Module 5: Handling collections of objects: A simple introduction to the vector class

In this module we will cover

- **The need for collections**
- **Minimal Use of the vector class from the standard library**
- **Some methods and operators of the vector class.**

Aims of this module

There are very few areas of scientific coding where you do not immediately run in to the need to store and manipulate a collection of some object. This might be a set of data readings or a collection of ThreeVectors representing tracks in a detector ...etc.

In the abstract examples you might need a collection of BankAccounts administered by one bank, or a collection of books in a library.

In this module we introduce you to the simplest of a set of standard library classes supplied with C++, the vector class.

This is designed to add and remove elements to in an easy way, and has methods to let you manipulate the collection.

This module will also be your first example of use of an object where you ONLY know about its methods, but you neither know nor care how it works internally.

5.1 The need for collections

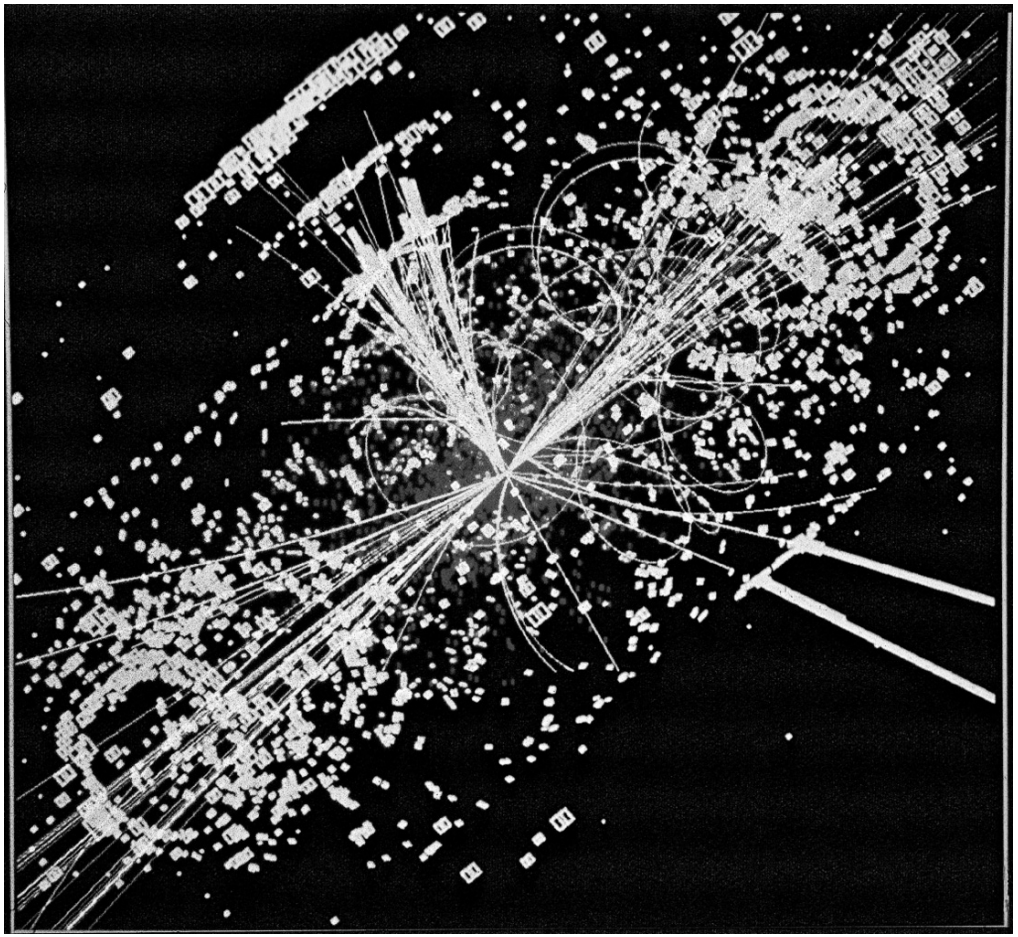
In almost all contexts (I know of no exceptions) you have to deal with collections of things.

