Class-OperatorOverloading

August 8, 2020

1 Overloading Operators

http://www.cplusplus.com/doc/tutorial/templates/

1.1 Table of Contents

1.2 header includes used in this notebook

[1]:
```
#include <iostream>
#include <string>

using namespace std;
```

1.3 Overloading operators

- classes, essentially, define new types to be used in C++ code
- for fundamental type such as int, float, certain operations have been defined using various operators
  - such operators such as (+, -, ==, >, etc.) have unambiguous meaning
  - c++ basic string class template has most of these operators overloaded. see: https://en.cppreference.com/w/cpp/string/basic_string
- classes (user defined types) on the other hand doesn't support these operators out-of-the box
- C++ allows most operators to be overloaded so that their behavior can be defined for just about any type...

1.3.1 Overloadable operators

- +, -, %, *, /, =, +=, *=, /=, %=
- <<, >>, <<=, >>=,
- <, >, ==, != <=, >=,
- ++, –

- &, ^, !, |
- ~, &=, ^=, |=
- &&, ||
- [ ], ( ), ->, ->*
- new, delete, new[], delete[]
- Syntax to overload operators:

```
type operator sign (parameters) {
    // sign is the operator symbol being overloaded
    // function body
}
```

- operators may be overloaded in two forms:
  - either as **member function** or as a **non-member function**

1.4 Overloading operators with non-member functions

- simply define functions that overload operator for some class type!
- some operators can NOT be overloaded as non-member functions
  - e.g., =, +=, [ ], -=, ( ), ->, etc.
- some operators can be ONLY overloaded as non-member functions
  - e.g., input insertion (>>) and output extraction operator (<<)

NOTE: if you see error in notebook, see complete code provided in demo-programs/operator-overloading/Rectangle.cpp

1.5 Friend functions

- friend functions can access private members of a class
- class can declare functions as friends, but functions can't declare themselves as a friend to class
- friend functions are not member of a class
- application of friend function:
  - if members are private (they typically are), friend functions can help overload operators as non-member functions
- use keyword `friend` in front of function name while marking a function as friend inside any class definition

For example see: demo-programs/operator-overloading/RectangleFriend.cpp

[2]:
```
// non-member function overload example
class Rectangle {
    friend void printRectangleFriend(const Rectangle& r);
    private:
        float length, width;
    public:
        // default and overloaded constructor
        Rectangle(float length=0, float width=0) {
```

```
            if (length < 0)
                length = 0;
            if (width == 0)
                width = 0;
            this->length = length;
            this->width = width;
        }

        float findArea() const {
            return this->length*this->width;
        }

        float findPerimeter() const {
            return 2*(this->length + this->width);
        }
};
```

[3]:
```
void printRectangle(const Rectangle& r) {
    //cout << "length = " << r.length << endl; // can't do this!!
    //cout << "width = " << r.width << endl; // can't do this too!!
    cout << "area = " << r.findArea() << endl; // can do this!
    cout << "perimeter = " << r.findPerimeter() << endl; // can do this!!
}
```

[4]:
```
void printRectangleFriend(const Rectangle& r) {
    cout << "length = " << r.length << endl; // can't do this!!
    cout << "width = " << r.width << endl; // can't do this too!!
    cout << "area = " << r.findArea() << endl; // can do this!
    cout << "perimeter = " << r.findPerimeter() << endl; // can do this!!
}
```

[5]:
```
Rectangle smallRect = {4, 2};
```

[6]:
```
printRectangleFriend(smallRect);
```

```
length = 4
width = 2
area = 8
perimeter = 12
```

1.6 Overloading operator with member functions

- some operators may be loaded in two forms: either as a member or as a non-member
  - e.g., +, -, *, /, >, <, ==, etc.
- some operators can be overloaded only as member functions
  - e.g., =, +=, [ ], -=, ( ), ->, etc.

NOTE: See demo-programs/operator-overloading/RectangleMember.cpp for working fully example!

[7]:
```
// overloading with member functions example
class Rectangle1 {
    private:
        float length, width;

    public:
        // default and overloaded constructor
        Rectangle1(float length=0, float width=0) {
            if (length < 0)
                length = 0;
            if (width == 0)
                width = 0;
            this->length = length;
            this->width = width;
        };

        float findArea() const {
            return this->length*this->width;
        }

        float findPerimeter() const {
            return 2*(this->length + this->width);
        }

        // overload + operator
        Rectangle1 operator+(const Rectangle1& rhs) {
            Rectangle1 temp;
            temp.length = this->length + rhs.length;
            temp.width = this->width + rhs.width;
            return temp;
        }

        // overload [] operator
        float operator[](unsigned int index) {
            if (index == 0)
                return this->length;
            else
                return this->width;
        }
};
```

[8]:
```
Rectangle1 r1 = {20, 10};
Rectangle1 r2 = {10, 5};
```

[9]:
```
Rectangle1 r = r1 + r2;
```

[10]:
```
cout << "length = " << r[0] << " width = " << r[1] << endl;
```

```
length = 30 width = 15
```

1.7 Constant objects and methods

- when an object of a class is qualified as a const object:

```
const SomeClass someObject;
```

```
- the access to its data members from outside the class is restricted to read-only, as if all
- Note: constructor is still called and is allowed to initialize and modify these data members
```

[ ]:
```
class SomeClass {
  public:
    int x;
    SomeClass(int val=0) { this->x = val;}
    int getX() { return x; }
};
```

[ ]:
```
const SomeClass someObj(10);
```

[ ]:
```
someObj.x = 100; // not allowed becuase someObj is const
```

[ ]:
```
cout << "x = " << someObj.x << endl; // x is public member
```

[ ]:
```
cout << "x = " << someObj.getX() << endl; // getX is not marked constant!
```

[ ]:
```
class MyClass {
  public:
    int x;
    MyClass(int val=0) { this->x = val;}
    int getX() const { return x; }
};
```

[ ]:
```
const MyClass myObj(20);
```

[ ]:
```
cout << "x = " << myObj.getX() << endl;
```

[ ]:
```
// passing const class objects to function is very common
void myPrint(const MyClass& obj) {
    cout << "x = " << obj.getX() << endl;
}
```

[ ]:
```
// myObj is passed by const ref
myPrint(myObj);
```

See /demo-programs/operator-overloading/ComplexNumber/ for complete example working with complex numbers and operator overloading

[ ]: