

# Introduction to Programming using C++

## Lecture Five: Introducing Classes

Carl Gwilliam

[gwilliam@hep.ph.liv.ac.uk](mailto:gwilliam@hep.ph.liv.ac.uk)

<http://hep.ph.liv.ac.uk/~gwilliam/cppcourse>

# The Next Step

- If you have made it this far ...

**Congratulations! You can program in C++!**

- But if you stop here you might as well have learnt a less syntactically complicated language.
- The next half of the course will be spent introducing **Object Orientated** (OO) programming.

# Types Revisited

- A type defines the information that can be held by an object.
- A type limits the actions that can be performed on the object.
- A type is a method of determining the amount of memory that needs to be allocated for a new object.

// C++ built-in types

int a; //integer

float b; //real

double c; //real

bool d; //boolean

char e; //character

float height = 1.76; // float type

unsigned integer counter = 5; // type limits to int above 0

counter = counter / 3.6; // int conversion of rhs

# Extending Types

- Types are not just a method of expressing the precision of a value. A type is a **category**.
- The key to OO programming in C++ is the ability to create user defined types.

```
// some declarations  
int x, y;  
Particle electron, muon;  
Department physics, chemistry;  
Car myFiesta;
```

- If the new types are declared and defined in the correct manner then all of the above statements are valid C++.

# Is there a need for new types?

- You have already written programs to model several real-world situations without need for new object types.

*"Write a program to calculate the probability of tossing 8 heads from 10 throws of a coin."*

$$P(r, p, n) = p^r (1 - p)^{n-r} \frac{n!}{r!(n-r)!}$$

↑  
use input stream to get r=8,  
p=0.5 and n=10

↑  
create a function to calculate  
factorials

↑  
use pow() function  
supplied by math.h

*"Investigate what the initial speed and acceleration of a car should be in order for it to travel 100 metres in under 20 seconds."*

$$s = \frac{at^2}{2} + v_i t$$

iterate time from 1s to 20s

calculate distance until get to 100m or t=20s

# Modeling Complex Systems

- Can the same programming logic be applied to the following complex systems?
  - Car engine performance
  - Resource optimisation within a company
  - Particle interaction with detector material

Yes! Split problem into small group of calculations and store relevant values in variables that have labels relevant to problem.

BUT:

- OO programming allows the creation of new types that more accurately reflect the scenario to be modeled.
- A problem can be modeled in terms of objects and the interaction between different objects.

This seems contrary to our obsession with reducing any situation into a series of abstract mathematical statements...

# Layers of Abstraction

- Object Orientated programming is just a higher level of abstraction than you are currently used to.
- Your programs are already written in a high-level language. Can you answer these questions? Do you care?
  - What is the bit sequence for the number 3?
  - What memory addresses are being used to store your variables?
  - How does program interpret keyboard signals into ASCII values?

objects  
modules  
statements  
assembler  
language  
bitwise operations



# Layers of Abstraction

- Dealing with objects and object interaction enables programs to be designed on a more intuitive level.
- C++ provides the syntax to allow you to manipulate user defined objects in the same way that integers and floating numbers are manipulated.

```
Planet a,b; // information determined  
a.force(b); // by class declaration
```

```
mass1 = getMass(1); // functions called to  
force = getForce (G,mass1,mass2,dist); //retrieve properties
```

```
mass1 = density1 * volume1; // statements establishing  
mass2 = density2 * volume2; // values of properties  
force = (G * mass1 * mass2) /pow(dist,2);
```

ABSTRACTION





# Introducing Classes

- User defined types are referred to as **classes**.
- An object is an **instance** of class. A class is a category of objects. Several objects of a certain class can be **instantiated** (just like built in types).

```
int x; // equivalent  
Car myFiesta, myFerrari; // statements
```

- A car can be declared in the same way that an integer can be declared. The variable **myFiesta** is associated with an object of type **Car**.
- An object instance is not a single value. Math operators have to be assigned a new meaning in this context.

```
float someValue = myFerrari - myFiesta;
```

# Declaring a Class

- A simple **car** class will be designed to introduce the basic concepts of class construction.
- In the first iteration, a **car** object consists of two characteristics, the age and retail price of the car.
- A car class is declared using the **class** keyword followed by the name of the class.
- The variables declared inside the block are associated with this class.
- These are known as **member variables** or **member data**.
- myFiesta (previous slide) is an object that contains two integers

```
// car declaration
class Car {
public:
    int ageInYears;
    int priceInGBP;
}; // semi-colon
    // terminated
```

