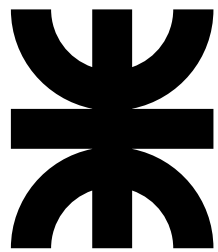
Informática II

Sobrecarga de operadores en C++

Gonzalo F. Pérez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2018 –

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

Ejemplos

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

Introducción

Sobrecarga de operadores

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

Ejemplos

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

Introducción

Sobrecarga de operadores

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

Ejemplos

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

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;
std::cout << pi;
int num = 10;
std::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;
std::cout << pi;
int num = 10;
std::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;
std::cout << pi;
int num = 10;
std::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);
std::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;
std::cout << pi;
int num = 10;
std::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);
std::cout << c;
```

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

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;
std::cout << pi;
int num = 10;
std::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);
std::cout << c;
```

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

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;
std::cout << pi;
int num = 10;
std::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);
std::cout << c;
```

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

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 (+)

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

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

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 (+)

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

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

```
Complejo Complejo::operator+(Complejo op2)
{
  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
```

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

```
objeto2 += objeto1
```

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
```

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

```
objeto2 += objeto1
```

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 puede sobrecargarse

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

Sobrecarga de operadores

Operadores que puede sobrecargarse

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

Operadores que no puede sobrecargarse

.	.*	::	?:	sizeof

Sobrecarga de operadores

Operadores que puede 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)

- Cuando se implementa la función operador como una función miembro, el operando más a la izquierda (o el única operando) debe ser un objeto de la clase (o una referencia a un objeto de la clase)

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)

- Cuando se implementa la función operador como una función miembro, el operando más a la izquierda (o el única operando) debe ser un objeto de la clase (o una referencia a un objeto de la clase)
- Si el operando izquierdo tiene que ser un objeto de una clase diferente o un tipo predefinido, la función operador debe implementarse como una 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)

- Cuando se implementa la función operador como una función miembro, el operando más a la izquierda (o el única operando) debe ser un objeto de la clase (o una referencia a un objeto de la clase)
- Si el operando izquierdo tiene que ser un objeto de una clase diferente o un tipo predefinido, la función operador debe implementarse como una 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 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.

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

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

Código fuente `fig18_03.c` (D&D 4° ed)

```
class NumeroTelefonico { // (800) 555-1212



  private:
    char codigoArea[4]; // código de área de tres dígitos y null
    char intercambio[5]; // código de área de intercambio y null
    char linea[5]; // 4 dígitos para la línea y null
};
```

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

Código fuente `fig18_03.c` (D&D 4° ed)

```
class NumeroTelefonico { // (800) 555-1212
  friend ostream &operator<<(ostream & , const NumeroTelefonico & );


  private:
    char codigoArea[4]; // código de área de tres dígitos y null
    char intercambio[5]; // código de área de intercambio y null
    char linea[5]; // 4 dígitos para la línea y null
};
```

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

Código fuente `fig18_03.c` (D&D 4° ed)

```
class NumeroTelefonico { // (800) 555-1212
  friend ostream &operator<<(ostream & , const NumeroTelefonico & );
  friend istream &operator>>(istream & , NumeroTelefonico & );

  private:
    char codigoArea[4]; // código de área de tres dígitos y null
    char intercambio[5]; // código de área de intercambio y null
    char linea[5]; // 4 dígitos para la línea y null
};
```

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

Código fuente `fig18_03.c` (D&D 4° ed)

```
class NumeroTelefonico { // (800) 555-1212
  friend ostream &operator<<(ostream & , const NumeroTelefonico & );
  friend istream &operator>>(istream & , NumeroTelefonico & );

  private:
    char codigoArea[4]; // código de área de tres dígitos y null
    char intercambio[5]; // código de área de intercambio y null
    char linea[5]; // 4 dígitos para la línea y null
};
```

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

Sobrecarga de los operadores << y >>

Implementación de la función operador de extracción de flujo

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


}
```

Sobrecarga de los operadores << y >>

Implementación de la función operador de extracción de flujo

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


}
```

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

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

Sobrecarga de los operadores << y >>

Implementación de la función operador de extracción de flujo

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

  return entrada; // habilita cin >> a >> b >> c;
}
```

La función operador de extracción de flujo **operator>>** tiene como argumento

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

```
cin >> telefono1 >> telefono2;
```

Sobrecarga de los operadores << y >>

Implementación de la función operador de extracción de flujo

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

  return entrada; // habilita cin >> a >> b >> c;
}
```

