Technical Module : Pointers

In this module we will cover

- **Pointers to primitives and objects**
- **Argument passing using pointers**
- **Memory allocation using new**
- **Memory freeing using delete**
- **An important use of a destructor**

You will probably be confused by the end of this. Particularly on the relation between pointers and references

Aims of this module

In this module we teach you about the use of pointers.

It will at first seem that pointers are the same as references, just more obtuse to use. In many cases this is true, but there are situations where you need to use pointers.

In this module we first explain pointers and then show them in use for dynamic memory allocation.

1 Pointers

Pointers were invented as part of C long before references were ever invented in C++

Pointers, as the name suggests, allow you to indirectly refer to a primitive or object.

References are in many cases a simpler and more obvious way to indirectly refer to a primitive or object. Therefore in many situations you no longer need pointers where they would have been used in C.

However pointers pervade C++ programs, so we must learn about them.

[You might like to note that pointers have been abolished in Java. Only "references" exist, but confusingly they are similar to but not the same as references in C++.]

In its simplest form a pointer is declared and used like this

ptrToInt is NOT an integer

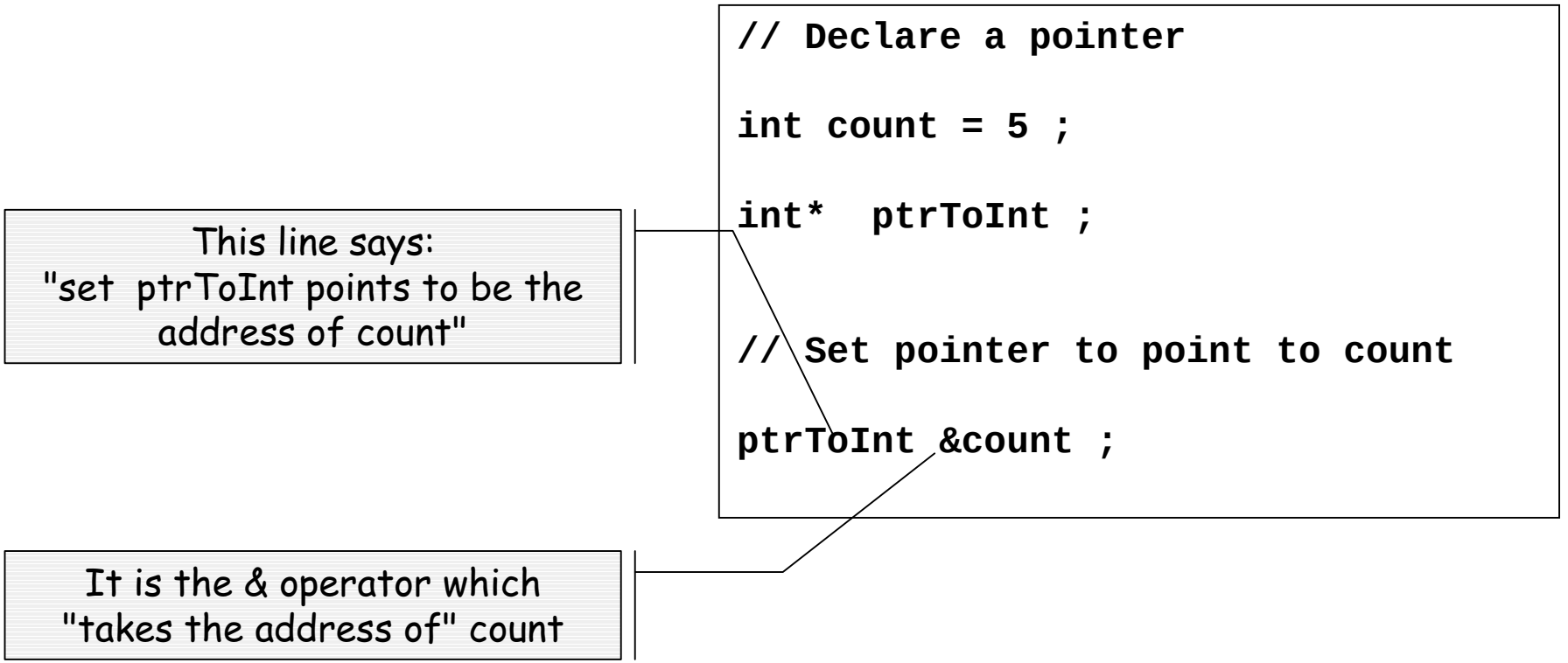
It is a thing which can POINT to an integer

```
// Declare a pointer

int*  ptrToInt ;



// Use a pointer

*ptrToInt = 3 ;
```

ptrToInt is "dereferenced" using the * operator.