# Accessing Member Variables

- Member variables are accessed by the **dot** operator.
- The dot operator is placed after name of the object and followed by the name of the desired member variable.

```
myFiesta.ageInYears = 5;
```

- Some common misunderstandings:
  - This statement is trying to assign an age to the idea of a car, rather than age of particular car (does **int = 5;** make any sense?)

```
Car.ageInYears = 5;
```
  - This statement is accessing age without the dot operator. The car member variables only exist within the **scope** of the car class (else how do we know if want age of myFiesta or myFerrari?) .

```
ageInYears = 5;
```

# Simpler Data Structures

- “The car class has no advantages over using a one dimensional array.”
  - An array can only hold variables of the same type whilst classes can contain variables of **multiple** types.
- “The class is just being written as a data structure. A C-style **struct** can be used for this purpose.”
  - True, but classes are much more powerful than **structs**.
  - E.g. Unlike structs, members of a class are private by default, which encourages encapsulation (see later).

```
int Car = {5,2000}
```

```
class Car {  
public:  
    int ageInYears;  
    int priceInGBP;  
};
```

```
struct Car {  
    int ageInYears;  
    int priceInGBP;  
} //no semi-colon
```

# Member Functions

- A class is a collection of member variables and a set of related functions, known as **member functions** or **class methods**.
- Member functions are also accessed using the **dot** operator.
- They are distinguished from member variables by the use of **parentheses**.
- Like other functions they can take arguments

```
class Car {  
    public:  
        int ageInYears;  
        int priceInGBP;  
        void brake();  
        void accelerate();  
};
```

Member functions have a role specific to the class they are declared in. They are primarily used to access and manipulate class member data.

```
Car myFiesta;  
myFiesta.brake();
```

# Writing Member Functions

- The difference between a normal function definition and a member function is the form of the function name.
- The member function name needs to be prefixed with the class it belongs to followed by the **scope resolution operator** `::`.
- This syntax indicates to the compiler that the function is **local** to the **Car** class and can only be accessed through a **Car** object.

```
// brake member function  
void Car::brake() {  
    cout << "Slowing down" << endl;  
}
```

# Private and Public Member data

- All members of a class are declared **private** by default.
- If **public** keyword is removed from class declaration the member variables and methods cannot be accessed from outside the class.

“Why should the member data be private? I created the object so I should have access to all of its information.”

- Think of the declaration as the acquisition of an object rather than the creation of an object.
- The inner workings are hidden from you for your own protection! You don't need to know how the object works, all you need is an **interface** to the object. This is known as **encapsulation** or **information hiding**.

```
class Car {  
    int ageInYears;  
    int priceInGBP;  
    void brake();  
    void accelerate();  
};  
  
Car myFiesta;  
// error here  
myFiesta.ageInYears = 5;
```

# Accessor Methods

- **Private** member data can be modified by calling **public** member functions known as accessor methods. Accessor methods are commonly known as “getters” and “setters”.

“Why go the trouble hiding the member data? It is easier for me to access the data directly through the dot operator rather than through this level of indirection.”

```
class Car {  
public:  
    void brake();  
    void accelerate();  
    void setAge(int age);  
    int getAge();  
    void setPrice(int price);  
    int getPrice();  
private:  
    int ageInYears;  
    int priceInGBP;  
};
```

- An accessor method will allow you to change the name of a member variable. This change is hidden from the user because of encapsulation.



# A Point Class

- The remainder of this lecture will be used to construct a point class.

```
int main() {  
    Point a, b;  
    :  
}
```

- To design the interface, think in terms of the point object:
  - What is my identity?
  - What messages can you give me?
  - What is my distance from the origin?
  - What is my distance from another point in space?

# Point class declaration

- The initial interface design contains just accessor methods.
- The member data is private and the accessor functions are public.
- The class **declaration**, or interface, is put into a separate **header** file (this allows to minimise the dependencies between different pieces of code).

```
// point class
// declaration
class Point {
public:
    void setX(float x);
    void setY(float y);
    void setZ(float z);
    float getX();
    float getY();
    float getZ();
private:
    float itsX;
    float itsY;
    float itsZ;
};                                     [Point.h]
```

# Preprocessor Macros

- Before the compiler interprets code it is passed through the **preprocessor**, whose task is to find preprocessor directives, or **macros**.
- Preprocessor macros are indicated by lines beginning with a hash, **#**.
  - In fact, we have already been using the **#include** macro.
- Macros can also be used for other purposes, such as defining constants, although they are generally considered bad practice.
- One place where macros are useful is to prevent a header file being included multiple times.
- For more information on macros see:

```
#ifndef MYHEADER_H
#define MYHEADER_H
// code
:
#endif
```

