NETB156 Object-Oriented Programming

Lasko M. Laskov

New Bulgarian University,
Informatics Department

♦

Lecture 10: Operator overloading. Automatic memory management

Outline

Operators as shorthand

Operators can be viewed as shorthand

- We are used to operators with *fundamental data types*.
- In this case they are intuitive, for example because of similarity with mathematics.
- Let `a`, `b` and `c` are integers. Which expression is intuitive?

```
a = b + c;
```

```
assign(a, sum(b, c));
```

In the expressions above:

- The operators are more intuitive and the expression is easier to read, remember and reproduce.
- On the other hand, operators act of shorthand of functionalities, that can be expressed as functions with the given arguments.

Operators and objects

Operators are intuitive also in some STL classes

- In some cases operators are quite intuitive when used with representatives of STL classes.
- Let `p` and `q` are `string` objects.
- Concatenation can be written in both ways.

```
p + q;
```

```
p.append(q);
```

⋆ In both expressions concatenation is a *member function* of the class invoked using the object `p`.

⋆ The mechanism that allowed the new meaning to be assigned to the operator is called *operator overloading*.

Operator overloading definition

Definition of a new meaning for operators

- A mechanism that assigns new meaning of a given operator that determines its actions with the representatives of a given class.

From a technical point of view

- Definition of a function whose name is `operator` followed by the operator symbol.
- It can be both a member and non-member function, but one of the operands must always be a class representative.
- You cannot redefine the operators on fundamental data types.

As *ad hoc* polymorphism

Operator overloading as *ad hoc* polymorphism

⋆ A given operator has different implementations depending on the data type of its operands.

ad hoc polymorphism

- a type of polymorphism;
- a polymorphic function has different implementations for each different set of data types of its arguments;
- for example, *function overloading* in C++.

Disadvantages of operator overloading

Not always intuitive:

- operators are concise as being just symbols;
- their concrete meaning can be not that evident in some cases;
- just recall << and >> used with objects `cout` and `cin`.

Their intuitive meaning can be substituted:

- the meaning that is supposed to be carried by an operator can be substituted by the programmer;
- for example, you can implement subtraction of complex numbers using the + operator.

Overloadable operators

Operators in the language

- are predefined set, and it cannot be changed;
- their precedence and associativity is fixed;
- the set of overloadable operators is given blow.

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

Overloaded operator functions

Operators can be overloaded in two ways:

- as non-member functions, separately from any class;
- as member functions of the given class.

Operator non-member functions

- in the case of a binary operator (two operands);
- both left and right operands are defined as function parameters.

Operator class member functions

- again, a binary operator;
- the left operand is always the implicit parameter.

Operator non-member functions

Overload an operator as non-member function

- separate from any class definition;
- at least one of the function parameters must be a class instantiation;
- often they are defined to be `friend` functions of the respective class.

```
1  return_type operator oper_symbol(par1, ...)
2  {
3      statement1;
4      ...
5      statementn;
6  }
```

Example

Overload the << operator

- to push data into a stack of integers;
- then we can use the operator instead of the `push()` member function.

```
1 void operator <<(stack<int> &st, int data)
2 {
3     st.push(data);
4 }
```

```
1 stack <int> st;
2 st << 42;
```

⋆ It is subjective how intuitive is the usage of the operator in this example.

Operator member functions

Overload an operator as member function

- part of the class definition itself;
- it means we must modify the class;
- the left operand is always the implicit parameter of the member function.

```
1  return_type Class_Name::operator oper_symbol
2                                   (par1, ...)
3  {
4      statement1;
5      ...
6      statementn;
7  }
```

