Friend Functions and Friend Classes

Friend Classes

C++ allows you to declare another class to be a "friend" of the current class to make it easier to access variables. OOP purists have criticized this feature as weakening the principles of encapsulation and information hiding. You can do everything you might want to do without using the "friend" feature. For example, Java is not friendly. There is no friend feature. However, this feature can make some code easier to write (at least in the short term; arguably not in the long term). You may also see other code that uses friends, so it is good to know what it means.

Consider the following Employee class:

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

class Employee
{
 private:
   string name;
   double salary;
 public:
   Employee();
   Employee(string n, double s);
   string getName();
   double getSalary();
};

Employee::Employee() : name(""), salary(0) {}
Employee::Employee(string n, double s): name(n), salary(s) {}
string Employee::getName() { return name; }
double Employee::getSalary() { return salary; }
```

We could add more functions but that will suffice to keep it simple. In this case we are doing our standard encapsulation with private variables.

Now let's say we'd like to write a "Boss" class where the boss is the supervisor of some employee:

```
class Boss
{
 public:
   Boss();
   void giveRaise(double amount);
 private:
   Employee e;
};
```

```
Boss::Boss() {}

void Boss::giveRaise(double amount)
{
   e.salary = e.salary + amount;
   cout << e.salary << endl;
}
```

This code won't work because the Boss class is not allowed to access the salary variable in the Employee class. The standard fix is to write an accessor and mutator method/function for the Employee class so we can get and set the variable. For example, if we had a `setSalary(double newSalary)` method defined in the Employee class then in the `giveRaise` method we could call `getSalary` to get the current salary, add the raise amount, then use `setSalary` to set the new amount.

However, one can argue that a boss should intrinsically have access to private variables in the Employee class. If you agree, then the friend modifier allows this to happen. Inside the Employee class we designate that Boss is Employee's friend. It doesn't matter if we put it in the private or public areas:

```
class Employee
{
 private:
   friend class Boss;
   string name;
   double salary;
 public:
   Employee();
   Employee(string n, double s);
   string getName();
   double getSalary();
};
```

It's that simple, now the code will compile and the Boss class is allowed to access private variables and functions inside the Employee class.

Making a global function a friend

You can also specify a specific global function to be a friend, rather than an entire class. To do this use the "friend" keyword in front of the function declaration inside the class. This makes the function technically not a member of the class, so we have to pass any object that we want to work on as a parameter instead of invoking it with the dot operator. The following gives access to the "raiseSalary" function only:

```
class Employee
{
 private:
   string name;
   double salary;
   // This is a global function, not member function!
   friend void raiseSalary(double a, Employee &e);
 public:
```

```
   Employee();
   Employee(string n, double s);
   string getName();
   double getSalary();
};


// Note: No Employee::  because raiseSalary is NOT a member of the
// Employee class!  It is a "friend" as a global function
// Because of this, we have to pass the object as a parameter
// (e.g. raiseSalary(amt, empObj)
// instead of using it to invoke the function (e.g. empObj.raiseSalary)

void raiseSalary(double a, Employee &e)
{
  e.salary += a;  // Normally not allowed to access e.salary
}
```

If we wanted to call this function it would look like this, we have to specify the object as the second parameter. It is passed by reference so we can change the employee rather than send in a copy.

```
void Boss::giveRaise(double amount)
{
   raiseSalary(amount,e);
   cout << e.getSalary() << endl;
}
```

This looks a little funky because raiseSalary is really defined in the global namespace, not in the scope of the Employee object.

Operator Overloading – Overloading as a member function

One place where friend functions are commonly used is when we overload operators. This is a handy feature that is not yet possible in Java although it has been discussed for future versions.

In "normal" overload we can write multiple methods and C++ invokes the one that matches the parameter list. We have been doing this for constructors.

You can also overload operators, like == or * or +. This can be really useful when you want to run your own code to determine what equality means or what + should do when you "add" two classes together.

Let's return to our simple version of the Employee class:

```
class Employee
{
 private:
   string name;
   double salary;
 public:
```

```
   Employee();
   Employee(string n, double s);
   string getName();
   double getSalary();
};

Employee::Employee() : name(""), salary(0) {}
Employee::Employee(string n, double s): name(n), salary(s) {}
string Employee::getName() { return name; }
double Employee::getSalary() { return salary; }
```

What if we would like to compare two Employee objects? C++ will give an error message because we haven't defined how to compare to Employee objects:

```
 Employee e1("Bob",50);
 Employee e2("Bob",50);
 cout << (e1 == e2) << endl;
```

One we can define the == operator is as a member function or member operator. This treats the == like any other function that is a member of the class!

```
class Employee
{
 private:
   string name;
   double salary;
 public:
   Employee();
   Employee(string n, double s);
   string getName();
   double getSalary();
   bool operator==(Employee& e2);    // Overloaded == operator as member
};

// Note that bool is a member function, not a global function
// We don't need "friend" since it is a member function and has
// access to private variables
bool Employee::operator==( Employee& e2)
{
  // Use this-> to access the calling object
  return (this->name == e2.name);
}
```

When we use == on two Employee objects, the object on the left side of the == is the calling object, and the object on the right side of the == is the object passed in as e2:

```
 Employee e1("Bob",50);
 Employee e2("Bob",50);
 cout << (e1 == e2) << endl;
```

In this case, e1 becomes the calling object so access this->name inside the == operator is accessing e1.name and e2 becomes e2.name. In other words, the compiler translates (e1 == e2) into e1.==(e2).

This is a good way to overload operators for a class but it has a shortcoming. The problem is that the variable on the left hand side of == has to be an Employee object. If it was something else we are trying to typecast into Employee then it won't work. For example, if we overload the + operator and want to allow e1 + 10, then e1 + 10 would work but 10 + e1 would fail because 10 is not a class.

We can get around this problem by overloading the + operator as a global function instead.

Operator Overloading – Overloading as a global function

To demonstrating overloading an operator as a global function let's revert back to our original Employee class:

```
class Employee
{
 private:
   string name;
   double salary;
 public:
   Employee();
   Employee(string n, double s);
   string getName();
   double getSalary();
};

Employee::Employee() : name(""), salary(0) {}
Employee::Employee(string n, double s): name(n), salary(s) {}
string Employee::getName() { return name; }
double Employee::getSalary() { return salary; }
```

We can define a global operator == that operates on Employee objects like this:

```
bool operator==(Employee& e1, Employee& e2)
{
  return (e1.getName() == e2.getName());
}
```

The following code will now run and give the same result as when we overloaded == as a member function/operator. The object on the left side of == is passed in as e1 and the object on the right side of == is passed in as e2.

```
 Employee e1("Bob",50);
 Employee e2("Bob",50);
 cout << (e1 == e2) << endl;
```

Although we didn't need it in this example, it is not uncommon to make an overloaded operator a friend. This allows the operator to access private variables. If we wanted to make it a friend then our code would look like this:

```
class Employee
{
 private:
   string name;
   double salary;
 public:
   Employee();
   Employee(string n, double s);
   string getName();
   double getSalary();
   friend bool operator==(Employee&e1, Employee &e2);
};

bool operator==(Employee& e1, Employee& e2)
{
  // Since this is a friend we can access the private variables
  return (e1.name == e2.name);
}
```

Here is another example where we override the += operator so it adds together the salaries and stores the sum into the Employee on the left hand side:

```
class Employee
{
 private:
   string name;
   double salary;
 public:
   friend bool operator==(Employee& e1, Employee& e2);
   friend void operator+=(Employee& e1, Employee& e2);
   Employee();
   Employee(string n, double s);
   string getName();
   double getSalary();
};

void operator+=(Employee& e1, Employee& e2)
{
  e1.salary += e2.salary;
}

int main()
{
 Employee e1("Bob",50);
 Employee e2("Bill",50);
 e1 += e2;
 cout << e1.getSalary() << " " << e2.getSalary() << endl;   // 100 and 50
}
```

Sometimes you don't want to change an object's value. For example, you might want to add two items together but not change e1 or e2. Here is another example:

```
class Employee
{
 private:
   string name;
   double salary;
 public:
   friend bool operator==(Employee& e1, Employee& e2);
   friend void operator+=(Employee& e1, Employee& e2);
   friend Employee operator+(Employee& e1, Employee& e2);
   Employee();
   Employee(string n, double s);
   string getName();
   double getSalary();
};
```

```
Employee operator+(Employee& e1, Employee& e2)
{
  Employee temp;
  temp.name = e1.name + " and " + e2.name;
  temp.salary = e1.salary + e2.salary;
  return temp;
}

int main()
{
 Employee e1("Bob",50);
 Employee e2("Bill",50);
 Employee e3 = e1 + e2;
 cout << e3.getSalary() << " " << e3.getName() << endl;
  // Outputs "100 Bob and Bill"
}
```

There's a bit going on under the hood. When we invoke +, the temp employee gets the concatenated name and sum of the salaries. When it is returned, the temp Employee is copied (using C++'s semantics for the assignment operator, i.e. it copies the name and salary) to the e3 variable. We then output e3. e1 and e2 are unchanged.

Here is one more example where we overload the + operator so it takes a string and then an employee, and concatenates the name onto the employee. For example, "Jill" + e1 gives us a new employee with e1's salary but the name is now "Jill" + e1.name.

```
class Employee
{
 private:
   string name;
   double salary;
 public:
```

```
   friend bool operator==(Employee& e1, Employee& e2);
   friend void operator+=(Employee& e1, Employee& e2);
   friend Employee operator+(Employee& e1, Employee& e2);
   friend Employee operator+(string name, Employee &e);
   Employee();
   Employee(string n, double s);
   string getName();
   double getSalary();
};

Employee operator+(string name, Employee &e)
{
  Employee temp;
  temp.name = name + " and " + e.name;
  temp.salary = e.salary;
  return temp;
}

int main()
{
 Employee e1("Bob",50);
 Employee e2 = "Jill" + e1;
 cout << e2.getSalary() << " " << e2.getName() << endl;
  // Outputs "50 Jill and Bob"
}
```

Const Confusion

Often we might use the const keyword with an overloaded operator if we don't want an operand to change.

C++ lets us use the keyword const to make things constant and unchangeable, but it can be used in many different ways that can be confusing.

const int x = 3; and
int const x = 3;

both make x a constant that is an integer set to 3. You can't change x. In general, the thing to the left is what the const applies to, unless there is nothing there, in which case the const applies to the thing to the right.

Const can also be used for a return type. Consider this:

```
int& foo()
{
       static int x = 0;
       return x;
}
```

main:

```
cout << foo()++ << endl;
cout << foo() << endl;
```

This outputs 0 and then 1, because the ++ changes the return value which references the static variable. If you want to return a reference for efficiency purposes but don't want it changeable then you can make it const:

```
const int& foo()
```

Const can also be used with parameter passing. You might want to pass something by reference to save memory, but really don't want the parameter to be changed. In this case you can make it const:

```
void foo(const big_class &parameter)
```

Finally, const can be used in OOP to specify that a function can't change any instance (member) variables in the object. To do this stick const at the end of the function in the header and implementation file:

```
class Fraction {
            public:
                  void print() const;
…
void Fraction::print() const
{
      cout << m_numerator << "/" << m_denominator << endl;
}
```

You can combine these to be extra confusing!

```
const int& myfunction(const MyClass& parm) const
```