This line says:
"set the thing which ptrToInt points to to be 3"

Now we get confusing: here is how you set a pointer to point to an existing variable:

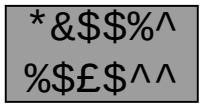


Annoyingly and very confusingly, this use of & has absolutely nothing to do with the use of & in the context of references !
You WILL BE confused !

Pictorial examples:

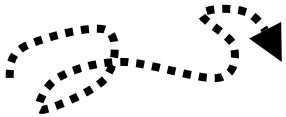
Pointer to an integer:

```
int a ;
```



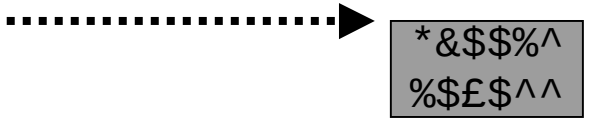
a is an integer - space has been allocated and probably contains garbage

```
int* pa ;
```



pa is a pointer to an integer, but doesn't point to anything sensible yet

```
pa = &a ;
```



pa now points to a, remember & means "take address of".

```
*pa = 10 ;
```

This places value 10 in the thing pointed to

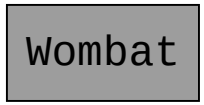
It works the other way around, i.e.

```
int b ;
b = *pa ;
```

2 Pointers to objects

In the same way (with one minor additional complication) one may use pointers to objects like this:

```
// Make a Thing object

Thing wombat ;

// Make a pointer to a Thing

Thing*  ptrToAThing ;


// Now set the pointer to point at wombat

ptrToAthing = &wombat ;
```



The minor complication is that when you have a pointer to an object you normally use the -> operator to invoke its methods

Instead of this (which works fine)

```
// make a pointer to a thing
Thing*  ptrToAThing ;

//.... set it to point at some sensible Thing object...

// Now use the printContents method in an "obvious" fashion

(*ptrToAthing).printContents()
```

you normally do this:

```
// make a pointer to a thing
Thing*  ptrToAThing ;

//.... set it to point at some sensible Thing object...

// Now use the printContents method using -> operator

ptrToAthing->printContents()
```

At this points students normally ask why ? what is the point of pointers ? what do they do that references dont do ?

As we said earleir, in many cases the answer is nothing because pointers were invented in C before C++ references.

If we tried to cover every way of using pointers in abstact examples now it would undoubtedly confuse you more than the good it would do.

(you would be unique if this were not the case !)

