

More on Classes

We discuss the problems of comparing, copying, passing, outputting, and destructing objects.

7.1 Pointers and References

In C++ functions, you can pass parameters by value or by address/reference. Pass-by-value uses a separate copy of the variable; changing this copy does not affect the variable in the calling function. Pass by value is inefficient if the object is large.

Pass-by-reference/address provides access to the original variable; changing the variable in the function does affect the variable in the calling function.

In C, pass-by-reference/address is achieved by pointers. This is still used. For example, we saw that arrays are implicitly passed this way. C++ introduced the idea of references or aliases. This is a simplified version of a pointer.

In C++, a function can return an object or a pointer to it. However, it is also common to have pass-by-reference arguments that are changed.

It is also possible to return a reference to an object. However, note that an object created using `new` exists in global memory; the `new` operator returns a pointer (and `malloc` and `calloc` are not needed). On the other other, an object created with a declaration exists in local memory; in particular it is automatically destroyed at the end of its scope. Thus one gets a compiler warning if one returns a reference to a local variable. Nevertheless, there are times when return a reference can be used. For example, a nice way to print an object is to overload the `<<` operator.

7.2 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:

```
Fraction operator+(const Fraction & other) const;
```

If the user has some code where two fractions are added, e.g. `A+B`, then this function is called on `A`, with `B` as the argument. In the actual code for the function, the data members of the first fraction are accessed direct; those of the second are accessed with `other.` notation.

7.3 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.

```
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

```
Foo X,Y;
if( X==Y )
```

the boolean condition invokes `X.operator==(Y)`

7.4 Outputting a Class

Output of a class can be achieved by overloading the stream insertion operator `<<`. This is usually a separate function which needs to be made a *friend* of your class: a friend has access to private variables and functions.

```
class Foo {
    private:
        int bar1,bar2;
    friend ostream &operator<< (ostream &, const Foo &);
};

ostream &operator<< (ostream &out, const Foo &myFoo)
{
    out << myFoo.bar1 << ":" << myFoo.bar2 << endl;
    return out;
}
```

Note that the arguments are passed by reference, and the stream itself is returned by reference (so that it works with successive `<<`).

7.5 Copying and Cloning

When a class is passed by value (into or out of a function), a copy is made using the *copy constructor*. Often the default compiler-inserted copy constructor is fine. However, if the class has pointers to other classes, then probably one should create a copy constructor oneself.

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

7.6 Destructors

In C++, anything that is created implicitly (such as passing by value or declaring a variable) is automatically deleted when no longer in use. However, anything created with a `new` command must be released to the system to avoid memory leaks. The `delete` command takes a pointer and deletes what the pointer points to.

A class should have a *destructor*; this has the name of the class preceded by a tilde, and is called to properly release memory. For example, our linked list classes will need a destructor. This frees up the nodes when the linked-list object is deleted.

7.7 Sample Code: Fraction.cpp

We create a class called `Fraction`. 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`. Note that the compiler is called by

```
g++ Fraction.cpp TestFraction.cpp
```

The fraction is stored in simplest form.

```
// Fraction.h - wdg 2009
#ifndef FRACTION_H
#define FRACTION_H

#include <iostream>
```

```

using namespace std;

class Fraction
{
public:
    Fraction(int whole);
    Fraction(int n,int d);
    Fraction(const Fraction & other);
    Fraction operator+(const Fraction & other) const;
    Fraction operator*(const Fraction & other) const;
    bool operator==(const Fraction & other ) const;

    friend ostream & operator<< (ostream & out, const Fraction & fraction);

private:
    int numer;
    int denom;
};

#endif



---


// Fraction.cpp - wdg 2009
#include "Fraction.h"

int gcd(int a,int b)
{
    if(b==0)
        return a;
    else
        return gcd (b, a%b) ;
}

Fraction::Fraction(int whole) : numer(whole), denom(1)
{
}

Fraction::Fraction(int n,int d)
{
    if(d<0) {

```

```

        d=-d; n=-n;
    }
    int h = gcd(n<0?-n:n,d);
    numer=n/h;
    denom=d/h;
}

Fraction::Fraction(const Fraction & other) : numer(other.numer), denom(other.denom)
{
}

ostream & operator<< (ostream & out, const Fraction & fraction)
{
    if(fraction.denom==1)
        out << fraction.numer;
    else
        out << fraction.numer << "/" << fraction.denom;
    return out;
}

Fraction Fraction::operator+(const Fraction & other) const
{
    int newNumer = numer*other.denom + denom*other.numer;
    int newDenom = denom * other.denom;
    return Fraction(newNumer,newDenom);
}

Fraction Fraction::operator*(const Fraction & other) const
{
    int newNumer = numer*other.numer;
    int newDenom = denom * other.denom;
    return Fraction(newNumer,newDenom);
}

bool Fraction::operator==(const Fraction & other) const
{
    return (numer==other.numer && denom==other.denom);
}

```

```

#include <iostream>

```

```
using namespace std;
#include "Fraction.h"

int main( )
{
    Fraction F(1), G(3,-5), H(7,28), I(H);
    F = F+G;
    G = I*H;
    cout << F << " " << G << " " << H << " " << I << endl;
    cout << boolalpha << (F==I);
    return 0;
}
```