C++ Programming Language

Operator Overloading

TABLE OF CONTENTS (HIDE)

Operator overloading means that the operation performed by the operator depends on the *type* of operands provided to the operator. For example, (a) the bit left-shift operator `<<` is overloaded to perform stream insertion if the left operand is a `ostream` object such as `cout`; (b) the operator `*` could means multiplication for two numbers of built-in types or indirection if it operates on an address. C++ lets you extend operator overloading to user-defined types (classes).

Operator overloading is similar to function overloading, where you have many versions of the same function differentiated by their parameter lists.

1. Overloaded Operators in the `string` class

As an example, the C++ `string` class (in header `<string>`) overloads these operators to work on `string` objects:

- String comparison (`==`, `!=`, `>`, `<`, `>=`, `<=`): For example, you can use `str1 == str2` to compare the contents of two `string` objects.
- Stream insertion and extraction (`<<`, `>>`): For example, you can use `cout << str1` and `cin >> str2` to output/input `string` objects.
- Strings concatenation (`+`, `+=`): For example, `str1 + str2` concatenates two `string` objects to produce a new `string` object; `str1 += str2` appends `str2` into `str1`.
- Character indexing or subscripting `[]`: For example, you can use `str[n]` to get the `char` at index *n*; or `str[n] = c` to modify the `char` at index *n*. Take note that `[]` operator does not perform index-bound check, i.e., you have to ensure that the index is within the bounds. To perform index-bound check, you can use `string`'s `at()` member function.
- Assignment (`=`): For example, `str1 = str2` assigns `str2` into `str1`.

Example

```
1  /* Test overloaded operators in the C++ string class
2     (TestStringOverloadOperators.cpp) */
3  #include <iostream>
4  #include <iomanip>
5  #include <string>    // needed to use the string class
6  using namespace std;
7
```

```
 8  int main() {
 9     string msg1("hello");
10     string msg2("HELLO");
11     string msg3("hello");
12
13     // Relational Operators (comparing the contents)
14     cout << boolalpha;
15     cout << (msg1 == msg2) << endl;  // false
16     cout << (msg1 == msg3) << endl;  // true
17     cout << (msg1 < msg2)  << endl;  // false (uppercases before lowercases)
18
19     // Assignment
20     string msg4 = msg1;
21     cout << msg4 << endl;  // hello
22
23     // Concatenation
24     cout << (msg1 + " " + msg2) << endl;  // hello HELLO
25     msg3 += msg2;
26     cout << msg3 << endl;  // helloHELLO
27
28     // Indexing
29     cout << msg1[1] << endl;     // 'e'
30     cout << msg1[99] << endl;    // garbage (no index-bound check)
31  // cout << msg1.at(99) << endl; // out_of_range exception
32  }
```

Notes: The relational operators (==, !=, >, <, >=, <=), +, <<, >> are overloaded as non-member functions, where the left operand could be a non-`string` object (such as C-string, `cin`, `cout`); while =, [ ], += are overloaded as member functions where the left operand must be a `string` object. I shall elaborate later.

2. User-defined Operator Overloading

2.1 "operator" Functions

To overload an operator, you use a special function form called an *operator function*, in the form of `operatorΔ()`, where `Δ` denotes the operator to be overloaded:

```
return-type operatorΔ(parameter-list)
```

For example, `operator+()` overloads the + operator; `operator<<()` overloads the << operator. Take note that `Δ` must be an existing C++ operator. You cannot create you own operator.

2.2 Example: Overloading '+' Operator for the `Point` Class as Member Function

In this example, we shall overload the `'+'` operator in the `Point` class to support addition of two `Point` objects. In other words, we can write `p3 = p1+p2`, where `p1`, `p2` and `p3` are `Point` objects, similar to the usual arithmetic operation. We shall construct a new `Point` instance `p3` for the sum, without changing the `p1` and `p2` instances.

`Point.h`

```
 1  /* The Point class Header file (Point.h) */
 2  #ifndef POINT_H
 3  #define POINT_H
 4
 5  class Point {
 6  private:
 7     int x, y; // Private data members
```

```
 8
 9  public:
10     Point(int x = 0, int y = 0); // Constructor
11     int getX() const; // Getters
12     int getY() const;
13     void setX(int x); // Setters
14     void setY(int y);
15     void print() const;
16     const Point operator+(const Point & rhs) const;
17           // Overload '+' operator as member function of the class
18  };
19
    #endif
```

Program Notes:

- We overload the + operator via a member function `operator+()`, which shall add this instance (left operand) with the `rhs` operand, construct a new instance containing the sum and and return it *by value*. We cannot return by reference a local variable created inside the function, as the local variable would be destroyed when the function exits.
- The `rhs` operand is passed by reference for performance.
- The member function is declared `const`, which cannot modify data members.
- The return value is declared `const`, so as to prevent it from being used as *lvalue*. For example, it prevents writing `(p1+p2) = p3`, which is meaningless and could be due to misspelling `(p1+p2) == p3`.

