Chapter 4 - Structs and Classes

C++ has two data structures which resemble Python classes - structs and classes.

They are so similar that, rather than talk about them separately, I will discuss classes and then come back and talk about the trivial differences between classes and structs.

Before we delve into the guts of c++ classes, we should probably talk about what they are about. The goal of the Class construct is to allow a developer to model a user defined type which behaves like a built in type. In order to achieve this goal, C++ and Python both provide a rich set of operators which may be specialized to achieve parity between custom and built in types.

In Python, you can implement a wide variety of dunder methods to customize mathematical operations, comparison, and more. (by dunder methods, I mean double underscore methods. `__add__` `__sub__` `__setattr__` etc)

In C++, you can customize all of the mathematical operations (`+` `-` `*` `/` `+=` `-=` `*=` `/=` `++` `--`), comparison (`==` `<` `>` `<=` `>=` `!=`), assignment (`=`), and more.

Both languages provide these facilities so that you may create data types for yourself which behave just like the built in ones.

Great. Lets get going. . . .

Classes

Since this is targeted at folks who are Python savvy, let's compare c++ classes to Python classes.

class Keyword

Like in Python, in C++, a class is defined using the class keyword:

Python

```
class Person(object):
```

C++

```
class Person {
};
```

Self Reference

In Python, each instance method has an explicit reference to itself, commonly spelled out as *self*, which all instance methods are passed as their first parameter.

C++ has an implicit reference to itself, called *this* which is a pointer to itself (we cover pointers elsewhere). However, you rarely need to use *this*, as C++ is usually smart enough to recognize references to member functions and data.

For completeness sake (hopefully this isn't confusing), consider a greet method defined in a person class in Python and C++, which we assume has an instance variable called *myname*:

Python

```
...
class Person(object):
    # ...
    # init code etc here
    def greet(self, other):
        print "Hello {}, my name is {}".format(other, self.myname )
```

C++

Here is the c++ implementation. Notice that we define the greet method "inline".

```
class Person {
    // ...
    // other implementation here
    void greet(const std::string& other) {
        cout << "Hello "<< other << ", my name is " << myname << endl;
    }
};
```

Constructor

Python has the initializer, otherwise known to true geeks as the dunderinit (`__init__`). Technically, this is an initializer, not a constructor, but let's gloss over that.

In C++, you have one or more constructor member functions. I say one or more, because C++ supports overloaded functions. So, for any function, you can redefine it multiple times, as long as the parameters differ in each version. This added flexibility is useful, because C++ is strictly typed. Oh, and like Python, C++ also supports default parameters... so, getting back to the constructor, let's take a look at how it is formed.

Adding to our person class:

```
Person {
    std::string firstname, lastname;
public:
    Person() : firstname(""), lastname("") {}; // default constructor
    Person(const std::string& fname, const std::string& lname) const : firstname(fname), las
}
```

So, let's talk about this. There are a couple of interesting things going on here. First, we have something called the *member initializer list*. it appears after the parens and before the brackets. It starts with a colon. See it?

```
somefunction() : <intializier list> {}
```

The initializer list is a comma separated list of data members which may be initialized from the parameter list of the function, or any constants, using method or uniform initialization notation [1][] (assuming you are compiling with C++11 support).

Operator Overloading

In C++ you can define custom behavior for a wide variety of operators. Doing so is known as *operator overloading*. Let's take a look at how this works by looking at overloading addition.

Binary & Unary Operators

In general, binary operators (operators operating on two items) and unary operators (operators operating on one item) are overloaded by defining class methods which return an instance of the class in question, are named "operator" followed by the actual operator in question, and take a const reference to an instance of the class. So for addition, the declaration looks like this (assuming we have a Point class):

```
class Point {
    ...
public:
    ...
    Point operator+(const Point& rhs);
    Point operator++(const Point& rhs);
};
```

Python would be similar, although Python does not have `++` or `--` operators..

```
class Point(object):
    ...
    def __add__(self, other):
```

```
    ...
```

Relational Operators

Relational operators return a boolean, and take a const reference to the containing class. So, continuing our example of Point:

```
class Point {
    ...
public:
    ...
    Point operator+(const Point& rhs);
    Point operator++(conts Point& rhs);
    bool operator==(const Point& rhs);
};
```

Once again, Python looks like this (foregoing implementation):

```
class Point(object):
    ...
    def __add__(self, other):
        ...
    def __eq__(self, other):
        ...
```

Inheritance

Specifying one or more superclasses in C++ is pretty simple. After the **class** keywork and class name, add a colon, followed by a comma separated list of the super classes.