Example

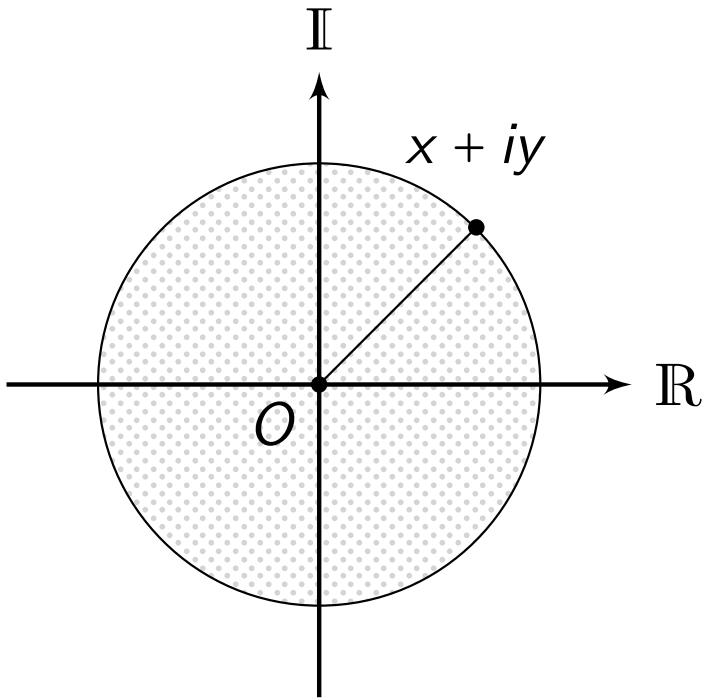
Overload the << operator

- the same example as above, but this time the operator function is part of the class;
- let us use our own implementation `Stack`

```
1  class Stack
2  {
3  public:
4      void operator <<(int data);
5      ...
6  };
```

```
1  void Stack::operator <<(int data)
2  {
3      push(data);
4  }
```

Complex numbers



Definition

Complex numbers are numbers of the form $x + iy$, where:

- $x, y \in \mathbb{R}$;
- $i = \sqrt{-1}$ is the imaginary unit.

Class Complex definition

- Definition of the basic class features.

```
1  class Complex
2  {
3  public:
4      Complex();
5      Complex(const Complex& z);
6      void set(double real, double imag);
7      Complex sum(const Complex& z);
8      void print() const;
9  private:
10     double real;
11     double imag;
12 };
```

Overloading the + operator

- Instead of using the member function `sum()`.

```
1  Complex Complex::operator +(const Complex& z)
2  {
3      Complex result;
4      result.real = real + z.real;
5      result.imag = imag + z.imag;
6      return result;
7  }
```

Overloading << operator

- This time the operator function cannot be part of the class.
- The reason is that the left operand must be of type `ostream`.
- However, the function can be defined as a **`friend`** of the class `Complex`.

```
1  ostream& operator <<(ostream& out,
2                       const Complex& z)
3  {
4      cout << z.real << " + i." << z.imag;
5      return out;
6  }
```

Memory management and classes

Memory management

- In C++ memory management is in the hands of the programmer.
- Is is both power and drawback of the language.
- Class encapsulation allows to hide the dynamic memory management from the class users.

For correct dynamic management, a class must provide:

- default constructor;
- copy constructor;
- destructor;
- overloaded assignment operator.

Class String

- To demonstrate above we will implement our own class `String`.
- The class contains dynamically allocated array of characters.

```
1  class String
2  {
3  public:
4      String();
5      String(const String& str);
6      ~String();
7      String& operator =(const String& str);
8  private:
9      char* arr;
10     int len;
11 };
```

Default constructor

- Default constructor is automatically triggered when an new object (with no input parameters) is created.
- In this case an empty string means a `NULL` pointer and zero characters.

```
1  String::String()
2  {
3      arr = NULL;
4      len = 0;
5  }
```

Copy constructor

- Invoked when a copy of the object is needed.
- For example when the object is passed as a function parameter by value.

```
1  String::String(const String& str)
2  {
3      len = str.len;
4      arr = new char[len + 1];
5      for (int i = 0; i < len; i++)
6      {
7          arr[i] = str.arr[i];
8      }
9      arr[len] = '\0';
10 }
```

Destructor

- It is automatically invoked when the object is destroyed.
- It determines how the memory dynamically allocated for the object is freed.

```
1  String::~String()
2  {
3      if (arr)
4      {
5          delete[] arr;
6          arr = 0;
7      }
8  }
```

Overloaded assignment operator

- Copies the contents of an object into another.

```
1  String& String::operator =(const String& str)
2  {
3      if (this != &str)
4      {
5          if (arr)
6          {
7              delete[] arr; arr = 0;
8          }
9          len = str.len;
10         arr = new char[len + 1];
11         for (int i = 0; i < len; i++)
12             arr[i] = str.arr[i];
13         arr[len] = '\0';
14     }
15     return *this;
16 }
```