Module 4:
Operator Overloading

Prof. 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++
- 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.

Operators

- We can do the following for built-in types

```
void main()
{
            int   a, b;
            int   c = a + b;
}
```

- We define classes, we also want to do the same for two objects, like below

```
void main()
{
            MyString   str1, str2;
            MyString   str3 = str1 + str2;
}
```

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)
    {
            ...
    }
    ...
}
```

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

Two types of operator

- Independent operator

```
Fraction operator +( Fraction  p1, Fraction  p2 );
```

  - Does not belong to any class
  - Number of arguments = operator n-nary.

- Class operator

```
Fraction Fraction::operator +( Fraction  p );
```

  - A method of class
  - Number of arguments = operator n-nary - 1

- **They act the same!!**

Member and non-member functions

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

Limitations for operators

- We cannot create a new operator (we redefine instead)
- We cannot redefine operators for build-in types
- We cannot change operator n-nary
- We cannot change operator precedence order
- 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

Example

```
class Array
{
    int* elements;
    int length;
    int operator[](const int index)
    {
        if (index >= 0 && index < length)
            return this->elements[index];
        else
            throw (index);

    }
};
```

Practice

- Fraction operator+ (const Fraction &ps)
- Fraction operator+ (const int x)

Example

```
class Fraction
{
   int numerator;
   int denominator;
bool operator==(const Fraction &ps)
   {
      int result = this->numerator * ps.denominator - this->denominator * ps.numerator;
      if (result == 0)
         return true;
      else
         return false;

   }
};
```

Special operators

- Assignments (=, +=, -=, *=, /=, …):
  - Provide operator += for Fraction.
  - n-nary?
  - Return result?

```
Fraction&  Fraction::operator +=( const Fraction &p );
```

- Practice:
  - Fraction& operator=(const Fraction &ps)
  - Fraction& operator+=(const Fraction &ps)
  - Fraction& operator+=(const int x)

Example

```
class Fraction
{
    int *numerator;
    int *denominator;

    Fraction& operator=(const Fraction &ps) //Toán tử gán bằng
    {
        if (this == &ps) //Tránh a = a
            return *this;
        delete numerator; //Xóa vùng nhớ cũ
        delete denominator;
        numerator = new int;// Tạo lại vùng nhớ mới
        denominator = new int;
        *this->numerator = *ps.numerator;//Gán giá trị cho vùng nhớ mới
        *this->denominator = *ps.denominator;
        return *this;
    }
```

Special operators

- Increasing / Decreasing (++, --):
  - Provide operator ++ for Fraction
  - n-nary?
  - Return result?
  - Prefix vs. posfix?

- Practice:
  - Fraction& Fraction::operator ++( ); // Prefix.
  - Fraction Fraction::operator ++(int x); // Posfix, fake argument.

Example

```
//Toán tử tiền tố ++a
  Fraction& operator++()
  {
     //Do việc xử lý xong mới gán, nên chỉ cần xử lý và trả về chính nó
     this->numerator = this->numerator + this->denominator;
     return *this;
  }

  //Toán tử hậu tố a++
  Fraction operator++(int x)
  {
     //Gọi phương thức sao chép, chép giá trị trước
     Fraction result(*this);
     //Tiến hành xử lý trực tiếp trên đối tượng hiện tại
     this->numerator = this->numerator + this->denominator;
     //Trả về đối tượng sao chép, không thực hiện xử lý
     return result;
  }
```

Friend function

- Operator +
  - Provide operator + for Fraction
  - Use independent operator

```
Fraction  operator + ( const Fraction &p1, const Fraction &p2);
```

  - How to access *private members*?

- Operator <<
  - Provide operator << for Fraction

```
Fraction  p( 1, 3 );
cout  <<  p;
```

  - Which class operator << belongs to?

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

Practice

- Let's define and implement a Fraction class which represents a fraction number with the following operators
  - Arithmetic: +, *
  - Comparison: >, <, ==, >=, <=, !=
  - Assignment: =, +=, *=
  - Increasing / Decreasing: ++, -- (add/subtract 1 unit)
  - Type-cast: (float), (int)
  - Input/Output: >>, <<

Practice

- Define and implement a Vector class with necessary operators
  - Dot product: $A.B = |A||B|\cos\theta$
  - Hadamard product: $(A.B)_i = A_i B_i$

Practice

- Define and implement a Matrix class with necessary operators
  - Matrix product: $A[m\times n].B[n\times p] = C[m\times p]$
  - Hadamard product: $(A.B)_{ij} = A_{ij}B_{ij}$
  - Remark:
    - ~~int operator[](const int i, const int j)~~
    - int operator()(const int i, const int j)