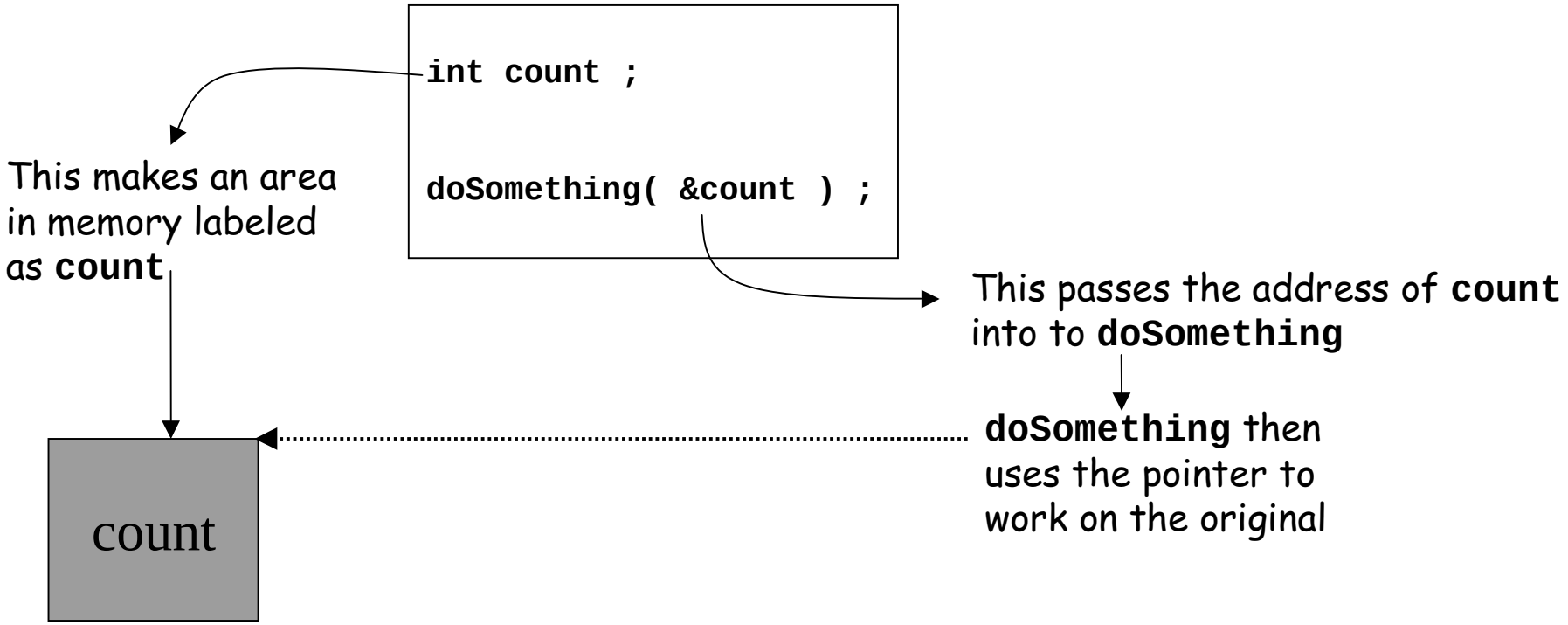
We therefore cover only two very common uses now:

argument passing
dynamic memory allocation

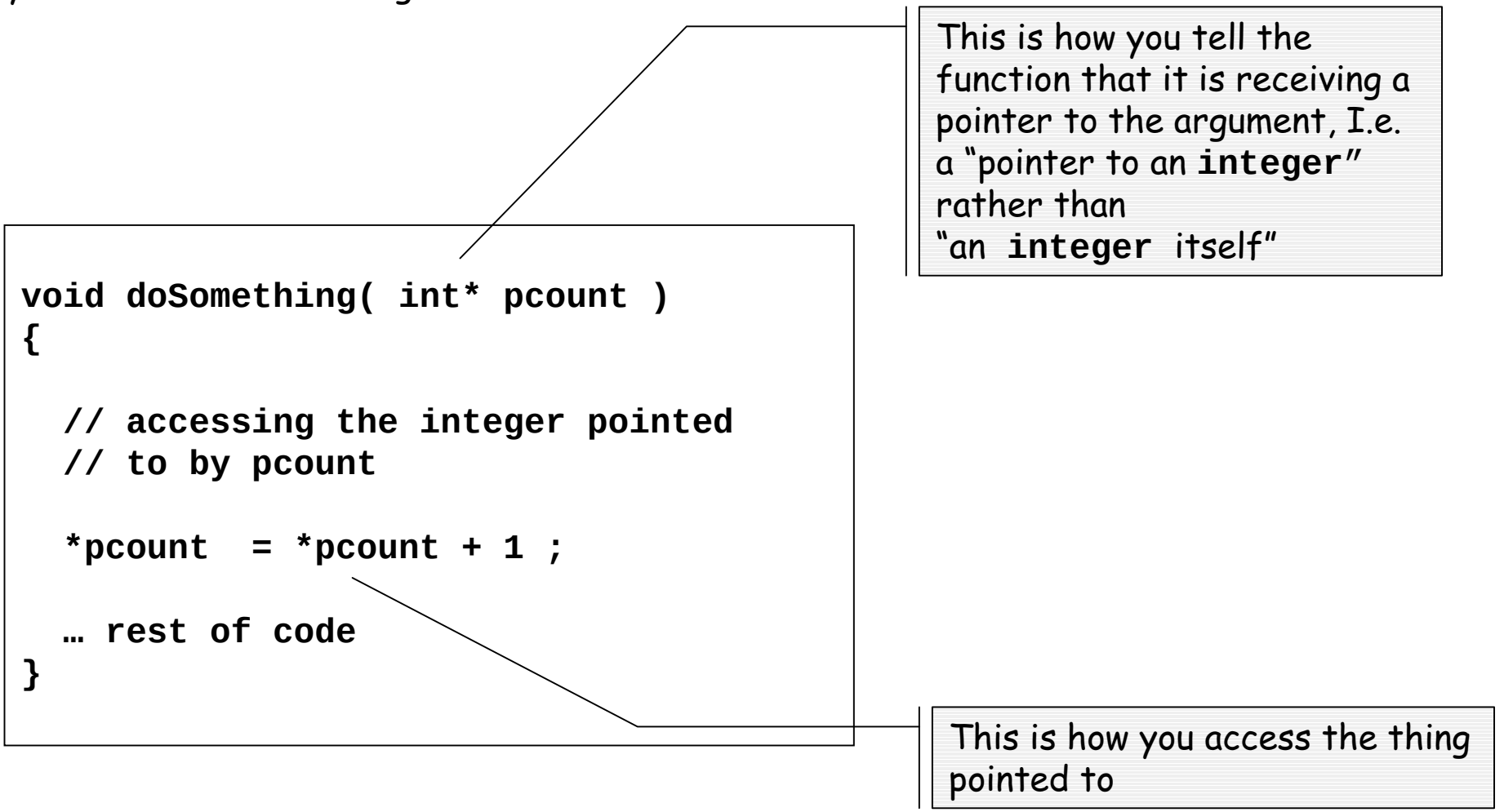
and defer any other use of pointers to well defined situations where they might arise, and where hopefuly the purpose of use wouldbe clear.

