Friends and Inheritance

Permission levels

C++ has the following permission levels for instance variables and class member functions:

> **private:** only visible within a class
> **protected:** (not covered yet) visible to class and to extending classes
> **public:** visible anywhere the class is visible

At this point, the permission levels in C++ behave just like the permission levels in Java. However, there is an exception to the permission rules – *friend functions* and *friend classes*. By declaring a class or outside function to be your "friend", you give that class or function access to your private and protected instance variables and functions.

Friend functions

A *friend function* of a class is a function that:

- Can access that class's private and protected members
- Is declared inside that class's definition with the keyword "`friend`"
- Is NOT a member of that class

Example

To see how the friend modifier works, let's revisit the `Vector3` class:

```
class Vector3 {
     private:
          int x, y, z;
     public:
          Vector3(void);
          Vector3(int, int, int);
};

double length(const Vector3&);
```

Here, we have defined an outside function, `length`, that takes a `Vector3` reference and returns its length. The trouble is that since the `length` function is not part of the `Vector3` class, it can't access the private variables `x`, `y`, and `z`. Instead, let's make `length` a friend of `Vector3`:

```
class Vector3 {
     private:
          int x, y, z;
     public:
          Vector3(void);
```

```
        Vector3(int, int, int);
        friend double length(const Vector3&);
};
```

Notice that the `length` function is now declared inside the `Vector3` class, beginning with the modifier "`friend`". However, it is NOT a member of `Vector3`.

Notice also that the `length` function takes a `Vector3` reference as an argument. In fact, a friend function will ALWAYS take an object with the type of its friend class – otherwise, there would be no point in declaring the friend relation.

When implementing a friend function, put it in the same `.cpp` file as the class it's friends with. However, the friend function is NOT part of the class, so leave off the scope operator. Here's the implementation of `Vector3` and `length`:

```
Vector3::Vector3(void) {
    x = 0;
    y = 0;
    z = 0;
}

Vector3::Vector3(int x, int y, int z) {
    this->x = x;
    this->y = y;
    this->z = z;
}

double length(const Vector3& v) {
    //assume math.h is included for sqrt
    return sqrt(v.x*v.x + v.y*v.y + v.z+v.z);
}
```

Notice that in the implementation:

- We do not repeat the friend keyword
- The scope operator is not included since the friend function is not part of the class
- The friend function can access private instance variables

Here is an example of using `Vector3` and `length`:

```
Vector3 v(1, 2, 3);
int l = length(v);
```

We use `length` like a C function, since it does not belong to a class.

Friends and operator overloading
In the operator overloading section, we saw how when we overload operators as non-member functions, we can no longer access private instance variables. For example, we tried to overload the + operator for the `Vector3` class like this:

```
class Vector3 {
     private:
          int x, y, z;
     public:
          Vector3(void);
          Vector3(int, int, int);
};

Vector3 operator+(const Vector3&, const Vector3&);
```

The trouble with this implementation is that `operator+` could not access the private variables `x`, `y`, and `z`. The solution is to make `operator+` a friend of `Vector3`:

```
class Vector3 {
     private:
          int x, y, z;
     public:
          Vector3(void);
          Vector3(int, int, int);
          friend Vector3 operator+(const Vector3&,
                                   const Vector3&);
};
```

The implementation will look like this:

```
Vector3::Vector3(void) {
     x = 0;
     y = 0;
     z = 0;
}

Vector3::Vector3(int x, int y, int z) {
     this->x = x;
     this->y = y;
     this->z = z;
}

//We can access private variables x, y, z
Vector3 operator+(const Vector3& v1, const Vector3& v2) {
     Vector3 result(v1.x+v2.x, v1.y+v2.y, v1.z+v2.z);
     return result;
}
```

And we can use `Vector3` like this:

```
Vector3 v1(1, 2, 3);
Vector3 v2(4, 5, 6);

Vector3 result = v1 + v2;
```

It is a good idea to make any overloaded non-member function a friend of the associated class.