<http://www.cplusplus.com/doc/tutorial/preprocessor.html>

# Accessor Implementation

- The **implementation** of the accessor functions, along with other class methods, are placed in a **source** file, separate from the header file and the main program.
- The implementation is totally separate from the interface.

```
#include "point.h"
void Point::setX(float x) {
    itsX = x;
}
float Point::getX() {
    return itsX;
}
[Point.cpp]
```

```
#include "point.h"
int main() {
    Point a;
    // set coordinates for point a
    a.setX(2); a.setY(1); a.setZ(5);
    // display coordinates
    cout << "vector coordinates a: "
         << a.getX() << ", " << a.getY()
         << ", " << a.getZ() << endl;
}
[UsePoint.cpp]
```

- A user of the point class will just need to **include** the header file at the top of the program.
- The coordinates of the point are set and retrieved in **main()**.

# Distance of Point from Origin

- A public class method is included that returns the distance of a point instance from the origin.

```
// declare point class
class Point {
public:
    :
    float distanceFromOrigin();
private:
    :
};
```

[Point.h]

```
:
// get distance from origin
float Point::distanceFromOrigin() {
    float distance = sqrt( pow(itsX,2)
        + pow(itsY,2) + pow(itsZ,2) );
    return distance;
}
:
```

[Point.cpp]

- Note that **private** member variables can be used in **public** class methods from the same class.

# Distance between Points

- What is the points distance from another point in space?

```

: [Point.cpp]
// distance between points
float Point::distanceFromPoint(float x, float y, float z) {
    float difX = itsX - x;
    float difY = itsY - y;
    float difZ = itsZ - z;
    float distance = sqrt( pow(difX,2)
                           + pow(difY,2)
                           + pow(difZ,2) );
    return distance;
}
:
```

```

: [UsePoint.cpp]
int main() {
    :
    float x = b.getX();
    float y = b.getY();
    float z = b.getZ();
    float dist =
    a.distanceFromPoint(x,y,z);
    :
}
```

- In this first iteration the **x**, **y** and **z** coordinates of one of the Point's is passed into the new Point's membre function. There's a better way...

# Passing Objects into Functions

- The point object can be passed directly into the member function.
- The member variables from another point obj have to be accessed via the dot operator.
- N.B. another object's member data can be accessed directly by an object of the **same** class **without** calling public accessor func.
- C++ is not a 'pure' object orientated language! (there's a compromise to enhance functionality)

```
class Point { [Point.h]
public:
    :
    float distanceFromPoint(Point anotherPoint);
    :
};
```

```
// distance between points
float Point::distanceFromPoint(Point anotherPoint) {
    float difX = itsX - anotherPoint.itsX;
    float difY = itsY - anotherPoint.itsY;
    float difZ = itsZ - anotherPoint.itsZ;
    float distance = sqrt( pow(difX,2) + pow(difY,2) +
                           pow(difZ,2) );
    return distance;
}
```

**[Point.cpp]**

```

:
int main() {
:
    float distance = a.distanceFromPoint(b);
:
}

```

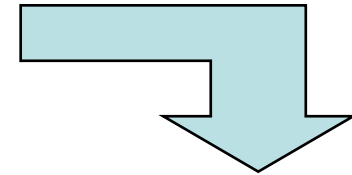
**[UsePoint.cpp]**

# Overloading Class Methods

- The two distance functions have different input args. Therefore a single member function **distance** can be overloaded (polymorphism).
- Overloading similar functions simplifies the class interface

```
class Point {  
public:  
:  
    float distanceFromOrigin();  
    float distanceFromPoint(Point anotherPoint);  
:  
};
```

[Point.h]

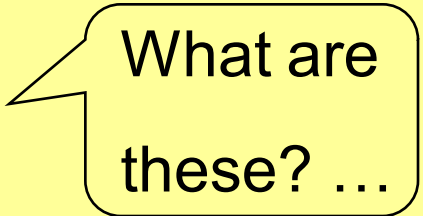


```
class Point {  
public:  
:  
    float distance();    // from origin  
    float distance(Point anotherPoint);  
:  
};
```



# Point Class Declaration

```
// point class declaration
class Point {
public:
    Point(float x, float y, float z);    // constructors
    Point();
    ~Point();                            // destructor
    void setX(float x);                  // accessors
    void setY(float y);
    void setZ(float z);
    float getX();
    float getY();
    float getZ();
    float distance();
    float distance(Point anotherPoint);
private:
    float itsX;                          // member variables
    float itsY;
    float itsZ;
};
```



What are  
these? ...