

Chapter 6

6.1 Operator overloading

6.2 Class templates

6.3 Modular programming

Miscellaneous

6.1 Operator overloading

You have already seen how regular functions can be overloaded (Chapter 3, Section 3.6.2) and how constructors can be overloaded (Chapter 4, Section 4.4.1). C++, also lets you overload standard operators such as: `+`, `-`, `*`, `>`, `<`, `=`, `++`, `>>`, `<<`, etc. Overloading an operator lets you redefine an operator such that it can be applied to objects of your own data type and not only to basic data types such as integers. For example, if we want to compare two objects of type *Date* (defined in example 4.4.1) to find which one of them represents an earlier date, we would have to pass two objects of type *Date* to a function (defined by us), which will then compare the three data members (*day*, *month* and *year*) of each object and return the result. Instead, we can overload the less than operator, which will enable us to directly compare two objects of type *Date* in the form: `if (d1 < d2)...` where `d1` and `d2` are instances of type *Date*. To overload an operator we will need to write a method or a regular function that will redefine the operator to perform a certain action each time it is applied to certain types objects. The name of the function must be composed of the keyword **operator** followed by the operator we want to overload (e.g. **operator<**), followed by an argument list. It is a common practice to implement operator overloading as a method of the class type the operator will be applied on.

Example 6.1.1 – Overloading the < (less than) operator for class Date

```

class Date
{
private:
    int day;
    int month;
    int year;
public:
    Date ( ) { day=0,month=0,year=0; }
    Date (int d, int m, int y ) { day=d,month=m,year=y; }
    ~ Date ( ){};
    void set_date ( int d, int m, int y ) {day=d,month=m,year=y;}
    int get_day( ) {return day;}
    int get_month( ) { return month;}
    int get_year( ) { return year;}
    bool operator<( Date );           // “operator<” prototype
};

// “operator<” implementation
bool Date::operator <(Date d)
{
    int date1=get_year() * 365 + get_month() * 30 + get_day();
    int date2=d.get_year() * 365 + d.get_month() * 30 + d.get_day();

    if (date1<date2)
        return 1;
    else
        return 0;
}

```

You probably wonder why only one parameter is passed to the method that overloads the less than operator. After all the less than operator is a binary operator which is supposed to compare two items. The answer is that since the operator was implemented as a method, the first parameter (the one that will be left to the operator) is implied. That is, there is no need to explicitly specify it in the method. You may consider the first parameter as the object that invokes the **operator<** method. For this reason the value assigned to **date1** in the example above involves invoking methods without the need to explicitly specify the instance they belong to.

Example 6.1.2 - Using the less than (<) operator:

```

void main()
{
    Date date1(10,8,01);           // creating two objects of type Date and initializing
    Date date2(15,7,02);           // them by invoking the second constructor.

    if (date1<date2)               // date1 invokes the "operator<" method which
                                   // is passed date2 to determine if date1
                                   // precedes date2.

    cout<<"date1 precedes date2"<<endl;
}

```

Overloading the insertion (<<) I/O operator

The two I/O operators- *insertion* (<<) and *extraction* (>>), are frequently overloaded enabling us to input and output data belonging to objects of our own class type. For example, whenever we want to output the values of the data members stored in a *Date* object, i.e. output the date; we would have to write:

```
cout<< date1.get_month() <<"/"<< date1.get_day() <<"/"<< date1.get_year();
```

Instead, if we overload the *insertion* operator we could just write: `cout<<date1;` Because of compiler issues and the scope of the pre-defined `ostream`, `istream` objects supplied to us by the compiler, overloading the *insertion* and *extraction* operators must be implemented as global functions and not as methods of a particular class type. The function receives a reference to a stream (`istream` in the case of the extraction operator, and `ostream` in the case of the insertion operator), and a reference to an object we want to input or output. It then returns a reference to the same stream that is passed. The following function illustrates how the *insertion* operator can be overloaded enabling us to output objects of type *Date*.

```

ostream& operator <<(ostream& out, Date &d)
{
    return out<<"the date is: "<<d.get_month()<<"/"<<d.get_day()<<"/"<<d.get_year();
}

```

It is customary to put the implementation of the overloaded insertion and extraction operators in the same header file as the class definition they are applied to.

Example 6.1.3 - Using the << overloaded operator:

```

void main()
{
    Date date1(10,8,01);
    Date date2(15,7,02);

    cout<<date1;      // will out put 8/10/1
    cout<<date2;      // will out put 7/15/2
}

```

Overloadable operators:

Only the following operators can be overloaded:

+	-	*	/	%	^	&	
~	!	,	=	<	>	<=	>=
++	--	<<	>>	==	!=	&&	
+=	-=	*=	/=	%=	^=	&=	=
<<=	>>=	[]	()	->	->*	new	delete

However, some of these operators may only be overloaded as member functions within a class. This holds for the '=', the '[]', the '()' and the '->' operators. While others such as '<<' and '>>' should be overloaded as global functions.

6.2 Class Templates (not ready)**6.3 Modular Programming** (not ready)