Here we are in C++:

```
class Man : BaseMan {
...
}
```

And Python:

```
class Man(BaseMan):
    ...
```

There are, however, some key differences between how inheritance works in C++ and Python which you need to be aware of. For one, **constructors are not inherited**. That is a big one. You need to define your constructors instead of relying on your superclass's constructors.

Initializing superclass

In C++, you initialize your superclass by calling its constructor in your initializer list, or in the body of your constructor.

The following c++ code:

```
class BasePerson {
    string m_name;
public:
    BasePerson(std::string name) : m_name{name} {}
};

class Person : BasePerson {
    int m_age;
    public:
    Person(std::string name, int age) : BasePerson{name}, m_age{age} {}
};
```

Looks like this in python:

```
class BasePerson:
    def __init__(self, name):
        self.m_name = name

class Person(BasePerson):
    def __init__(name, age):
        super(Person, self).__init__(name)
        self.m_age = age
```

Access Specifiers - Private, Public, Protected

One thing you might have fussed with in Python a bit is the notion of privacy. You probably have prefixed variables with an underscore as a way of telling the rest of the world that a variable is part of an implementation, and shouldn't be mucked with. You might have even gone so far as to use two underscores in order to invoke name mangling. Well, no offense to Python, but its implementation of privacy is second rate and you can get around it. Privacy in Python is what you might call a "gentlemen's agreement". And I don't know about you, but I'm not always a gentleman when I program.

In a C++ class, all of your variables and functions are defined under an access specifier. And there are three of them:

- public
- private
- protected

Public data and functions may be accessed by anyone. Instance methods, external callers, you name it.

Private data and functions may only be accessed internally by other functions within the class.

Lastly, protected data and functions may only be accessed by internal functions / data or by classes which inherit from the class in whith the protected data.

Oh, and access specifiers may be used more than once. Anything following an access specifier belongs to it. And class data is private by default. So if you see data or methods but no specifier, that data and those methods are private....

Example Time:

```
// person defined with all public methods
class Person {
 public:
    // constructor
    Person(std::string name) : name(name) {};
    std::string name
};
```

With Person above, all data is accessible outside of the class.

```
//
class PrivatePerson {
private:
    std::string name;
public:
    PrivatePerson(std::string name) : name(name) {};
};
```

However, with PrivatePerson above, name is private, and thus inaccessible from outside of the class.

```
//
class ProtectedPerson {
protected:
    std::string name;
public:
    ProtectedPerson(std::string nm) : name(nm) {}
};

class Man : public ProtectedPerson {
    public:
    void greet() { std::cout << "Hi, my name is "<< name << std::endl;}
}
```

Likewise , ProtectedPerson above only makes name accessible from descendants. For example, the Man class.

Trying to use them in main....

```
//
// main funciton

int main() {
    // this works
    Person dude("The Dude");
    std::cout << dude.name << std::endl;

    // this wont compile
    PrivatePerson privateDude("The Private Dude");
    std::cout << privateDude.name << std::endl;

    // protected
    Man theDude("The Dude");
    theDude.greet();
    // this doesnt work either
    theDude.name = "Shelly";

    return 0;
};
```

Exercise

Ok, its exercise time. Using all of the knowledge gained thus far, implement a Vector class. This class will represent a 3 dimensional vector, in the mathematical sense (not to be confused with std::vector, which is an array-ish construct).
3 dimensional vectors represent a direction. They have 3 coordinates in x,y,z space which indicate the terminal end of a line segment starting at 0,0,0. In addition to a direction, a vector has a magnitude, or strength, which is simply the length of the vector.

To model all this, we will need to store the terminal end of the vector, which we will represent with three doubles (x, y, z).
You may or may not remember vectors from your trig class, but you can perform a number of specialized operations between vectors. You can add them together, subtract one from the other, and multiply one by a scalar value. You can also perform a cross product between them, which produces a vector which is orthogonal (perpendicular) to each one, and perform a dot product between them, which produces a scalar value (equivalent to the length of the first times the length of the second times the cosine of the angle between them.)

