Module 3:
Algorithm steering elements

In this module we will cover:

- If , elseif, else
- Switch and enumerated constants
- For
- While and Do

Aims of the module

The previous modules have concentrated on programming philosophy. As such they have until now only touched upon very basic C++ language components.

In this module we extend this to cover C components which are in practice needed in almost all programming situations. These are the constructs needed to make branching decisions in programs.

Nono of this has anything sprecific to do with OO programming, but you cant do many more useful exercises until this material is covered.

1.1 if, else if, else

One of the most common operations you want to do is take different actions as a result of some test.

We have already seen examples of this in the previous modules.

The simplest example is the

`if`

construct shown here:

Here is the test

The **`a > b`** form is fairly obvious.

We will cover more forms in a moment

```
// Example of if usage

if( a > b )
{
  .. any code you like
}
```

If the test is passed then the code in {..} is executed

This can be made more useful by extending it with the **optional**

`else if, else`

```
// Example of if usage

if( a > b )
{
   .. any code you like
}

else if( a < c )
{
   .. any code you like
}

else
{
   .. Any code you like
}
```

Only if the first test fails will this second (optional) test be performed.

If it passes the following code in {..} is executed.

You can have many of these

Optionally you can specify this "catch all" to be executed if all the other tests fail.

Finally we need to look at some of the forms of test which can appear:

```
// Examples of tests in if()

// Greater or less than are obvious
if( a > b ) {..}
if( a < b ) {..}

// also greater than or equal to ..
if( a >= b ) {..}
if( a <= b ) {..}

// Equals to is a funny one
if( a = = b ) {..}

// Not equal to
if( a != b ) {..}

// You can AND or OR two tests
if( (test1) && (test2) ) {..}
if( (test1) || (test2) ) {..}
```

Private Study:

Familiarise with the forms of expression which can appear inside an if(..)

In fact it is more general than it looks. The rule is that anything can appear in the test which evaluates to a logical true or false conclusion.

```
// use of general logical value int valid ;

bool valid ;

.. some code which sets a value of valid ..

if( valid )
{
  // this will be executed if valid is true
}

// or you might see

if( !valid )
{
  // executed if valid is false
}
```

What you don't see in the previous examples is that expressions like **a > b** actually evaluate inside C++ to true or false

This means that any function which returns `true` or `false` can be used in the test directly.

For example, lets invent a Particle class in our minds. Suppose this has a method which returned true if the particle was massless, and false otherwise.

Lets call this method isMassless( )

Then you could write the following code:

```
// Use of a function in a test

Particle p ;


.. Some code which sets the attributes of p to something


if( p.isMassless() )
{
  // This code will be executed if p is massless
}
```

1.2 SWITCH

Another construct which allows branching is via use of the SWITCH / CASE statements.

However first you have to be introduced to a new data type known as an

"enumerated constant"

[It will probably take you a few goes to get your head around this one - so expect to cover it in private study and in surgery]

"enumerated constants"

```
// Example of setting up a list of enumerated
constants

enum ParticleType { ELECTRON, MUON, TAU, QUARK };


// Now we can declare variables which can
only be set to the symbolic values in the list

enum ParticleType myParticleType ;


// This is allowed
myParticleType = ELECTRON ;

// These are not allowed
myParticleType = GRAVITON ;
myParticleType = 10 ;
```

This makes a variable called:

myParticleType

which can only be set to the values specified in the list associated with

enum ParticleType

Now we can see the switch/case statements in action using the enumerated list on the last slide

The switch looks at the value of this

The possible cases are all in the {..}

A stupidity - you have to explicitly break at the end of what you want to do for each case

