

C++ Programming

Classes, Classes, Classes

Classes, Classes, Classes

- Classes and Objects
- Constructors and Destructors
- Inheritance
- Multiple Inheritance
- Friends

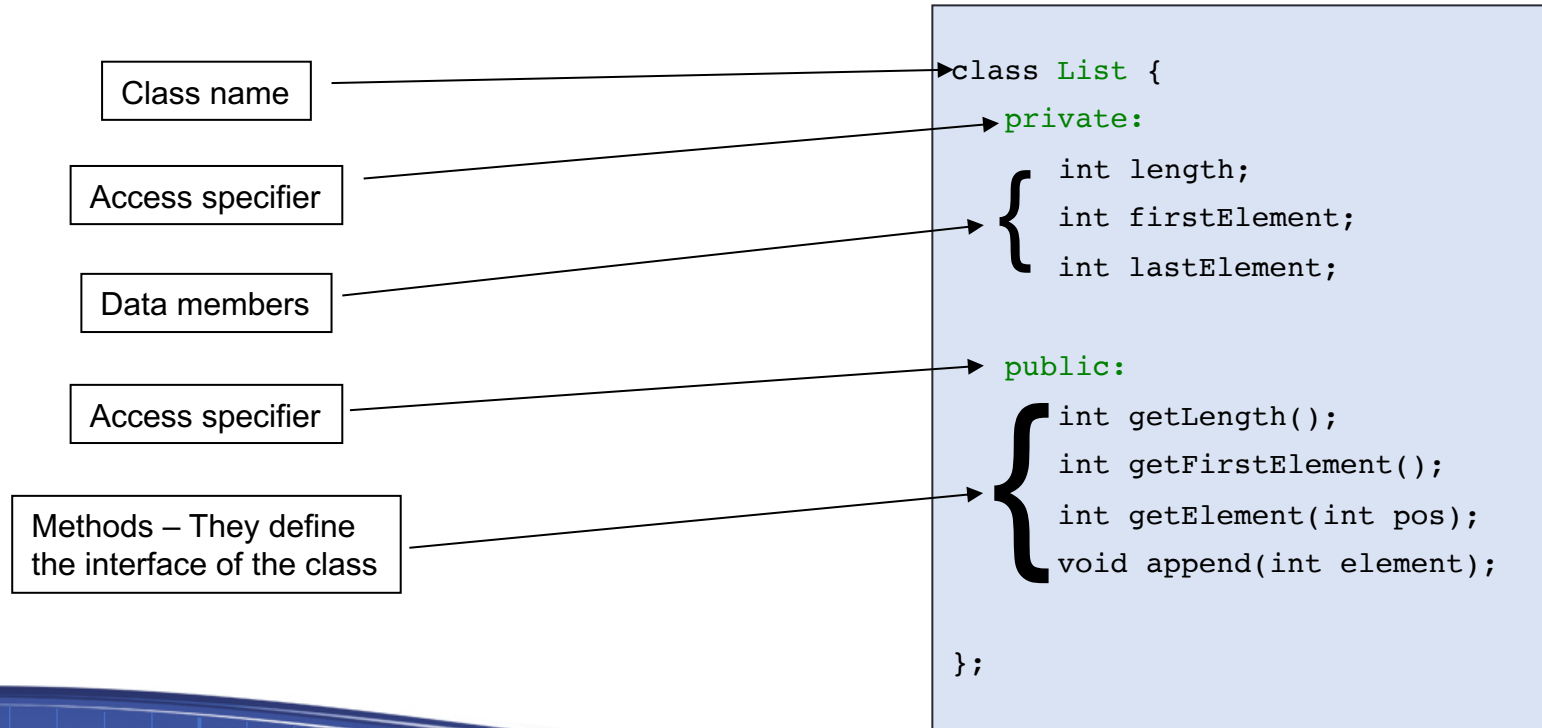
Classes and Objects

- Recall that in C++, classes are an expanded concept of structs
- Classes contain both data members and functions
- An object is an instantiation of a class
- Classes are defined using the keyword `class`

Classes and Objects

```
class class_name{  
    access_specifier_1: // more on this soon  
        member1;  
    access_specifier_2: // more on this soon  
        member2;  
  
    ...  
} object_names;           // optional; we do not need  
                           // to create objects now
```

Classes and Objects – An Example



Hello World, Now with Objects!