`Point.cpp`

```
 1  /* The Point class Implementation file (Point.cpp) */
 2  #include "Point.h"
 3  #include <iostream>
 4  using namespace std;
 5
 6  // Constructor - The default values are specified in the declaration
 7  Point::Point(int x, int y) : x(x), y(y) { } // Using initializer list
 8
 9  // Getters
10  int Point::getX() const { return x; }
11  int Point::getY() const { return y; }
12
13  // Setters
14  void Point::setX(int x) { this->x = x; }
15  void Point::setY(int y) { this->y = y; }
16
17  // Public Functions
18  void Point::print() const {
19     cout << "(" << x << "," << y << ")" << endl;
20  }
21
22  // Member function overloading '+' operator
23  const Point Point::operator+(const Point & rhs) const {
24     return Point(x + rhs.x, y + rhs.y);
25  }
```

Program Notes:

- The function allocates a new `Point` object with the sums of `x`'s and `y`'s, and returns this object by `const` value.

`TestPoint.cpp`

```
 1  #include "Point.h"
 2  #include <iostream>
```

```
using namespace std;

int main() {
   Point p1(1, 2), p2(4, 5);
   // Use overloaded operator +
   Point p3 = p1 + p2;
   p1.print();  // (1,2)
   p2.print();  // (4,5)
   p3.print();  // (5,7)

   // Invoke via usual dot syntax, same as p1+p2
   Point p4 = p1.operator+(p2);
   p4.print();  // (5,7)

   // Chaining
   Point p5 = p1 + p2 + p3 + p4;
   p5.print();  // (15,21)
}
```

Program Notes:

- You can invoke the overloaded operator via `p1+p2`, which will be translated into the dot operation `p1.operator+(p2)`.
- The `+` operator supports chaining (cascading) operations, as `p1+p2` returns a `Point` object.

2.3 Restrictions on Operator Overloading

- The overloaded operator must be an existing and valid operator. You cannot create your own operator such as $\oplus$.
- Certain C++ operators cannot be overloaded, such as `sizeof`, dot (`.` and `.*`), scope resolution (`::`) and conditional (`?:`).
- The overloaded operator must have at least one operands of the user-defined types. You cannot overload an operator working on fundamental types. That is, you can't overload the `'+'` operator for two `int`s (fundamental type) to perform subtraction.
- You cannot change the syntax rules (such as associativity, precedence and number of arguments) of the overloaded operator.

3. Overloading Operator via "`friend`" non-member function

3.1 Why can't we always use Member Function for Operator Overloading?

The member function `operatorΔ()` can only be invoked from an object via the dot operator, e.g., `p1.operatorΔ(p2)`, which is equivalent to `p1 Δ p2`. Clearly the left operand `p1` should be an object of that particular class. Suppose that we want to overload a binary operator such as `*` to multiply the object `p1` with an `int` literal, `p1*5` can be translated into `p1.operator*(5)`, but `5*p1` cannot be represented using member function. One way to deal with this problem is only allow user to write `p1*5` but not `5*p1`, which is not user friendly and break the rule of commutativity. Another way is to use a non-member function, which does not invoke through an object and dot operator, but through the arguments provided. For example, `5*p1` could be translated to `operator+(5, p1)`.

In brief, you cannot use member function to overload an operator if the left operand is not an object of that particular class.

3.2 "`friend`" Functions

A regular non-member function cannot directly access the private data of the objects given in its arguments. A special type of function, called `friend`s, are allowed to access the private data.

A "friend" function of a class, marked by the keyword `friend`, is a function defined outside the class, yet its argument of that class has unrestricted access to all the class members (`private`, `protected` and `public` data members and member functions). Friend functions can enhance the performance, as they eliminate the need of calling public member functions to access the private data members.

3.3 Example: Overloading << and >> Operators of `Point` class using non-member `friend` Functions

In this example, we shall overload << and >> operators to support stream insertion and extraction of `Point` objects, i.e., `cout << aPoint`, and `cin >> aPoint`. Since the left operand is not a `Point` object (`cout` is an `ostream` object and `cin` is an `istream` object), we cannot use member function, but need to use non-member function for operator overloading. We shall make these functions `friend`s of the `Point` class, to allow them to access the private data members directly for enhanced performance.

