# Module : Technical topics

**In this module we have grouped together several unrelated technical topics.**

**We call them "technical" as they have not much to do with the main theme of "Object Oriented" thinking. Some items for example have no relevance in Java.**

**They must be covered as the course progresses in order to develop technical competence in the way we write C++ code.**

**These slides will be covered as necessary during the course. However it would be a very good idea for students to try to appraise these topics in advance by themselves using these slides and the course text books.**

- **main( )**
- **#include files : user files and system files**
- **Organisation of classes into .h and .cpp files.**
- **The C pre-processor**
- **References**
- **const**
- **scope**
- **namespaces**
- **casting / conversion**

Important:

You are expected to look into these topics yourself outside of the lectures !

We cover them here only to show you what you need to know.

By all means ask questions in lectures and in surgery, but do not assume that just because you have seen it once now then that is all you need to do.

These items are defined as part of your non-contact hour course work.

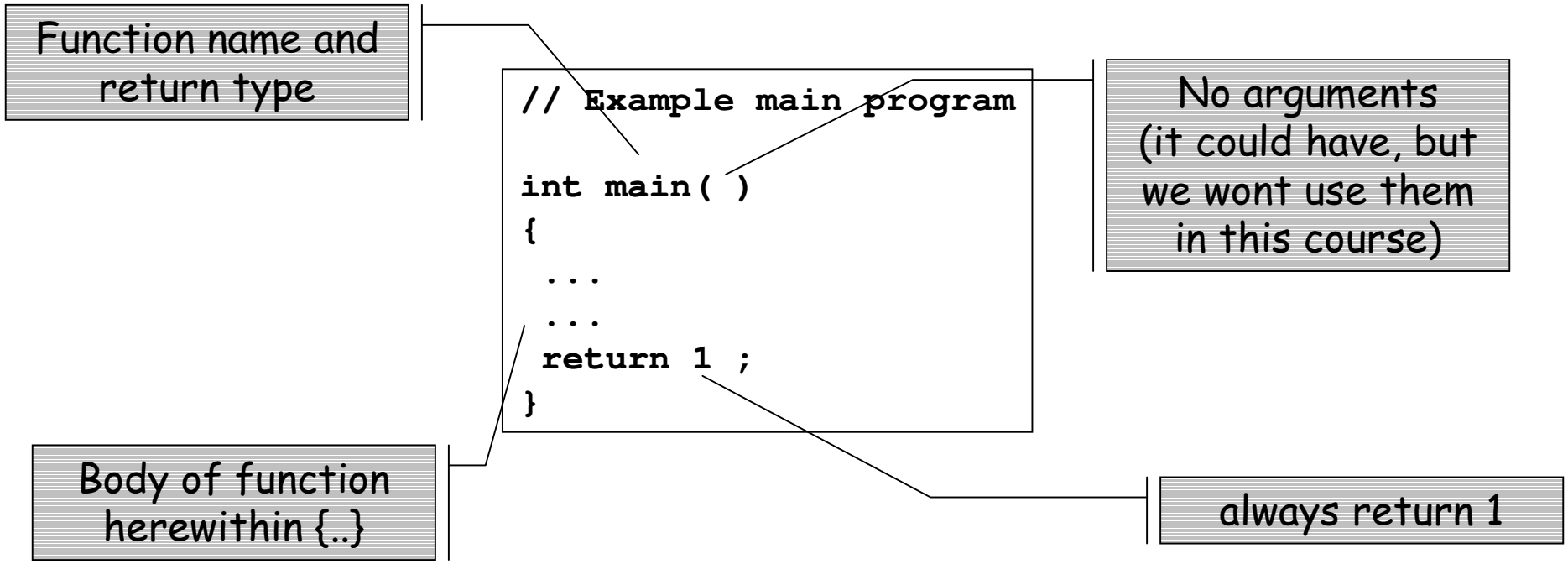
# Technical 1:

# main( )

**In order to build an executable program there must exist somewhare in the code which is linked together a function called**

```
int main( )
```

**which returns an integer (for success or failure).**



**There must be exactly 1 and only 1 main( ) function**

**If there are more the linker will complain. This is a common problem whioch arises if you include two files, each of which has a main( ) function.**

# Technical 2:

# #include

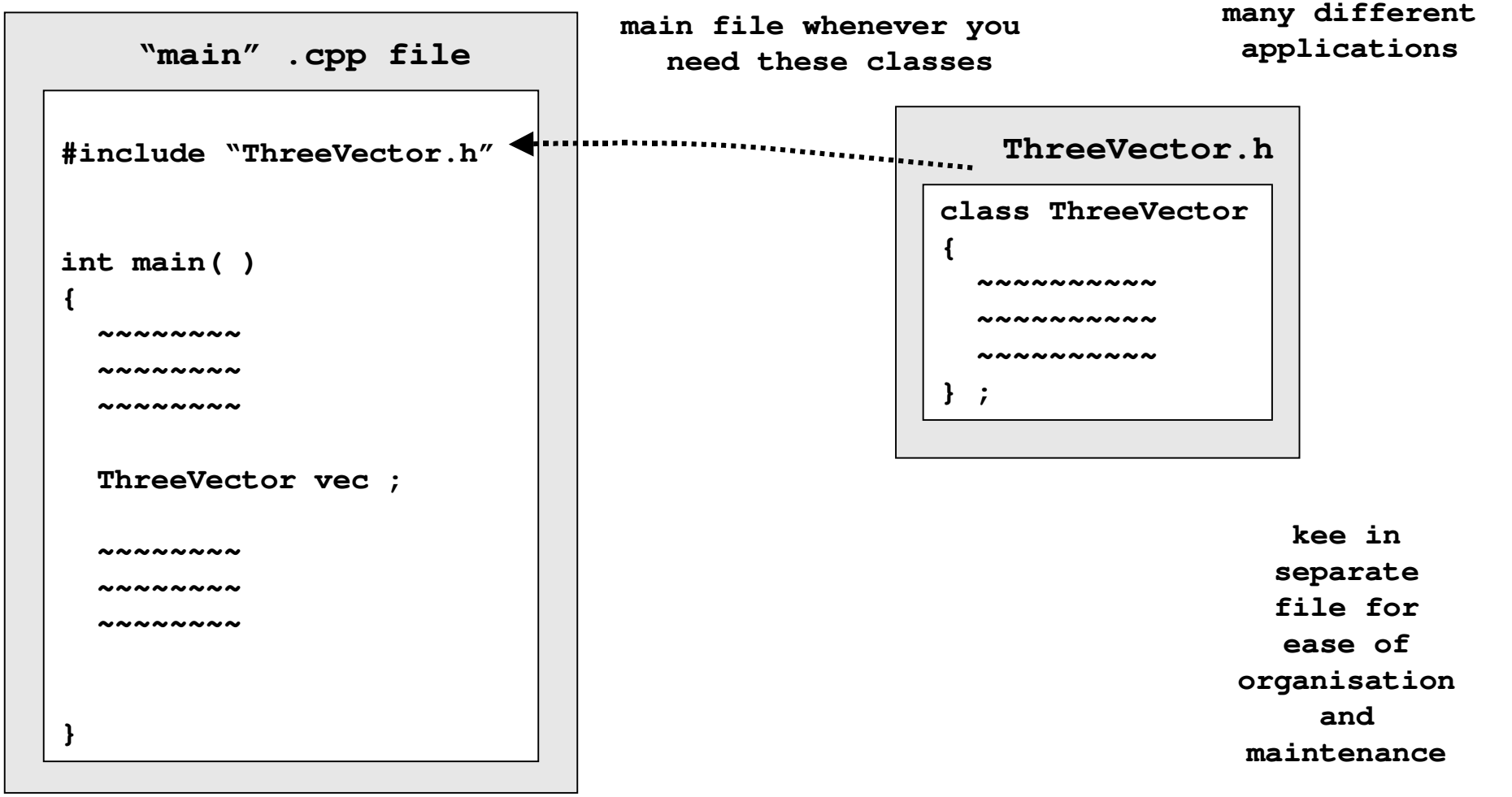
- The #include directive in any file tells the pre-processor to include the contents of that file at this point.

- This happens before the compiler sees the file. In other words by the time the compiler runs it is as if there was one big file with all the #included replaced by the relevant code.

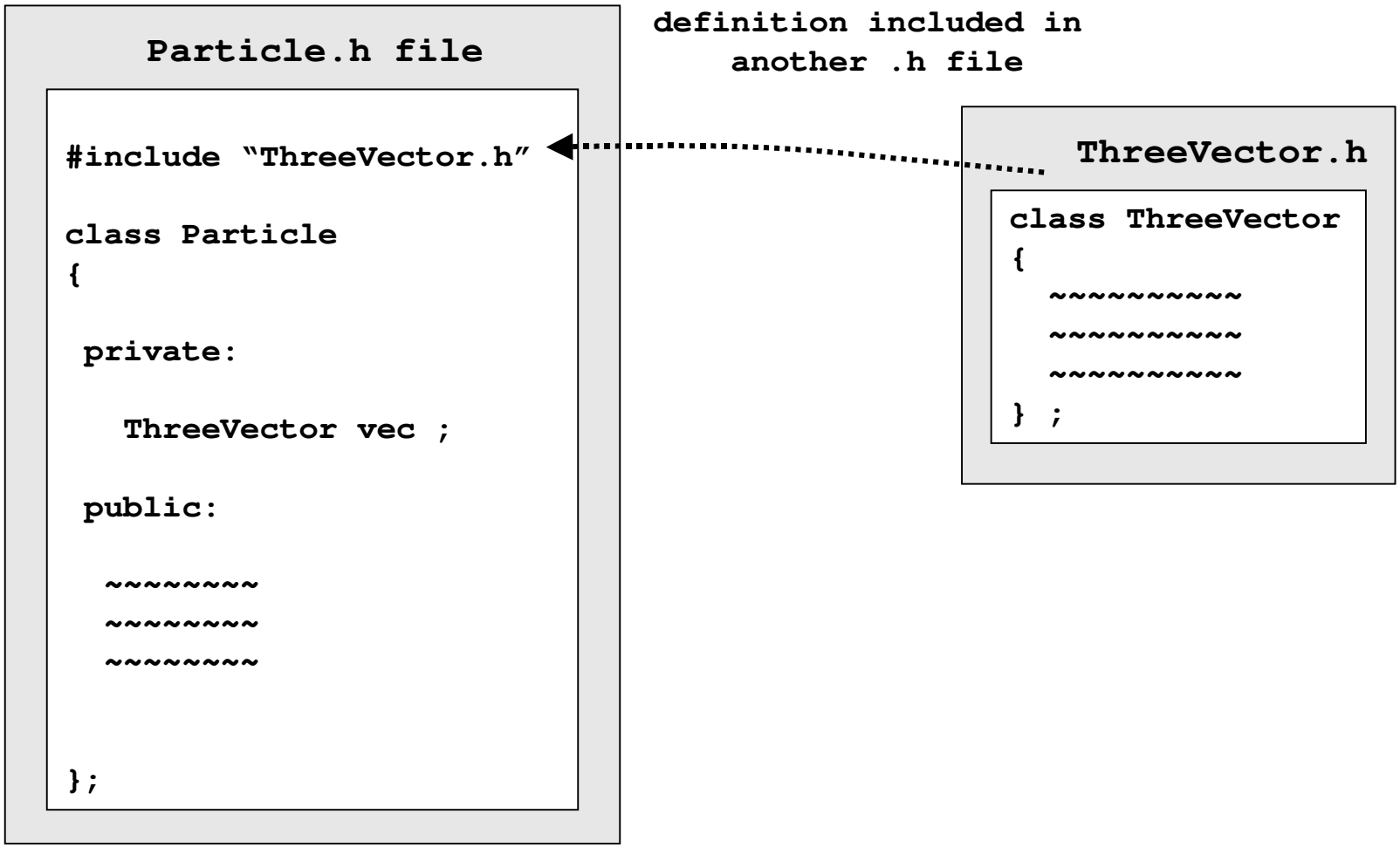
- #include directives normally appear at the top of a file.

- The following slides show the most common uses of #include files

## Use of a class in a main function



## Use of a class by another class in its declaration



- There is another way to #include things.

- This is used when you are including "files known to the system". These are typically files like iostream, or cmath.

- The system keeps these in a special directory which it knows about

- You tell the pre-processor to include from there using the <..> format:

```
#include <iostream>
#include <cmath>

int main()
{

  ... your code using ...

  return 1 ;
}
```

# Technical 3:

# Organisation of classes into .h and .cpp files

Earlier in the course modules we wrote a classes completely in one file.

This means that we wrote something like this in a single .h file

In particular we wrote all the code for each method directly inline as part of the class { ..} structure

**ThreeVector.h file**

```
class ThreeVector
{
 private:
   double px, py, pz ;

 public:

  double magnitude( )
   {
     ~~~~~~~~
   }

  double dotProduct( ThreeVector rhs )
   {
     ~~~~~~~~
   }

   ...other methods....
};
```

**In practice C++ classes are not written this way.**

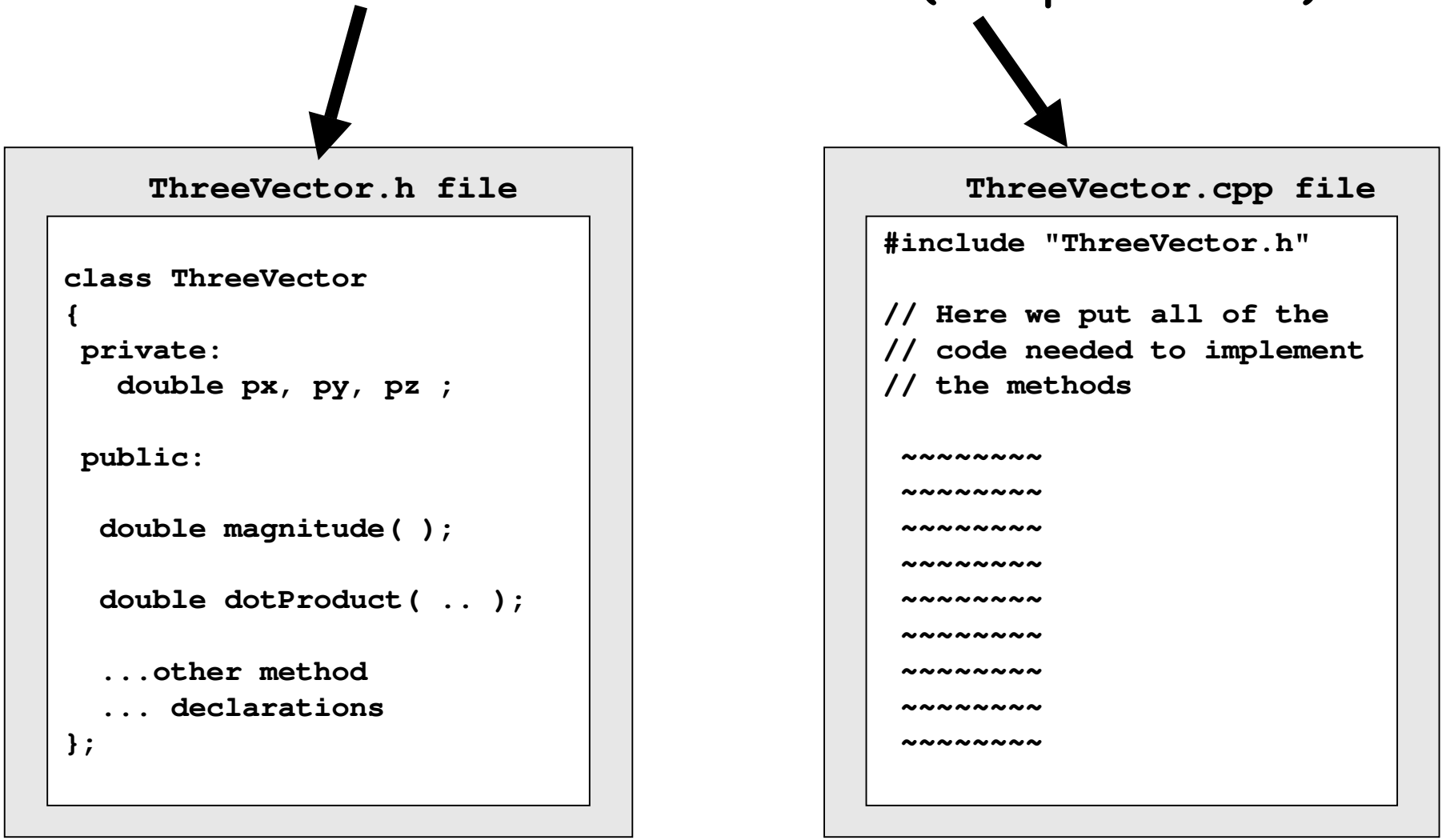
**There are several reasons, but the simplest is just clarity.**

**You can well imagine that if there are several complex methods in a class, then the .h file becomes unreadable, and it is difficult to see at a glance what methods the class possesses**

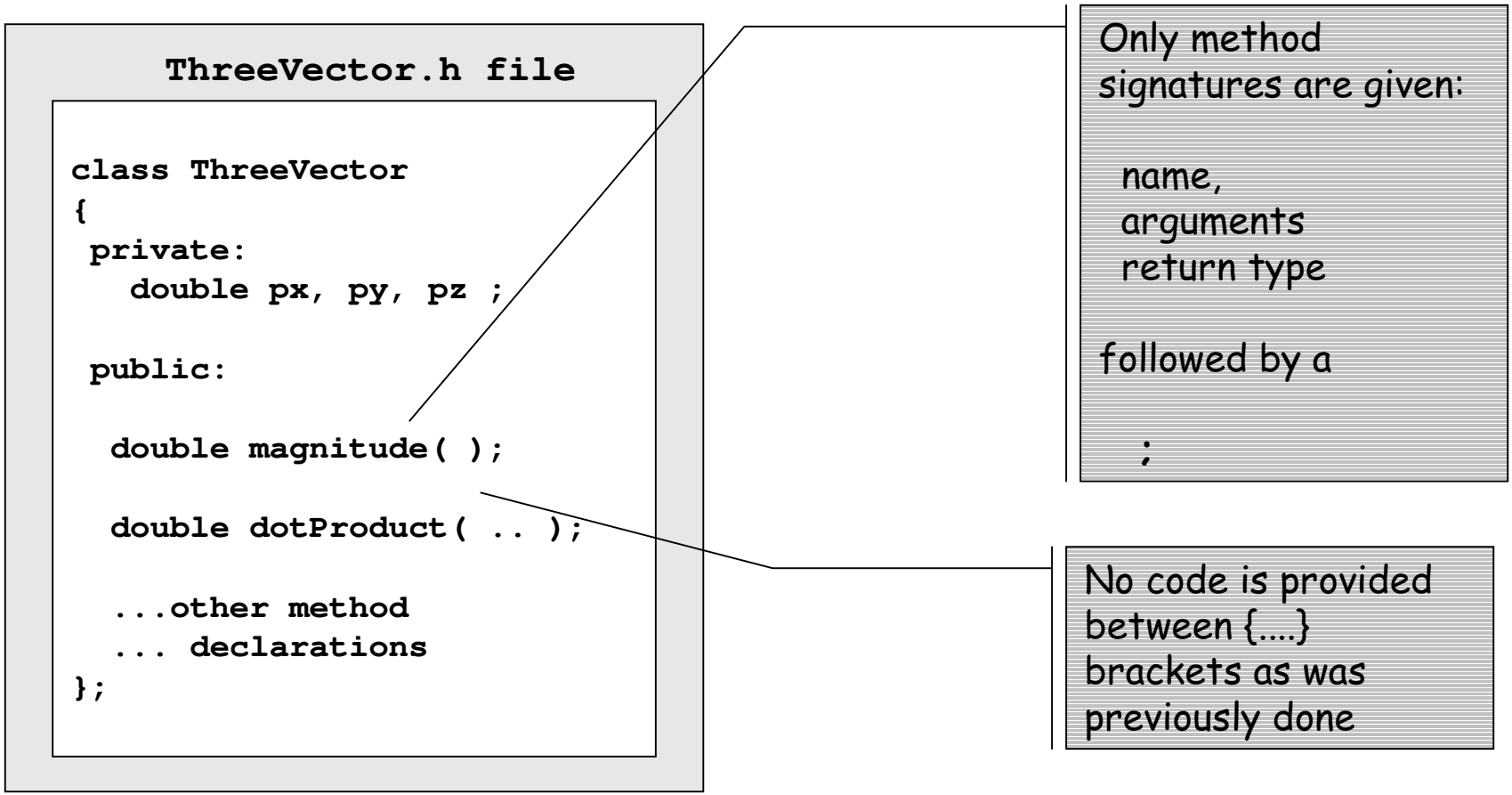
**(..although in Jave you can only do it this way...)**

This is avioded by splitting the

declaration from the definition (or implementation)



## The declaration (.h) file looks like this:




**In other words we promise that a method will be provided with the given signature, but this tells the compiler that it will be found somewhare else in its travels.**

**ThreeVector.h file**

```
class ThreeVector
{
 private:
   double px, py, pz ;

 public:

  double magnitude( );

  double dotProduct( .. );

  ...other method
  ... declarations
};
```

**This .h file is now short and readable. It tells the programmer very clearly what methods the class provides.**

**It can also be included as before into any other files which want to use the ThreeVector class.**

**Such files only need the declaration of the method signatures to work, and do not rely in any way on the implementation of those methods.**

## The definition (.cpp) file looks like this:
## (I call this the implementation file also)

**ThreeVector.cpp file**

```
#include "ThreeVector.h"


// Here is the code for the magnitude method

double ThreeVector::magnitude()
{
  return sqrt( px*px + py*py + pz*pz ) ;
}


// Here is code for dotProduct method

double ThreeVector::dotProduct( ThreeVector rhs )
{
  return ( px*rhs.px + py*rhs.py + pz*rhs.pz ) ;
}
```

The .cpp file always #includes the .h file

**ThreeVector.cc file**

```
#include "ThreeVector.h"


// Here is the code for the magnitude method

double ThreeVector::magnitude()
{
  return sqrt( px*px + py*py + pz*pz ) ;
}


// Here is code for dotProduct method

double ThreeVector::dotProduct( ThreeVector rhs )
{
  return ( px*rhs.px + py*rhs.py + pz*rhs.pz ) ;
}
```

Here is something really new:

**ThreeVector.cc file**

```
#include "ThreeVector.h"


// Here is the code for the magnitude method

double ThreeVector::magnitude()
{
  return sqrt( px*px + py*py + pz*pz ) ;
}


// Here is code for dotProduct method

double ThreeVector::dotProduct( ThreeVector rhs )
{
  return ( px*rhs.px + py*rhs.py + pz*rhs.pz ) ;
}
```

**This is because you have to tell the compiler which class this magnitude method is for**

**Remember you can write a magnitude method for many classes (eg a ComplexNumber class)**

**When the compiler reaches this file it has no other way of knowing which one you are supplying.**

**Therefore you tell it by prefixing the method name with the classname::**

**For a complete and detailed example look now at the ThreeVector files supplied by the course tutor**

**Ignore the things called constructors for now**
**These are covered in the next module**

**From this point on you should write all of your classes in this way**

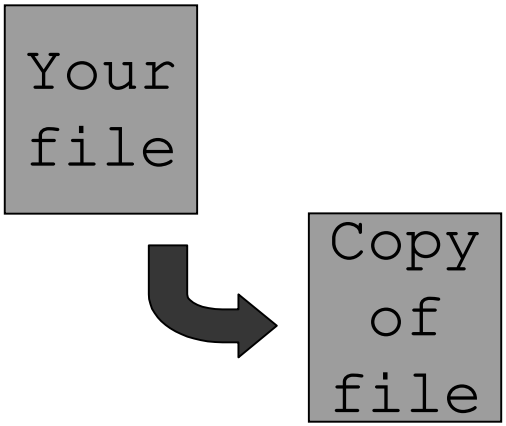
**You should re-write all of the Comton analysis classes you have written to be like this now.**

# Technical 4:

# The C pre-processor

**The C pre-processor is invoked each time you compile a file.**
**It takes actions on the file BEFORE actually trying to compile it**

**Think of it as starting by taking the file you have passed it, and effectively copying it to a temporary version that it is going to edit for you.**




**Wherever it encounters a # directive it takes some action and edits the copy**

**Only when it has finished with all of the # directives will it actually then compile the copy of the file**

**#define**

## BEFORE PRE-PROCESSOR ACTION

```
#define XXXX YYYY


~~~~~~~
~~~~~~~

~~~~~XXXX~~~~

~~~~~~~

int a = XXXX;

~~~~~~~
```

**#define**

## AFTER PRE-PROCESSOR ACTION

```
~~~~~~~
~~~~~~~

~~~~~YYYY~~~~

~~~~~~~

int a = YYYY;

~~~~~~~
```

**NOTE: no semi-colons are needed. Thw # directives are NOT language statements**

**#include**

## BEFORE PRE-PROCESSOR ACTION

```
#include "Wombat.h"

#include "Wombat.cpp"

~~~~~~~~
...your code....
~~~~~~~~
```

**filename: Wombat.h**

```
//--------------
class Wombat {
  float tailLenght ;
  float cuteness ;
  ....
}  ;
```

**filename: Wombat.cpp**

```
//..............
void Wombat::cuddle( )
{
  // code to cuddle wombat
  return ;
}
```

#include

## AFTER PRE-PROCESSOR ACTION

```
class Wombat {
  float tailLength;
  float cuteness;
  ....
}  ;

void Wombat::cuddle( )
{
  // code to cuddle wombat
  return ;
}

~~~~~~~
...your code....
~~~~~~~
```

filename: Wombat.h

```
//--------------
class Wombat {
  float tailLength;
  float cuteness;
  ....
}  ;
```

filename: Wombat.cpp

```
//..............
void Wombat::cuddle( )
{
  // code to cuddle wombat
  return ;
}
```

## #ifndef /#endif

### BEFORE PRE-PROCESSOR ACTION

```
#include "ThreeVector.h"

#include "ThreeVector.h"
```

filename: threevec.h

```
#ifndef __threevec__

#define __threevec__ 1
//--------------
class ThreeVector {
   ...
   ...
} ;

#endif
```

## #ifndef /#endif

## AFTER PRE-PROCESSOR PERFORMS #include(s)

```
#ifndef __threevec__

#define __threevec__ 1
//--------------
class ThreeVector {
   ...
   ...
} ;
#endif


#ifndef __threevec__

#define __threevec__ 1
//--------------
class ThreeVector {
   ...
   ...
} ;

#endif
```

filename: threevec.h

```
#ifndef __threevec__

#define __threevec__ 1
//--------------
class ThreeVector {
   ...
   ...
} ;

#endif
```

## #ifndef /#endif

## AFTER PRE-PROCESSOR HITS FIRST #ifndef/#endif

```
#define __threevec__ 1
//--------------
class ThreeVector {
   ...
   ...
} ;


#ifndef __threevec__

#define __threevec__ 1
//--------------
class ThreeVector {
   ...
   ...
} ;

#endif
```

## #ifndef /#endif

## AFTER PRE-PROCESSOR HITS SECOND #ifndef/#endif

```
#define __threevec__ 1
//--------------
class ThreeVector {
   ...
   ...
} ;
```

## #ifndef /#endif

**In other words**

```
#ifndef __threevec__

#define __threevec__ 1

//---------------
class ThreeVector {
   ...
   ...
} ;

#endif
```

**is a scheme to make sure that this file NEVER get included twice.**

**[It is not important to follow it exactly now, merely to recognise the formation, and know why you need it]**

# Technical 5:

# References

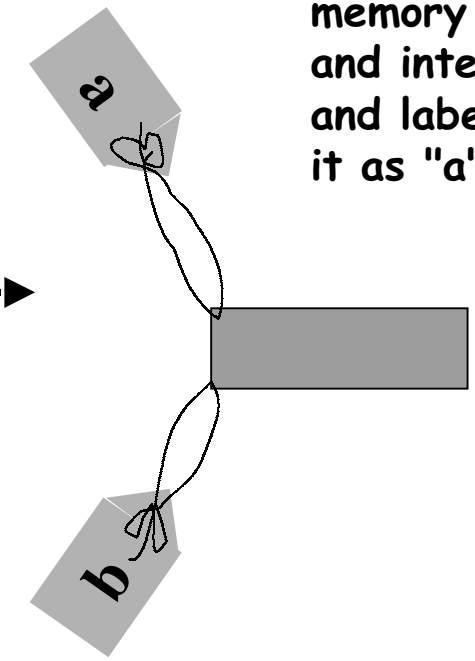
**C++ introduces a new way of referring to an instance of a variable or object**

**This is via a**

# reference

In the simplest form references get introduced like this:

```
// create an integer

int a ;



// Now create another name for the
// same integer

int&  b = a ;
```




grab memory for and integer and labels it as "a"

"b" is just another label for the same memory

`b` is a "reference".

`int&` means "make a label to reference an integer"

`a` and `b` are synonyms for the same thing

**WARNING !**

**- the use of & here has**

**a completely different meaning to**

**the use of & in code to "take the address of"**
**as appears with pointers**

**This is designed to confuse you**

- **`a and b` can both be used just like any ordinary integer**

```
int a;
int & b = a;


a = 25 ;

...


b = 41 ;   //same thing as putting a = 41
```

**`a` and `b` are synonyms for the same variable**

```
int a;
int & b = a;


a = 5 ;
b = 4 ;
```

What is the value of a ?

```
int a;
int & b = a;
int c ;

b = 10 ;
c = 5 ;
a = c ;
```

What is the value of b ?

Exactly the same rules apply for objects

```
// create a Thing

Thing t ;



// Now create another name for the
// same Thing

Thing& r = t ;
```

`r` and `t` are synonyms for the same object. You can write:

```
    t.methodName(..)
or
    r.methodName(..)
```

**Before you wonder what the point of this is, please note**

**(i) you almost never use it in this simple way to make two names for the same thing in one bit of code**

**in fact it would be a particularly stupid thing to do !**

**we only show this to see how it works in the simplest situation**

**(ii) it is most commonly used for argument passing as we will see on the next slides...**

**(iii) it is also used extensively in the context of operator overloading, which is covered in a later section, and with copy constructors.**

Consider a hypothetical Matrix class, and a function called normalise (details are irrelevant here) which, unsuprisingly, normalises an matrix

```
Matrix myMat ;

// normalise the matrix...

normalise( myMat ) ;
```

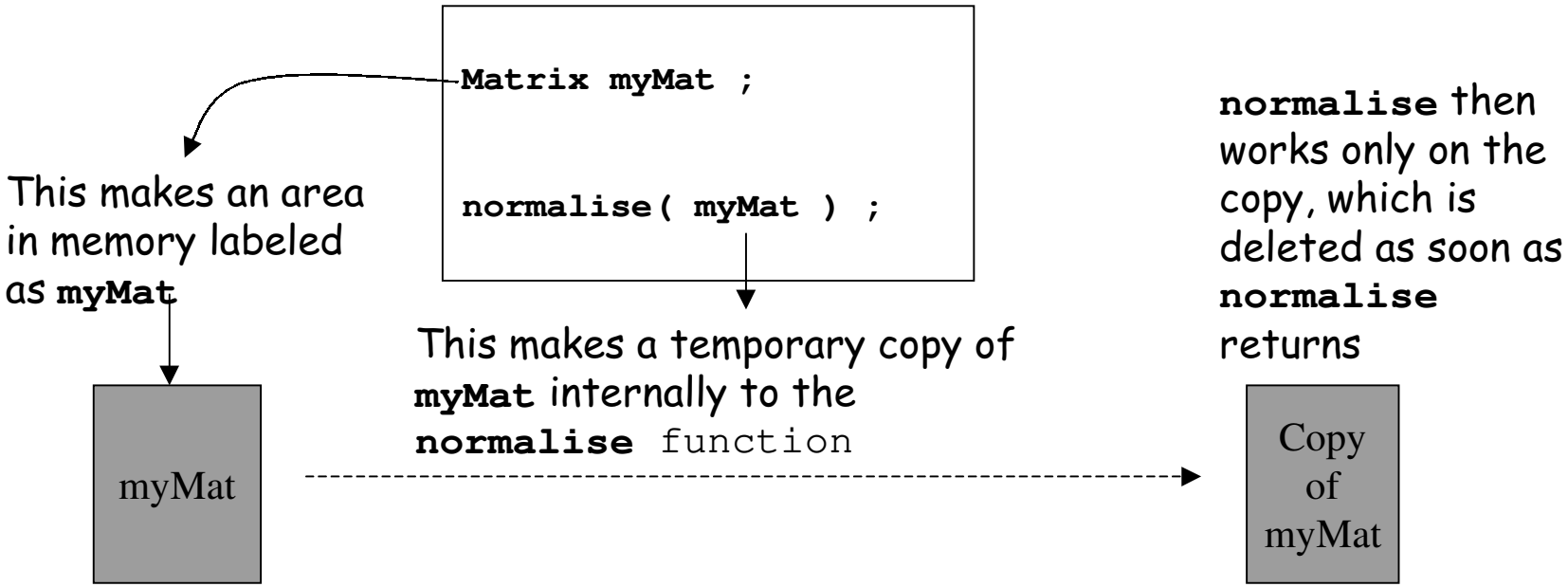
You might think the normalise funtion could look like this:

```
void normalise( Matrix m )
{

  // some code to normalise

}
```

However this WONT do what you want it to do.
!

**To understand why we have to go over a detail of how C++ works**

The problem arises because when you pass arguments to functions, you actually pass a copy of the thing in question



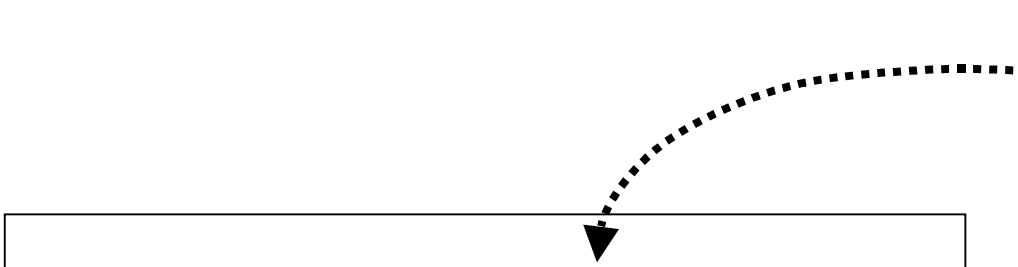

As a result, if you want to CHANGE the thing you are passing, you cannot send it in as an argument like this

```
Matrix myMat ;

// normalise the matrix...

normalise( myMat ) ;
```

So in this example, myMat is not changed by the call to normalise

```
void normalise( Matrix m )
{

  // some code to normalise

}
```

```
Matrix myMat ;

// set m to something...

normalise( myMat ) ;
```



```
void normalise( Matrix& m )
{

  // some code to normalise

}
```

If instead you modify the normalise function like this

then only a reference to `myMat` is passed

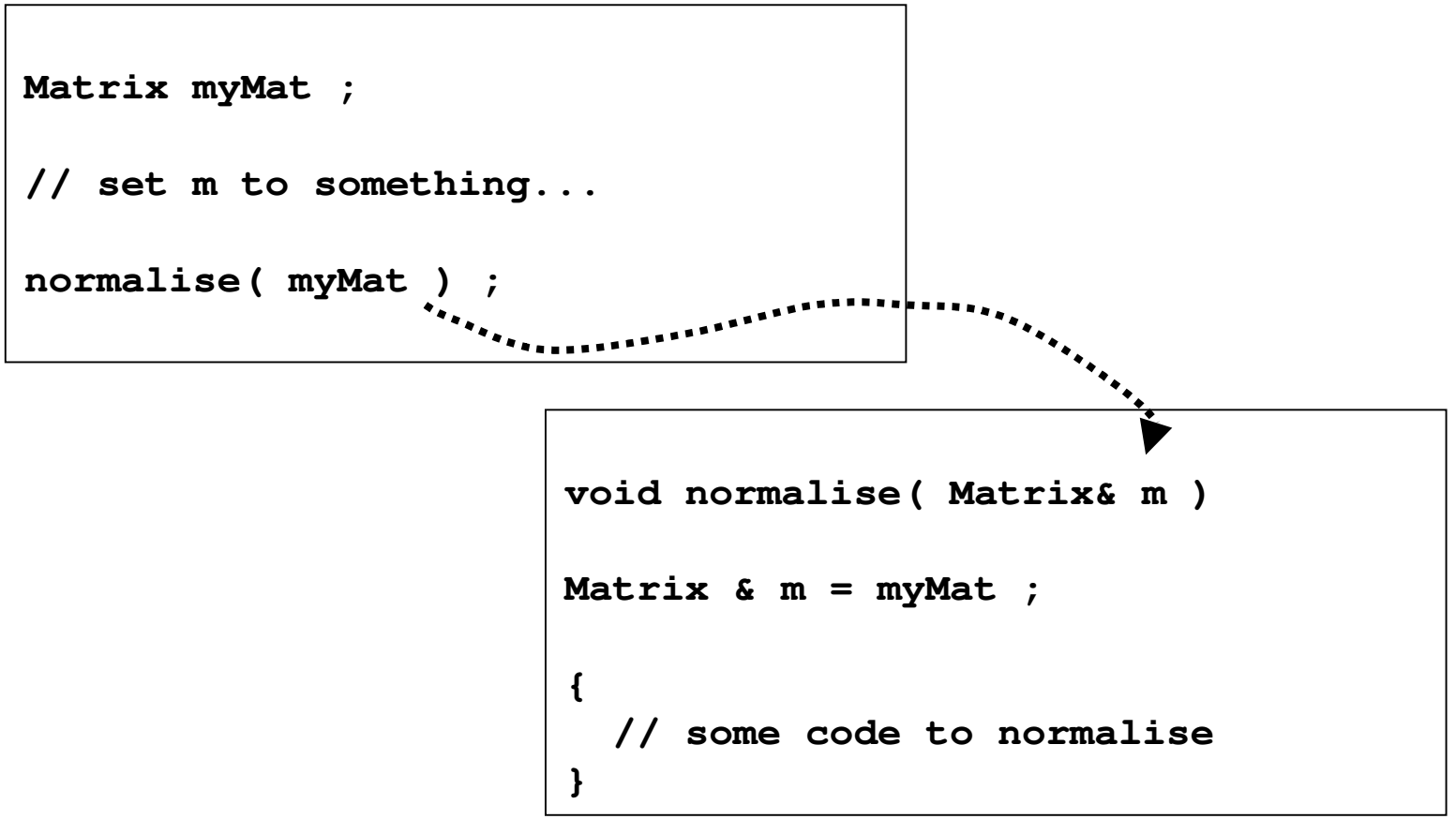
`m` becomes a synonym for `myMat` itself

⇒ `myMat` can be modified

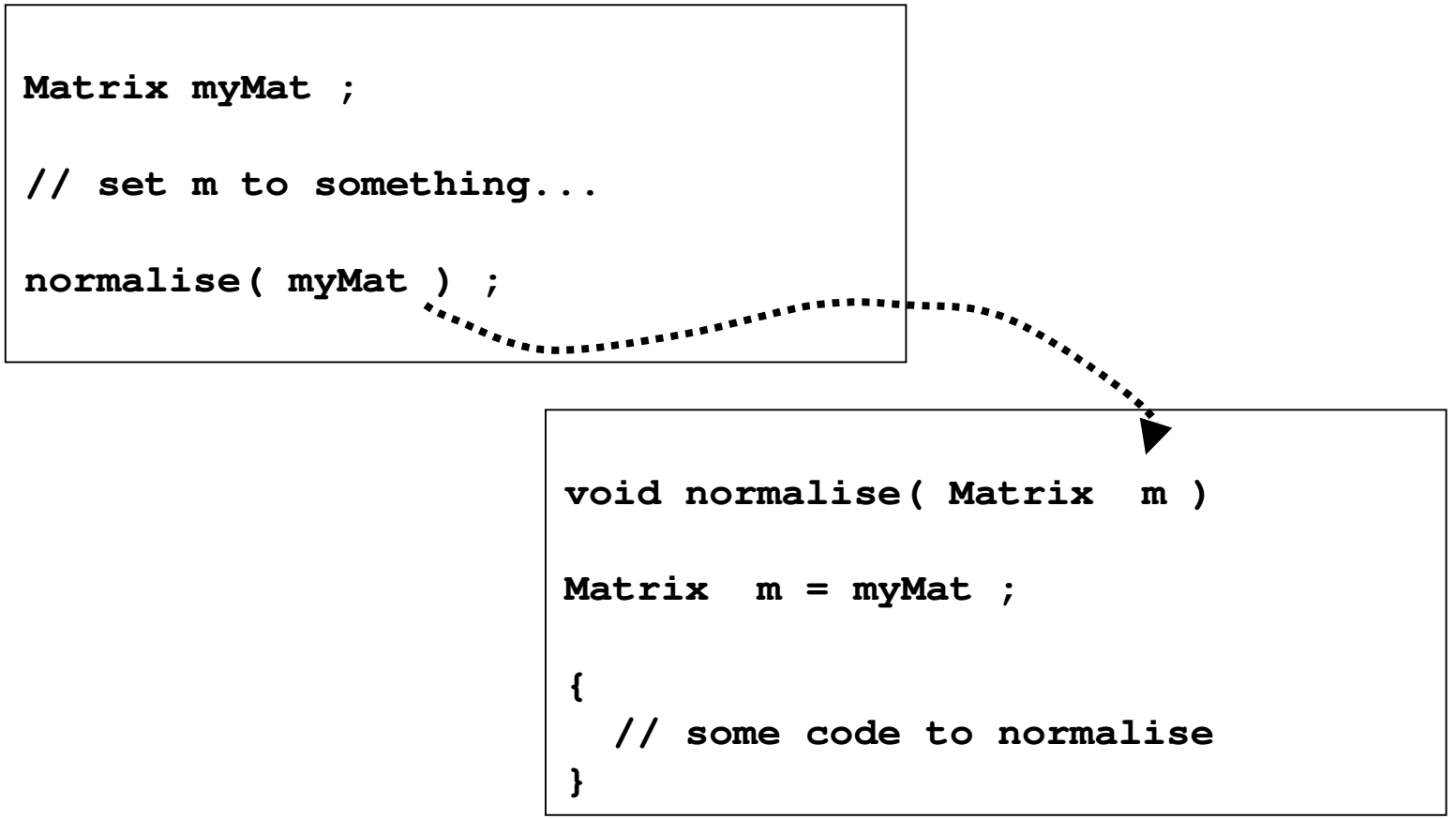
⇒ no copying of large matrix

If this seems confusing, it may be helpful to keep the following simple picture in mind:

When you pass any argument to a function, imagine the following "hidden line" being inserted in the function itself



If yuo did not have the reference in it would be like this:



**A key point is that:**

- **you only needed to modify the code for the method**
- **you did not need to modify the code which used it**

**In fact as the user of a method you dont care whether internally the method uses a copy of the object or a reference !**

**You will see exactly this technique used in copy constructors and operator overloading.**

**YOU should also consider using it in your own functions in order to avoid making temporary copies of large objects.**

# Initialisation of references

**References must always be set to refer to a legitimate item.**
**You cannot declare an "empty" reference.**

**In other words you cannot do this:**

```
int a = 10;

int& b ;

b = a ;
```

**You are obliged to do this**

```
int a = 10;

int& b = a ;
```

# Technical 6:

# const

There are often occasions when you want to make a symbol so that it can be set once but never changed thereafter

for example PI :

```
// Example of scope something you want to
// keep constant

int PI = 3.14159 ;

...
...

float degrees ;
float radians ;

degrees = 360 * radians / 2 / PI ;
```

However as written there is nothing stopping PI being overwritten with another value.