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

class HelloClass {
public:
    void Hello() {
        cout << "Hello World!" << endl;
    }
};

int main() {
    HelloClass helloObject;
    helloObject.Hello();
}
```

Access Specifiers

- Classes have access specifiers to limit and restrict access to the various data members and member functions of the class
- An access specifier is one of the following keywords:
 - `private`: private members of a class are accessible only from within other members of the same class (this is the default for classes)
 - `public`: public members are accessible from anywhere where the class and its instantiated objects are visible
 - `protected`: protected members are accessible from other members of the same class and members of their derived classes

Access Specifiers

```
class Rectangle {  
    int width, height;  
    public:  
        Rectangle(int, int);  
        int area() {return (width * height);}  
};
```

Members `width` and `height` are private by default; they can only be accessed by other members of the same class. (This said it is better form to explicitly put a `private` designator to make things clear, even though it isn't required.)

Any of the public members of objects from this class can be accessed as if they were normal functions or normal variables by simply inserting a dot (`.`) between the object name and member name.

Access Specifiers – An Example

```
class Triangle {  
    private:  
        float base, height, area;  
    public:  
        void setBase(float val) {  
            base = val; }  
  
        float getBase() {  
            return base; }  
  
        void setHeight(float val) {  
            height = val; }  
  
        float getHeight() {  
            return height; }  
  
        void calculateArea() {  
            area = (base*height / 2); }  
};
```

```
int main() {  
    Triangle *tr1 = new Triangle;  
    Triangle tr2;  
  
    tr1->setBase(2.0);  
    tr1->setHeight(3.0);  
    tr1->calculateArea();  
  
    tr2.setBase(1.0);  
    tr2.setHeight(2.0);  
    tr2.calculateArea();  
  
    tr2.base = 10.0;  
    tr2.base = tr1->height;  
    height = base;  
}
```

Correct. The data members base, height, area, can all be directly accessed from code within the class.

We cannot access a private data member from code that is defined outside of the class. We need here to use the accessor setBase() applied to object tr2. Also, we would need to use getHeight() on object tr1 to access its height.

This also does not make sense. Data members height, base must be used in conjunction with a specific object.

Objects and Their Creation

- Class definitions are a form of type definition
- As noted earlier, when we define a variable of a class type, we say that this variable denotes an object (i.e. an instance) of this class
- If we define the variable as a pointer to a class type, we haven't actually created an object yet; to create the object (i.e. allocate memory for it) we use the C++ `new` operator
 - We saw an example of this with the `Triangle` class earlier
 - Let's take a second look at this ...

Objects and Their Creation

```
class Triangle {  
    private:  
        float base, height, area;  
    public:  
        void setBase(float val) {  
            base = val; }  
  
