Example: The MyComplex Class

The MyComplex class is simplified from the C++ STL's complex template class. I strongly recommend that you study the source code of complex (in the complex header) - you can download the source code for GNU GCC.

MyComplex.h

```
 1/*
 2 * The MyComplex class header (MyComplex.h)
 3 * Follow, modified and simplified from GNU GCC complex template class
 4 */
 5#ifndef MY_COMPLEX_H
 6#define MY_COMPLEX_H
 7
 8#include <iostream>
 9
10class MyComplex {
11private:
12   double real, imag;
13
14public:
15   explicit MyComplex (double real = 0, double imag = 0); // Constructor
16   MyComplex & operator+= (const MyComplex & rhs); // c1 += c2
17   MyComplex & operator+= (double real);           // c += double
18   MyComplex & operator++ ();                      // ++c
19   const MyComplex operator++ (int dummy);         // c++
20   bool operator== (const MyComplex & rhs) const;   // c1 == c2
21   bool operator!= (const MyComplex & rhs) const;   // c1 != c2
22
23   // friends
24   friend std::ostream & operator<< (std::ostream & out, const MyComplex & c); // out << c
25   friend std::istream & operator>> (std::istream & in, MyComplex & c);        // in >> c
26   friend const MyComplex operator+ (const MyComplex & lhs, const MyComplex & rhs); // c1
27   friend const MyComplex operator+ (double real, const MyComplex & rhs);  // double + c
28   friend const MyComplex operator+ (const MyComplex & lhs, double real);  // c + double
29};
30
31#endif
32
```

Program Notes:

- I prefer to list the private section before the public section in the class declaration to have a quick look at the internal of the class for ease of understanding.
- I named the private data members real and imag, that potentially crash with the function parameters. I resolves the crashes via this-> pointer if needed. Some people suggest to name private data members with a trailing underscore (e.g., real_, imag_) to distinguish from

the function parameters. As private members are not expose to the users, strange names are acceptable. The C++ compiler uses leading underscore(s) to name its variables internally (_xxx for data members, __xxx for local variables).

- The constructor is declared `explicit`. This is because a single-argument constructor can be used for implicit conversion, in this case, from `double` to `MyComplex`, e.g.,

```
// Without explicit
MyComplex c = 5.5;  // Same as MyComplex c = (MyComplex)5.5;
The keyword explicit disables implicit conversion.
// With explicit
MyComplex c = 5.5;
   // error: conversion from 'double' to non-scalar type 'MyComplex' requested
MyComplex c = (MyComplex)5.5;   // Okay
```

Avoid implicit conversion, as it is hard to track and maintain.

- The constructor sets the default value for `real` and `imag` to 0.
- We overload the stream insertion operator `<<` to print a `MyComplex` object on a `ostream` (e.g., `cout << c`). We use a non-member friend function (instead of member function) as the left operand (`cout`) is not a `MyComplex` object. We declare it as friend of the `MyComplex` class to allow direct access of the private data members. The function return a reference of the invoking `ostream` object to support cascading operation, e.g. `cout << c << endl;`.
- We overload the prefix increment operator (e.g., `++c`) and postfix increment operator (e.g., `c++`) as member functions. They increases the real part by 1.0. Since both prefix and postfix operators are unary, a dummy `int` argument is assigned to postfix `operator++()` to distinguish it from prefix `operator++()`. The prefix operator returns a reference to this object, but the postfix returns a value. We shall explain this in the implementation.
- We overload the plus operator `+` to perform addition of two `MyComplex` objects, a `MyComplex` object and a `double`. Again, we use non-member friend function as the left operand may not be a `MyComplex` object. The `+` shall return a new object, with no change to its operands.
- As we overload the `+` operator, we also have to overload `+=` operator.
- The function's reference/pointer parameters will be declared `const`, if we do not wish to modify the original copy. On the other hand, we omit `const` declaration for built-in types (e.g., double) in the class declaration as they are passed by value - the original copy can never be changed.
- We declare the return values of `+` operator as `const`, so that they cannot be used as *lvalue*. It is to prevent meaningless usages such as `(c1+c2) = c3` (most likely misspelling `(c1 + c2) == c3`).
- We also declare the return value of `++` as `const`. This is to prevent `c++++`, which could be interpreted as `(c++)++`. However, as `C++` return by value a temporary object (instead of the original object), the subsequent `++` works on the temporary object and yields incorrect output. But `++++c` is acceptable as `++c` returns this object by reference.

