Module 7:
Constructors and Destructors:

In this module we will cover:

- **Constructors**
  - **Default constructor**
  - **Other constructors**
  - **Copy constructor**
- **Quirks about constructors**
- **Destructor**

Aims of this module

We have previously introduced the concept of a “class” as a construct which combines member variables and methods in a formal way.

In this module you will learn about a new and vitally important part of a class which determines how an object gets made.

This all revolves around the idea of "Constructors" which are special methods which get called automatically to initialise objects.

Associated with these are "Destructors" which get called when an object goes out of scope or is deleted.

These are fundamental new concepts in OO and you must use these from early on. Therefore this module covers their use in some detail.

7.1 Constructors

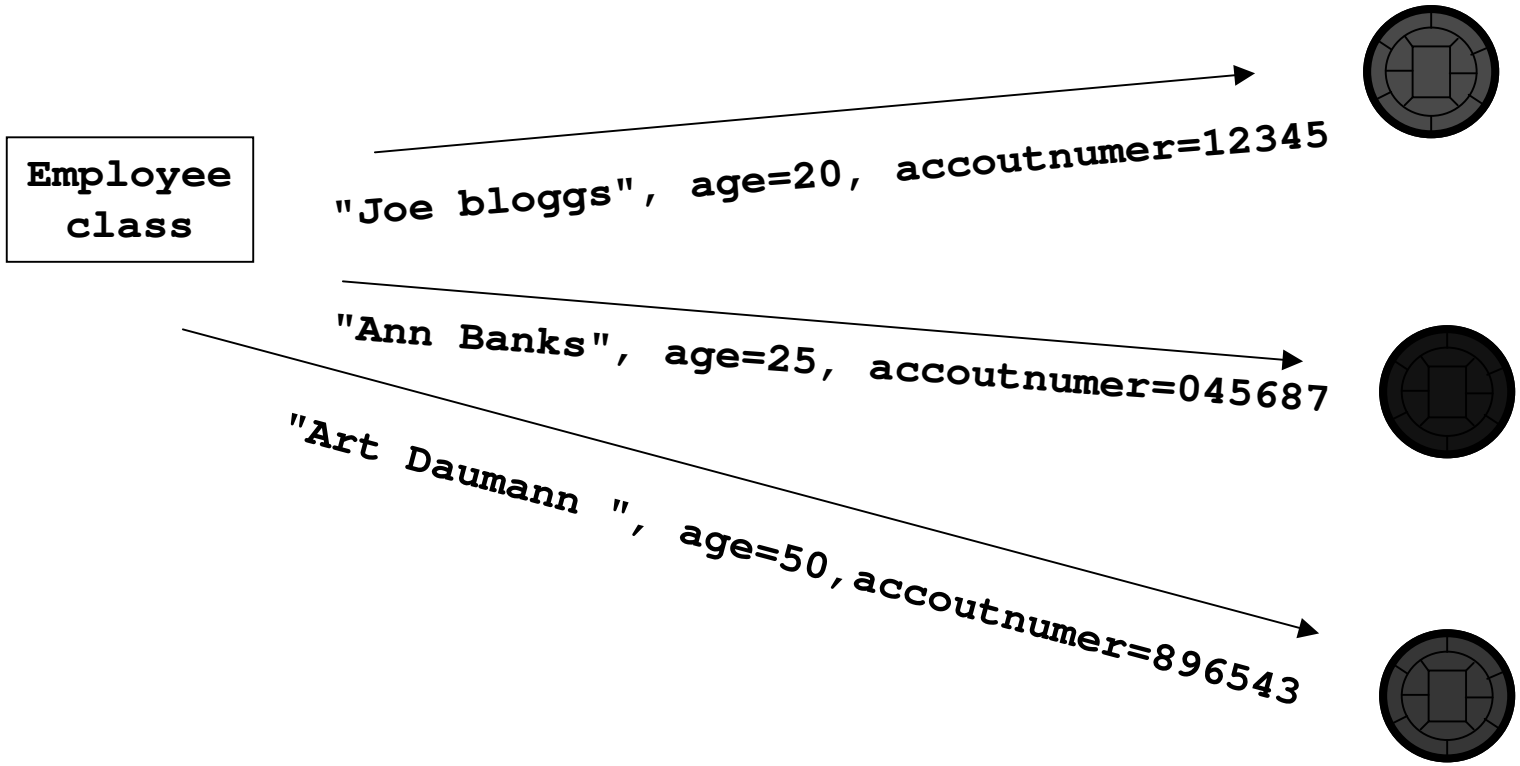
Throughout the course so far we have seen the idea that as soon as you make an Object then you should remember to initialise is so that its member variables are set to something sensible. I.e

```
// Make a ThreeVector and initialise the px,py,pz

ThreeVector vec ;

vec.initialise( 0.8, 0.4, 1.3 )
```

If you DO NOT do something like this then you may find that your internal variables are full of some random garbage like -1111111111345678, or whatever happened to be left in the memory from a previous use.