Our vector will need to provide the following capabilities. It should:

- support addition and subtraction between vectors
- support cross product and dot products between vectors

- support scalar multiplication
- provide a normalize function

Write unit tests to validate that your class works using google test.

Coda - Using Google Test

Ok, you want to be a good citizen and practice this whole TDD thing right? But how do you go about setting up google test with cmake?

First, create an appropriate structure for your project.

```
vector_eg/
   src/
   lib/
   tests/
```

- src is where your source code will go.
- lib is where your google test framework will go.
- tests is where your tests will go.

Step 1 - install google test

google test should ship with a googletest directory chock full of goodness, including a CMakeLists.txt file. Go ahead and simply recursively copy googletest into lib.

Step 2 - Set Up src

In src, create a CMakeLists.txt file. Set it up to compile the code we will be creating shortly in the same directory. This time around we need to create a shared library output, instead of an executable. The test will be the executable. Our code will be the library. Doing this is every bit as simple as creating an executable. Simply call `add_library` instead of `add_executable`:

```
project(vector)
file(GLOB  cpps *.cpp)
file(GLOB hpps *.hpp)
add_library(vector SHARED ${cpps} ${hpps})
```

Now go ahead and create your Vector.cpp and Vector.h files in src.

Step 3 - set up your tests

In tests/ add another CMakeLists.txt file to compile your tests, which we will be adding shortly.

```
include_directories(${gtest_SOURCE_DIR}/include ${gtest_SOURCE_DIR})

# add files
file(GLOB tests_cpp *.cpp)

# add project name
add_executable(${PROJECT_NAME} ${tests_cpp})

target_link_libraries(${PROJECT_NAME} gtest gtest_main)
target_link_libraries(${PROJECT_NAME} vector)
```

This CMakeLists.txt file is a bit more involved. First, it includes directories from the googletest framework, which we will need.

Then it uses glob to grab all of the cpp files in the tests directory.

Then it creates an executable, adding all of the cpp files from the prior step.

The final two steps are to link in google test, and then link in our own vector library (which we set up in a previous CMakeLists.txt file)

Bringing it all together

Ok, we have our individual CMakeLists.txt files, but we need to set up a top level CMakeLists.txt file to call each of the CMakeLists.txt files we created *in the correct order*.

So, under our project's root directory, create a new CMakeLists.txt file:

```
cmake_minimum_required(VERSION 2.6.2)

project(unit_tests)
add_subdirectory(src)
add_subdirectory(lib/googletest)
add_subdirectory(tests)
```

We require a minimum cmake version as usual.
Then we give the project a name.
And finally, we add the subdirectories with our CMakeLists.txt files in the order we want them to be executed. We don't really care if our source or the google test framework is compiled first. We want both of them taken care of before compiling tests. . . .

TDD - Test Driven Development

Ok. Lets give this a go. Test Driven Development is a technique where you create each test and then add just enough to your source code to get it to pass. While I cannot say that I always faithfully practice test driven development, I can say that the more you interleave your testing, the easier it is on you.

Go to your src/ directory. You should have created Vector.cpp and Vector.h by now. If you used an IDE which provides scaffolding, you even have an empty class. Lets add a default constructor to get this rolling.

Vector.h

```
class Vector {
public:
   double x, y, z;
   Vector();
}
```

Ok. Let's jump into the tests/ directory and create a tests.cpp file.

All google tests need to include gtest to work. Since we are testing Vector, we also need to include Vector.h:

```
#include "gtest/gtest.h"
#include "../src/Vector.hpp"
```

Now it is time to write our first actual test. Gtest has a TEST macro which we use for each test.

```
TEST( <name of group>, <name of test>) {
...
}
```

So for our Vector class:

```
TEST(vector, default_constructor) {

}
```

Your tests are all going to look the same. You are going to write a bit of code to exercise a single function or method, then you are going to test the results. Gtest has a number of functions that will help you with this. Here are a few of them:

- EXPECT_EQ(rhs, lhs)
- EXPECT_DOUBLE_EQ(rhs, lhs)
- EXPECT_FLOAT_EQ(rhs, lhs)
- EXPECT_TRUE(expression)
- EXPECT_FALSE(expression)

Ok. Lets write our first test:

The Default Constructor.

