Computing and Statistical Data Analysis
Lecture 6

Introduction to classes and objects:

- Declaring classes
- A `TwoVector` class
- Creating objects
- Data members
- Member functions
- Constructors
- Defining the member functions
- Pointers to objects

Glen Cowan
RHUL Physics

Classes

A class is something like a user-defined data type. The class must be declared with a statement of the form:

```
class MyClassName {
  public:
    public function prototypes and
    data declarations;
    ...
  private:
    private function prototypes and
    data declarations;
    ...
};
```

Typically this would be in a file called `MyClassName.h` and the definitions of the functions would be in `MyClassName.cc`.

Note the semi-colon after the closing brace.

For class names often use UpperCamelCase.

A simple class: `TwoVector`

We might define a class to represent a two-dimensional vector:

```
class TwoVector {
  public:
    TwoVector();
    TwoVector(double x, double y);
    double x();
    double y();
    double r();
    double theta();
    void setX(double x);
    void setY(double y);
    void setR(double r);
    void setTheta(double theta);
  private:
    double m_x;
    double m_y;
};
```

Class header files

The header file must be included (`#include "MyClassName.h"`) in other files where the class will be used.

To avoid multiple declarations, use the same trick we saw before with function prototypes, e.g., in `TwoVector.h` :

```
#ifndef TWOVECTOR_H
#define TWOVECTOR_H

class TwoVector {
  public:
     ...
  private:
     ...
};

#endif
```

Objects

Recall that variables are instances of a data type, e.g.,

```
double a;     // a is a variable of type double
```

Similarly, objects are instances of a class, e.g.,

```
#include "TwoVector.h"
int main() {
  TwoVector v;  // v is an object of type TwoVector
```

(Actually, variables are also objects in C++. Sometimes class instances are called “class objects” -- distinction is not important.)

A class contains in general both:

variables, called “data members” and

functions, called “member functions” (or “methods”)

Data members of a **`TwoVector`** object

The data members of a **`TwoVector`** are:

```
...
private:
  double m_x;
  double m_y;
```

Their values define the “state” of the object.

Because here they are declared **`private`**, a **`TwoVector`** object’s values of **`m_x`** and **`m_y`** cannot be accessed directly, but only from within the class’s member functions (more later).

The optional prefixes **`m_`** indicate that these are data members. Some authors use e.g. a trailing underscore. (Any valid identifier is allowed.)

The constructors of a `TwoVector`

The first two member functions of the `TwoVector` class are:

```
...
public:
  TwoVector();
  TwoVector(double x, double y);
```

These are special functions called constructors.

A constructor always has the same name as that of the class.

It is a function that is called when an object is created.

A constructor has no return type.

There can be in general different constructors with different signatures (type and number of arguments).

The constructors of a **`TwoVector`**, cont.

When we declare an object, the constructor is called which has the matching signature, e.g.,

```
TwoVector u;    // calls TwoVector::TwoVector()
```

The constructor with no arguments is called the "default constructor". If, however, we say

```
TwoVector v(1.5, 3.7);
```

then the version that takes two **`double`** arguments is called.

If we provide no constructors for our class, C++ automatically gives us a default constructor.

Defining the constructors of a `TwoVector`

In the file that defines the member functions, e.g., `TwoVector.cc`, we precede each function name with the class name and `::` (the scope resolution operator). For our two constructors we have:

```
TwoVector::TwoVector() {
  m_x = 0;
  m_y = 0;
}
TwoVector::TwoVector(double x, double y) {
  m_x = x;
  m_y = y;
}
```

The constructor serves to initialize the object.

If we already have a `TwoVector v` and we say

```
TwoVector w = v;
```

this calls a "copy constructor" (automatically provided).

The member functions of **`TwoVector`**

We call an object's member functions with the "dot" notation:

```
TwoVector v(1.5, 3.7);    // creates an object v
double vX = v.x();
cout << "vX = " << vX << endl;  // prints vX = 1.5
...
```

If the class had public data members, e.g., these would also be called with a dot. E.g. if **`m_x`** and **`m_y`** were public, we could say

```
double vX = v.m_x;
```

We usually keep the data members private, and only allow the user of an object to access the data through the public member functions. This is sometimes called "data hiding".

If, e.g., we were to change the internal representation to polar coordinates, we would need to rewrite the functions **`x()`**, etc., but the user of the class wouldn't see any change.