3 Using pointers to pass primitive arguments

Before C++ references were invented, pointers HAD to be used in C to pass arguments which you wanted to change. You can still do this in C++ like this:

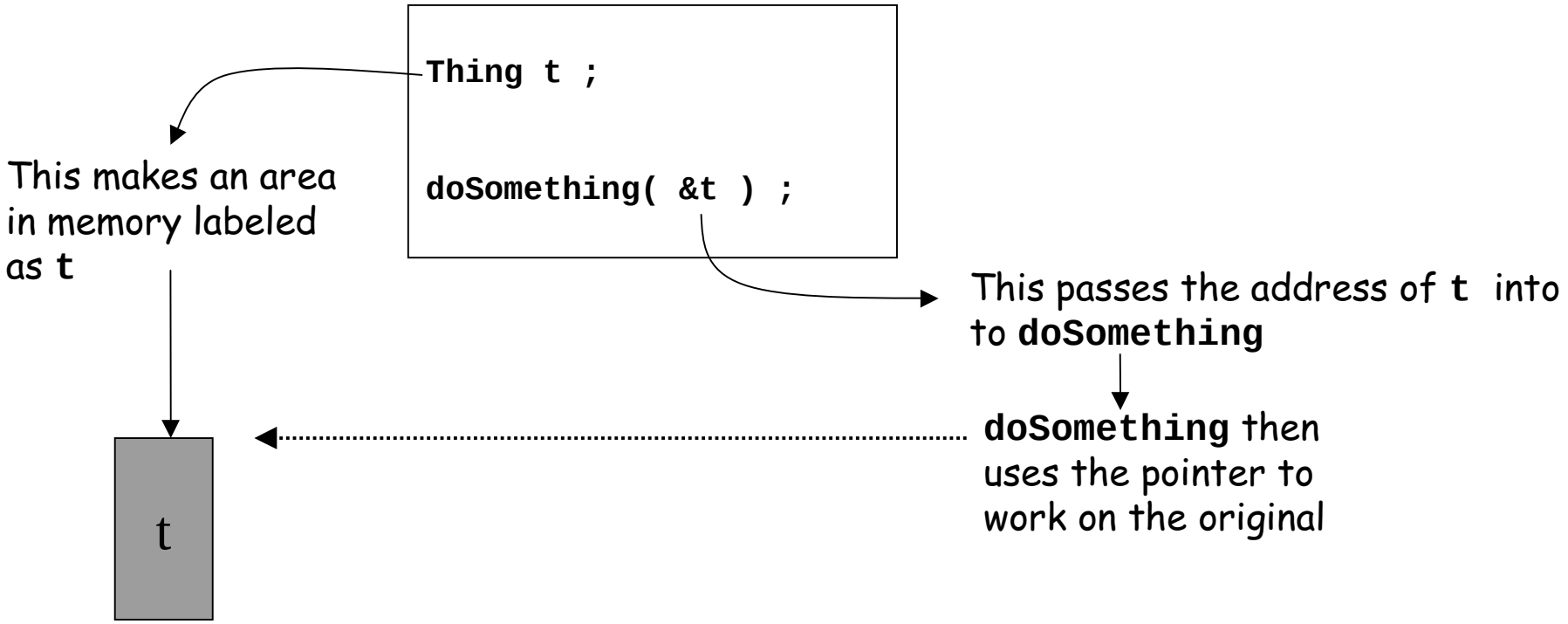


Inside the function `doSomething`
you have to make the arguments match this



4 Using pointers to pass objects

This is identical to passing primitives (it can be used instead of passing a reference, but there is generally no point in doing so)



If you are passing a user defined data type you probably want to use its methods. You normally do that like this:

```
// Declare user defined data type

Thing t ;


// Call a function to change it:

doSomething( & t ) ;
```

```
void doSomething( Thing* ptr )
{

  // accessing a method
  // through the pointer

  ptr->initialise( 10.) ;

  ... rest of code
}
```

This is how you access methods of the thing pointed to.

5 Memory allocation using "new"

So far the only way we have ever created instances of data types is by declaring them in a piece of code

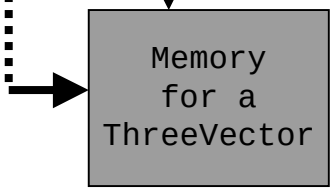
An alternative in C++ to explicitly create things in a free area of memory using

new

vec is declared as a pointer to a **ThreeVector** (but at this stage it doesnt point to anything - it is un-initialised).

At this point an area of memory is carved off and protected.
vec is then set to point to it

```
/* Example of use of new */

ThreeVector* vec ;

vec = new ThreeVector  ;
```



Think of **new** as a function which returns a pointer to a new instance of the data type specified after it

As an example of where you might use pointers with new, here is a a puzzle for the whiteboard

Which of the following works/doesn't work ?
Explain why ?

```
// Code A
main()
{
  ThreeVector vec ;

  vec = gimmeAVector( ) ;

  dump( vec ) ;
}
```

```
// Code A
ThreeVector gimmeAVector()
{
  ThreeVector v ;

  return v ;
}
```

```
// Code B
main()
{
  ThreeVector* pvec ;

  pvec = gimmeAVector( ) ;

  dump2( pvec ) ;
}
```

```
// Code B
ThreeVector* gimmeAVector()
{
  ThreeVector* pv ;

  return pv ;
}
```

```
// Code C
main()
{
  ThreeVector* pvec ;

  pvec = gimmeAVector( ) ;

  dump( pvec ) ;
}
```

```
// Code C
ThreeVector* gimmeAVector()
{
  ThreeVector v ;

  return &v ;
}
```

```
// Code D
main()
{
  ThreeVector* pvec ;

  pvec = gimmeAVector( ) ;

  dump( pvec ) ;
}
```

```
// Code D
ThreeVector* gimmeAVector()
{
  ThreeVector* pv ;

  pv = new ThreeVector  ;

  return pv ;
}
```

Very typically pointers are used like this, i.e.

-when you want to create something long lived inside a function

- when you want to return this thing to the caller without it going out of scope when you terminate

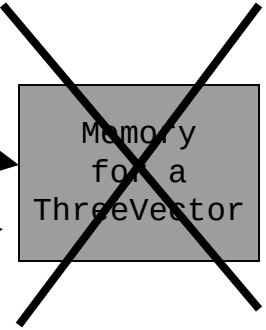
```
// Code D
ThreeVector* gimmeAVector()
{
  ThreeVector* pv ;

  pv = new ThreeVector  ;

  return pv ;
}
```

6 Memory de-allocation using "delete"

Very important !

