

# Introduction to Programming using C++

## Lecture Six: Classes in Practice

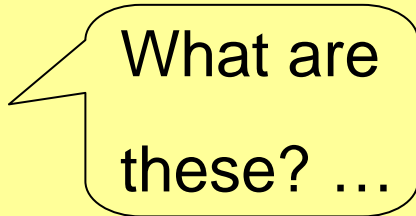
Carl Gwilliam

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

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

# 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? ...

# Constructors

- After a variable has been declared it has to be assigned a value before it can be used in an expression..

```
int i;           // declared
i = 7;          // initialised
int j = 7;       // declared & initialised
Point a;         // declared
a.setX(7.0);     // tedious initialisation
```

- Class objects are initialised by the use of a **constructor**: whenever an object of a class is created, its constructor is called
- Remember, the built-in types can also be initialised by the constructor method:

```
int j = 7        // normal initialisation
int j(7);        // constructor initialisation
```

# Constructors

- A constructor is a special public member function:
  - The function has the same name as the class
  - There is no return type and a **return** statement is not required

```
class Name {  
public:  
    Name();  
    :  
};      [Name.h]
```

```
:  
Name::Name() {  
    :  
}  
:      [Name.cpp]
```

- A default constructor is provided by the compiler (only if you don't provide any) but this is overwritten if a constructor is supplied in the class declaration.

# Using Constructors

- Declaring a constructor enables the member data to be initialised when the class object is instantiated.
- Constructors are preferred to multiple accessor methods (“set” methods) if the member variables are initialised but never reassigned.

```
// point class declaration      [Point.h]
class Point {
public:
```

```
    Point(float x, float y, float z);
```

```
    :
```

```
private:
```

```
    :
```

```
}
```

```
# include “Point.h”
```

```
int main() {
```

```
    Point a(1,2,1);
```

```
    :
```

```
}      [UsePoint.h]
```

```
:
// point constructor
```

```
Point::Point(float x, float y, float z)
```

```
{
```

```
    itsX = x; itsY = y; itsZ = z;
```

```
}
```

```
:
```

```
[Point.cpp]
```

# Overloading Constructors

- Like all functions, constructors can be **overloaded**. They are distinguished by the number and type of input arguments.
- The use of different constructors allows member data to be initialised in multiple ways.
- A constructor with no arguments does not have to be called with empty parentheses.

```
// point class declaration [Point.h]
class Point {
public:
    Point(float x, float y, float z);
    Point();
    :
}
```

```
# include "Point.h"
int main() {
    Point a(1,2,1);
    Point b; //origin
} [UsePoint.h]
```

```
:
// point constructors
Point::Point(float x, float y, float z)
{
    itsX = x; itsY = y; itsZ = z;
}
Point::Point() {
    itsX = 0; itsY = 0; itsZ = 0;
}
: [Point.cpp]
```

# Destructors

- Every time a constructor is declared it is advisable to provide a destructor.
- The destructor also has the same name as the class but with a **tilde (~)** in front

```
// point class declaration
class Point {
public:
    Point(float x, float y, float z);
    Point();
    ~Point();
    :
}                                     [Point.h]
```

```
// point class declaration
Point::~~Point() { }                [Point.cpp]
```

```
                                     [UsePoint.cpp]
:
int main() {
    if (i > 1) {
        Point a(1,2,1);
        :
    } // out of scope so
    : // destructor called
}
```

- The class destructor is called when the object goes out of scope or delete is called (see later).

