

C++ Programming

Classes, Classes, Classes

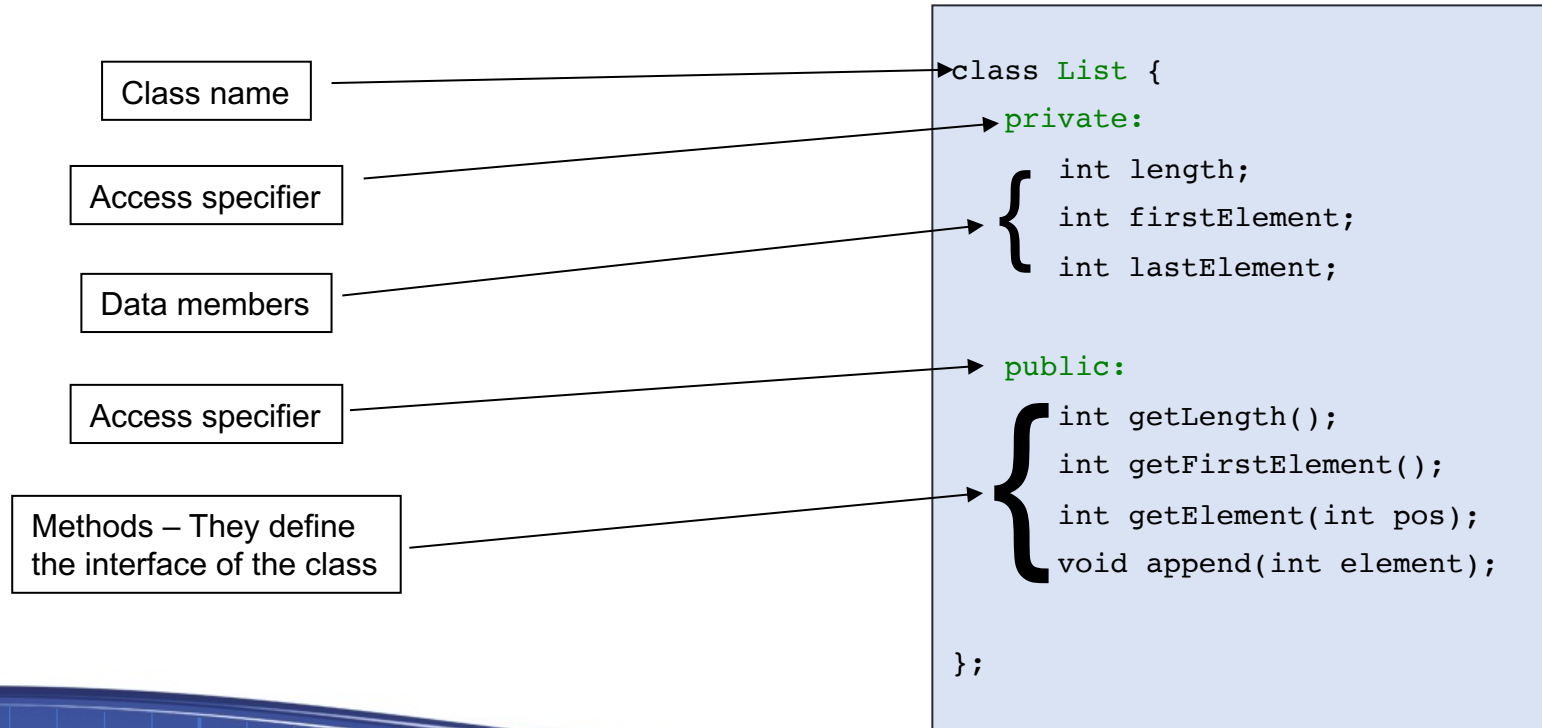
Classes and Objects

- 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();
}
```

Hello World, Now with Objects, Even Better!

```
#ifndef HELLO_CLASS_H
#define HELLO_CLASS_H

class HelloClass {
public:
    void Hello();
};

#endif
```

#include

```
#include <iostream>
#include "HelloClass.h"
using namespace std;

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

#include

```
#include "HelloClass.h"

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

Hello World, Now with Objects, Even Better!

- From a command line, building this in one step would look like:

```
> g++ HelloMain.cpp>HelloClass.cpp -o>HelloWorld
> ./HelloWorld
```

- Or we can do it in a few steps like:

```
> g++ -c>HelloClass.cpp
> g++ -c>HelloMain.cpp
> g++>HelloMain.o>HelloClass.o -o>HelloWorld
> ./HelloWorld
```

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);}  
};
```