When you have finished with memory you must `delete` it in order to give the memory back to the system

```
// Example of use of new and delete
// in a function

void myFunction {

  ThreeVector* pvec ;

  pvec = new ThreeVector  ;

  ....
  ....

  delete pvec  ;

  return;
}
```

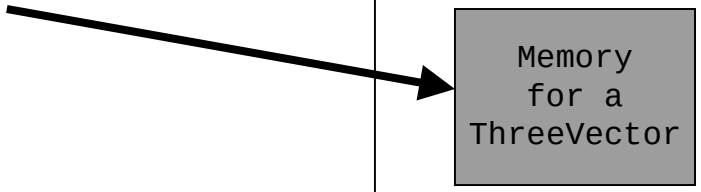


If you fail to do this you will cause a

"memory leak"

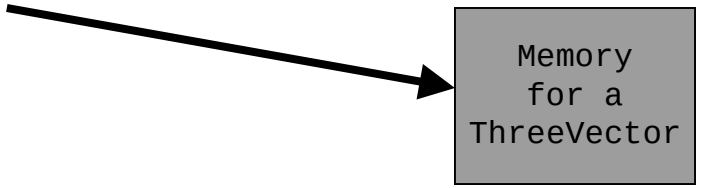
The computer memory gets filled up with orphaned memory, and eventually falls over.

```
// Example of use of new and delete
// in a function

void myFunction {

  ThreeVector* pvec ;

  pvec = new ThreeVector  ;


  ....
  ....

  // forgot to delete( pvec ) ;

  return;
}
```



The function returns, and the pointer goes out of scope.

But the memory remains allocated with nothing pointing to it !



7 An important use of the destructor

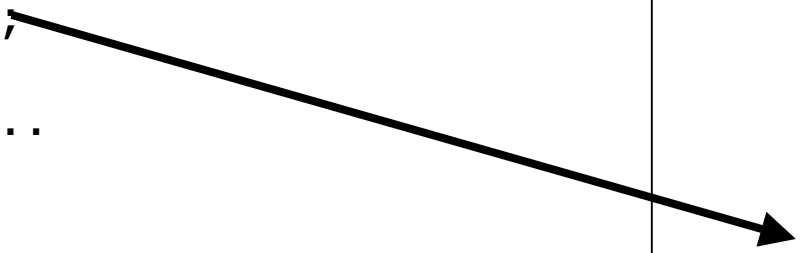
The most common situation where a memory leak occurs arises when you allocate some memory using new within an object, then forget to delete it before the the object gets destroyed

Here is an example of the sort of problem which can arise.

```
// The .h file for a badly written class

class BadClass
{
  private:
    Thing* ptr ;

  public:
    BadClass( ) ;

}
```

```
// The .cc file with constructor

BadClass::BadClass( )
{
  ptr = new Thing ;
}
```

```
// some piece of code which makes a
// BadClass instance

void mycode() {

  BadClass b ;

  ... use it...


  return ;
}
```



When you make a `BadClass`, then

(i) memory for a BadClass is allocated

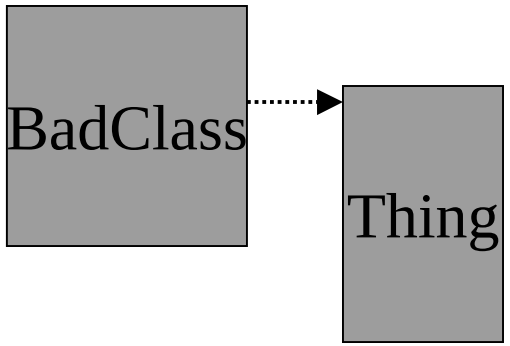
(ii) the BadClass constructor is called which uses new to allocate memory for a `Thing`, and sets the pointer to it

```
// some piece of code which makes a
// BadClass instance

void mycode() {

  BadClass b ;

  ... use it...


  return ;
}
```

When you make a `BadClass`, then

(i) memory for a BadClass is allocated

(ii) the BadClass constructor is called which uses new to allocate memory for a `Thing`, and sets the pointer to it

