Modern C++ - Operators

14. User defined operators

A good and clear interface is in the primary importance when defining a class. As documentation is frequently behind the actual code the only real information about it is the source itself. Suppose one want to use the Date class we defined in the previous chapter. The natural way would be something like the following:

```
 1 #include <iostream>     // standard header files
 2 #include "date.h"       // date.h for date class
 3
 4 int main()
 5 {
 6   date d1( 2016, 4);  // this should be 2016.04.01
 7   date d2 = d1;       // copy constructor,
 8   d2 += 40;           // add 40 days
 9   d1 = d2;            // assignment, now d1 is 2016.05.11
10
11   std::cout << "d1++ == " << d3++ << std::endl; // 2016.05.11
12   std::cout << " d1  == " <<  d3  << std::endl; // 2016.05.12
13   std::cout << "++d2 == " << ++d3 << std::endl; // 2016.05.12
14   std::cout << " d2  == " <<  d3  << std::endl; // 2016.05.12
15
16   d3.setDate(2016,3,1);
17   std::cout << "d3 == " << d3 << std::endl; // 2016.03.1
18   return 0;
19 }
```

This is more concise, natural and understandable than the interface we provided earlier for the Date class.

Overloaded operators

User defined operators can be implemented as *member functions* or as *namespace functions*. However, not all operators can be non-member: the *assignment* operators (**operator=**, **operator+=** etc.), the *function call* operator (**operator()**), the *index* operator (**operator[]**) and the *array-pointer* operator (**operator->**) must be defined as member function.

The *scope* operator (**::**) and *dot* operator (**.**) can not be overloaded at all.

When an operator is overloaded as a member, its leftmost operand will be the object to call it. When it is a member function, the arguments are just passed to the function in an ordinary way:

```
// when operator+ and operator~ are a member functions:
a+b  ->  a.operator+(b)
~a   ->  a.operator~()

// when operator+ and operator~ are namespace functions:
a+b  ->  operator+(a,b)
~a   ->  operator~(a)
```

Overloading some operators can be misleading. Logical operators (**||** and **&&**) are *shortcut* operators, i.e. they strictly evaluate first the left operand and then – when it is required – evaluates the right operand. However, when the left operand is true in logical *or*, or false in logical *and*, the right hand side operand is not evaluated at all. This is not possible when the user defines his own operators as *all* operads are evaluated before passing them to the function (here the user defined logical operator). the same problem stands when one overloads the comma (**,**) operator.

We can still use the other operators. We declare the appropriate operators in the class header:

```
 1 class Date
 2 {
 3 public:
 4   // public methods as earlier ...
 5
 6   Date& operator++();       // pre-fix increment
 7   Date  operator++(int);    // post-fix increment
 8   Date& operator+=( int n); // add n days
 9   /* operator--(), ... */
10 private:
11   void checkDate( int y, int m, int d);
12   int year;
13   int month;
14   int day;
15 };
```

As for the built-in *integral* types (**int**, etc.) we want to define separate *pre-fix*

increment and *post-fix increment* **++** operators with different semantics. The C++ convention is that the **operator++** with no parameter is the pre-fix increment, and the one with an **int** parameter is the post-fix version. The latter usually does not use its parameter which is optimized out from the actual code by the compiler.

In the cpp file we implement the three new methods:

```
 1 Date& Date::operator++()
 2 {
 3   return next();
 4 }
 5 Date Date::operator++(int)
 6 {
 7   Date old(*this);
 8   next();
 9   return old;
10 }
11 Date& Date::operator+=( int n)
12 {
13   return add(n);
14 }
```

We have to explain the difference between the implementations of the two increment functions. The semantics of the pre-fix increment is to *first* increment the value and *then* returning the new value. This is straightforward with the existing **next** function. The post-fix increment have to increment the value and *then* returning the old value. But how can we return with the old value? We have to store it somewhere. We use the variable **old** to store the old value. Then we increment the current value and at the end we return with the old value.