# Const Member Functions

- **Const** can be applied:
  - before a variable declaration to indicate that the value will remain constant until it goes out of scope.
  - before a pointer declaration to indicate that the pointer will always refer to the same object until pointer goes out of scope.
  - AND **after** a member function.
- **Const** after a member function means that the member data can't be modified by that function.
  - This is a safeguard for both developer and user against future modifications to the member function.

```
:
const double PI = 3.14;
double* const px = &x;
:
// declare point class
class Point {
public:
    :
    float getX() const;
    :
};                                     [Point.h]
```

```
:
// get x coordinate
float Point::getX() const {
    itsX++; // error
    return itsX;
}
:                                     [Point.cpp]
```



# Inline Member Functions

- It is common to have some class methods only a few statements in length.
- The definition of these functions can be moved to the interface (header) without loss of generality.
- Member functions that are declared and defined in the same statement are called **inline** functions.
- Making functions inline increases the program efficiency if they're called frequently. Efficiency is not increased if a large function is made inline.

```
class Point { // declare point class
public:
    Point(float x, float y, float z);
    Point();
    ~Point();
    void setX(float x) {itsX = x;}
    void setY(float y) {itsY = y;}
    void setZ(float z) {itsZ = z;}
    float getX() const {return itsX;}
    float getY() const {return itsY;}
    float getZ() const {return itsZ;}
    float distance();
    float distance(Point anotherPoint);
:
};
```

# A Line Class

- We can now apply same design to create a **line** object.
- A line is defined as a point of origin in space, and a direction in x, y and z.
- A line can be constructed in three ways:
  - Start from the origin and give explicit x,y and z direction
  - Give a specific starting **point** and x,y and z direction
  - Define the line as the distance between two **points**
- The initial design indicates that a **point** object is required in the construction of a **line** object.
- The idea of objects being constructed from other objects is pivotal to OO thinking. Otherwise classes remain as elaborate data structures.

# Constructing a Line Class

```
class Line {                                     [Line.h]
public:
    Line(Point o, float x, float y, float z);
    Line(float x, float y, float z);
    Line(Point a, Point b);
    ~Line();
    float getXdir();
    float getYdir();
    float getZdir();
private:
    Point the_origin;
    float itsXdir;
    float itsYdir;
    float itsZdir;
};
```

```
#include "Point.h" [Line.cpp]
#include "Line.h"
// constructor for two points
Line::Line(Point a, Point b) {
    the_origin = a;
    itsXdir = b.getX() - a.getX();
    itsYdir = b.getY() - a.getY();
    itsZdir = b.getZ() - a.getZ();
}
```

```
: #include "Point.h"
   #include "Line.h"
   int main() {
       Point a(1,0,1), b(3,2,6);
       Line l(a,b);
       :
   } [UseLine.cpp]
```

- To use one class within another we must **include** the **header** file & put **source** file in the compile command

# Object Initialisation

- Whenever an object of a class is created, its constructor AND the constructors for all objects that belong to it are called. This is done before the class's own constructor is called
- By default, the constructors invoked are the default ones.
- You can assign a data member to something specific inside the class's constructor by
  - Creating a new point and then assign it to the object.
  - Creating anonymous point obj & assign straight to data member.

```
class Rectangle {  
    :  
private:  
    Point itsLowerLeft;  
    Point itsLowerRight;  
    Point itsUpperLeft;  
    Point itsUpperRight;  
};
```

```
Rectangle::Rectangle() {  
    Point itsLowerLeft(0,0,0); // local  
    itsLowerRight(1,0,0);      // error  
    Point ul(0,1,0);  
    itsUpperLeft = ul;          // ok  
    itsUpperRight = Point(1,1,0); // better  
}  
:
```

# Initialisation Lists

- Initialisation lists can be used to call a non-default constructor for an object belonging to a class **directly** when an object of that class is created.
  - this is faster as it avoids an extra assignment step

```
Name::Name(input_args):  
    memVar1(init_expr),...,memVarN(init_expr)  
{  
    statements;  
}
```

```
// constructor with start and direction  
Line::Line(Point o, float x, float y, float z):  
    the_startPoint(o),  
    itsXdir(x),  
    itsYdir(y),  
    itsZdir(z)  
{ ... }
```

- The order of the objects in the initialisation list should be that in which they were declared (compiler will reorder)

# Pointers to Objects

- A pointer can be used to point to an object as well as the built-in types.
- An object's member variables and functions are then accessed through the **indirection** operator (->).
  - This is equivalent to dereferencing the pointer to get the object itself and then using the dot operator (see next slide).
- Pointers enable objects to be passed by 'reference' to non-member and member functions => Allows the object to be modified through a function call.
- The efficiency of the program is increased if a large object is passed by reference rather than by value (as only the address is passed).