Sobrecarga de los operadores << y >>

Implementación de la función operador de extracción de flujo

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

  return entrada; // habilita cin >> a >> b >> c;
}
```

- Suponiendo un objeto de la clase `NumeroTelefonico` llamado `telefono`

Sobrecarga de los operadores << y >>

Implementación de la función operador de extracción de flujo

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

  return entrada; // habilita cin >> a >> b >> c;
}
```

- Suponiendo un objeto de la clase `NumeroTelefonico` llamado `telefono`
- Cuando el compilador encuentra la sentencia

```
cin >> telefono;
```

Sobrecarga de los operadores << y >>

Implementación de la función operador de extracción de flujo

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

  return entrada; // habilita cin >> a >> b >> c;
}
```

- Suponiendo un objeto de la clase `NumeroTelefonico` llamado `telefono`
- Cuando el compilador encuentra la sentencia

  ```
  cin >> telefono;
  ```

  se genera una llamada a la función operador

  ```
  operator>>(cin, telefono);
  ```

Sobrecarga de los operadores << y >>

Implementación de la función operador de extracción de flujo

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

  return entrada; // habilita cin >> a >> b >> c;
}
```

- Suponiendo un objeto de la clase `NumeroTelefonico` llamado `telefono`
- Cuando el compilador encuentra la sentencia

  ```
  cin >> telefono;
  ```

  se genera una llamada a la función operador

  ```
  operator>>(cin, telefono);
  ```

- En esta llamada a la función operador
  1. el parámetro referencia `entrada` se vuelve un alias de `cin` y

Sobrecarga de los operadores << y >>

Implementación de la función operador de extracción de flujo

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

  return entrada; // habilita cin >> a >> b >> c;
}
```

- Suponiendo un objeto de la clase `NumeroTelefonico` llamado `telefono`
- Cuando el compilador encuentra la sentencia

  ```
  cin >> telefono;
  ```

  se genera una llamada a la función operador

  ```
  operator>>(cin, telefono);
  ```

- 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 `telefono`

Sobrecarga de los operadores << y >>

Implementación de la función operador de extracción de flujo

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

  return entrada; // habilita cin >> a >> b >> c;
}
```

- Suponiendo un objeto de la clase **NumeroTelefonico** llamado **telefono**
- Cuando el compilador encuentra la sentencia

```
cin >> telefono;
```

se genera una llamada a la función operador

```
operator>>(cin, telefono);
```

- 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 **telefono**
- La función operador 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, NumeroTelefonico &num)
{
  // Implementación

  return entrada; // habilita cin >> a >> b >> c;
}
```

- Suponiendo un objeto de la clase `NumeroTelefonico` llamado `telefono`
- Cuando el compilador encuentra la sentencia

```
cin >> telefono;
```

se genera una llamada a la función operador

```
operator>>(cin, telefono);
```

- 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 `telefono`
- La función operador devuelve una referencia de `istream` (`entrada`), o sea, `cin`.
- El primer operando no un objeto de la clase `NumeroTelefonico`

Sobrecarga de los operadores << y >>

```
1  istream &operator>>(istream &entrada, NumeroTelefonico &num)
2  {
3    entrada.ignore(); // ingnora (
4    entrada >> setw(4) >> num.codigoArea; // el código de área de entrada
5    entrada.ignore(2); // ignora ) y el espacio
6    entrada >> setw(4) >> num.intercambio; // introduce intercambio
7    entrada.ignore(); // ignora el guión (-)
8    entrada >> setw(5) >> num.linea; // introduce línea
9
10   return entrada; // habilita cin >> a >> b >> c;
11 }
```

Sobrecarga de los operadores << y >>

```
1  istream &operator>>(istream &entrada, NumeroTelefonico &num)
2  {
3    entrada.ignore(); // ingnora (
4    entrada >> setw(4) >> num.codigoArea; // el código de área de entrada
5    entrada.ignore(2); // ignora ) y el espacio
6    entrada >> setw(4) >> num.intercambio; // introduce intercambio
7    entrada.ignore(); // ignora el guión (-)
8    entrada >> setw(5) >> num.linea; // introduce línea
9
10   return entrada; // habilita cin >> a >> b >> c;
11 }
```

```
1  int main()
2  {
3    NumeroTelefonico telefono; // crea el objeto telefono
4
5    cout << "Introduzca un número de teléfono de la forma (123) 456-7890:\n";
6
7    // cin >> telefono invoca a la función operator>> al
8    // ejecutar la llamada operator>>(cin, telefono).
9    cin >> telefono;
10
11   return 0;
12 }
```

Sobrecarga de los operadores << y >>

Implementación de la función operador de inserción de flujo