MyComplex.cpp

```
1/* The MyComplex class implementation (MyComplex.cpp) */
2#include "MyComplex.h"
3
4// Constructor
```

```
 5MyComplex::MyComplex (double r, double i) : real(r), imag(i) { }
 6
 7// Overloading += operator for c1 += c2
 8MyComplex & MyComplex::operator+= (const MyComplex & rhs) {
 9   real += rhs.real;
10   imag += rhs.imag;
11   return *this;
12}
13
14// Overloading += operator for c1 += double (of real)
15MyComplex & MyComplex::operator+= (double value) {
16   real += value;
17   return *this;
18}
19
20// Overload prefix increment operator ++c (real part)
21MyComplex & MyComplex::operator++ () {
22  ++real;   // increment real part only
23  return *this;
24}
25
26// Overload postfix increment operator c++ (real part)
27const MyComplex MyComplex::operator++ (int dummy) {
28   MyComplex saved(*this);
29   ++real;  // increment real part only
30   return saved;
31}
32
33// Overload comparison operator c1 == c2
34bool MyComplex::operator== (const MyComplex & rhs) const {
35   return (real == rhs.real && imag == rhs.imag);
36}
37
38// Overload comparison operator c1 != c2
39bool MyComplex::operator!= (const MyComplex & rhs) const {
40   return !(*this == rhs);
41}
42
43// Overload stream insertion operator out << c (friend)
44std::ostream & operator<< (std::ostream & out, const MyComplex & c) {
45   out << '(' << c.real << ',' << c.imag << ')';
46   return out;
47}
48
49// Overload stream extraction operator in >> c (friend)
```

```
50std::istream & operator>> (std::istream & in, MyComplex & c) {
51   double inReal, inImag;
52   char inChar;
53   bool validInput = false;
54   // Input shall be in the format "(real,imag)"
55   in >> inChar;
56   if (inChar == '(') {
57      in >> inReal >> inChar;
58      if (inChar == ',') {
59         in >> inImag >> inChar;
60         if (inChar == ')') {
61            c = MyComplex(inReal, inImag);
62            validInput = true;
63         }
64      }
65   }
66   if (!validInput) in.setstate(std::ios_base::failbit);
67   return in;
68}
69
70// Overloading + operator for c1 + c2
71const MyComplex operator+ (const MyComplex & lhs, const MyComplex & rhs) {
72   MyComplex result(lhs);
73   result += rhs;  // uses overload +=
74   return result;
75   // OR return MyComplex(lhs.real + rhs.real, lhs.imag + rhs.imag);
76}
77
78// Overloading + operator for c + double
79const MyComplex operator+ (const MyComplex & lhs, double value) {
80   MyComplex result(lhs);
81   result += value;  // uses overload +=
82   return result;
83}
84
85// Overloading + operator for double + c
86const MyComplex operator+ (double value, const MyComplex & rhs) {
87   return rhs + value;   // swap and use above function
88}
```

Program Notes:

- The prefix ++ increments the real part, and returns this object by reference. The postfix ++ saves this object, increments the real part, and returns the saved object by value. Postfix operation is clearly less efficient than prefix operation!
- The + operators use the += operator (for academic purpose).

- The friend functions is allow to access the private data members.
- The overloaded stream insertion operator << outputs "`(real,imag)`".
- The overloaded stream extraction operator >> inputs "`(real,imag)`". It sets the `failbit` of the `istream` object if the input is not valid.

`TestMyComplex.cpp`

```
 1/* Test Driver for MyComplex class (TestMyComplex.cpp) */
 2#include <iostream>
 3#include <iomanip>
 4#include "MyComplex.h"
 5
 6int main() {
 7   std::cout << std::fixed << std::setprecision(2);
 8
 9   MyComplex c1(3.1, 4.2);
10   std::cout << c1 << std::endl;  // (3.10,4.20)
11   MyComplex c2(3.1);
12   std::cout << c2 << std::endl;  // (3.10,0.00)
13
14   MyComplex c3 = c1 + c2;
15   std::cout << c3 << std::endl;  // (6.20,4.20)
16   c3 = c1 + 2.1;
17   std::cout << c3 << std::endl;  // (5.20,4.20)
18   c3 = 2.2 + c1;
19   std::cout << c3 << std::endl;  // (5.30,4.20)
20
21   c3 += c1;
22   std::cout << c3 << std::endl;  // (8.40,8.40)
23   c3 += 2.3;
24   std::cout << c3 << std::endl;  // (10.70,8.40)
25
26   std::cout << ++c3 << std::endl; // (11.70,8.40)
27   std::cout << c3++ << std::endl; // (11.70,8.40)
28   std::cout << c3   << std::endl; // (12.70,8.40)
29
30// c1+c2 = c3;  // error: c1+c2 returns a const
31// c1++++;      // error: c1++ returns a const
32
33// MyComplex c4 = 5.5;  // error: implicit conversion disabled
34   MyComplex c4 = (MyComplex)5.5;  // explicit type casting allowed
35   std::cout << c4 << std::endl; // (5.50,0.00)
36
37   MyComplex c5;
38   std::cout << "Enter a complex number in (real,imag): ";
39   std::cin >> c5;
```

```
40   if (std::cin.good()) {  // if no error
41      std::cout << c5 << std::endl;
42   } else {
43      std::cerr << "Invalid input" << std::endl;
44   }
45   return 0;
46}
```