`Point.h`

```
1  /* The Point class Header file (Point.h) */
2  #ifndef POINT_H
3  #define POINT_H
4
5  #include <iostream>
6
7  // Class Declaration
8  class Point {
9  private:
10    int x, y;
11
12 public:
13    Point(int x = 0, int y = 0);
14    int getX() const; // Getters
15    int getY() const;
16    void setX(int x); // Setters
17    void setY(int y);
18
19    friend std::ostream & operator<<(std::ostream & out, const Point & point);
20    friend std::istream & operator>>(std::istream & in, Point & point);
21 };
22
23 #endif
```

Program Notes:

- Friends are neither `public` or `private`, and can be listed anywhere within the class declaration.
- The `cout` and `cin` need to be passed into the function by reference, so that the function accesses the `cout` and `cin` directly (instead of a clone copy by value).
- We return the `cin` and `cout` passed into the function by reference too, so as to support cascading operations. For example, `cout << p1 << endl` will be interpreted as `(cout << p1) << endl`.
- In <<, the reference parameter `Point` is declared as `const`. Hence, the function cannot modify the `Point` object. On the other hand, in >>, the `Point` reference is non-const, as it will be modified to keep the input.
- We use fully-qualified name `std::istream` instead of placing a "`using namespace std;`" statement in the header. It is because this header could be included in many files, which would include the `using` statement too and may not be desirable.

Point.cpp

```
1  /* The Point class Implementation file (Point.cpp) */
2  #include <iostream>
3  #include "Point.h"
4  using namespace std;
5
6  // Constructor - The default values are specified in the declaration
7  Point::Point(int x, int y) : x(x), y(y) { } // using member initializer list
8
9  // Getters
10 int Point::getX() const { return x; }
11 int Point::getY() const { return y; }
12
13 // Setters
14 void Point::setX(int x) { this->x = x; }
15 void Point::setY(int y) { this->y = y; }
16
17 ostream & operator<<(ostream & out, const Point & point) {
18    out << "(" << point.x << "," << point.y << ")";  // access private data
19    return out;
20 }
21
22 istream & operator>>(istream & in, Point & point) {
23    cout << "Enter x and y coord: ";
24    in >> point.x >> point.y;  // access private data
25    return in;
26 }
```

Program Notes:

- The function definition does not require the keyword `friend`, and the *`ClassName`*`::` scope resolution qualifier, as it does not belong to the class.
- The `operator<<()` function is declared as a friend of `Point` class. Hence, it can access the private data members `x` and `y` of its argument `Point` directly. `operator<<()` function is NOT a friend of `ostream` class, as there is no need to access the private member of `ostream`.
- Instead of accessing private data member `x` and `y` directly, you could use public member function `getX()` and `getY()`. In this case, there is no need to declare `operator<<()` as a friend of the `Point` class. You could simply declare a regular function prototype in the header.

  ```
  // Function prototype
  ostream & operator<<(ostream & out, const Point & point);

  // Function definition
  ostream & operator<<(ostream & out, const Point & point) {
     out << "(" << point.getX() << "," << point.getY() << ")";
     return out;
  }
  ```

  Using `friend` is recommended, as it enhances performance. Furthermore, the overloaded operator becomes part of the extended public interface of the class, which helps in ease-of-use and ease-of-maintenance.

TestPoint.cpp

```
1  #include <iostream>
2  #include "Point.h"
3  using namespace std;
4
5  int main() {
```

```
   Point p1(1, 2), p2;

   // Using overloaded operator <<
   cout << p1 << endl;    // support cascading
   operator<<(cout, p1);  // same as cout << p1
   cout << endl;

   // Using overloaded operator >>
   cin >> p1;
   cout << p1 << endl;
   operator>>(cin, p1);  // same as cin >> p1
   cout << p1 << endl;
   cin >> p1 >> p2;      // support cascading
   cout << p1 << endl;
   cout << p2 << endl;
}
```

The overloaded >> and << can also be used for file input/output, as the file IO stream `ifstream/ofstream` (in `fstream` header) is a subclass of `istream/ostream`. For example,

```
#include <fstream>
#include "Point.h"
using namespace std;

int main() {
   Point p1(1, 2);

   ofstream fout("out.txt");
   fout << p1 << endl;

   ifstream fin("in.txt"); // contains "3 4"
   fin >> p1;
   cout << p1 << endl;
}
```

4. Overloading Binary Operators

All C++ operators are either *binary* (e.g., `x + y`) or *unary* (e.g. `!x`, `-x`), with the exception of *tenary* conditional operator (`? :`) which cannot be overloaded.