Pre-fix increment and **operator+=** returns with a reference to themselves. This is cheap and straightforward. However, as the post-fix operator returns with a local variable, that operator must return a value.

This behavioral difference exists even for the operators defined for the built-in types.

Operators on input and output streams

Input and output operators are natural targets of operator overloading. In the standard **<iostream>** header the standard library provides a number of overloding versions for these operators for the built-in and libraries. Some of these are implemented as member functions of the **std::ostream** and **std::istream** standard classes.

However, when we want to overload the input and output operators for a custom type we have to consider that a member operator is always called on the *left operand*. Here that would be an object of the standard input/output classes. Naturally, we have no right to modify any standard class. Therefore, our only option is to implement the input and output operators as namespace functions.

```
1 #include <iostream>  // to declre istream and ostream
2 class Date
3 {
4 public:
5   // public and private methods as earlier ...
6 };
7 // namespace operators for input/output
8 std::istream& operator>>( std::istream& is, Date& d);
9 std::ostream& operator<<( std::ostream& os, const Date &d);
```

Notice the signature and return value of the operators. The leftmost parameter is the stream object we target for i/o. It is always passed bey reference. Input/output streams are non-copyable classes (however they can be moved since C++11), therefore we pass them by reference. naturally, the second parameter for input operation will be changed on the call, so it is also passed by reference. The output operator, however, is not intending to change its parameter.

The return value in both cases is the reference to the leftmost parameter. The reason is that these operators can be – and frequently is – used in chain, therefore each calls should return the actual (an d perhaps modified) stream object for the next i/o operation:

```
Date d1(2016), d2(2015);
std::cout << d1 << d2;
// means:
operator<<( operator<<( std::cout, d1), d2);
```

The implementation of the operators are strateforward. Notice the return statement!

```
 1 std::istream& operator>>( std::istream& is, Date& d)
 2 {
 3   d.get( is);
 4   return is;  /* important to chained reads */
 5 }
 6 std::ostream& operator<<( std::ostream& os, const Date &d)
 7 {
```

```
 8   d.put( os);
 9   return os;    /* important to chained writes */
10 }
```

Namespace vs. member operator

We want to further improve our Date class. Some of the clients want to compare two dates, i.e. the *earlier* date is the *smaller*. As the comparision operators can be defined either as member operators or as namespace operators, we have two choices.

```
1 Date d1(2016), d2(2015);
2 d1 < d2     -->  d1.operator<(d2)  if member operator
3 d1 < d2     -->  operator<(d1,d2)  if namespace operator
```

Some of the object-oriented textbooks would suggest to use the member function choice, as that would better express the cohesion between the data structure and the operations executed on it.

In this case, the users can use the Date class in the following ways:

```
 1 #include <iostream>
 2 #include "date.h"
 3
 4 int main()
 5 {
 6   Date d1(2016), d2(2015);
 7   //...
 8   if ( d1 < d2 )          // ...  means: d1.operator<(d2);
 9   else if ( d1 < 2000 )   // .... means: d1.operator<(2000)
10   else if ( 2000 < d2 )    // SYNTAX ERROR! no 2000.operator<(d2)
11   // ...
12 };
```

In line 8. we compare two Date objects, the earlier date is the smaller. In line 9. we compare d1 to 2000.1.1. The reason of this behavior is that Date constructor accepts a single integer as a constructor parameter (the year), and defaults the month and day parameter to 1. A constructor of class **X** taking a single parameter of type **Y** can be considered as a conversion operator from **X** to **Y**. when passing the integer argument *2000*, the compiler first checks whether we have an overloaded version with exactly these parameters. When there is no match, but it is possible to convert the argument(s), then the automatic conversion will be happened.

However, we have got syntax error on **2000 < d1**. The reason is that automatic conversion happens only in function argument position, e.g. when we are passing an integer to the **Date::operator<(Date)**.

Therefore, we argue for implementing commutative operators as namespace functions. When we implement the less operation as a namespace function: **bool operator<(Date, Date)** the two operands are in a *simmetric* position, the same conversions will be applied for them. However, when we choose the member function implementation, the left operand will be in a special environment: no conversion will be applied for it.