        float getBase() {  
            return base; }  
  
        void setHeight(float val) {  
            height = val; }  
  
        float getHeight() {  
            return height; }  
  
        void calculateArea() {  
            area = (base*height / 2);}  
};
```

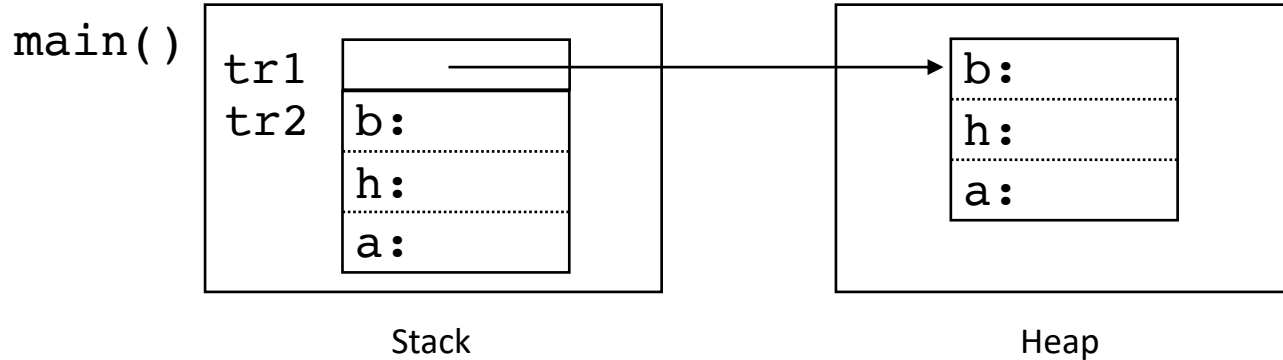
```
int main() {  
  
    Triangle *tr1 = new Triangle;  
    Triangle tr2;  
  
    tr1->setBase(2.0);  
    tr1->setHeight(3.0);  
    tr1->calculateArea();  
  
    tr2.setBase(1.0);  
    tr2.setHeight(2.0);  
    tr2.calculateArea();  
  
}
```

Objects and Their Creation

- Things to keep in mind with objects ...
 - An object of the class `Triangle` corresponds, models, and denotes a specific `Triangle` that has a specific name (e.g. `tr2`)
 - Each object has all the data members and member functions defined by the class of which it is an instance of (e.g. `height`, `base`, `area`)
 - While each object from a class has the same set of data members and member functions, each object has its own values in its data members
 - For instance, in this example, each object of the class `Triangle` has a data member called `height`, but each object may have a different value in its data member `height`

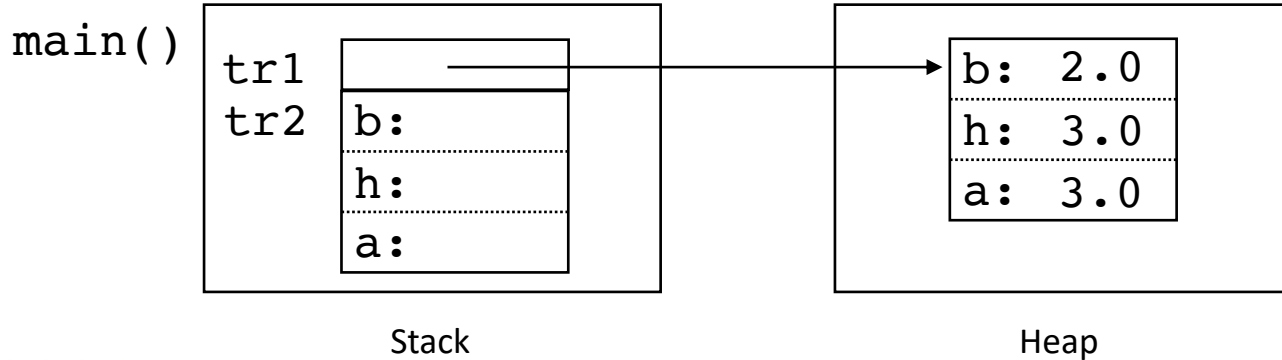
Objects and Their Creation