Suppose that we wish to overload the binary operator `==` to compare two `Point` objects. We could do it as a *member function* or *non-member function*.

1. To overload as a *member function*, the declaration is as follows:

   ```
   class Point {
   public:
      bool operator==(const Point & rhs) const;  // p1.operator==(p2)
      ......
   };
   ```

   The compiler translates "`p1 == p2`" to "`p1.operator==(p2)`", as a member function call of object `p1`, with argument `p2`.

   Member function can only be used if the left operand is an object of that particular class.

2. To overload as a *non-member function*, which is often declared as a `friend` to access the private data for enhanced performance, the declaration is as follows:

   ```
   class Point {
      friend bool operator==(const Point & lhs, const Point & rhs); // operator==(p1,p2)
   ```

```
   ......
};
```

The compiler translates the expression "`p1 == p2`" to "`operator==(p1, p2)`".

5. Overloading Unary Operators

Most of the unary operators are prefix operators, e.g., `!x`, `-x`. Hence, prefix is the norm for unary operators. However, unary increment and decrement come in two forms: prefix (`++x`, `--x`) and postfix (`x++`, `x--`). We to a mechanism to differentiate the two forms.

5.1 Unary Prefix Operator

Example of unary prefix operators are `!x`, `-x`, `++x` and `--x`. You could do it as a non-member function as well as member function. For example, to overload the prefix increment operator `++`:

1. To overload as a non-member `friend` function:

```
class Point {
   friend Point & operator++(Point & point);
   ......
};
```

   The compiler translates "`++p`" to "`operator++(p)`".

2. To overload as a member function:

```
class Point {
public:
   Point & operator++();  // this Point
   ......
};
```

   The compiler translates "`++p`" to "`p.operator++()`".

You can use either member function or non-member friend function to overload unary operators, as their only operand shall be an object of that class.

5.2 Unary Postfix Operator

The unary increment and decrement operators come in two forms: prefix (`++x`, `--x`) and postfix (`x++`, `x--`). Overloading postfix operators (such as `x++`, `x--`) present a challenge. It ought to be differentiated from the prefix operator (`++x`, `--x`). A "dummy" argument is therefore introduced to indicate postfix operation as shown below. Take note that postfix `++` shall save the old value, perform the increment, and then return the saved value by value.

1. To overload as non-member `friend` function:

```
class Point {
   friend const Point operator++(Point & point, int dummy);
};
```

   The compiler translates "`pt++`" to "`operator++(pt, 0)`". The `int` argument is strictly a *dummy value* to differentiate prefix from postfix operation.

2. To overload as a member function:

```
class Point {
public:
   const Point operator++(int dummy);  // this Point
```

```
   ......
};
```

The compiler translates "pt++" to "pt.operator++(0)".

5.3 Example: Overloading Prefix and Postfix ++ for the Counter Class

Counter.h

```
/* The Counter class Header file (Counter.h) */
#ifndef COUNTER_H
#define COUNTER_H
#include <iostream>

class Counter {
private:
   int count;
public:
   Counter(int count = 0);   // Constructor
   int getCount() const;     // Getters
   void setCount(int count); // Setters
   Counter & operator++();              // ++prefix
   const Counter operator++(int dummy); // postfix++

   friend std::ostream & operator<<(std::ostream & out, const Counter & counter);
};

#endif
```

Program Notes:

- The prefix function returns a reference to this instance, to support chaining (or cascading), e.g., ++++c as ++(++c). However, the return reference can be used as lvalue with unexpected operations (e.g., ++c = 8).
- The postfix function returns a const object by value. A const value cannot be used as lvalue. This prevents chaining such as c++++. Although it would be interpreted as (c++)++. However, (c++) does not return this object, but an temporary object. The subsequent ++ works on the temporary object.
- Both prefix and postfix functions are non-const, as they modify the data member count.