# Pointers to Objects

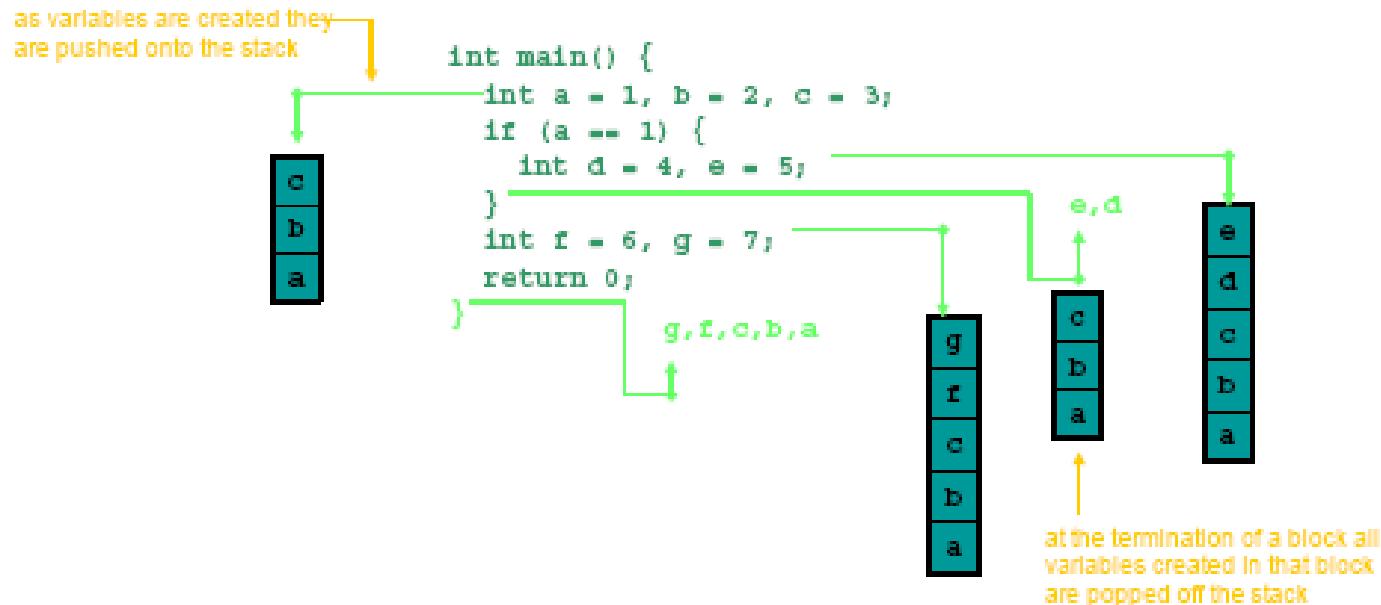
```
float getDist(Point* pPoint) { //pointer
    return pPoint->distance(); //indirection
    // equivalent to (*pPoint).distance();
}
float getDist(Point& rPoint) { //reference
    return rPoint.distance(); //no indirection
}

int main() {
    Point a(2,1,3);
    //pointer to object of type Point
    Point* b = &a;
    // reference to object of type Point
    Point& c = a;
    float dist = b->distance();
    cout << "distance b: " << dist << endl;
    cout << "distance b: " << getDist(b) << endl;
    cout << "distance c: " << getDist(c) << endl;
    return 0;
}
```

- We can also use references with objects.
- These have the same advantages, but without the need for the use of an indirection operator.
- So why so we need pointers?

