Arreglo & ) const; // compara igualdad
12
13
14
15
16
17
18    private:
19        int tamano; // Tamaño del arreglo
20        int *ptr; // Puntero al primer elemento del arreglo
21};
```

Ejemplo: clase Arreglo (básica)

```
1 class Arreglo {
2     public:
3         Arreglo(int = 10); // constructor predeterminado
4         ~Arreglo(); // destructor
5
6         int obtenerTamano() const;
7         void imprimir() const;
8
9         // Sobrecarga de operadores
10        const Arreglo &operator=(const Arreglo & ); // asigna arreglos
11        bool operator==(const Arreglo & ) const; // compara igualdad
12
13        // Determina si dos arreglos no son iguales y devuelve true,
14        // de lo contrario devuelve false (utiliza operator==).
15        bool operator!=(const Arreglo &derecha) const
16            { return ! ( *this == derecha) ; }
17
18    private:
19        int tamano; // Tamaño del arreglo
20        int *ptr; // Puntero al primer elemento del arreglo
21};
```

Sobrecarga de los operadores << y >>

- Los operandos de inserción y extracción de flujo están sobrecargados en la biblioteca de entrada/salida para poder operar con cada tipo de dato predefinido, incluyendo cadenas, punteros, etc.

```
double pi = 3.14159;  
cout << pi;  
int num = 10;  
cout << num;
```

Sobrecarga de los operadores << y >>

- Los operandos de inserción y extracción de flujo están sobrecargados en la biblioteca de entrada/salida para poder operar con cada tipo de dato predefinido, incluyendo cadenas, punteros, etc.

```
double pi = 3.14159;  
cout << pi;  
int num = 10;  
cout << num;
```

- Estos operadores se pueden sobrecargar para operar con tipos de datos definidos por el usuario

```
Complejo c(2,3);  
cout << c;
```

Sobrecarga de los operadores << y >>

```
class Complejo {  
  
    public:  
        Complejo(double = 0.0, double = 0.0);  
  
    private:  
        double parteReal;  
        double parteImaginaria;  
};
```

Sobrecarga de los operadores << y >>

```
class Complejo {  
    friend istream &operator>>(istream & , Complejo & );  
    friend ostream &operator<<(ostream & , const Complejo & );  
  
    public:  
        Complejo(double = 0.0, double = 0.0);  
  
    private:  
        double parteReal;  
        double parteImaginaria;  
};
```

Sobrecarga de los operadores << y >>

```
class Complejo {  
    friend ostream &operator>>(ostream & , Complejo & );  
    friend ostream &operator<<(ostream & , const Complejo & );  
  
    public:  
        Complejo(double = 0.0, double = 0.0);  
  
    private:  
        double parteReal;  
        double parteImaginaria;  
};
```

- ▶ Función operador: `operator>>` y `operator<<`
- ▶ Operadores implementados como función amiga (no miembro)
- ▶ `cin` es de la clase `istream` y `cout` es de la clase `ostream`
- ▶ Valor de retorno: `istream &` y `ostream &` (referencias)

Sobrecarga de los operadores << y >>

Implementación de la función operador de extracción de flujo

```
istream &operator>>(istream &entrada, Complejo &num)
{
    // Implementación
}
```

Sobrecarga de los operadores << y >>

Implementación de la función operador de extracción de flujo

```
istream &operator>>(istream &entrada, Complejo &num)
{
    // Implementación
}
```

La función operador de extracción de flujo `operator>>` tiene como argumento:

1. una referencia a `istream` llamada `entrada`,

Sobrecarga de los operadores << y >>

Implementación de la función operador de extracción de flujo

```
istream &operator>>(istream &entrada, Complejo &num)
{
    // Implementación
}
```

La función operador de extracción de flujo `operator>>` tiene como argumento:

1. una referencia a `istream` llamada `entrada`,
2. una referencia a `Complejo` llamada `num`, y

Sobrecarga de los operadores << y >>

Implementación de la función operador de extracción de flujo

```
istream &operator>>(istream &entrada, Complejo &num)
{
    // Implementación
    return entrada; // habilita cin >> c1 >> c2;
}
```

La función operador de extracción de flujo `operator>>` tiene como argumento:

1. una referencia a `istream` llamada `entrada`,
2. una referencia a `Complejo` llamada `num`, y
3. devuelve una referencia a `istream` para permitir la llamada en cascada

`cin >> c1 >> c2;`

Sobrecarga de los operadores << y >>

Implementación de la función operador de extracción de flujo