Friend classes

A *friend class* is very similar to a friend function. However, it lets an entire class have access to another class's private and protected members.

If you want class `F` to be given private/protected access to class `C`, add the line:

```
friend class F;
```

in the public declarations of class `C`.

Example

Friend classes are very useful when developing data structures. When we have a stack, queue, or linked list that is made up of nodes, it make sense to let the stack, queue, or linked list see the private and protected members of the nodes.

Here's an example using a linked stack. First, the class definitions:

```
class Node {
     private:
          int data;
          Node *next;
     public:
          Node(int d) {data = d; next = NULL;}
          friend class Stack;
};

class Stack {
     private:
          Node *top;
     public:
          Stack(void);
          ~Stack();
          void push(int);
          int pop(void);
};
```

And here is the implementation of `Stack`:

```
Stack::Stack(void) {
	top = NULL;
}

Stack::~Stack() {
	while (top != NULL) pop();
}

void Stack::push(int val) {
	Node *n = new Node(val);
	if (top == NULL) top = n;
	else {
		n->next = top;
		top = n;
	}
}

int Stack::pop(void) {
	if (top == NULL) throw "Empty stack";
	else {
		Node *temp = top;
		int val = top->data;
		top = top->next;
		delete temp;
		return val;
	}
}
```

Notice that `Stack` is able to access the private `Node` variables next and data because `Stack` is a friend of `Node`.

Caveats
While friend classes and friend functions are handy tools, they should not be overused. Giving all sorts of classes and functions "friend access" destroys the idea of data encapsulation, and can be just as bad as making everything in a class public. Think carefully before adding any friends.

Other things to consider:

- Friendship is not reciprocal (if `A` is a friend of `B`, `B` is not necessarily a friend of `A`)
- Friendship is not transitive (friends of my friends are not necessarily my friends)
- Friendship is not inherited (children of my friends are not necessarily my friends)

Inheritance

The idea of *inheritance* in C++ is the same as the idea of inheritance in Java. We have a general form of the class, and we want to reuse some of its data and function when defining a more specific version.

A “general class” is also called a base class, parent class, and super class.

A “specialized class” is also called a derived class, child class, and sub class.

Example

To get an idea for how inheritance works in C++, let’s look at an example. First, we will write a class that represents a general person:

```
class Person {
     protected:
          string name;
          int age;
     public:
          Person(string, int);
          void print(void);
};

Person::Person(string name, int age) {
     this->name = name;
     this->age = age;
}

void Person::print(void) {
     cout << name << " " << age << endl;
}
```

Notice that we have made the instance variables `name` and `age` protected so any class that extends `Person` will inherit these variables.

Now, here is the class definition for a more specific kind of person – a student:

```
class Student: public Person {
     private:
          string major;
     public:
          Student(string, int, string);
};
```

Notice the “`: public Person`” on the first line. This mean that `Student` extends `Person`, and inherits all private and protected members of `Person`.

Constructors in child classes

Before we implement the `Student` class, we need a way to call the parent constructor (`Person`) from the child constructor. In Java, we accomplished this with the keyword `super`. However, there is no corresponding keyword in Java. Instead. the parent constructor is called like this:

```
Student::Student(string name, int age, string major) : Person(name, age) {
	this->major = major
}
```

The "`: Person(name, age)`" at the end of the constructor header calls the `Person` constructor, which initializes the `name` and `age` variables.

If you leave off this call to the parent constructor, then the compiler will automatically insert a call to the no-argument parent constructor. This is OK if you have no parent constructor or if you defined a no-argument parent constructor. If not, you will get a compiler error.