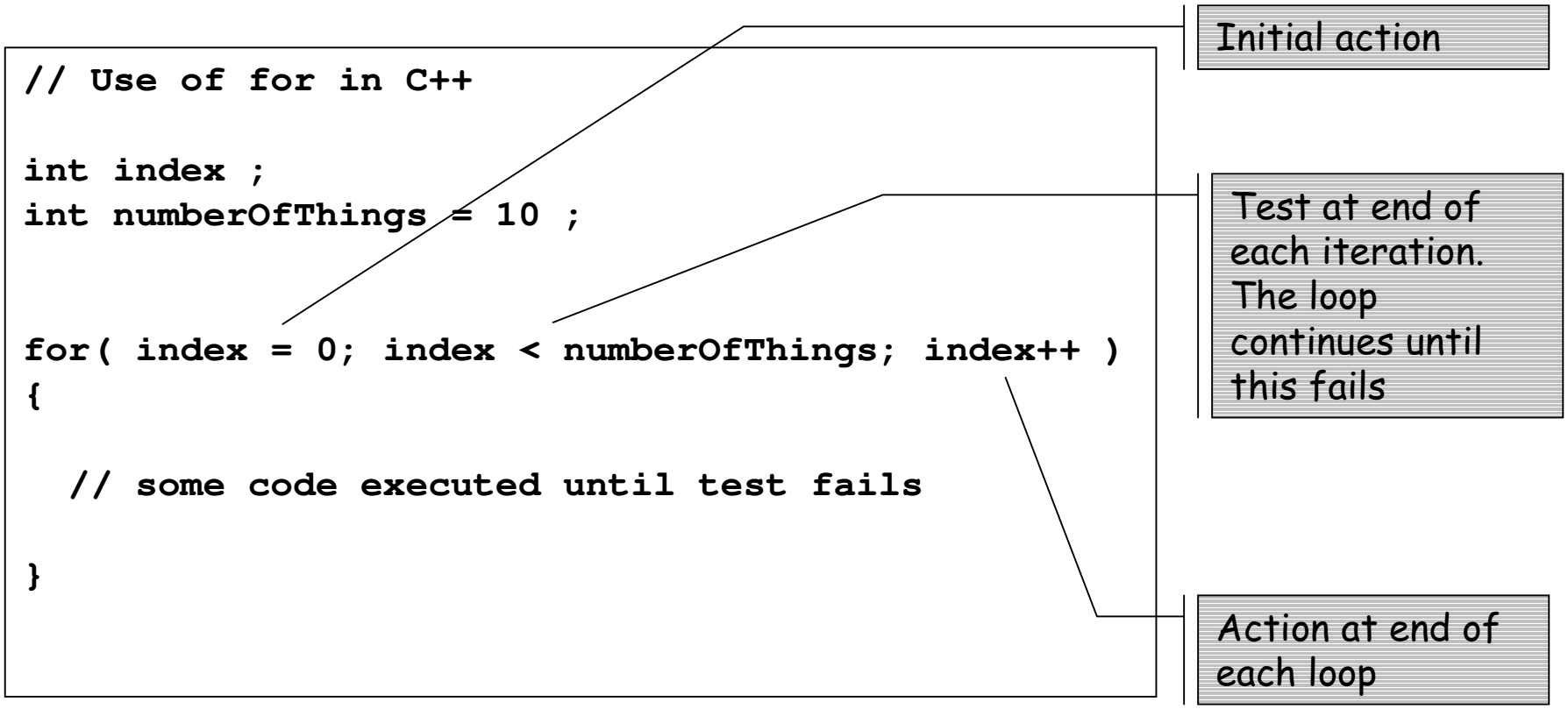
Catch-all in case none match.

```
switch( myParticleType )
{
  case ELECTRON:
    std::cout << "The particle was an electon" ;
    break ;

  case MUON:
    std::cout << "The particle was an muon "  ;
    break ;

  case TAU:
    std::cout << "The particle was an tau " ;
    break ;

  case QUARK:
    std::cout << "The particle was an quark ";
    break ;

  case DEFAULT:
    std::cout << "I havnt a clue what it was ";
    break ;
}
```

1.3 FOR

Now we move to constructs used to repeat a piece of code until some condition is satisfied.

One way to do this is with the **for** statement:



This can be made arbitrarily obtuse and complicated !!!!!!!

All you need to note at this stage is that for loops are not limited to simple integer indicies. You can do anything in the three fields which satistfy the requirements of the previous slide, I.e.

```
(initial action ; logical test ; action at end of loop)
```

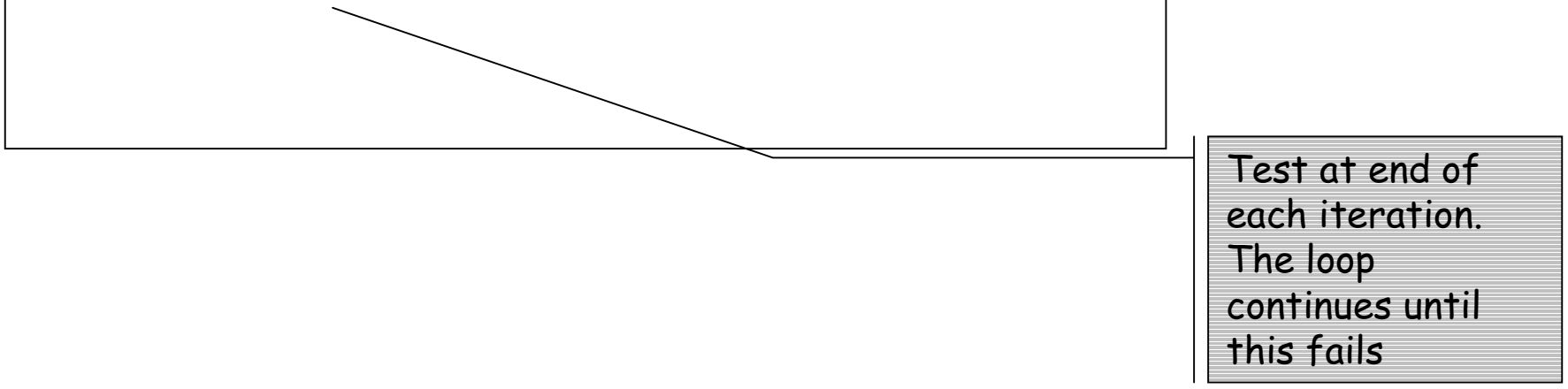
1.5 do / while

Another way to repeat code is to use the while or do statements:

```
// Use of while in


while( any test )
{

  // some code executed until test fails

}
```

Test at beginning of each iteration. The loop continues until this fails

```
// Use of do in C


do
{

  // some code executed until test fails

} while( any test )
```



Summary of Module 3: Algorithm steering elements

- **The if statement**

  if { }

  else if { }

  else{ }

- **The switch statement**

  enumerated constants

  switch / case

- **The for staatment**

  for(... ; ... ; ...) { }

The do and while statements

while(test) { }

do { } while(test)