To complete the task, first we declare the appropriate operators in date.h:

```
 1 class Date
 2 {
 3   //...
 4 };
 5 bool operator<(const Date &d1, const Date &d2);
 6 bool operator<=(const Date &d1, const Date &d2);
 7 bool operator>(const Date &d1, const Date &d2);
 8 bool operator>=(const Date &d1, const Date &d2);
 9 bool operator==(const Date &d1, const Date &d2);
10 bool operator!=(const Date &d1, const Date &d2);
```

And then we implement the methods in date.cpp:

```
 1 bool operator<( Date d1, Date d2)
 2 {
 3   return (d1.getYear() <  d2.getYear())
 4       || (d1.getYear() == d2.getYear()
 5                   && d1.getMonth() <  d2.getMonth())
 6       || (d1.getYear() == d2.getYear()
 7                   && d1.getMonth() == d2.getMonth()
 8                   && d1.getDay()   <  d2.getDay());
 9 }
10
11 bool operator==( date d1, date d2) { return !(d1<d2 || d2<d1); }
12 bool operator!=( date d1, date d2) { return d1<d2 || d2<d1; }
13 bool operator<=( date d1, date d2) { return !(d2<d1); }
14 bool operator>=( date d1, date d2) { return !(d1<d2); }
15 bool operator>( date d1, date d2)  { return d2<d1; }
```

To implement the other 5 operators using the less operator is a safety startegy: we hear can ensure the consistency between the operators. In this case we do not need to bother with efficiency: such trivial function bodies are optimized very well by the C++ compiler, we will hardly see the performance difference compared to the direct implementations not using the less operator.

Inline functions

We can improve the optimization possibilities defining the operators (and any other methods) in the date.h header file as *inline function*.

```
 1 class Date
 2 {
 3   //...
 4 };
 5 inline bool operator<(const Date &d1, const Date &d2)
 6 {
 7   return (d1.getYear() <  d2.getYear())
 8       || (d1.getYear() == d2.getYear()
 9                   && d1.getMonth() <  d2.getMonth())
10       || (d1.getYear() == d2.getYear()
11                   && d1.getMonth() == d2.getMonth()
12                   && d1.getDay()   <  d2.getDay());
13 }
14 inline bool operator==( date d1, date d2) { return !(d1<d2 || d2<d1); }
15 inline bool operator!=( date d1, date d2) { return d1<d2 || d2<d1; }
16 inline bool operator<=( date d1, date d2) { return !(d2<d1); }
17 inline bool operator>=( date d1, date d2) { return !(d1<d2); }
18 inline bool operator>( date d1, date d2)  { return d2<d1; }
```

C++ compilers use a large variety of optimizations. One of the most powerful tool is *inlining*, e.g. the compiler replaces the function call with the body of the called function. Calling a function is not always cheap: arguments should be copied to the stack, registers may have to be saved, etc. For certain small functions even the generated code of the function call is longer than the called function body itself. In these situations inlining may lead to significant performance gain.

When we place a function definition into a header file, the function body will be visible for the compiler in every place we call the function. in this case we must not repeat the function definition in the source (data.cpp) file. The **inline** keyword has two effects:

1. Give a hint for the compiler to inline the function. Note that inlining is not

mandatory for the compiler, e.g. function recursion, and other specific situations may restrict inlining (or make it partial). Also, when we set a function pointer addressing an inline function, the function must be generated as usual, and can be called via the function pointer.

2. The *linkage name* of the function will be specific to the compilation unit. When a function definition is given in a header included to multiple compilation units, the function will be compiled into each object files idependently. In a normal case, when we attempt to link these objects, we get *ambigous reference* error. To avoid this, every function instance in every different object files must have different linkage name.

Functions defined inside the class curly brackets are automatically inline. In this case we do not write the **inline** keyword.

```
1 class Date
2 {
3   int getYear() const { return year; }
4   // ...
5 };
```