Counter.cpp

```
/* The Counter class Implementation file (Counter.cpp) */
#include "Counter.h"
#include <iostream>
using namespace std;

// Constructor - The default values are specified in the declaration
Counter::Counter(int c) : count(c) { } // using member initializer list

// Getters
int Counter::getCount() const { return count; }

// Setters
void Counter::setCount(int c) { count = c; }

// ++prefix, return reference of this
Counter & Counter::operator++() {
   ++count;
   return *this;
}
```

```
20
21  // postfix++, return old value by value
22  const Counter Counter::operator++(int dummy) {
23     Counter old(*this);
24     ++count;
25     return old;
26  }
27
28  // Overload stream insertion << operator
29  ostream & operator<<(ostream & out, const Counter & counter) {
30     out << counter.count;
31     return out;
32  }
```

Program Notes:

- The prefix function increments the `count`, and returns this object by reference.
- The postfix function saves the old value (by constructing a new instance with this object via the copy constructor), increments the `count`, and return the saved object by value.
- Clearly, postfix operation on object is less efficient than the prefix operation, as it create a temporary object. If there is no subsequent operation that relies on the output of prefix/postfix operation, use prefix operation.

`TestCounter.cpp`

```
1   #include "Counter.h"
2   #include <iostream>
3   using namespace std;
4
5   int main() {
6      Counter c1;
7      cout << c1 << endl;     // 0
8      cout << ++c1 << endl;   // 1
9      cout << c1 << endl;     // 1
10     cout << c1++ << endl;   // 1
11     cout << c1 << endl;     // 2
12     cout << ++++c1 << endl; // 4
13  // cout << c1++++ << endl; // error caused by const return value
14  }
```

Program Notes:

- Take note of the difference in `cout << c1++` and `cout << ++c1`. Both prefix and postfix operators work as expected.
- `++++c1` is allowed and works correctly. `c1++++` is disallowed, because it would produce incorrect result.

6. Example: Putting them together in `Point` Class

This example overload binary operator << and >> as non-member functions for stream insertion and stream extraction. It also overload unary ++ (postfix and prefix) and binary += as member function; and +, += operators.

`Point.h`

```
1   /* The Point class Header file (Point.h) */
2   #ifndef POINT_H
3   #define POINT_H
4   #include <iostream>
5
6   class Point {
7   private:
```

```
   int x, y;

public:
   explicit Point(int x = 0, int y = 0);
   int getX() const;
   int getY() const;
   void setX(int x);
   void setY(int y);
   Point & operator++();              // ++prefix
   const Point operator++(int dummy); // postfix++
   const Point operator+(const Point & rhs) const; // Point + Point
   const Point operator+(int value) const;           // Point + int
   Point & operator+=(int value);           // Point += int
   Point & operator+=(const Point & rhs); // Point += Point

   friend std::ostream & operator<<(std::ostream & out, const Point & point); // out << point
   friend std::istream & operator>>(std::istream & in, Point & point);        // in >> point
   friend const Point operator+(int value, const Point & rhs); // int + Point
};

#endif
```

Point.cpp

```
/* The Point class Implementation file (Point.cpp) */
#include "Point.h"
#include <iostream>
using namespace std;

// Constructor - The default values are specified in the declaration
Point::Point(int x, int y) : x(x), y(y) { }

// Getters
int Point::getX() const { return x; }
int Point::getY() const { return y; }

// Setters
void Point::setX(int x) { this->x = x; }
void Point::setY(int y) { this->y = y; }

// Overload ++Prefix, increase x, y by 1
Point & Point::operator++() {
   ++x;
   ++y;
   return *this;
}

// Overload Postfix++, increase x, y by 1
const Point Point::operator++(int dummy) {
   Point old(*this);
   ++x;
   ++y;
   return old;
}

// Overload Point + int. Return a new Point by value
const Point Point::operator+(int value) const {
   return Point(x + value, y + value);
}

// Overload Point + Point. Return a new Point by value
```

```
38  const Point Point::operator+(const Point & rhs) const {
39     return Point(x + rhs.x, y + rhs.y);
40  }
41
42  // Overload Point += int. Increase x, y by value
43  Point & Point::operator+=(int value) {
44     x += value;
45     y += value;
46     return *this;
47  }
48
49  // Overload Point += Point. Increase x, y by rhs
50  Point & Point::operator+=(const Point & rhs) {
51     x += rhs.x;
52     y += rhs.y;
53     return *this;
54  }
55
56  // Overload << stream insertion operator
57  ostream & operator<<(ostream & out, const Point & point) {
58     out << "(" << point.x << "," << point.y << ")";
59     return out;
60  }
61
62  // Overload >> stream extraction operator
63  istream & operator>>(istream & in, Point & point) {
64     cout << "Enter x and y coord: ";
65     in >> point.x >> point.y;
66     return in;
67  }
68
69  // Overload int + Point. Return a new point
70  const Point operator+(int value, const Point & rhs) {
71     return rhs + value;  // use member function defined above
72  }
```