Defining the member functions

Also in `TwoVector.cc` we have the following definitions:

```
double TwoVector::x() const { return m_x; }
double TwoVector::y() const { return m_y; }
double TwoVector::r() const {
  return sqrt(m_x*m_x  + m_y*m_y);
}
double TwoVector::theta() const {
  return atan2(m_y, m_x);          // from cmath
}
...
```

These are called “accessor” or “getter” functions.

They access the data but do not change the internal state of the object; therefore we include `const` after the (empty) argument list (more on why we want `const` here later).

More member functions

Also in **`TwoVector.cc`** we have the following definitions:

```
void TwoVector::setX(double x) { m_x = x; }
void TwoVector::setY(double y) { m_y = y; }
void TwoVector::setR(double r) {
  double cosTheta = m_x / this->r();
  double sinTheta = m_y / this->r();
  m_x = r * cosTheta;
  m_y = r * sinTheta;
}
```

These are "setter" functions. As they belong to the class, they are allowed to manipulate the **`private`** data members **`m_x`** and **`m_y`**.

To use with an object, use the "dot" notation:

```
TwoVector v(1.5, 3.7);
v.setX(2.9);        // sets v's value of m_x to 2.9
```

Pointers to objects

Just as we can define a pointer to type **`int`**,

```
int* iPtr;        //  type "pointer to int"
```

we can define a pointer to an object of any class, e.g.,

```
TwoVector* vPtr;  // type "pointer to TwoVector"
```

This doesn't create an object yet!   This is done with, e.g.,

```
vPtr = new TwoVector(1.5, 3.7);
```

`vPtr` is now a pointer to our object. With an object pointer, we call member functions (and access data members) with **`->`** (not with "**`.`**"), e.g.,

```
double vX = vPtr->x();
cout << "vX = " << vX << endl;  // prints vX = 1.5
```

Forgotten detail: the **`this`** pointer

Inside each object's member functions, C++ automatically provides a pointer called **`this`**. It points to the object that called the member function. For example, we just saw

```
void TwoVector::setR(double r) {
  double cosTheta = m_x / this->r();
  double sinTheta = m_y / this->r();
  m_x = r * cosTheta;
  m_y = r * sinTheta;
}
```

Here the use of **`this`** is optional (but nice, since it emphasizes what belongs to whom). It can be needed if one of the function's parameters has the same name, say, **`x`** as a data member. By default, **`x`** means the parameter, not the data member; **`this->x`** is then used to access the data member.

Wrapping up lecture 6

We've introduced classes -- these behave like a sort of user defined data type.

Objects are instances of classes. In addition to holding data they have a set of functions that can act on the data. This is what distinguishes object-oriented programming from "procedural programming".

Next week we will carry on with the `TwoVector` class, adding more member functions, e.g., `TwoVector::rotate(double alpha)`.

We will discuss dynamic memory allocation.

We will see standard C++ classes such as `vector` and `string`.

We will briefly discuss things such as operator overloading, templates and inheritance.

Extra slides

Namespaces

A namespace is a unique set of names (identifiers of variables, functions, objects) and defines the context in which they are used.

E.g., variables declared outside of any function are in the global namespace (they have global scope); and can be used anywhere.

A namespace can be defined with the `namespace` keyword:

```
namespace aNameSpace {
  double x = 1.0;
}
```

To refer to this `x` in some other part of the program (outside of its local scope), we can use

```
aNameSpace::x
```

`::` is the scope resolution operator.

The `std` namespace

C++ provides automatically a namespace called `std`.

It contains all identifiers used in the standard C++ library (lots!), including, e.g., `cin`, `cout`, `endl`, ...

To use, e.g., `cout`, `endl`, we can say:

```
using std::cout;
using std::endl;
int main(){
  cout << "Hello" << endl;
  ...
```

or we can omit `using` and say

```
int main(){
  std::cout << "Hello" << std::endl;
  ...
```

`using namespace std;`

Or we can simply say

```
using namespace std;
int main(){
  cout << "Hello" << endl;
  ...
```

Although I do this in the lecture notes to keep them compact, it is not a good idea in real code. The namespace **`std`** contains thousands of identifiers and you run the risk of a name clash.

This construction can also be used with user-defined namespaces:

```
using namespace aNameSpace;
int main(){
  cout << x << endl;          // uses aNameSpace::x
  ...
```