Lecture #09

Polymorphism (1-2)

SE271 Object-Oriented Programming (2020)

Yeseong Kim

Original slides from Prof. Shin at DGIST

Short Notice

- Again: Will take the midterm
  - 10/21 during the class time
  - Convention hall A, E1 building
  - Can leave after 11:30 pm
  - Will include the lecture materials we will learn on Wednesday
- HW2
  - New deadline: 10/25 Sunday
  - Try to do it even though the deadline is after the exam
  - Don’t try cheating anymore. F will be graded without any exception

Today's Topic

- Polymorphism: Overloading
  - –Operator Overloading
    - Operator <<
    - Operator []
    - Operator =

- Let's detour: std::string

Operator Overloading

- Operators in C++ are defined as functions – overloading
  - num1 @ num2;
    - return_type **operator**@(a, b);
    - return_type a.**operator**@(b);

- Operators (can be overloaded)
  - +, -, *, /, %
  - **=**, +=, -=, *=, /=, %=
  - ==, !=, >, >=, <, <=
  - ++, -- (postfix, prefix)
  - ^, &, |, ~, **<<**, >>
  - &&, ||, !
  - ->*, ->, (), **[]**, new, new[], delete[], delete

Example: operator overloading <<

- Complex type (23 + 7j)
  - **cout << Complex(23, 7);**
  - **23+7j** Cout : class std::basic_ostream<char>
  - **ostream @ Complex**

```
class Complex {
    int* m_r;  // real part
    int* m_i;  // imaginary part
public:
    ……
    friend ostream& operator<< (ostream& o,
Complex c)
};
```

```
ostream& operator<< (ostream& o, Complex c)
{
    o << *c.m_r << (*c.m_i < 0 ? "" : "+") << *c.m_i
<< "j" ;
    return o;
}

int main(){
    cout << Complex (23, 7);
    return 0;
}
```

Example: operator overloading []

- Vector: dynamic array
  - cout << val[1];
  - val[10] = 20;

```
class Vector {
    int* m_data;  // pointer to dynamic array
    int m_size;  // array size
public:
    Vector() : m_data(new int[100]{ 0, }),
m_size(100) {}
    ~Vector() { delete m_data; }
    //int operator[] (int index) const;
    int& operator[] (int);
};
```

```
int& Vector::operator[] (int index) {
    if (index >= m_size)
    {
        cout << "Error: index out of bound";
        exit(0);
    }
    return m_data[index];
}
int main(){
    Vector v;
    cout << v[1];
    v[1] = 20;
    cout << v[1];
}
```

Example: operator overloading =

- Complex type (23 + 7j)
  - Complex = Complex
  - a = b = c

```
class Complex {
    int* m_r;  // real part
    int* m_i;  // imaginary part
public:
    ……
     Complex operator=(Complex c);
};
```

```
Complex Complex::perator=(Complex c) {
    if (this == &c)
        return *this;
    delete m_r;
    delete m_i;
    m_r = new int(*c.m_r);
    m_i = new int(*c.m_i);
    return *this;
}
```

From Point of Memory View

```
Complex fn(Complex c) {
    return *this;
}

Complex t, o;
t = fn(o);
```

```
Return_type fn(Parmeter_type p, ...)
{
    ......
    return *this;
}
```

Example: operator overloading =

- Complex type (23 + 7j)
  - Complex = Complex
  - a = b;

```
class Complex {
    int* m_r;  // real part
    int* m_i;  // imaginary part
public:
    ......
   Complex & operator=(Complex & c);
};
```

```
Complex & operator=(Complex & c) {
    if (this == &c)
        return *this;
    delete m_r;
    delete m_i;
    m_r = new int(*c.m_r);
    m_i = new int(*c.m_i);
    return *this;
}
```

a = b + c; ???

r-value vs. l-value

- a = b + c

  temporary object (i.e. Complex(2, 3))

  Q) Complex & param = (b+c); // or Complex(2,3);

```
int a, b;
a = 2 + 3;
a = b = 2;
a = b + 3;
int& c = a;
int& d = 10
const int& e = 10;
```

- l-value vs. r-value
  - **r-value :** a temporary value that does not persist beyond the expression that uses it[1]
  - **l-value :** an object that persists beyond a single expression[1]
    - Has storage address

[1] "Lvalues and Rvalues (Visual C++)". Microsoft Developer Network. Retrieved 3 September 2016.

Example: operator overloading =

- Complex type (23 + 7j)
  - Complex = Complex
  - a = b + c;

```
class Complex {
    int* m_r;  // real part
    int* m_i;  // imaginary part
public:
    ......
    Complex & operator=(const Complex & c);
};
```

```
Complex & operator=(const Complex & c) {
    if (this == &c)
        return *this;
    delete m_r;
    delete m_i;
    m_r = new int(*c.m_r);
    m_i = new int(*c.m_i);
    return *this;
}
```

```
Complex c1, c2, c3;
c1 =-c2 + c3;
```

Overview of std::string

- A string is a sequence (array) of characters in C and C++
  - E.g., : “John” “Hello World”

- In C, we should use special functions to handle strings
  - E.g., strcmp(a, b) *!=* a == b

- In C++, we can nicely abstract the character array with std::string
  - #include <string>
  - std::string is a class! The power of object-oriented programming!
  - Now, you can use a == b

Creating a string from char*

- You can assign char* to std::string

```
std::string x1("Hello World!");
std::string x2 = "Hello World!";
std::string x3; x3 = "Hello World!";
std::string x3 = x4;
```

- Cf. you can check the type of a variable with the following way

```
auto x4 = "Hello World"; // ????
cout << typeid(x4).name() << endl;
```

Concatenating two strings (or char*)

- The operator + performs the concatenation of two strings
- You can mix the types in the concatenation as well!

```
std::string x1("Hello World!");
std::string x2 = "It is a beautiful world!";

cout << (x1 + " " + x2) << endl;
```

- You can also add with += operator

```
x1 += " " + x2;
cout << x1 << endl;
```

String comparison

- You can check whether two strings are same using ==

```
std::string x1("Hello World!");
std::string x2 = "It is a beautiful world!";

cout << (x1 + " " + x2) << endl;
```

  - Cf. want to see if they are different? Use !=

- The operator <, >, >=, <= is also supported – in an alphabetical order

```
std::string a = "apple";
std::string b = "banana";

cout << std::boolalpha << (a == b) << endl;
cout << std::boolalpha << (a < b) << endl;
```

Index Operator and substr function

- [] : Get the character at a position
- substr: get the substring for the given range

```
std::string alphabet = "ABCDEFG";
cout << alphabet[2] << endl;
cout << alphabet.substr(2, 4) << endl;
```

Find patterns

- find: find the index of the rightmost occurrence
- rfind: find the index of the leftmost occurrence

```
std::string foo = "My name is John. His name is quite nice.";
cout << foo.find("name") << endl;
cout << foo.rfind("name") << endl;
```

A short, unofficial homework

- Think about how the operator “<<“ works for std::string or char*
  - Imagine how you can implement ☺

References

- Learn c++
  - https://www.learncpp.com/
  - Chapter 6.11a
  - Chapter 9.3, 9.8
  - Chapter 15.1-3

ANY QUESTIONS?