```
Triangle *tr1 = new Triangle;  
Triangle tr2;
```



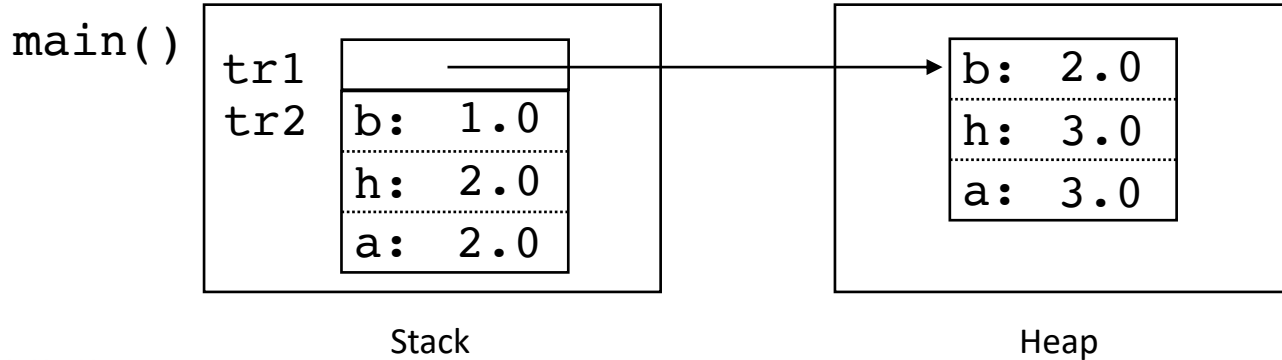
Objects and Their Creation

```
tr1->setBase(2.0);  
tr1->setHeight(3.0);  
tr1->calculateArea();
```



Objects and Their Creation

```
tr2.setBase(1.0);  
tr2.setHeight(2.0);  
tr2.calculateArea();
```



Objects and Their Destruction

- Objects created on the stack are destroyed when the function they were created in returns
- For dynamically created objects, as we are using the `new` operator to create them, we use the `delete` operator to destroy them
 - They are not automatically garbage collected as in some languages like Java
- When the program terminates, the heap is destroyed along with everything it contains; that said, it is still better to `delete` objects when you are done with them as destructors are not called when the heap is destroyed when the program ends (more on this in a bit ...)

Constructors

- A class can include a special function called a constructor
- A constructor is automatically called whenever a new object of this class is created (on the stack or on the heap via `new`)
- A constructor allows the class to initialize member variables, allocate storage, and so on
- Constructors are not strictly required in C++, but are strongly recommended as the proper way to initialize objects
 - If you do not provide one, a default one that takes no parameters and does nothing is implicitly created behind the scenes for you

Constructors

- The constructor function is declared just like a regular member function but with a name that matches the class name and without any return type
- If a constructor requires parameters, these parameters must be supplied or else an error will occur
- It is possible to define multiple constructors, providing a variety of ways to initialize objects

Constructors

- For example:

```
class Rectangle{  
    int width, height;  
    public:  
        Rectangle(int, int);    // constructor  
        int area() {return (width * height);}  
};
```

Constructors - Example

```
class Point {  
    private:  
        int x;  
        int y;  
    public:  
        // default constructor – no parameters  
        Point() {  
            x = 0;  
            y = 0;  
        }  
        // constructor with parameters  
        Point(int new_x, int new_y) {  
            x = new_x;  
            y = new_y;  
        }  
};
```

```
// Usage  
  
// call to our default constructor follows  
Point p;  
  
// call to constructor and initialize with two parameters  
Point q(10,20);  
  
// call to default constructor follows  
Point *r = new Point();  
  
// our default constructor is not called here.  
// Just assignment of object p to variable (object) s  
Point s = p;
```

Copy Constructors

- The copy constructor is a special constructor in C++ that creates an object by initializing it with another, previous initialized object of the same class
- The copy constructor can be used to:
 - Initialize one object from another of the same type
 - Copy an object to pass it as an argument to a function
 - Copy an object to return it from a function

Copy Constructors - Example

```
class Point {  
    private:  
        int x;  
        int y;  
    public:  
        ...  
        // copy constructor  
        Point(const Point &p) {  
            x = p.x;  
            y = p.y;  
        }  
};
```

// Usage