We can say the same thing in a more abstract way:

When you make an object it is unlikely to be of any use unless you give it "state" which distinguishes it from other objects:



So far we have always done this by invoking this special "initialise" method explicitly after making the object.

C++ formalises this by defining a "special method" which is invoked automatically for you when you make the object. This is called a

Constructor

To start we will see a so called:

Default Constructor

This doesnt actually do anything useful in the way of giving state, but shows the syntax.

Then we look at more useful ones which take arguments

The Default Constructor

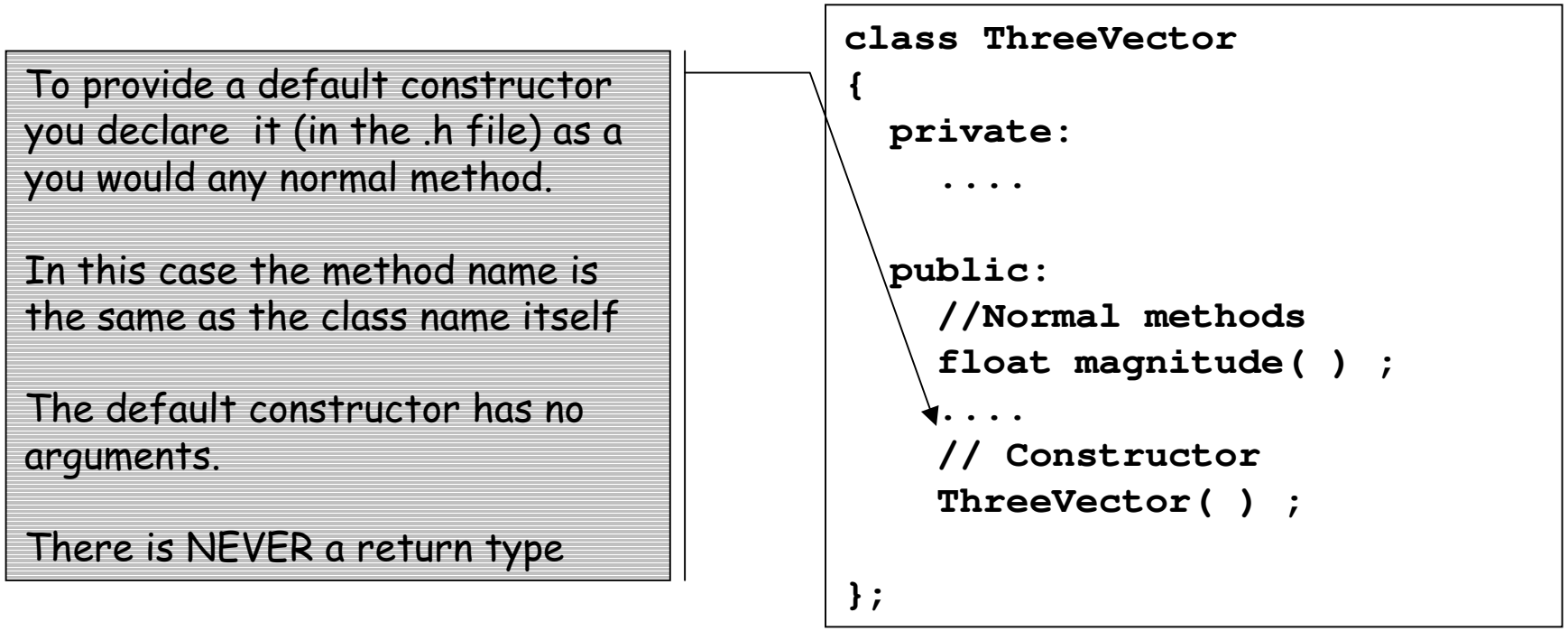
(no arguments)

To provide a default constructor you declare it (in the .h file) as a you would any normal method.

In this case the method name is the same as the class name itself

The default constructor has no arguments.