```
istream &operator>>(istream &entrada, Complejo &num)
{
    // Implementación
    return entrada; // habilita cin >> c1 >> c2;
}
```

- Definido c tipo Complejo, cuando el compilador encuentra la sentencia

cin >> c;

se genera una llamada a la función operador

operator>>(cin, c);

Sobrecarga de los operadores << y >>

Implementación de la función operador de extracción de flujo

```
istream &operator>>(istream &entrada, Complejo &num)
{
    // Implementación
    return entrada; // habilita cin >> c1 >> c2;
}
```

- Definido c tipo Complejo, cuando el compilador encuentra la sentencia

cin >> c;

se genera una llamada a la función operador

operator>>(cin, c);

- En esta llamada a la función operador:
 1. el parámetro referencia **entrada** se vuelve un alias de **cin** y
 2. el parámetro referencia **num** se vuelve un alias de **c**
- Devuelve una referencia de **istream** (**entrada**), o sea, **cin**.

Sobrecarga de los operadores << y >>

Implementación de la función operador de extracción de flujo

```
istream &operator>>(istream &entrada, Complejo &num)
{
    // Implementación
    return entrada; // habilita cin >> c1 >> c2;
}
```

- Definido c tipo Complejo, cuando el compilador encuentra la sentencia

cin >> c;

se genera una llamada a la función operador

operator>>(cin, c);

- En esta llamada a la función operador:
 1. el parámetro referencia **entrada** se vuelve un alias de **cin** y
 2. el parámetro referencia **num** se vuelve un alias de **c**
- Devuelve una referencia de **istream** (**entrada**), o sea, **cin**.

(el primer operando NO ES un objeto de la clase Complejo)

Sobrecarga de los operadores << y >>

Implementación de la función operador de inserción de flujo

```
1 ostream &operator<<(ostream &salida, const Complejo &num)
2 {
3     // Implementación
4     return salida; // habilita cout << c1 << c2;
5 }
```

Sobrecarga de los operadores << y >>

Implementación de la función operador de inserción de flujo

```
1 ostream &operator<<(ostream &salida, const Complejo &num)
2 {
3     // Implementación
4     return salida; // habilita cout << c1 << c2;
5 }
```

La función operador de inserción de flujo `operator<<` tiene como argumento:

1. una referencia a `ostream` llamada `salida`,
2. una referencia a `Complejo` llamada `num`, y
3. devuelve una referencia a `ostream`

Sobrecarga de los operadores << y >>

Implementación de la función operador de inserción de flujo

```
1 ostream &operator<<(ostream &salida, const Complejo &num)
2 {
3     // Implementación
4     return salida; // habilita cout << c1 << c2;
5 }
```

La función operador de inserción de flujo `operator<<` tiene como argumento:

1. una referencia a `ostream` llamada `salida`,
2. una referencia a `Complejo` llamada `num`, y
3. devuelve una referencia a `ostream`
4. Cuando el compilador encuentra la sentencia

`cout << c;`

se genera una llamada a la función operador

`operator<<(cout, c);`

Sobrecarga de los operadores << y >>

- ▶ Las funciones operador `operator>>` y `operator<<` se declaran en la clase `Complejo` como funciones amigas no miembros de la clase.

Sobrecarga de los operadores << y >>

- ▶ Las funciones operador `operator>>` y `operator<<` se declaran en la clase `Complejo` como funciones amigas no miembros de la clase.
- ▶ Esto es así ya que el objeto de la clase `Complejo` se utiliza en ambos casos como operando derecho. Para sobrecargar el operador como función miembro de la clase, el operando de la clase debe estar como operando izquierdo.

Sobrecarga de los operadores << y >>

- ▶ Las funciones operador `operator>>` y `operator<<` se declaran en la clase `Complejo` como funciones amigas no miembros de la clase.
- ▶ Esto es así ya que el objeto de la clase `Complejo` se utiliza en ambos casos como operando derecho. Para sobrecargar el operador como función miembro de la clase, el operando de la clase debe estar como operando izquierdo.
- ▶ Por cuestiones de rendimiento, conviene declarar las funciones operador no miembro como funciones amigas dado que tienen que acceder a los datos miembros de la clase, de otra forma, estas debería llamar a funciones `get`.

Ejemplo: clase **Arreglo**

Implementación de clase **Arreglo** como alternativa a los punteros del lenguaje C:

- ▶ No se puede imprimir un arreglo que no sea de **char** de forma directa (necesita un bucle).
- ▶ Tampoco se pueden cargar valores a un arreglo de forma directa (**cin**).
- ▶ No se pueden comparar dos arreglos con los operadores de igualdad o relacionales.
- ▶ Cuando se pasa a una función de propósito general se debe pasar el tamaño del arreglo.
- ▶ No se puede asignar un arreglo a otro arreglo (en C son punteros **const**).
- ▶ Etc. etc.

Ver definición de clase y luego ejemplo de código fuente **fig18_04.cpp** (D&D)

Ejemplo: clase Arreglo

