Module 5: Operator Overloading

Dr. Tran Minh Triet

Acknowledgement

- Slides
  - Course CS202: Programming Systems
    Instructor: MSc. Karla Fant,
    Portland State University
  - Course CS202: Programming Systems
    Instructor: Dr. Dinh Ba Tien,
    University of Science, VNU-HCMC
  - Course DEV275: Essentials of Visual Modeling with UML 2.0
    IBM Software Group

Outline

- What is function overloading?
- Operator overloading in C++
- Operator overloading in C#
- Overloading **cin** and **cout**

Overloading

- There are many different "definitions" for the same name
- In C++, overloading functions are differentiated by their signatures (i.e. number/types of arguments)
- ***Note:*** the return type is not considered in differentiating overloading functions.

Operator Overloading

- To define operator implementations for our new user-defined types
- For example, operators such as +, -, *, / are already defined for built-in types
- When we have a new data type, e.g. **CFraction**, we need to define new operator implementations to work with it.

Operators can be overloaded in C++

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

- Operator :: or . or .* cannot be defined by users.
- Operators sizeof, typeid, ?: cannot be overloaded.
- Operators =, ->, [], () can only be overloaded by non-static functions

Overloading guidelines

- Do what users expect for that operator.
- Define them if they make logical sense. E.g. subtraction of dates are ok but not multiplication or division
- Provide a complete set of properly related operators: a = a + b and a+= b have the same effect

Syntax

- Declared & defined like other methods, except that the keyword **operator** is used.

  <returned-type> **operator** <op>(<arguments>)

Example:

```
bool CFullName::operator==(const CFullName& rhs)
{
   return       ((m_sFirstName==rhs.m_sFirstFName) &&
                (m_sSurname==rhs.m_sSurName));
}
```

Operators in use

```
int main()
{
   CFullName s1, s2;
   if (s1 == s2) //s1.operator==(s2)
   {
         ...
   }
   ...
}
```

Exercise

- Implement a **CFraction** class with basic arithmetic operators: **+**, **-**, *****, **/**
- Remember to handle:

```
CFraction x, y;
y = x + 5;
y = 5 + x;
```

- Implement prefix and postfix increment:
  x++ and ++x. Hint: using dummy int

Notes about Op overloading

- Subscript operators often come in pair

```
const A&      operator[] (int index) const;
A&            operator[] (int index);
```

- Maintain the usual identities for x == y and x != y
- Prefix/Postfix operators for ++ and --
  - Prefix returns a reference
  - Postfix return a copy

Member and non-member functions

```
int main()
{
   CFullName s1, s2;
   if (s1 == s2)
          // member: s1.operator==(s2)
          // or non-member: operator==(s1, s2)
   {
          ...
   }
   ...
}
```

The keyword: friend

- With the keyword **friend**, you grant access to other functions or classes
- Friend functions give a flexibility to the class. It doesn't violate the encapsulation of the class.
- Friendship is "directional". It means if class A considers class B as its friend, it doesn't mean that class B considers A as a friend.

Example

```
class CDate
{
   public:
         ...
         friend void doSomething();
   private:
         int m_iDay, m_iMonth, m_iYear;
}
```

- In **doSomething()**, we can have access to private data members of the class **CDate**

Friend functions

- Friend functions is called like **f(x)** while member functions is called **x.f()**
- Use member functions if you can. Only choose friend functions when you have to.
- Sometimes, friend functions are good:
  - Binary infix arithmetic operators, e.g. **+, -**
  - Cannot modify original class, e.g. ostream

Friend functions

```
class CSample
{
    private:
        int m_a, m_b;
    public:
        friend int Compute(CSample x);
}
```

Friend functions

```
int Compute(CSample x)
{
   return x.m_a+x.m_b;
}

main()
{
   CSample x;
   ...
   cout << "The result is:" << Compute (x);
}
```

Overloading cin and cout

- We do not have access to the istream or ostream code → cannot overload << or >> as member functions
- They cannot be members of the user-defined class because the first parameter must be an object of that type
- Operators << and >> must be non-members, but it needs to access to private data members → make them friend functions

Typical syntax

- The general syntax for insertion and extraction operator overloadings:

```
ostream& operator<<(ostream& out, const CFraction& x)
{
   out << x.numerator << " / " << x.denominator;
   return out;
}

istream& operator>>(istream& in, CFraction& x);
```

Exercises

- Implement insertion and extraction operators for **CFraction** and **CDate** class

Operator Overloading In C#

Source: **Operator Overloading In C#**
By Rajesh (rajeshvs@msn.com)
URL: http://www.csharphelp.com/archives/archive135.html

Operator Overloading In C#

- All C# binary operators can be overloaded.

  +, -, *, /, %, &, |, <<, >>

- All C# unary operators can be overloaded.

  +, -, !, ~, ++, --, true, false

- All relational operators can be overloaded, but only as pairs.

  ==, !=, <, >, <= , >=

Operator Overloading In C#

- Compound assignment operators can be overloaded.
- In C#, compound assignment operators are automatically overloaded when the respective binary operator is overloaded.

  +=, -=, *=, /=, %=

- These operators cannot be overloaded:

  &&, ||

  () (Conversion operator)

  =, . , ?:, ->, new, is, as, sizeof

Operator Overloading In C#

- Operator functions must be public and static.
- They can take only value arguments.
- The ref and out parameters are not allowed as arguments to operator functions.

Operator Overloading In C#

- The general form of an operator function:

public static <return_type> operator <op> (<argument list>)

- For overloading the unary operators, there is only one argument and for overloading a binary operator there are two arguments.
- At least one of the arguments must be a user-defined type such as class or struct type.

Operator Overloading In C#

- The general form of operator function for unary operators is as follows.

  public static <return_type> operator <op>
  (<Type> t)

  { /* Statements */ }

  where Type must be a class or struct.

- The <return type> can be any type except void for unary operators like +, ~, ! and dot (.). but the <return type> must be the type of <Type> for ++ and --

Operator Overloading In C#

- An overloaded binary operator must take two arguments, at least one of them must be of the type class or struct, in which the operation is defined.
- Overloaded binary operators can return any value except the type void.
- The general form of a overloaded binary operator is as follows.

```
public static <return_type> operator <op>
(Type1 t1, Type2 t2)        { /*Statements*/ }
```