There is NEVER a return type

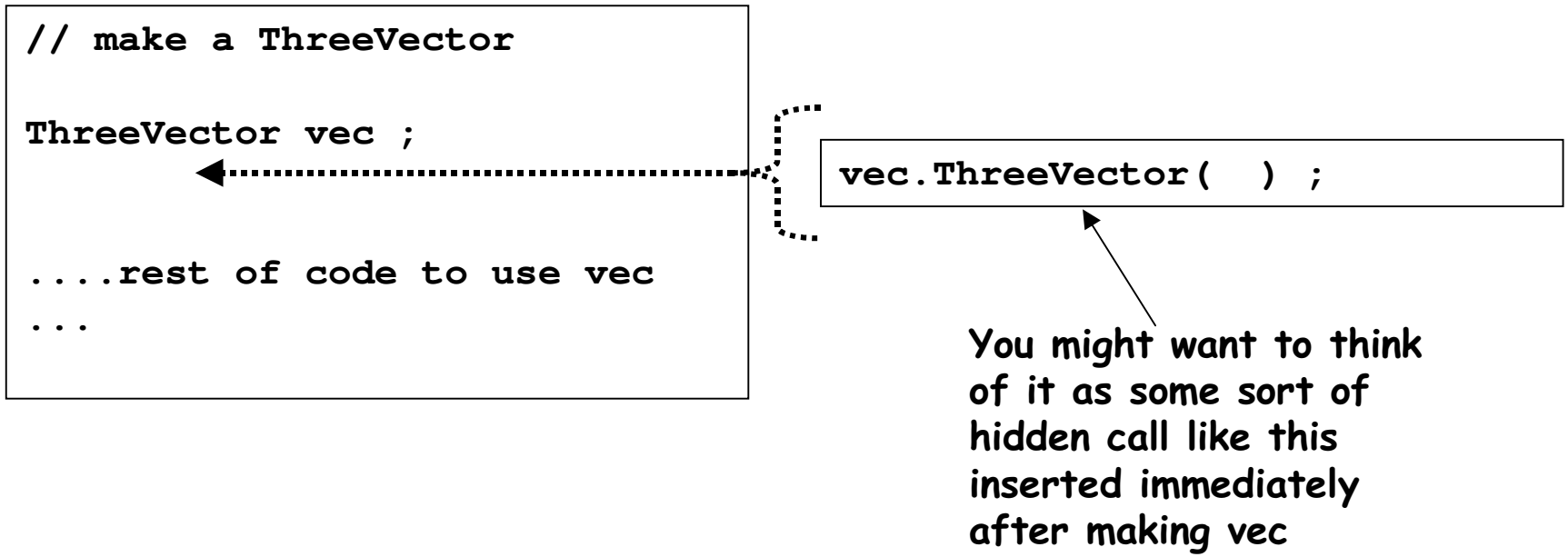


You write the code for the method as you would for a normal method, except that there is no return type.

This is of course normally sitting in a .cpp file

```
....

// Magnitude method
float ThreeVector::magnitude( )
{
   ....
   ....
}


// Default constructor
ThreeVector::ThreeVector( )
{
  px = 0 ;
  py = 0 ;
  pz = 0 ;
}
```

Now, whenever you make a ThreeVector, this default constructor will automatically be called for you



Other Constructors

(ones which take some usefel arguments)

```
class ThreeVector
{
  private:
     ....

  public:
     //Normal methods
     float magnitude( ) ;
     ....
     // Constructors
     ThreeVector( ) ;
     ThreeVector( float x, float y, float z ) ;
};
```

Here is an extra constructor which takes three floating point arguments.

We would use this to initialise to a specific (x,y,z) rather than just (0,0,0).

Here is the extra code in the .cpp file
for the constructor taking three floats

```
....

// Default constructor
ThreeVector::ThreeVector( )
{
  px = 0 ;
  py = 0 ;
  pz = 0 ;
}

// Extra constructor
ThreeVector::ThreeVector( float x, float y, float z )
{
  px = x ;
  py = y ;
  pz = z ;
}
```

Suppose you want to make a ThreeVector and initialise it to (0.8, 0.4, 1.2).

To cause the appropriate constructor to be invoked when you make the ThreeVector you simply do this:

```
// make a ThreeVector

ThreeVector vec( 0.8, 0.4, 1.2 )  ;


....rest of code to use vec ...
```

Note the format: you are <u>not</u> supplying arguments to a method called vec

⇒ vec is the variable name !