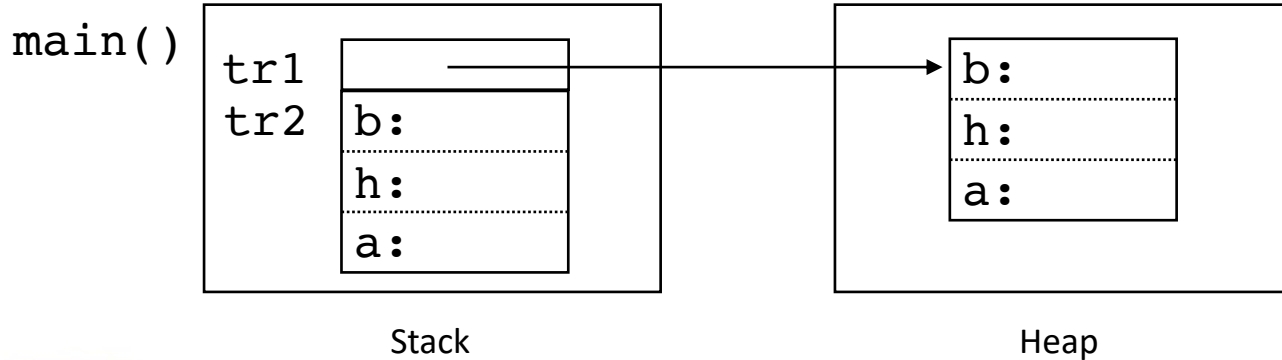
```
int main() {  
  
    Triangle *tr1 = new Triangle;  
    Triangle tr2;  
  
    tr1->setBase(2.0);  
    tr1->setHeight(3.0);  
    tr1->calculateArea();  
  
    tr2.setBase(1.0);  
    tr2.setHeight(2.0);  
    tr2.calculateArea();  
  
}
```

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`

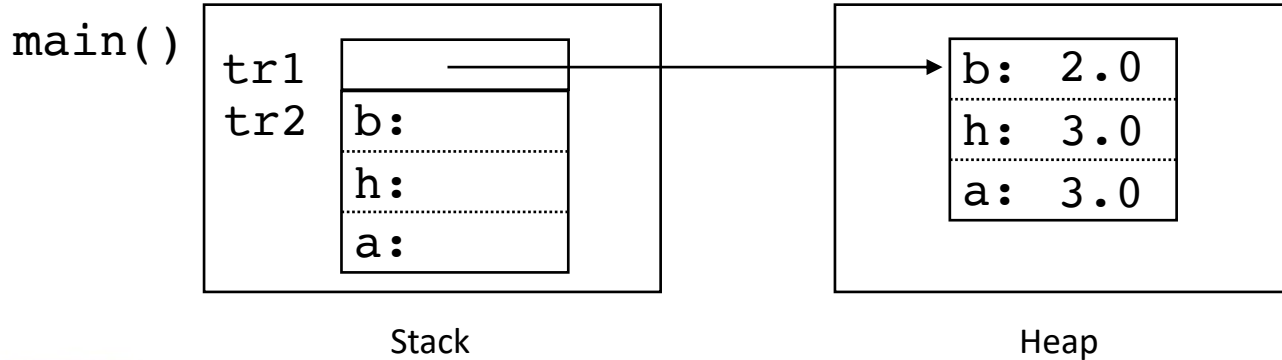
Objects and Their Creation

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



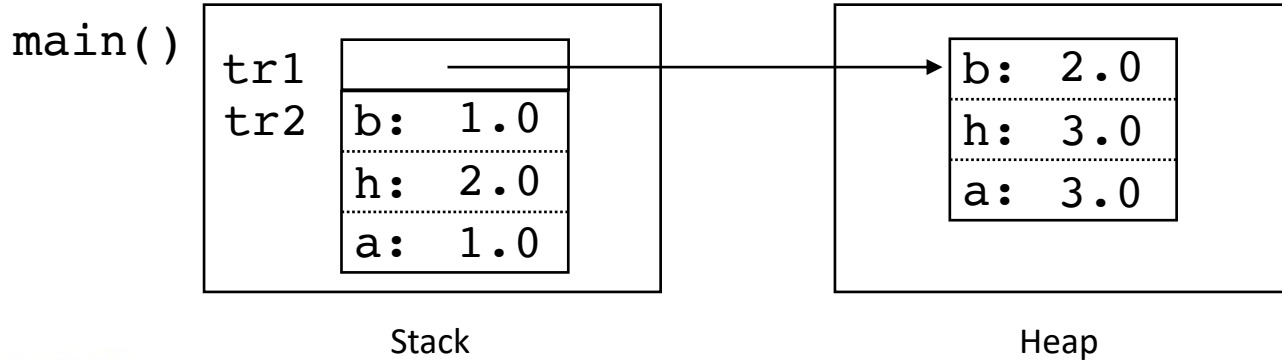
Objects and Their Creation

```
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

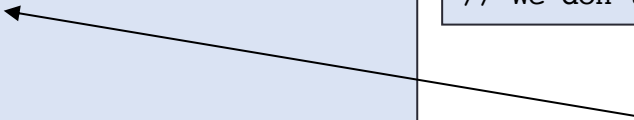
- If you do not declare a copy constructor, the compiler will typically give you one implicitly that does a simple member-wise copy of the source object
 - You can ask it not to do this, if you don't want it
- If the simple implicit copy constructor does not suffice to initialize the object properly 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.