```
1 ostream &operator<<(ostream &output, const NumeroTelefonico &num)
2 {
3   output << "(" << num.codigoArea << ")"
4     << num.intercambio << "-" << num.linea;
5   return output; // habilita cout << a << b << c;
6 }
```

Sobrecarga de los operadores << y >>

Implementación de la función operador de inserción de flujo

```
1 ostream &operator<<(ostream &output, const NumeroTelefonico &num)
2 {
3   output << "(" << num.codigoArea << ")"
4     << num.intercambio << "-" << num.linea;
5   return output; // habilita cout << a << b << c;
6 }
```

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

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

Sobrecarga de los operadores << y >>

Implementación de la función operador de inserción de flujo

```
1 ostream &operator<<(ostream &output, const NumeroTelefonico &num)
2 {
3   output << "(" << num.codigoArea << ")"
4     << num.intercambio << "-" << num.linea;
5   return output; // habilita cout << a << b << c;
6 }
```

La función operador de inserción de flujo **operator<<** tiene como argumento

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

   ```
   cout << telefono;
   ```

   se genera una llamada a la función operador

   ```
   operator<<(cout, telefono);
   ```

Sobrecarga de los operadores << y >>

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

Sobrecarga de los operadores << y >>

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

- Esto es así ya que el objeto de la clase `NumeroTelefonico` 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 `NumeroTelefonico` como funciones amigas no miembros de la clase
- Esto es así ya que el objeto de la clase `NumeroTelefonico` 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`

Sobrecarga de operadores unarios

Se puede sobrecargar un operador unario para una clase como

1. una función miembro (no estática) 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 (no estática) 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 (no estática) 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 (no estática) 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

Se puede sobrecargar un operador unario para una clase como

1. una función miembro (no estática) 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 & );
    . . .
};
```

- Las funciones miembros que implementan operadores sobrecargados no deben ser estáticas
- Las funciones miembros estáticas solo pueden acceder a datos miembros estáticos de la clase

Sobrecarga de operadores unarios

Sobrecarga del operador unario `!` como función miembro no estática sin argumentos

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

Sobrecarga de operadores unarios

Sobrecarga del operador unario `!` como función miembro no estática 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);
```

Sobrecarga de operadores binarios

Se puede sobrecargar un operador binario para una clase como

1. una función miembro no estática 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 no estática 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 no estática 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 no estática 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

- Sobrecarga del operador binario **+=** como función miembro no estática

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

Sobrecarga de operadores binarios

- Sobrecarga del operador binario `+=` como función miembro no estática

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

Si `y` y `z` son objetos de la clase `Cadena`, la sentencia `y += z` se convierte en una llamada a la función miembro `operator+=`

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

Sobrecarga de operadores binarios

- Sobrecarga del operador binario `+=` como función miembro no estática

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

Si `y` y `z` son objetos de la clase `Cadena`, la sentencia `y += z` se convierte en una llamada a la función miembro `operator+=`

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

- Sobrecarga del operador binario `+=` como función no miembro de la clase

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

Sobrecarga de operadores binarios

- Sobrecarga del operador binario `+=` como función miembro no estática

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

Si `y` y `z` son objetos de la clase `Cadena`, la sentencia `y += z` se convierte en una llamada a la función miembro `operator+=`

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

- Sobrecarga del operador binario `+=` como función no miembro de la clase

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

Si `y` y `z` son objetos o referencias de la clase `Cadena`, la sentencia `y += z` se convierte en una llamada a la función miembro `operator+=`

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

Ejemplo: clase `Arreglo`

Implementación de clase `Arreglo` como alternativa a los punteros del lenguajes C

Ejemplo: clase `Arreglo`

Implementación de clase `Arreglo` como alternativa a los punteros del lenguajes C

- No se puede imprimir un arreglo que no sea de `char` de forma directa (necesita un bucle)

Ejemplo: clase `Arreglo`

Implementación de clase `Arreglo` como alternativa a los punteros del lenguajes 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`)

Ejemplo: clase `Arreglo`

Implementación de clase `Arreglo` como alternativa a los punteros del lenguajes 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

Ejemplo: clase `Arreglo`

Implementación de clase `Arreglo` como alternativa a los punteros del lenguajes 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

Ejemplo: clase `Arreglo`

Implementación de clase `Arreglo` como alternativa a los punteros del lenguajes 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`)

Ejemplo: clase `Arreglo`

Implementación de clase `Arreglo` como alternativa a los punteros del lenguajes 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.

Ejemplo: clase `Arreglo`

Implementación de clase `Arreglo` como alternativa a los punteros del lenguajes 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 4° ed.)

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
24
25
26
27 };
```