TestPoint.cpp

```
 1  #include <iostream>
 2  #include "Point.h"
 3  using namespace std;
 4
 5  int main() {
 6     Point p1(1, 2);
 7     cout << p1 << endl;   // (1,2)
 8
 9     Point p2(3,4);
10     cout << p1 + p2 << endl; // (4,6)
11     cout << p1 + 10 << endl; // (11,12)
12     cout << 20 + p1 << endl; // (21,22)
13     cout << 10 + p1 + 20 + p1 << endl; // (32,34)
14
15     p1 += p2;
16     cout << p1 << endl; // (4,6)
17     p1 += 3;
18     cout << p1 << endl; // (7,9)
19
20     Point p3;  // (0,0)
21     cout << p3++ << endl; // (0,0)
22     cout << p3 << endl;   // (1,1)
```

```
   cout << ++p3 << endl; // (2,2)
}
```

7. Implicit Conversion via Single-argument Constructor & Keyword "explicit"

In C++, a *single-argument* constructor can be used to implicitly convert a value to an object. For example,

```
#include <iostream>
using namespace std;

class Counter {
private:
   int count;
public:
   Counter(int c = 0) : count(c) { }
         // A single-argument Constructor which takes an int
         // It can be used to implicitly convert an int to a Counter object
   int getCount() const { return count; }    // Getter
   void setCount(int c) { count = c; } // Setter
};

int main() {
   Counter c1; // Declare an instance and invoke default constructor
   cout << c1.getCount() << endl;  // 0

   c1 = 9;
     // Implicit conversion
     // Invoke single-argument constructor Counter(9) to construct a temporary object.
     // Then copy into c1 via memberwise assignment.
   cout << c1.getCount() << endl;  // 9
}
```

This implicit conversion can be confusing. C++ introduces a keyword "`explicit`" to disable implicit conversion. Nonetheless, you can still perform explicit conversion via type cast operator. For example,

```
#include <iostream>
using namespace std;

class Counter {
private:
   int count;
public:
   explicit Counter(int c = 0) : count(c) { }
      // Single-argument Constructor
      // Use keyword "explicit" to disable implicit automatic conversion in assignment
   int getCount() const { return count; }    // Getter
   void setCount(int c) { count = c; } // Setter
};

int main() {
   Counter c1; // Declare an instance and invoke default constructor
   cout << c1.getCount() << endl;  // 0

// Counter c2 = 9;
     // error: conversion from 'int' to non-scalar type 'Counter' requested

   c1 = (Counter)9;  // Explicit conversion via type casting operator
```

```
23      cout << c1.getCount() << endl;  // 9
24  }
```

8. 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
10  class MyComplex {
11  private:
12     double real, imag;
13
14  public:
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 + c2
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
5  MyComplex::MyComplex (double r, double i) : real(r), imag(i) { }
6
7  // Overloading += operator for c1 += c2
8  MyComplex & 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)
15 MyComplex & MyComplex::operator+= (double value) {
16    real += value;
17    return *this;
18 }
19
20 // Overload prefix increment operator ++c (real part)
21 MyComplex & MyComplex::operator++ () {
22   ++real;   // increment real part only
```

```
  return *this;
}

// Overload postfix increment operator c++ (real part)
const MyComplex MyComplex::operator++ (int dummy) {
   MyComplex saved(*this);
   ++real;  // increment real part only
   return saved;
}

// Overload comparison operator c1 == c2
bool MyComplex::operator== (const MyComplex & rhs) const {
   return (real == rhs.real && imag == rhs.imag);
}

// Overload comparison operator c1 != c2
bool MyComplex::operator!= (const MyComplex & rhs) const {
   return !(*this == rhs);
}

// Overload stream insertion operator out << c (friend)
std::ostream & operator<< (std::ostream & out, const MyComplex & c) {
   out << '(' << c.real << ',' << c.imag << ')';
   return out;
}

// Overload stream extraction operator in >> c (friend)
std::istream & operator>> (std::istream & in, MyComplex & c) {
   double inReal, inImag;
   char inChar;
   bool validInput = false;
   // Input shall be in the format "(real,imag)"
   in >> inChar;
   if (inChar == '(') {
      in >> inReal >> inChar;
      if (inChar == ',') {
         in >> inImag >> inChar;
         if (inChar == ')') {
            c = MyComplex(inReal, inImag);
            validInput = true;
         }
      }
   }
   if (!validInput) in.setstate(std::ios_base::failbit);
   return in;
}