To avoid this you can declare a symbol to be

# const

```
// Example of scope something you want to
// keep constant

const int PI = 3.14159 ;

...
...

float degrees ;
float radians ;

degrees = 360 * radians / 2 / PI ;
```

## The following WILL NOT work
## (the compiler disallows it)

```
// Example of scope something you want to
// keep constant

const int PI = 3.14159 ;

...
...

float degrees ;
float radians ;

degrees = 360 * radians / 2 / PI ;

...

PI = 25 ;
```

## Similarly you can declare a REFERENCE to be `const`

```
// Example of a const reference


int a = 5 ;

const int& b = a ;

...
...

a = 7 ;      // this is allowed

b = 8 ;      // this is NOT allowed
```

As usual, the relevance of const references is not the simple example shown, but in argument passing.

VERY often you see the following:

```
// Example of a method which receives a const  reference

float ThreeVector::dotProduct( const ThreeVector& otherVector )
{

  float prodsq ;

  prodsq = x*otherVector.x + y*otherVector.y + z*otherVector.z ;

  return sqrt( prodsq ) ;

}
```

The purpose of this is:
(i) the argument is passed by reference to avoid a copy
(ii) the reference is made const to promise that the method will not change the argument in any way

- const references are ALWAYS used in copy constructors

```
// Copy constructor

ThreeVector::ThreeVector( const ThreeVector& source )
{

  x = source.x ;
  y = source.y ;
  z = source.z ;

}
```

(in my opinion this is one of the most obtuse looking bits of C++ code)

- We cover it here so that you recognise it when you see it

- I DO NOT expect you to do this in any course exercises if you do not wish to.

# Technical 7:

# scope

We have already implicitly used the concept of `scope`, even though the word hasn't been explicitly mentioned.

Scope refers to how long something exists for after it is declared.

Just because you might type

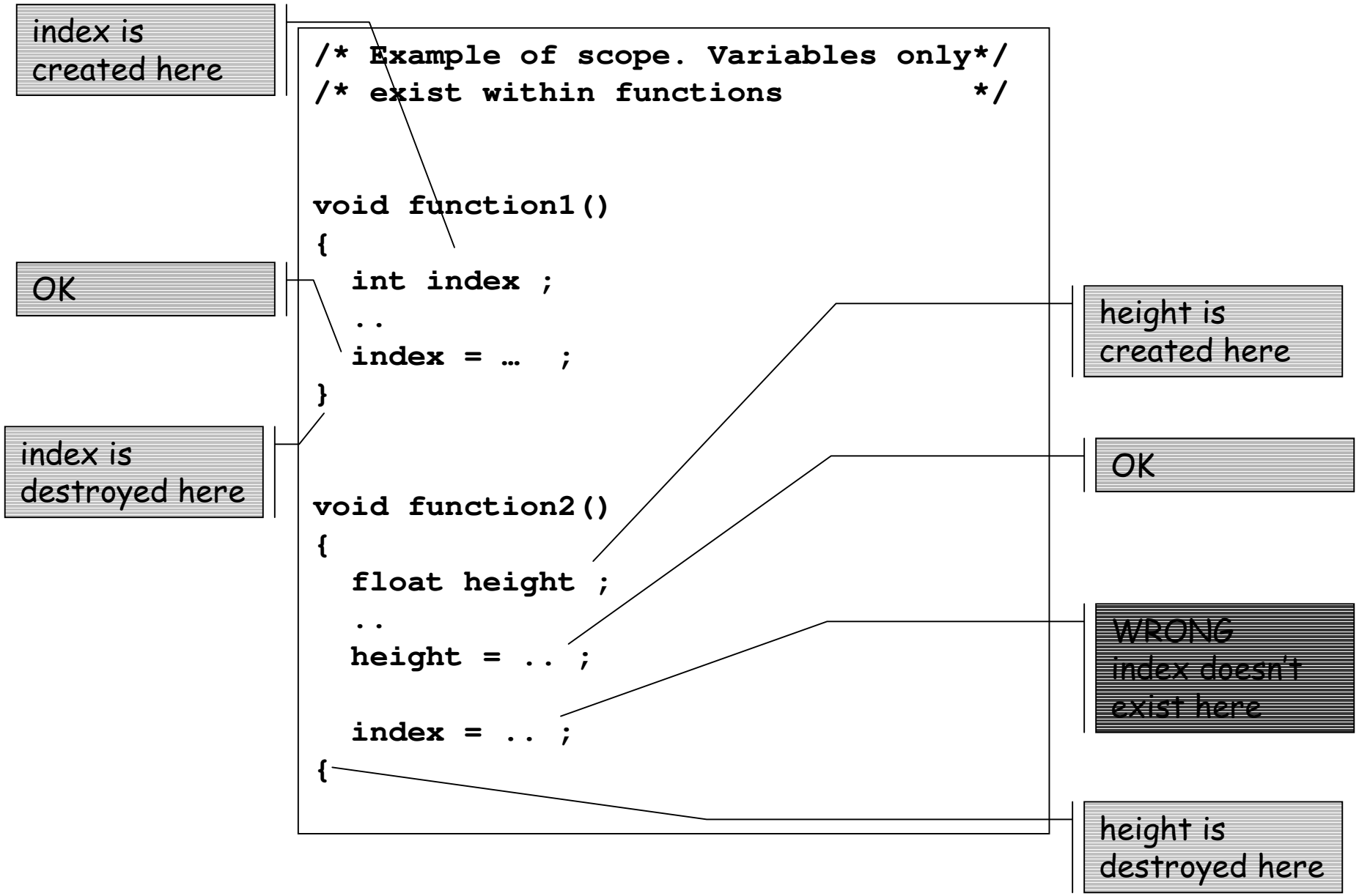
```
int index ;
```

somewhere in one file, doesn't mean that the name "index" is known in every other file included in the whole program

- far from it in fact !