```
1 class Arreglo {
2     friend ostream &operator<<(ostream & , const Arreglo & );
3     friend istream &operator>>(istream & , Arreglo & );
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23 };
```

Ejemplo: clase Arreglo

```
1 class Arreglo {
2     friend ostream &operator<<(ostream & , const Arreglo & );
3     friend istream &operator>>(istream & , Arreglo & );
4
5     public:
6         Arreglo(int = 10); // constructor predeterminado
7
8         ~Arreglo(); // destructor
9         int obtenerTamano() const; // valor de retorno
10
11
12
13
14
15
16
17
18
19
20
21
22
23 };
```

Ejemplo: clase Arreglo

```
1 class Arreglo {
2     friend ostream &operator<<(ostream & , const Arreglo & );
3     friend istream &operator>>(istream & , Arreglo & );
4
5     public:
6         Arreglo(int = 10); // constructor predeterminado
7         Arreglo(const Arreglo & ); // constructor de copia
8         ~Arreglo(); // destructor
9         int obtenerTamano() const; // valor de retorno
10
11
12
13
14
15
16
17
18
19
20
21
22     private: // falta completar
23 };
```

Ejemplo: clase Arreglo

```
1 class Arreglo {
2     friend ostream &operator<<(ostream & , const Arreglo & );
3     friend istream &operator>>(istream & , Arreglo & );
4
5     public:
6         Arreglo(int = 10); // constructor predeterminado
7         Arreglo(const Arreglo & ); // constructor de copia
8         ~Arreglo(); // destructor
9         int obtenerTamano() const; // valor de retorno
10
11         const Arreglo &operator=(const Arreglo & ); // asigna los arreglos
12         bool operator==(const Arreglo & ) const; // compara la igualdad
13
14
15
16
17
18
19
20
21
22     private: // falta completar
23 };
```

Ejemplo: clase Arreglo

```
1 class Arreglo {
2     friend ostream &operator<<(ostream & , const Arreglo & );
3     friend istream &operator>>(istream & , Arreglo & );
4
5     public:
6         Arreglo(int = 10); // constructor predeterminado
7         Arreglo(const Arreglo & ); // constructor de copia
8         ~Arreglo(); // destructor
9         int obtenerTamano() const; // valor de retorno
10
11         const Arreglo &operator=(const Arreglo & ); // asigna los arreglos
12         bool operator==(const Arreglo & ) const; // compara la igualdad
13
14         // Determina si dos arreglos no son iguales y
15         // devuelve true, de lo contrario devuelve false (utiliza operator==).
16         bool operator!=(const Arreglo &derecha) const
17             { return ! ( *this == derecha ); }
18
19
20
21
22     private: // falta completar
23 };
```

Ejemplo: clase Arreglo

```
1 class Arreglo {
2     friend ostream &operator<<(ostream & , const Arreglo & );
3     friend istream &operator>>(istream & , Arreglo & );
4
5     public:
6         Arreglo(int = 10); // constructor predeterminado
7         Arreglo(const Arreglo & ); // constructor de copia
8         ~Arreglo(); // destructor
9         int obtenerTamano() const; // valor de retorno
10
11         const Arreglo &operator=(const Arreglo & ); // asigna los arreglos
12         bool operator==(const Arreglo & ) const; // compara la igualdad
13
14         // Determina si dos arreglos no son iguales y
15         // devuelve true, de lo contrario devuelve false (utiliza operator==).
16         bool operator!=(const Arreglo &derecha) const
17             { return ! ( *this == derecha ); }
18
19         int &operator[](int ); // operador de subíndice
20         const int &operator[](int ) const; // operador de subíndice
21
22     private: // falta completar
23 };
```

