5.1 Operator Overloading

In general, the term *overloading* means having multiple functions with the same name (but different parameter lists). For example, this can be used to add the usual mathematical operators for a user-defined class. Thus, one might have the prototype for a member function that returns a fraction that is the sum of the current fraction and another fraction:

\[
\text{Fraction operator+}(\text{const Fraction} \; \& \; \text{other}) \; \text{const};
\]

If the user has some code where two fractions are added, e.g. \( A+B \), then this member function is called on \( A \), with \( B \) as the argument. That is, the compiler changes \( A+B \) to \( A.\text{operator+}(B) \); In the actual code for the function, the data members of the first fraction are accessed directly; those of the second are accessed with \text{other}. notation.

5.2 Equality Testing

To allow one to test whether two objects are equal, one should provide a function that tests for equality. In C++, this is achieved by overloading the \( == \) function. The argument to the \( == \) function is a *reference* to another such object.

```cpp
class Foo {
    int bar;
    bool operator== ( const Foo &other ) const
    {
        return (bar == other.bar);
    }
};
```

Most binary operators are *left-to-right associative*. It follows that when in the calling function we have

\[
\text{Foo } X,Y;
\text{if( } X==Y \text{ )}
\]

the boolean condition invokes \( X.\text{operator==}(Y) \)
5.3 Inputting or Outputting a Class

Output of a class can be achieved by overloading the stream insertion operator \texttt{\textless\textless}. This is usually a separate \textit{global function} (that is, not a member function). In order to access the private variables of your class, you usually need to make it a \textit{friend} of your class (by adding its prototype inside the class).

```cpp
class Foo {
    private:
        int bar1, bar2;
        friend ostream &operator\textless\textless (ostream &, const Foo &);
    
    ostream &operator\textless\textless (ostream &out, const Foo &myFoo)
    {
        out \textless\textless myFoo.bar1 \textless:\textless myFoo.bar2 \textless\textless endl;
        return out;
    }
}
```

Note that the arguments are passed by reference, and the stream itself is returned by reference (so that the operator works with successive \texttt{\textless\textless}).

One can use the same approach to read an object from the user. Usually the user data is read into a string and then parsed internally. This is to handle malformed data without crashing.

5.4 Copying and Cloning

When a class is passed by value (into or out of a function), a copy is made using the \textit{copy constructor}.

Often the default compiler-inserted copy constructor is fine. This provides a \textit{shallow copy}—only the declared variables are copied. For example, if the class contains the header pointer to a linked list, the pointer will be copied, but both the header in the new object and the old object will point to the same \texttt{Node} in memory. This is usually not what you want. Instead a \textit{deep copy} produces a \textit{completely separate object}.

```cpp
class Foo
{
    private:
        Bar *barPtr;
    public:
        Foo( const Foo &other ) {
```
barPtr = new Bar( *(other.barPtr) );

};

You should assume the deep copy is required unless otherwise specified.

Note that the code A=B uses the assignment operator. There is a fundamental trio:

either the default copy constructor, assignment operator and destructor are all okay, or you need to provide all three.

We see how to create an assignment operator next.

5.5 Class Assignment

Suppose we have defined a class Foo. If we write:

    Foo *bar1 = new Foo();
    Foo *bar2 = bar1;

then the pointer bar2 points to the same instance of Foo that bar1 does. In particular, only one object exists.

If we write:

    Foo bar1, bar2;
    // changes to the two objects
    bar2 = bar1;

then bar2 is now a copy of bar1. Unless you specify otherwise, this is done by the default assignment operator, which is a shallow copy—only the declared variables are copied. For example, if Foo contains a head pointer to a linked list, the pointer will be copied, but both the head in bar1 and the one in bar2 will point to the same Node in memory. This is usually not what you want.

To create your own assignment operator, start with:

    Foo & operator= (const Foo &other) {
        // make copies of other's members and assign them to this object
        return *this;
    }

The this pointer always refers to the object on which the member function is being invoked. (The function has to return the object, so that A=B=C works.) Now, one should first deallocate the old stuff in the object using the delete command. However, if the user writes bar=bar, assigning an object to itself, this can cause a problem. Thus, one adds a test to avoid any changes occurring in this case:

    if( this != &other ) {
        //
Sample Code

We create a class called Fraction. Note that the fraction is stored in simplest form.

In what follows we have first the header file Fraction.h, then the implementation file Fraction.cpp, and then a sample program that uses the class TestFraction.cpp.

<table>
<thead>
<tr>
<th>Fraction.h</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fraction.cpp</td>
</tr>
<tr>
<td>TestFraction.cpp</td>
</tr>
</tbody>
</table>