```
// call to one of our other constructors  
Point p(10,20);
```

```
// use our new copy constructor  
Point s = p;
```

The parameter is passed by reference but can not be altered inside the method because it is declared const.

Copy Constructors

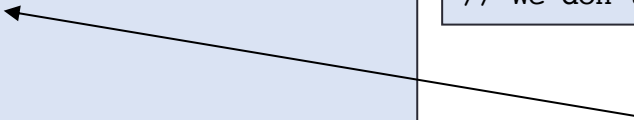
- If you do not declare a copy constructor, the compiler will typically give you one implicitly that does a simple member-wise copy of the source object
 - You can ask it not to do this, if you don't want it
- If the simple implicit copy constructor does not suffice to initialize the object properly on its own, however, you should define one for yourself
 - For example, if your objects contain pointers to other things, you might need to make copies of those other things too, instead of just copying the pointers ...

Destructors

- A class can also include a special function called a destructor
- A destructor is automatically called whenever an object of this class is being destroyed (when `delete` is used, or when a function returns and it has objects in its stack frame)
- A destructor allows the class to deallocate storage and so on
- Destructors are also not strictly required in C++, but are strongly recommended as the proper way to tear down objects
 - If you do not provide one, a default one that does nothing is implicitly created behind the scenes for you

Destructors - Example

```
class Point {  
    private:  
        int x;  
        int y;  
    public:  
        ...  
        // destructor  
        ~Point() {  
        }  
};
```



// Usage

```
// call to one of our constructors  
Point *p = new Point(10,20);  
Point q(20,10);
```

```
// destroy the dynamic object and call our new destructor  
delete p;
```

```
// q is on the stack and will be deleted when main() exits  
// We don't need to do anything special for that ...
```

This destructor does nothing special.
It simply has an empty body.

Inheritance

- One of the most important concepts in object-oriented programming is that of inheritance
- Inheritance allows us to define a class in terms of another class, which makes it easier to create, organize, and maintain an application
- This also provides an opportunity to reuse code functionality and accelerate implementation time (less time to code, less time to test, and so on)

Inheritance

- When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class
- This existing class is called the base class or superclass, and the new class is referred to as the derived class or a subclass of the base class

Inheritance

- A class can be derived from more than one class, which means it can inherit data and functions from multiple base classes (multiple inheritance)
- To define a derived class, we use a class derivation list to specify the base class(es); a class derivation list has the form:

```
class derived-class: access-specifier base-class
```

- In this case, `access-specifier` is one of `public`, `protected`, or `private`, and `base-class` is the name of a previously defined class
- If the access specifier is not used, then it is `private` by default

Inheritance

- If instead of

```
class B { ... };
```

we write:

```
class B: public A { ... };
```

... it means: B is a subclass of A

- For example:

```
class Mammal: public Animal { ... };
```

which means Mammal is a subclass of Animal

Inheritance – Example

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

// Base class
class Shape
{
protected:
    int sides;
public:
    int getSides()
        { return sides; }
    void setSides(int s)
        { sides = s; }
};
```

```
// Derived class
class Rectangle: public Shape
{
public:
    Rectangle()
        { sides = 4; }
};
```

```
int main()
{
    Rectangle rect;
    cout << "Sides: " << rect.getSides() << endl;
}
```

Inheritance and Access

	public member	protected member	private member
Access from members of own class	Yes	Yes	Yes
Access from members of a derived class	Yes	Yes	No
Access from non members (other code)	Yes	No	No

Inheritance and Access

- In public inheritance, all members are inherited with the same access levels as they had in the base class
- In protected inheritance, all public members in the base class are inherited as protected
- In private inheritance, all members in the base class are inherited as private
- Generally, most use cases for inheritance should follow public inheritance, even though in C++ private would be the default

Inheritance and Access – Example 1

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

// Base class
class Shape
{
protected:
    int sides;
public:
    int getSides()
        { return sides; }
    void setSides(int s)
        { sides = s; }
};
```

```
// Derived class
class Rectangle: public Shape
{
public:
    Rectangle()
        { sides = 4; }
};
```

```
int main()
{
    Rectangle rect;
    cout << "Sides: " << rect.getSides() << endl;
}
```

Inheritance and Access – Example 2

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

// Base class
class Shape
{
private:
    int sides;
public:
    int getSides()
        { return sides; }
    void setSides(int s)
        { sides = s; }
};
```

```
// Derived class
class Rectangle: public Shape
{
public:
    Rectangle()
        { setSides(4) }
};
```