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   private:
24     int tamanio; // tamaño del arreglo
25     int *ptr; // apuntador al primer elemento del arreglo
26     static int cuentaArreglo; // # de Arreglos instanciados
27 };
```

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 obtenerTamanio() const; // valor de retorno
10
11
12
13
14
15
16
17
18
19
20
21     static int obtenerCuentaArreglo(); // Devuelve la cuenta de los
22                                        // arreglos instanciados.
23   private:
24     int tamanio; // tamaño del arreglo
25     int *ptr; // apuntador al primer elemento del arreglo
26     static int cuentaArreglo; // # de Arreglos instanciados
27 };
```

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 obtenerTamanio() const; // valor de retorno
10
11
12
13
14
15
16
17
18
19
20
21     static int obtenerCuentaArreglo(); // Devuelve la cuenta de los
22                                        // arreglos instanciados.
23   private:
24     int tamanio; // tamaño del arreglo
25     int *ptr; // apuntador al primer elemento del arreglo
26     static int cuentaArreglo; // # de Arreglos instanciados
27 };
```

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 obtenerTamanio() 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     static int obtenerCuentaArreglo(); // Devuelve la cuenta de los
22                                        // arreglos instanciados.
23   private:
24     int tamanio; // tamaño del arreglo
25     int *ptr; // apuntador al primer elemento del arreglo
26     static int cuentaArreglo; // # de Arreglos instanciados
27 };
```

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 obtenerTamanio() 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     static int obtenerCuentaArreglo(); // Devuelve la cuenta de los
22                                        // arreglos instanciados.
23   private:
24     int tamanio; // tamaño del arreglo
25     int *ptr; // apuntador al primer elemento del arreglo
26     static int cuentaArreglo; // # de Arreglos instanciados
27 };
```

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 obtenerTamanio() 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     static int obtenerCuentaArreglo(); // Devuelve la cuenta de los
22                                        // arreglos instanciados.
23   private:
24     int tamanio; // tamaño del arreglo
25     int *ptr; // apuntador al primer elemento del arreglo
26     static int cuentaArreglo; // # de Arreglos instanciados
27 };
```

Constructor de copia

- Se invoca cuando es necesario copiar un objeto a otro, por ejemplo:

Constructor de copia

- Se invoca cuando es necesario copiar un objeto a otro, por ejemplo:
  1. en llamada a funciones por valor,

Constructor de copia

- Se invoca cuando es necesario copiar un objeto a otro, por ejemplo:
  1. en llamada a funciones por valor,
  2. cuando se devuelve por valor un objeto desde una función, o

Constructor de copia

- Se invoca cuando es necesario copiar un objeto a otro, por ejemplo:
  1. en llamada a funciones por valor,
  2. cuando se devuelve por valor un objeto desde una función, o
  3. cuando se inicializa un objeto que es copia de otro de la misma clase

Constructor de copia

- Se invoca cuando es necesario copiar un objeto a otro, por ejemplo:
  1. en llamada a funciones por valor,
  2. cuando se devuelve por valor un objeto desde una función, o
  3. cuando se inicializa un objeto que es copia de otro de la misma clase
- Se llama en una definición cuando se instancia y se inicializa un objeto

```
Arreglo enteros3(enteros1);
```

Constructor de copia

- Se invoca cuando es necesario copiar un objeto a otro, por ejemplo:
  1. en llamada a funciones por valor,
  2. cuando se devuelve por valor un objeto desde una función, o
  3. cuando se inicializa un objeto que es copia de otro de la misma clase
- Se llama en una definición cuando se instancia y se inicializa un objeto

```
Arreglo enteros3(enteros1);
```

  o de forma equivalente

```
Arreglo enteros3 = enteros1;
```

Constructor de copia

- Se invoca cuando es necesario copiar un objeto a otro, por ejemplo:
  1. en llamada a funciones por valor,
  2. cuando se devuelve por valor un objeto desde una función, o
  3. cuando se inicializa un objeto que es copia de otro de la misma clase
- Se llama en una definición cuando se instancia y se inicializa un objeto

  ```
  Arreglo enteros3(enteros1);
  ```

  o de forma equivalente

  ```
  Arreglo enteros3 = enteros1;
  ```

El constructor de copia debe tener como parámetro una referencia al objeto a copiar, no un objeto (llamada por valor). De lo contrario, el constructor de copia dará como resultado una *recursividad infinita*, ya que en una llamada por valor se debe pasar una copia del objeto al constructor copia lo que da como resultado que se ejecute el constructor copia de manera recursiva.