Using inherited objects

Here is an example of using the `Person` and `Student` classes:

```
Person *p1;
Person *p2;
Student *s;

p1 = new Person("Fred", 20);
p2 = new Student("Bob", 18, "Psychology");
s = new Student("Jane, 20, "Statistics");

p1->print();
p2->print();
s->print();
```

Notice that we can store a `Student` object in a `Person` pointer (since a `Student` is a `Person`), and that we can call the `print` function on all our objects (since `Student` inherits `print`).

However, we cannot store a `Person` in a `Student` pointer (since a person is not necessarily a student):

```
//Illegal!
Student *s1 = new Person("Ted", 30);
```

Function overriding

Recall from Java that *function overriding* is when we have two functions with the same name, parameters, and return type – but one version is in the parent class and one version is in the child class.

Example

Our `Person` class had a `print` function – let's override it in the `Student` class:

```
class Student: public Person {
     protected:
          string major;
     public:
          Student(string, int, string);
          void print(void);
};
```

Here is the overridden implementation of `print`:

```
void Student::print(void) {
     cout << name << " " << age << " " << major << endl;
}
```

Suppose we try to use the `print` function as follows:

```
Person *p1;
Person *p2;
Student *s;

p1 = new Person("Fred", 20);
p2 = new Student("Bob", 18, "Psychology");
s = new Student("Jane, 20, "Statistics");

p1->print();
p2->print();
s->print();
```

What will be printed by the three print statements?

1) **`p1->print()`** – calls print in `Person`, prints "**Fred 20**"
2) **`p2->print()`** – calls print in `Person`, prints "**Bob 18**"
3) **`s->print()`** – calls print in `Student`, prints "**Jane 20**"

Notice that when we have a `Student` object stored in a `Person` pointer (`p2`), the `Person` `print` function still gets called – this is very different behavior than in Java. To get Java's behavior, where the child version is called, we need to use something called a *virtual function*. We will discuss virtual functions in the next section.

Access to an overridden base function
If we have a child variable and call an overridden function, it will call the version in the child class. However, we can force a call to the parent version. Here's an example:

```
Student *s = new Student("Tom", 18, "Open");

s->print();           //Calls print in Student
s->Person::print();   //Calls print in Person
```

(If we have a regular, non-pointer object, we do the same thing but with a `.` instead of an `->`.)

We can also call the parent function from inside the child class. For example, we could write the `Student print` function so that it first called the `Person print` function:

```
void Student::print() {
     //calls print in Person
     Person::print();

     cout << major << endl;
}
```

Destructors with inheritance
When a destructor for a child class is invoked, it first calls the destructor for the parent class – you do not need to call it yourself. Consequently, your child destructor should only release memory specific to the child class, since other memory will be released by the parent destructor.

Multiple inheritance
In Java, a child class can only have one parent. In C++, however, you can write a class that extends several other classes (*multiple inheritance*).

The syntax for multiple inheritance is as follows:

```
class Child: public A, public B, ... {

};
```

Here, `Child` extends `A` and `B` (and possible more classes if we want). Child inherits the private and protected members from every class it extends.

Be very careful when using multiple inheritance! If functions and variables in the different parent classes have the same name, this can become very confusing.

Casting objects

It is sometimes useful to cast from a child object to a parent object. To do this in C++, we use the `dynamic_cast` function. Here's an example:

```
Student *s = new Student("Tom", 20, "Physics");

Person *p = dynamic_cast<Person*>(s);
```

This line casts `s` to a `Person` object, and returns a pointer to the result. We can also use the `dynamic_cast` function to determine the type of an object. For example:

```
Person *p;

//p is initialized to be either a new Person or a new Student

if (dynamic_cast<Student*>(p) != NULL) {
    //We know p points to a Student object

    Student *s = dynamic_cast<Student*>(p);

    //Now we can treat s like a student
}
```

The `dynamic_cast` function returns `NULL` if the object we are trying to cast does not have the type we expect. So, we can try casting to the child version. If the cast fails, we know we have a parent object. If it succeeds, we know we have a child object.