// Overloading + operator for c1 + c2
const MyComplex operator+ (const MyComplex & lhs, const MyComplex & rhs) {
   MyComplex result(lhs);
   result += rhs;  // uses overload +=
   return result;
   // OR return MyComplex(lhs.real + rhs.real, lhs.imag + rhs.imag);
}

// Overloading + operator for c + double
const MyComplex operator+ (const MyComplex & lhs, double value) {
   MyComplex result(lhs);
   result += value;  // uses overload +=
   return result;
}
```

```
84
85  // Overloading + operator for double + c
86  const 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
6   int 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;
```

```
   } else {
      std::cerr << "Invalid input" << std::endl;
   }
   return 0;
}
```

9. Dynamic Memory Allocation in Object

If you dynamically allocate memory in the constructor, you need to provide your own destructor, copy constructor and assignment operator to manage the dynamically allocated memory. The defaults provided by the C++ compiler do not work for dynamic memory.

9.1 Example: `MyDynamicArray`

`MyDynamicArray.h`

```
/*
 * The MyDynamicArray class header (MyDynamicArray.h)
 * A dynamic array of double elements
 */
#ifndef MY_DYNAMIC_ARRAY_H
#define MY_DYNAMIC_ARRAY_H

#include <iostream>

class MyDynamicArray {
private:
   int size_;   // size of array
   double * ptr;  // pointer to the elements

public:
   explicit MyDynamicArray (int n = 8);         // Default constructor
   explicit MyDynamicArray (const MyDynamicArray & a); // Copy constructor
   MyDynamicArray (const double a[], int n);    // Construct from double[]
   ~MyDynamicArray();                           // Destructor

   const MyDynamicArray & operator= (const MyDynamicArray & rhs); // Assignment a1 = a2
   bool operator== (const MyDynamicArray & rhs) const;     // a1 == a2
   bool operator!= (const MyDynamicArray & rhs) const;     // a1 != a2

   double operator[] (int index) const;  // a[i]
   double & operator[] (int index);      // a[i] = x

   int size() const { return size_; }    // return size of array

   // friends
   friend std::ostream & operator<< (std::ostream & out, const MyDynamicArray & a); // out << a
   friend std::istream & operator>> (std::istream & in, MyDynamicArray & a);        // in >> a
};

#endif
```

Program Notes:

- In C++, the you cannot use the same name for a data member and a member function. As I would like to have a public function called `size()`, which is consistent with the C++ STL, I named the data member `size_` with a trailing underscore, following C++'s best practices. Take note that leading underscore(s) are used by C++ compiler for its internal variables (e.g., _xxx for data members and __xxx for local variables).

- As we will be dynamically allocating memory in the constructor, we provide our own version of destructor, copy constructor and assignment operator to manage the dynamically allocated memory. The defaults provided by the C++ compiler do not work on dynamic memory.
- We provide 3 constructors: a default constructor with an optional size, a copy constructor to construct an instance by copying another instance, and a construct to construct an instance by copying from a regular array.
- We provide 2 version of indexing operators: one for read operation (e.g., `a[i]`) and another capable of write operation (e.g., `a[i] = x`). The read version is declared as a `const` member function; whereas the write version return a reference to the element, which can be used as *lvalue* for assignment.

MyDynamicArray.cpp

```
1  /* The MyDynamicArray class implementation (MyDynamicArray.cpp) */
2  #include <stdexcept>
3  #include "MyDynamicArray.h"
4
5  // Default constructor
6  MyDynamicArray::MyDynamicArray (int n) {
7     if (n <= 0) {
8        throw std::invalid_argument("error: size must be greater then zero");
9     }
10
11    // Dynamic allocate memory for n elements
12    size_ = n;
13    ptr = new double[size_];
14    for (int i = 0; i < size_; ++i) {
15       ptr[i] = 0.0;  // init all elements to zero
16    }
17 }
18
19 // Override the copy constructor to handle dynamic memory
20 MyDynamicArray::MyDynamicArray (const MyDynamicArray & a) {
21    // Dynamic allocate memory for a.size_ elements and copy
22    size_ = a.size_;
23    ptr = new double[size_];
24    for (int i = 0; i < size_; ++i) {
25       ptr[i] = a.ptr[i];  // copy each element
26    }
27 }
28
29 // Construct via a built-in double[]
30 MyDynamicArray::MyDynamicArray (const double a[], int n) {
31    // Dynamic allocate memory for a.size_ elements and copy
32    size_ = n;
33    ptr = new double[size_];
34    for (int i = 0; i < size_; ++i) {
35       ptr[i] = a[i];  // copy each element
36    }
37 }
38
39 // Override the default destructor to handle dynamic memory
40 MyDynamicArray::~MyDynamicArray() {
41    delete[] ptr;  // free dynamically allocated memory
42 }
43
44 // Override the default assignment operator to handle dynamic memory
45 const MyDynamicArray & MyDynamicArray::operator= (const MyDynamicArray & rhs) {
46    if (this != &rhs) {  // no self assignment
47       if (size_ != rhs.size_) {
48          // reallocate memory for the array
```