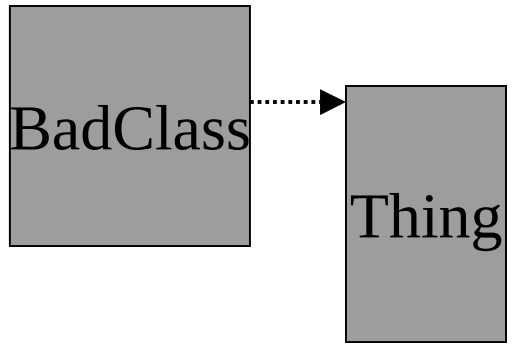


```
// some piece of code which makes a
// BadClass instance

void mycode() {

  BadClass b  ;

  ... use it...


  return ;
}
```

When a BadClass object goes out of scope, then it gets destroyed, but this only deletes the BadClass itself

The Thing is left hanging around with nothing pointing to it

This is a MEMORY LEAK

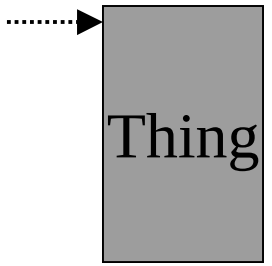


```
// some piece of code which makes a
// BadClass instance

void mycode() {

  BadClass b  ;

  ... use it...


  return ;
}
```

When a BadClass object goes out of scope, then it gets destroyed, but this only deletes the BadClass itself

The Thing is left hanging around with nothing pointing to it

This is a MEMORY LEAK



You should instead supply a destructor

```
~BetterClass( )
```

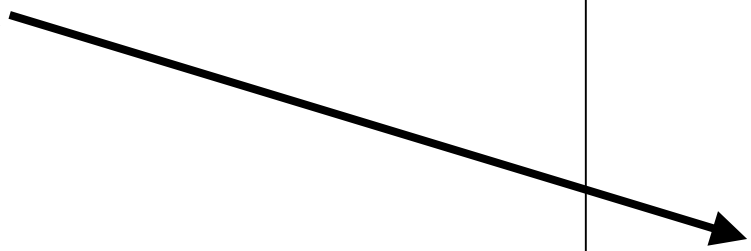
This destructor gets called immediately before a BetterClass object gets destroyed

```
// The .h file for
// A beter written class

class BetterClass
{
  private:
    Thing* ptr ;

  public:
    BetterClass( ) ;
    ~BetterClass( ) ;

}
```

It deletes the Thing explicitly before the BetterClass object goes out of existence

```
// In the .cc file

BetterClass::~BetterClass( )
{
  delete ptr ;
}
```

```
// some piece of code which makes a
// BetterClass instance

void mycode() {

  BetterClass b;

  ... use it...


  return ;
}
```



When you make a `BetterClass`, then

(i) memory for a `BetterClass` is allocated

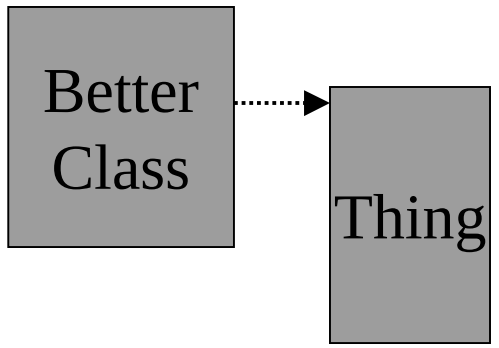
(ii) the `BetterClass` constructor is called which uses `new` to allocate memory for a `Thing`, and sets the pointer to it

```
// some piece of code which makes a
// BetterClass instance

void mycode() {

  BetterClass b;

  ... use it...


  return ;
}
```

When you make a `BetterClass`, then

(i) memory for a BetterClass is allocated

(ii) the BetterClass constructor is called which uses new to allocate memory for a `Thing`, and sets the pointer to it