Because sides is private, we can't reference it directly in the derived class. We need to use setSides() instead.

```
int main()
{
    Rectangle rect;
    cout << "Sides: " << rect.getSides() << endl;
}
```

Inheritance and Access – Example 3

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

// Base class
class Shape
{
protected:
    int sides;
public:
    int getSides()
        { return sides; }
    void setSides(int s)
        { sides = s; }
};
```

```
// Derived class
class Rectangle: private Shape
{
public:
    Rectangle()
        { sides = 4; }
};
```

Because we inherited privately, the `getSides()` method is no longer publicly visible.

```
int main()
{
    Rectangle rect;
    cout << "Sides: " << rect.getSides() << endl;
}
```

As a result, this call will fail.

Inheritance of Constructors and Destructors

- Like other members, a base class's constructors and destructor are also passed down to derived classes during inheritance
- They are automatically called by constructors and destructor of the derived class when its constructors or destructor are called
 - Base class constructors are called first, but their destructors are called last
- Unless otherwise specified, the default constructor (the one taking no parameters) is called

Inheritance of Constructors and Destructors

```
#include <iostream>
using namespace std;
class Parent {
public:
    Parent() {
        cout << "In Parent default constructor" << endl;
    }
    ~Parent() {
        cout << "In Parent destructor" << endl;
    }
};

class Child: public Parent {
public:
    Child() {
        cout << "In Child default constructor" << endl;
    }
    ~Child() {
        cout << "In Child destructor" << endl;
    }
};

int main() {
    Parent p;
    Child c;
}
```

When executed, we get this output.
Note the order in which things are printed.

In Parent default constructor
In Parent default constructor
In Child default constructor
In Child destructor
In Parent destructor
In Parent destructor


Inheritance of Constructors and Destructors

```
#include <iostream>
using namespace std;
class Parent {
public:
    Parent() {
        cout << "In Parent default constructor" << endl;
    }
    Parent(int x) {
        cout << "In Parent other constructor" << endl;
    }
};

class Child: public Parent {
public:
    Child() {
        cout << "In Child default constructor" << endl;
    }
    Child(int x) {
        cout << "In Child other constructor" << endl;
    }
};

int main() {
    Child c1;
    Child c2(0);
}
```

Note that because we didn't specify which base class constructor to use, the second case still used the default even though a better match was there.



```
In Parent default constructor
In Child default constructor
In Parent default constructor
In Child other constructor
```

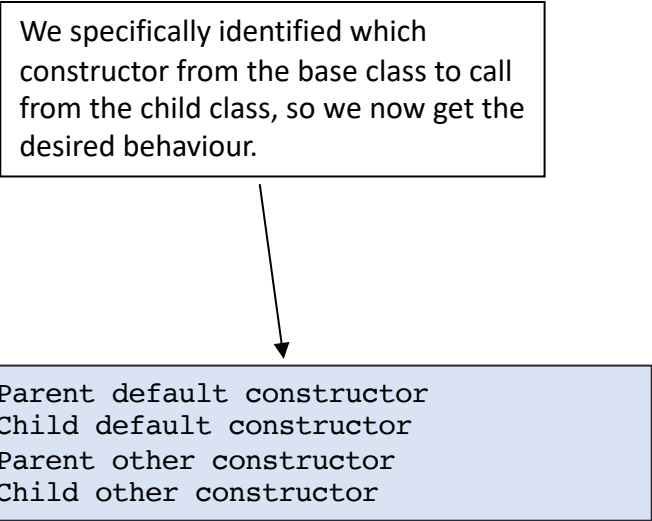
Inheritance of Constructors and Destructors