Well, first, what do we expect our vector to get initialized to without any user intervention? It could be anything sensible. I am going to choose a unit vector pointing down the z axis.

```
TEST(vector, default_constructor) {
    Vector v{};
    EXPECT_DOUBLE_EQ(0.0, v.x);
    EXPECT_DOUBLE_EQ(0.0, v.y);
    EXPECT_DOUBLE_EQ(1.0, v.z);
}
```

And that is it. Well, lets run our unit test. It should fail at this point, because we have not written a constructor. Let's fix that.

Navigate to the src/ directory and edit your Vector.cpp. Add a default constructor and initialize it to <0,0,1>. Run the test again. It should pass.

The rest of the Vector class

From here, we are going to continue to build out our Vector class. We can add other constructors - one which allows you to initialize the Vector from 3 doubles would make sense. Perhaps one which allow you to initiLize a vector from another vector as well.

- Vector(double x_, double y_, double z_);
- Vector(const Vector& rhs);

Next, we should look at overloading operators for addition and subtraction. Go ahead and implement the following operator overloads, being sure to follow the same pattern as outlined above with regard to tests:

- Vector operator+(const Vector& rhs);
- Vector& operator+=(const Vector& rhs);
- Vector operator-(const Vector& rhs);
- Vector& operator-=(const Vector& rhs);

As a reminder, the `+=` and `-=` operators return a reference to Vector. To achieve this, you have to dereference `this` in your return statement (`return *this`).

What about comparisons? Go ahead and implement the `==` operator, as well as the other comparison operators:

- bool operator==(const Vector& rhs) const;
- bool operator<(const Vector& rhs) const;
- bool operator<=(const Vector& rhs) const;
- bool operator>(const Vector& rhs) const;
- bool operator>=(const Vector& rhs) const;

Whew, that is a lot of boilerplate, eh? It takes work to make a class mimic a built in type. How did you implement the comparison operators? I would expect that you would have calculated the length of each vector and used that as a measure. If you haven't written a `length()` method, do it now:

- double length() const;

What else should we implement? Well, what do you remember about vectors? What can you do with them? From the dim recesses of your memory (or perhaps a quick google'ing) you should come up with a few things

- you can take two vectors' `dot product`
- you can take two vectors `cross product`
- you can `multiply` a vector by a double
- You can `normalize` a vector.

Why don't you implement those methods next. Overload multiplication for the dot product and the scalar multiplication. Use `cross` as the method name for cross product, and `normalize` for the method name for normalize. I am going to let you come up with the signatures. Remember to use `const` and references where appropriate...

A note on operations between heterogeneous types.

When you implement `operator*`, you may come to a realization: this only works when multiplying a vector by a double. It will not work multiplying a double by a vector.

```
Vector3d v{};
double d = 22.2;

auto v2 = v * d; //works
auto v3 = d * v; // fails
```

Why is the case? Well, let's think about it. `v * d` is simply syntactic sugar for `v1.operator*(d)`. Likewise, `d * v` is syntactic sugar for `d.operator*(v)`. But we don't have access to the double class in order to extend it to support multiplication against a Vector3d. So what do we do? Well, we can define overloaded operators as free friend functions instead of methods. In our case, it would look something like this:

```
class Vector3d {
    ...
    friend Vector3d operator*(const double d, const Vector3d& v);
};
...
operator*(const double d, const Vector3d& v) {
    return Vector3d{ v.x * d, v.y * d, v.z * d};
}
```

Structs

Really? After all that, we are going to tackle another concept? Yes, because it is so simple. A `struct` is just like a class, except that it defaults to `public` if

you don't specify an access modifier. That's it. So, feel free to use the `struct` keyword in C++ interchangeably with `class` . In C++s precursor language, C, `struct`s were an altogether different beast. You can google it if you are interested.

[1]: Method initialization uses parens, (eg age(1)), whereas uniform initialization uses braces (eg age{1}). C++11 allows you to initialize everything using braces, whereas before c++11, initialization notation was a mixed bag.