# The Stack

- All the variables & user-defined objects in the programs so far have been stored in an area of memory called the **stack**.



- This is a simple and effective method of managing memory. There are never any residual objects that remain on the stack at the end of the program.



# Limitations of the Stack

- For large programs holding variables on the stack there are two limitations:
  - Objects do not persist after the block they were declared in.
  - The stack is a relatively small area of memory. Programs that have to deal with a number of large objects will encounter problems with memory size (it's not infinite!).
- Global variables are a possible solution to the first limitation. But it is generally bad practice to use global variables for anything else than constants.
- Need an area of computer memory capable of containing persistent large objects that can still enforce the same access restrictions that the stack provides...

# The Heap

- The heap (or free store) is a large reserved area in computer memory able to store persistent objects.
- To get an address (i.e. pointer) for the allocated memory on the free store you have to use the **new** keyword.
- The class (or built-in type) after **new** tells the compiler how much memory to allocate to store the new object.

```
int* px = new int;           // int on heap
float* py = new float(5);    // float initialised to 5 on heap
float* par = new float[10];  // array of floats on heap
Point* pA = new Point(1,2,1); // Point object on heap (used earlier)
cout << "x coord of A: " << pA->getX() << endl;
```

- Class methods for an object on the free store are accessed through the indirection operator.
- Initialisation is done via the constructor formalism

# Lifetime of Objects on the Heap

- Objects created on the free store exist beyond scope of the block in which they're declared.
- However, the returned pointer to the object is located on the stack.
- To access the object from a function the ptr to the object must be passed to that function.
- Placing the ptr on stack enables func interface to remain intact.

```
int main() {
    getPoint(2,1,7);
    Point* b = getPoint2(3,4,3);
    // error
    cout << "x(a)= " << a->getX() << endl;
    //ok
    cout << "x(b)= " << b->getX() << endl;
    return 0;
}

void getPoint(float x, float y, float z) {
    // object persists but ptr not passed
    // back to main, so we can't use it
    Point* a = new Point(x,y,z);
    return;
}

Point* getPoint2(float x, float y, float z) {
    Point* b = new Point(x,y,z);
    return b;
}
```

# Memory Management

- What happens to the allocated memory once the pointer runs out of scope?
- C++ delegates the responsibility of allocating and freeing memory on the heap to the software developer.
- This is one of the most contentious issues surrounding C++. Other OO languages, such as Java, automatically de-allocate memory used on the heap.
- Unfortunately, you **will** have to use to free store. It is better to be aware of the traps and pitfalls associated with memory management now before you lose complete control...

# The **delete** Keyword

- Memory allocated for an object on the heap has to be freed when the object is no longer needed or, at the latest, before the object goes out of scope.
- The memory is freed by using the **delete** keyword.
- All objects stored on the heap (declared with **new** keyword) must have a corresponding **delete** statement.

```
int main() {  
    getDistance();  
    return 0;  
}  
  
void getDistance() {  
    Point* a = new Point(2,3,2);  
    Point* b = new Point(2,6,-4);  
    cout << "dist a to b= "  
        << a->distance(*b);  
    << endl;  
    delete a;    // clean-up  
    delete b;    // memory  
    return;  
}
```

# delete in Objects

- If a class has data members on the heap, the memory associated to them must be freed when an object of that class goes out of scope.
- This is done by placing delete statements in the class's **destructor**.

“I don't need to bother with that. The compiler doesn't even warn me. Will it really affect the running of my program?”

```

:                                     [Point.cpp]
// constructor
Point::Point(float x, float y, float z) {
    // value placed on heap
    itsX = new float(x);
    :
}
:
// destructor: delete memory on heap
Point::~~Point() {
    delete itsX;
    :
}
:
float Point::getX() {
    //deref ptr to get val
    return *itsX;
}
:

class Point {
public:
    :
private:
    float* itsX;
    float* itsY;
    float* itsZ;
};                                     [Point.h]
```