```
#include <iostream>
using namespace std;
class Parent {
public:
    Parent() {
        cout << "In Parent default constructor" << endl;
    }
    Parent(int x) {
        cout << "In Parent other constructor" << endl;
    }
};

class Child: public Parent {
public:
    Child() : Parent() {
        cout << "In Child default constructor" << endl;
    }
    Child(int x) : Parent(x) {
        cout << "In Child other constructor" << endl;
    }
};

int main() {
    Child c1;
    Child c2(0);
}
```

We specifically identified which constructor from the base class to call from the child class, so we now get the desired behaviour.



In Parent default constructor
In Child default constructor
In Parent other constructor
In Child other constructor

Multiple Inheritance

- Until now we have seen inheritance from one base class to one or more derived classes
- It is possible in C++ for one class to inherit from more than one class; this is referred to as multiple inheritance
- This can be done by simply listing multiple classes, separated by commas, in the base class section of the class definition (after the :)

Multiple Inheritance

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

class House {
protected:
    int residents;
public:
    int getResidents() {
        return residents;
    }
    void setResidents(int num) {
        residents = num;
    }
};

class Boat {
protected:
    float maxSpeed;
public:
    float getMaxSpeed() {
        return maxSpeed;
    }
    void setMaxSpeed(float max) {
        maxSpeed = max;
    }
};
```

A HouseBoat is both a House and a Boat, inheriting the members of both.

```
class HouseBoat: public House, public Boat {
};

int main() {
    HouseBoat hb;

    hb.setResidents(4);
    hb.setMaxSpeed(10.0);

    cout << "This dwelling has " << hb.getResidents();
    cout << " residents and a max speed of ";
    cout << hb.getMaxSpeed() << endl;
}
```

Friends of Classes

- Private and protected members of a class cannot be accessed from outside that class; however, this rule does not apply to “friends”
- Friends are functions or classes declared with the `friend` keyword
- A couple of things to keep in mind:
 - Friendship is not reciprocal; just because X is a friend of Y, that does not mean that Y is a friend of X, unless it is explicitly declared
 - Friendship is not transitive; a friend of a friend is not considered a friend, again unless explicitly declared

Friend Functions

- A non-member function can access the private and protected members of a class if it is declared a friend of that class
- That is done by including a declaration of this external function within the class, and preceding it with the keyword `friend`
 - Because of the way they are declared, such friend functions can appear as though they are members of the the class but they are not; they simply have access to private and protected members that they ordinarily wouldn't have access to

Friend Functions

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

class Rectangle {
private:
    int width, height;
public:
    Rectangle() {}
    Rectangle (int x, int y) : width(x), height(y) {}
    int area() {return width * height;}
    friend Rectangle doubleInSize(const Rectangle&);
};

Rectangle doubleInSize(const Rectangle& param){
    Rectangle res;
    res.width = param.width*2;
    res.height = param.height*2;
    return res;}

int main () {
    Rectangle foo;
    Rectangle bar(2,3);
    foo = doubleInSize(bar);
    cout << foo.area() << endl;
}
```

Note the interesting shorthand used to initialize the width and height data members. This still works!

Here we denote this function to be a friend of the class. We're not saying it's a member ... it's just a friend.

The friend function has direct access to private data members but is itself not a member of the class.

Friend Classes

- Similar to friend functions, a friend class is a class whose members have access to the private or protected members of another class

Friend Classes

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

class Square;

class Rectangle {
private:
    int width, height;
public:
    int area () {return (width * height);}
    void convert (Square a);
};

class Square {
friend class Rectangle;
private:
    int side;
public:
    Square (int a) {
        side = a;
    }
};

void Rectangle::convert (Square a) {
    width = a.side;
    height = a.side;
}
```

Note the forward declaration of the class Square. This is because of a circular dependency/chicken-and-the-egg situation here. The Rectangle class needs to know about Square to refer to it in its convert () method. But, Square needs to know about Rectangle to make it a friend. Since they both can't be declared first, we give the compiler an empty declaration of Square so it knows to expect it later on. We also have to defer the definition of convert () until Square has been defined, as this method needs Square's details.

```
int main () {
    Rectangle rect;
    Square sqr(4);

    rect.convert(sqr);
    cout << rect.area() << endl;
}
```