⇒ you are putting arguments after the variable name at the point it is created

This is the ONLY way you can make a specific constructor get invoked:

```
// make a ThreeVector

ThreeVector vec( 0.8, 0.4, 1.2 )  ;

....rest of code to use vec ...
```

You can NEVER call it yourself later like this:

```
// make a ThreeVector

ThreeVector vec ;

... some code...

vec.ThreeVector( 0.8, 0.4, 1.2 )  ;
```

This is not allowed. A thing gets constructed ONLY when it is made, not later on

Student exercise

·Write an Animal class which has member variables to represent:
-The food type (carnivore, omnivore, herbivore)
- the number of legs

·Supply the class with (at least) two constructors:

1. One which takes the only the food type
This should assume the animal has 4 legs

2. Another which takes the food type, but also the number of legs in case it is not 4. You might like to make this check that the value supplied is even and print an error message if not.

·Write some code to demonstrate the use of these constructors.

Hint: I would provide a dump() method for the Animal class which can be used to print out the member variable information after you have constructed a few animals

7.2 The copy constructor

There is a special constructor which is very often provided for a class.

It is called the "copy constructor"

Its purpose is to constuct a new object by making an exact copy of an existing object.

This example shows how it is invoked:

```
// Example of using the copy
// constructor

// make a Thing

Thing t1  ;

//...set t1 to something ....

// make a second Thing which is
// a copy of t1 by invoking the
// copy constructor

Thing t2( t1 ) ;
```

This is how you (could) declare a copy constructor

```
class ThreeVector
{
  private:
     ....

  public:
     //Normal methods
     float magnitude( ) ;
     ....
     // Default Constructor
     ThreeVector( ) ;

     // Copy constructor
     ThreeVector( ThreeVector source) ;

};
```

The copy constructor takes another ThreeVector as its argument

This is how you write the code for the copy constructor

The member variables of the new object are set to equal the member variables of the source object

```
// This is in the .cpp file


// Copy constructor
ThreeVector::ThreeVector( ThreeVector src )
{
  px = src.px ;
  py = src.py ;
  pz = src.pz ;
}
```

Note how they are accessed from the source, using

src.px

If you find this difficult, you might like to realise that you have already done something very like this !

You did it when writing the `dotProduct( )` method for the ThreeVector class. This method took another ThreeVector as an argument

Look at this and see the similarity if you need to.

Now you need to know about references !

go to special module on technical topics to find out what a reference is.

In practice copy constructors are always written using a const reference to the source.

This is because there is no need to waste the time and memory making a temporary copy.

Thus a copy constructor is always declared thus:

```
// This is the .h file

class ThreeVector
{
  private:
     ....

  public:
    //Normal methods
    ...
    // Copy constructor
    ThreeVector( const ThreeVector& source);
};
```

And written thus:

Note that the code itself is unchanged

```
// This is the .cpp file

// Copy constructor
ThreeVector::ThreeVector( const ThreeVector& src )
{
   px = src.px ;
   py = src.py ;
   pz = src.pz ;
}
```

7.3: Quirks about Constructors

IMPLICIT & EXPLICIT DEFAULT CONSTRUCTORS

If you ever want to make an object in the simplest way, i.e like this ....

```
// make a Thing

Thing t  ;
```

...then you have to realise that the compiler will only let you do this if there exists a default constructor:

`Thing( )`

i.e a constructor with no arguments which the system can call.

Thus you have to be aware of when a default constructor exists or not !

1. If you NEVER explicitly declare or write ANY constructors

i.e your .h file looks like this ....

```
class Thing
{

 // Normal variables
 float xxx ;

 // Normal methods

 void aService( ..) ;

 // But no constructors

};
```

...then the compiler in effect provides an implicit default constructor free.

You never see it, but it is as if

```
Thing( ) ;
```

exists.

```
// Then this always works

Thing t  ;
```

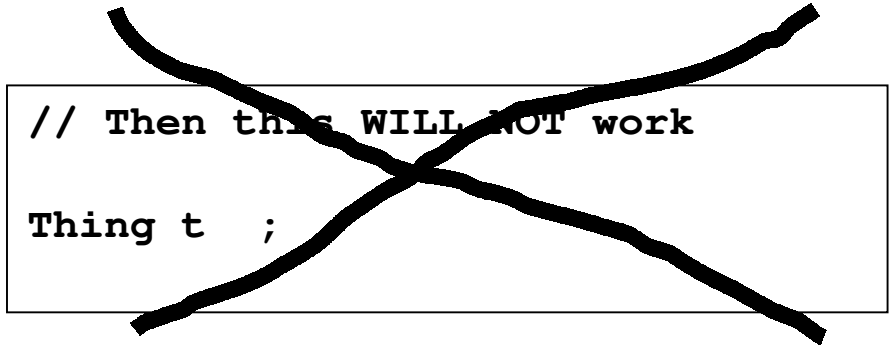
2. If you DO explicitly declare or write ANY constructor at all

i.e your .h file looks like this ....