A declaration is valid ONLY within the {…} in which it was declared.

```
/* Example of scope. The
variable can only exists in
between the brackets */

{
  int index ;

  ..
  ..

}
```


index is created here
/* Example of scope. Variables only*/
/* exist within functions */
void function1()
{
int index ;
..
index = … ;
}
OK
index is destroyed here
void function2()
{
float height ;
..
height = .. ;
index = .. ;
{
height is created here
OK
WRONG index doesn't exist here
height is destroyed here

If you want to have a variable known known to ALL functions in a file then one way is to declare it OUTSIDE of them all

This now works

In my opinion this is BAD

We wont declare variable outside functions in this course

```
/* Example of scope. */

/* Declare index globally to this file*/
int index ;


void function1()
{
  ..
  index = …  ;
}


void function2()
{
  float height ;
  ..
  height = .. ;

  index = .. ;
{
```

An important consequence of scope is that you CANNOT do this

it WONT work

`numberOfTimesCalled`

is created afresh each time the function is entered, and destroyed when it exits

So it only ever equals 1 !.

```
/* Example of scope. Variables only*/
/* exist within functions          */


void function1()
{
  int numberOfTimesCalled

  /* increment variable each time this */
  /* function is called                */

  numberOfTimesCalled++ ;



  .. rest of code ..



}
```

You can do what you want to do in 2 ways

(i) declare it in the file outside the function as shown previously

(ii) declare it as a

**static**

This will now work and is safe

```
/* Example of scope. Variables only*/
/* exist within functions          */


void function1()
{
  static int numberOfTimesCalled

  /* increment variable each time this */
  /* function is called                */

  numberOfTimesCalled++ ;



  .. rest of code ..



}
```

**Note that this scope rule {..} applies generally, not just within function {..}**

**The `for{..}` or `if{..}` constructs are the same.**

**In fact it is true for any `{..}` anywhere in the code**

This wont work

```
/* Scope within a file */


if( some test )
{
  int index ;
  index = …  ;
}

std::cout << index ;
```

This will work and is fine.

```
/* Scope within a file */

int index ;

if( some test )
{
  index = …  ;
}

std::cout << index ;
```

**To finish on scope**

**1. Read the course text book on scope**

**2. Declaring variables appropriately within functions is FINE, i.e outside of a for{..} loop if you want to subsequently use the variable.**

**3. Declaring static variables within functions is FINE**

**4. But declaring variables outside functions is BAD as it makes the variables visible to code in other files. Dont do this in this course.**

**5. If we want to pass data between functions we will use arguments.**

# Technical 7:

# namespace

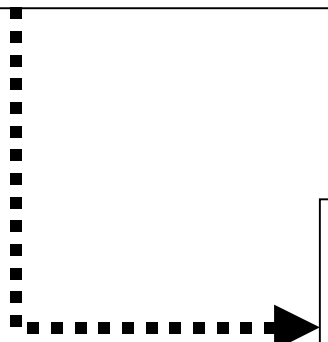
**namespaces are easy**

**They are simply a way of packaging classes (and other names) so that they have a further distinguishing name**

**This is most often done for reasons of organisation, or to avoid ambiguity.**

**Best shown by example:**

```
// a windowing class

class Master {
  int positionToDraw ;
  int size ;
  ...
};
```



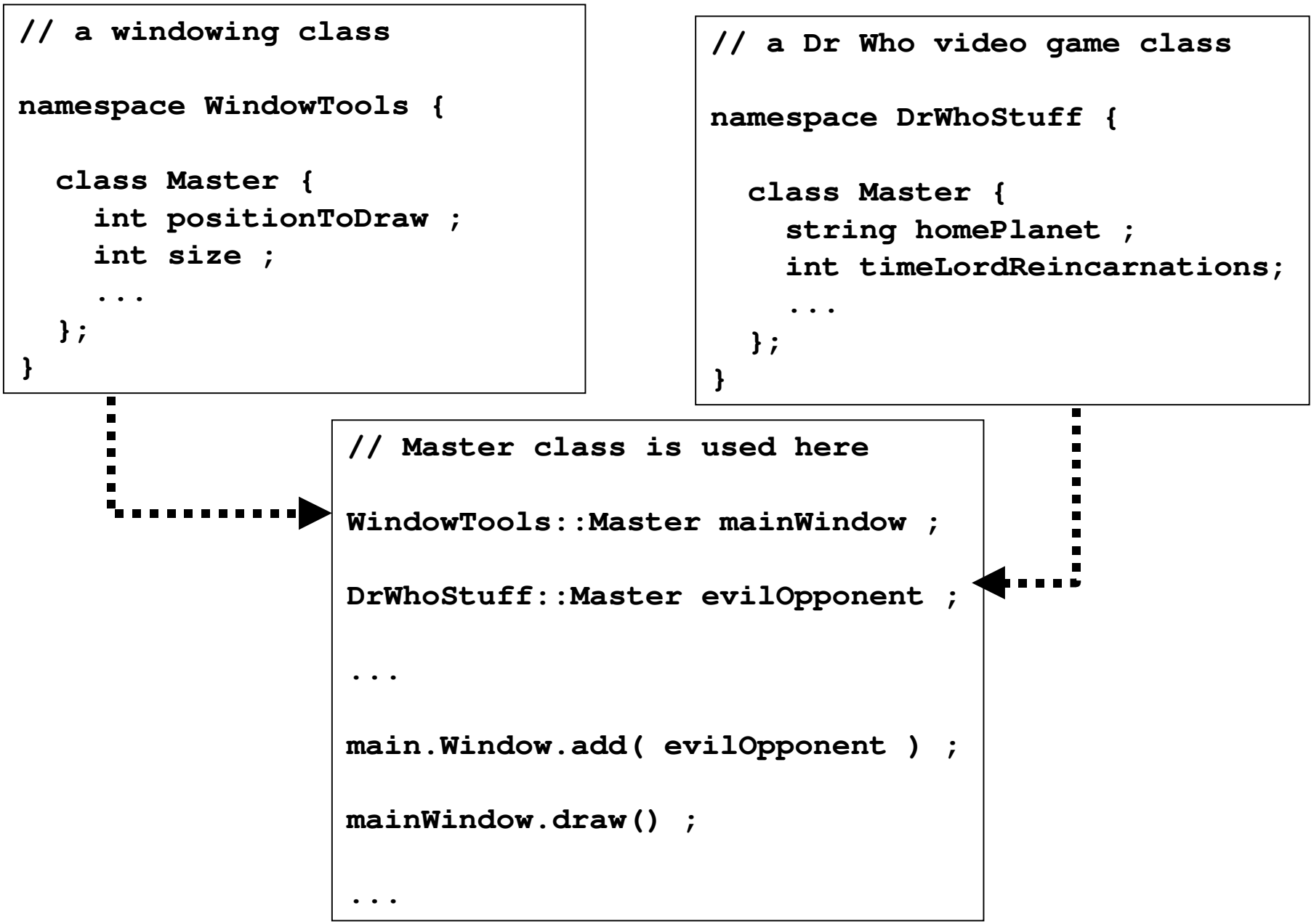
```
// Master class is used here

Master mainWindow ;



...



mainWindow.draw() ;

...
```

```
// a windowing class

class Master {
  int positionToDraw ;
  int size ;
  ...
};
```

```
// a Dr Who video game class

class Master {
  string homePlanet ;
  int timeLordReincarnations;
  ...
};
```

```
// Master class is used here

Master mainWindow ;

Master evilOpponent ;

...

main.Window.add( evilOpponent ) ;

mainWindow.draw() ;

...
```

Compiler has no way to work out what you want


// a windowing class
namespace WindowTools {
class Master {
int positionToDraw ;
int size ;
...
};
}
// a Dr Who video game class
namespace DrWhoStuff {
class Master {
string homePlanet ;
int timeLordReincarnations;
...
};
}
// Master class is used here
WindowTools::Master mainWindow ;
DrWhoStuff::Master evilOpponent ;
...
main.Window.add( evilOpponent ) ;
mainWindow.draw() ;
...

**We have put each of our classes inside a namespace**

**⇒ We can put as many classes as we want inthe WindowTools namespace**

**⇒ We can put as many classes as we want inside the DrWhoStuff namesapce**

**You have already seen this without knowing what it is:**

**⇒ cout, cin, endl are all inside the std:: namespace**

```
std::cout << " A message " << std::endl ;
```

# Technical 8:

# casting
# conversion