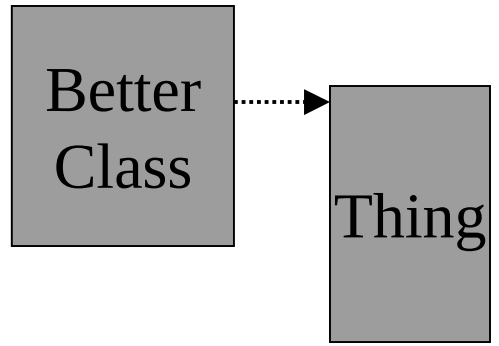


When a `BetterClass` object goes out of scope,

(i) the destructor is called first

(ii) then it gets destroyed

```
// some piece of code which makes a
// BadClass instance

void mycode() {

  BetterClass b  ;

  ... use it...


  return ;
}
```

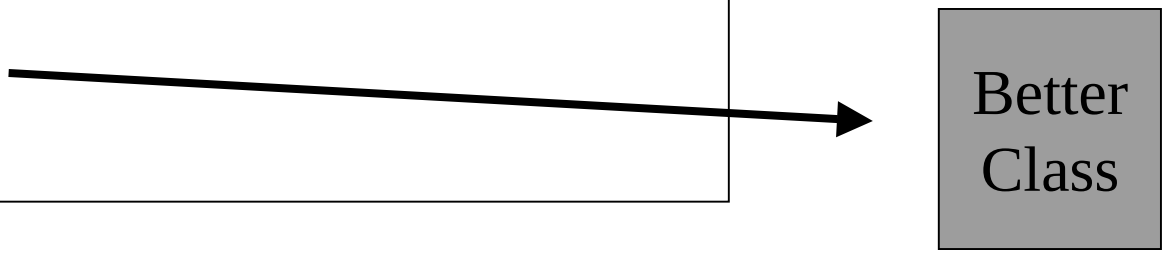


When a `BetterClass` object goes out of scope,

(i) the destructor is called first

(ii) then it gets destroyed

```
// some piece of code which makes a
// BadClass instance

void mycode() {

  BetterClass b  ;

  ... use it...


  return ;
}
```



```
// some piece of code which makes a
// BadClass instance

void mycode() {

  BetterClass b ;

  ... use it...


  return ;
}
```

When a `BetterClass` object goes out of scope,

(i) the destructor is called first

(ii) then it gets destroyed

Be careful

You only need to do this if your class is the only thing which points to some memory from inside itself.

You don't need to do this is you are simply storing a pointer to something passed into you from outside via a method. For in such a case the outside environment is by definition keeping track of the memory in question.

We repeat from an earlier module:

·You rarely need a destructor - although it is considered good practice to always provide one even if it does nothing.

·However always consider it if you have used the new(..) operator inside your class

Memory leaks are one pf the biggest problems of C++ (JAVA for instance does not have this problem)

8 this->

C++ provides a special pointer

Sometimes an object has need to refer to itself. Examples are:

- **to pass itself to a function it is calling**
- **to pass a pointer to itself to a function**
- **to return a copy of itself, or a pointer to itself.**

C++ provides a special inbuilt pointer in every object:

this->

this-> is actually very simple once you have grasped it.

Look at these two bits of code:

```
// To explicitly refer to yourself inside a method

ThreeVector::magnitude( )
{
   float magsq =  this->x * this->x
                + this->y * this->y
                + this->z * this->z ;

   return sqrt( magsq ) ;
}
```

These bits of code are Identical

```
// To explicitly refer to yourself inside a method

ThreeVector::magnitude( )
{
   float magsq =  x * x
                + y * y
                + z * z ;

   return sqrt( magsq ) ;
}
```

if you dont put the this-> in then the compiler does it for you !

Summary of Module : Pointers

In this module we have covered the following topics.

- **Pointers to primitive in-build types**
  - *** to de-reference a pointer**
  - **& to take the address of**
- **Pointers to objects**
  - **The same as for in-built types**
  - **In addition the -> operator for invoking methods**

- **Use of Pointers to pass arguments**
  - **A (not recommended) alternatice to references**
  - **Historical as pointers existed before references**

- **Use of new and delete**
  - **Dynamic allocation of memory**
  - **Returns a pointer to the allocated memory**
  - **You are responsible for keeping hold of the pointer**
  - **you must delete the allocation when finished.**

- **Important use for destructors**
  - **Destructors should delete any memory which an object allocates with new UNLESS responsibility for the pointer has been passed to some other object.**
- **this->**