...then the compiler NO LONGER provides an implicit default constructor.

It asssumes you know what you are doing, and have taken charge of constructors.

```
class Thing
{

 // Normal variables
 float xxx ;

 // Normal methods

 void aService( ..) ;

 // A constructor

 Thing( float initial ) ;

};
```



```
// Then this WILL NOT work

Thing t  ;
```

3. If you DO explicitly declare or write constructors, and you INCLUDE your own default constructor

i.e your .h file looks like this ....

```
Class Thing
{
 // Normal variables
 float xxx ;

 // Normal methods

 void aService( ..) ;

 // My default constructor
 Thing( ) ;

 // Possible other constructors
 Thing( .... ) ;

};
```

```
// Then this will work

Thing t  ;
```

IMPLICIT "COPY" CONSTRUCTOR

The compiler ALSO provides an implicit COPY constructor free.

What this means is that you can always do this..

```
// Create t1

Thing t1 ;

// Create t2 as a COPY of t1

Thing t2( t1 ) ;
```

`t2` is a direct copy of `t1` !

In effect the compiler has provided a constructor like this:

```
Thing( const Thing& source )
```

However is only does a bit by bit copy of the object. This may not be what you want ! Always therefore consider whether you need to write your own explicit copy constructor

INITIALISATION of MEMBER VARIABLES

You might already have noticed that you CANNOT do this..

```
class Thing
{

  float x = 0. ;
  int   y = 1 ;

}
```

.. the compiler throws a wobbly.

In order to initialise member varibles you must do it in the constructor like this....

```
Thing::Thing( )
{

  x = 0. ;
  y = 1 ;

}
```

INITIALISATION of CONST MEMBERs

Similarly you CANNOT do this..

```
class Thing
{

  const float x = 0. ;
  const int   y = 1 ;

}
```

Unfortunately .. NOR can you do this when the members are const....

```
Thing::Thing( )
{
  x = 0. ;
  y = 1 ;
}
```

In order to initialise `const` members you must do it in the constructor like this....

```
Thing::Thing( )

: x(0.),  y(1)

{
   ....
   ... rest of constructor ..
}
```

... or like this...

```
Thing::Thing( float xinit, int yinit )

: x( xinit ),  y( yinit )

{
   ....
   ... rest of constructor ..
}
```

This is NEW !!!!

- It is called a member initialisation list
- For now just be aware that they are needed for some operations
- I WILL NOT require you to know this for the course - its too obtuse

7.4: Destructor

The constructor has an opposite. It is called the:

Destructor

This gets called automatically when an Object goes out of existence [I.e. at he end of a {...} in which it was created.

Its function is to tidy up anytihng the object might have done before it gets destroyed.

To supply a destructor you add

~ClassName( )

as a new special method

```
// The .h file for

class Thing
{

  public:

    // A constructor
    Thing( ) ;

    // A destructor
    ~Thing( ) ;

};
```

Then you write this code to implement the destructor.

```
// In the .cpp file

Thing::~Thing( )
{
   ... code in here to tidy
   ... up Thing object
}
```

For the most part you don't need to supply a destructor.

It is however considered to be good practice to always write a destructor, even if it is empty to start with.

If you have allocated memory from within your Object, then you will probably have to write a destructor which explicitly deletes this memory.We will return to this later.

Student exercise to do in your own time (i.e. between now and next session)

- Modify your ThreeVector class to provide some sensible constructors

Write

- a default constructor to set everything to zero
- an obvious constructor to take x,y,z as arguments
- a copy constructor

- Write some code to demonstrate that you know how to use these constructors.

- We will look at it next session

util/
ThreeVector.h
ThreeVector.cpp

Summary of Module 7: Constructors and Destructors

In this module we have covered the following topics.

- **Constructors**
  - **Constructors are called automatically when you make an object**
  - **You can provide several constructors with different argument lists**
  - **The correct constructor is called according to the arguments to supply when making the object**
  - **Copy constructor**
- **Destructor**
  - **The destructor is called when an object get destroyed. Its function is to cleanly end the life of an object**