```
49          delete [] ptr;
50          size_ = rhs.size_;
51          ptr = new double[size_];
52       }
53       // Copy elements
54       for (int i = 0; i < size_; ++i) {
55          ptr[i] = rhs.ptr[i];
56       }
57    }
58    return *this;
59 }
60
61 // Overload comparison operator a1 == a2
62 bool MyDynamicArray::operator== (const MyDynamicArray & rhs) const {
63    if (size_ != rhs.size_) return false;
64
65    for (int i = 0; i < size_; ++i) {
66       if (ptr[i] != rhs.ptr[i]) return false;
67    }
68    return true;
69 }
70
71 // Overload comparison operator a1 != a2
72 bool MyDynamicArray::operator!= (const MyDynamicArray & rhs) const {
73    return !(*this == rhs);
74 }
75
76 // Indexing operator - Read
77 double MyDynamicArray::operator[] (int index) const {
78    if (index < 0 || index >= size_) {
79       throw std::out_of_range("error: index out of range");
80    }
81    return ptr[index];
82 }
83
84 // Indexing operator - Writable a[i] = x
85 double & MyDynamicArray::operator[] (int index) {
86    if (index < 0 || index >= size_) {
87       throw std::out_of_range("error: index out of range");
88    }
89    return ptr[index];
90 }
91
92 // Overload stream insertion operator out << a (as friend)
93 std::ostream & operator<< (std::ostream & out, const MyDynamicArray & a) {
94    for (int i = 0; i < a.size_; ++i) {
95       out << a.ptr[i] << ' ';
96    }
97    return out;
98 }
99
100 // Overload stream extraction operator in >> a (as friend)
101 std::istream & operator>> (std::istream & in, MyDynamicArray & a) {
102    for (int i = 0; i < a.size_; ++i) {
103       in >> a.ptr[i];
104    }
105    return in;
106 }
```

Program Notes:

- Constructor: [TODO]
- Copy Constructor:
- Assignment Operator:
- Indexing Operator:

TestMyDynamicArray.cpp

```
/* Test Driver for MyDynamicArray class (TestMyDynamicArray.cpp) */
#include <iostream>
#include <iomanip>
#include "MyDynamicArray.h"

int main() {
   std::cout << std::fixed << std::setprecision(1) << std::boolalpha;

   MyDynamicArray a1(5);
   std::cout << a1 << std::endl;  // 0.0 0.0 0.0 0.0 0.0
   std::cout << a1.size() << std::endl;  // 5

   double d[3] = {1.1, 2.2, 3.3};
   MyDynamicArray a2(d, 3);
   std::cout << a2 << std::endl; // 1.1 2.2 3.3

   MyDynamicArray a3(a2);   // Copy constructor
   std::cout << a3 << std::endl; // 1.1 2.2 3.3

   a1[2] = 8.8;
   std::cout << a1[2] << std::endl;  // 8.8
// std::cout << a1[22] << std::endl; // error: out_of_range

   a3 = a1;
   std::cout << a3 << std::endl; // 0.0 0.0 8.8 0.0 0.0

   std::cout << (a1 == a3) << std::endl;  // true
   std::cout << (a1 == a2) << std::endl;  // false

   const int SIZE = 3;
   MyDynamicArray a4(SIZE);
   std::cout << "Enter " << SIZE << " elements: ";
   std::cin >> a4;
   if (std::cin.good()) {
      std::cout << a4 << std::endl;
   } else {
      std::cerr << "Invalid input" << std::endl;
   }
   return 0;
}
```

Link to "C++ References & Resources"

Latest version tested: Cygwin/MinGW GCC 4.6.2
Last modified: May, 2013

Feedback, comments, corrections, and errata can be sent to Chua Hock-Chuan (ehchua@ntu.edu.sg) | HOME