(we use the word collection genericaly instead of list which means something specific)

- **A collection of intensities for each angle (from X-ray set)**
- **A collection of measured count rates for each angle (Compton exp)**
- **A collection of resistances for each temperature (from Helium fridge)**
- ....
- **Acollection of cross-sections measured in a particle physics experiment**
- ....
- **A collection of BankAccounts**
- **A collection of Books in a library**

An explicit which we will use later: "jet finding".

Here is a pictue of an event from a particle physics experiment at CERN:



You can clearly see that there are "jets" of particles close to each other.These arise from the decay of some particle.

We run jet finder algorithms on such events,

In order to do this we will need to store and manipulate collections of Particles.

5.2 The vector class from the STL

Imagine then, we need to store a large numer of `Particle` instances, each representing a particle found in the detector.

To do this you need to use the `vector` class as shown on the next slide

This is supplied in a standard C++ library. Its actually called the

"standard template library" or STL for short.

The STL is very important, and we cover it in detail in a later module. For now we are just going to get hands on experience of using vector without really telling you how it works.

This is an example where you use a class which someone else has supplied. You neither know nor care how it works !

You care only about what methods it provides to you - i.e. what it can do for you !

```
// Example of Use of the vector class
#include <iostream>
#include <vector>
#include "Particle.h"

int main()
{
  // Make a vector object to store Particles

  vector<Particle> particleStore ;

  // Set up a loop to find and store 10 particles
  for( int n=0; n < 10 ; n++ )
  {
    // Get parameters for next particle
    float px, py, pz ;
    std::cin >> px >> py >> pz ;

    // Make a particle and initialise it
    Particle p ;
    p.initialise( px, py, pz ) ;

    // Now store it in the vector
    particleStore.push_back( p ) ;
  }
}
```

Lots of things here !

```
// Example of Use of the vector class
#include <iostream>
#include <vector>
#include "Particle.h"

int main()
{
  // Make a vector object to store Particles

  vector<Particle> particleStore ;

  // Set up a loop to find and store 10 particles
  for( int n=0; n < 10 ; n++ )
  {
    // Get parameters for next particle
    float px, py, pz ;
    std::cin >> px >> py >> pz ;

    // Make a particle and initialise it
    Particle p ;
    p.initialise( px, py, pz );

    // Now store it in the vector
    particleStore.push_back( p ) ;
  }
}
```

Firstly you have to include the definition of the vector class

Next, you make an instance of the vector class. Here you make an object called:
particleStore
It is empty at this point.

There is a bit of new magic: you have to specify what type of thing you want to store in it. This is done using the
<Particle>
after the
vector
class name

```
// Example of Use of the vector class
#include <iostream>
#include <vector>
#include "Particle.h"

int main()
{
  // Make a vector object to store particles

  vector<Particle> particleStore ;

  // Set up a loop to find and store 10 particles
  for( int n=0; n < 10 ; n++ )
  {
    // Get parameters for next particle
    float px, py, pz ;
    std::cin >> px >> py >> pz ;

    // Make a particle and initialise it

    Particle p ;
    p.initialise( px, py, pz );

    // Now store it in the vector
    particleStore.push_back( p ) ;
  }
}
```

This is how you set up a loop to do something 10 times.

This was covered in an earlier module.

```
// Example of Use of the vector class
#include <iostream>
#include <vector>
#include "Particle.h"

int main()
{
  // Make a vector object to store Particle

  vector<Particle> particleStore ;

  // Set up a loop to find and store 10 particles
  for( int n=0; n < 10 ; n++ )
  {
    // Get parameters for next particle
    float px, py, pz ;
    std::cin >> px >> py >> pz ;

    // Make a particle and initialise it
    Particle p ;
    p.initialise( px, py, pz ) ;

    // Now store it in the vector
    particleStore.push_back( p ) ;
  }
}
```

This bit of code reads in 3 numbers from the keyboard.

The use of cin >> is new, but fairly obvious. It is covered properly later

This makes an instance of a **Particle** called **p** and initialises it with the numbers read in.

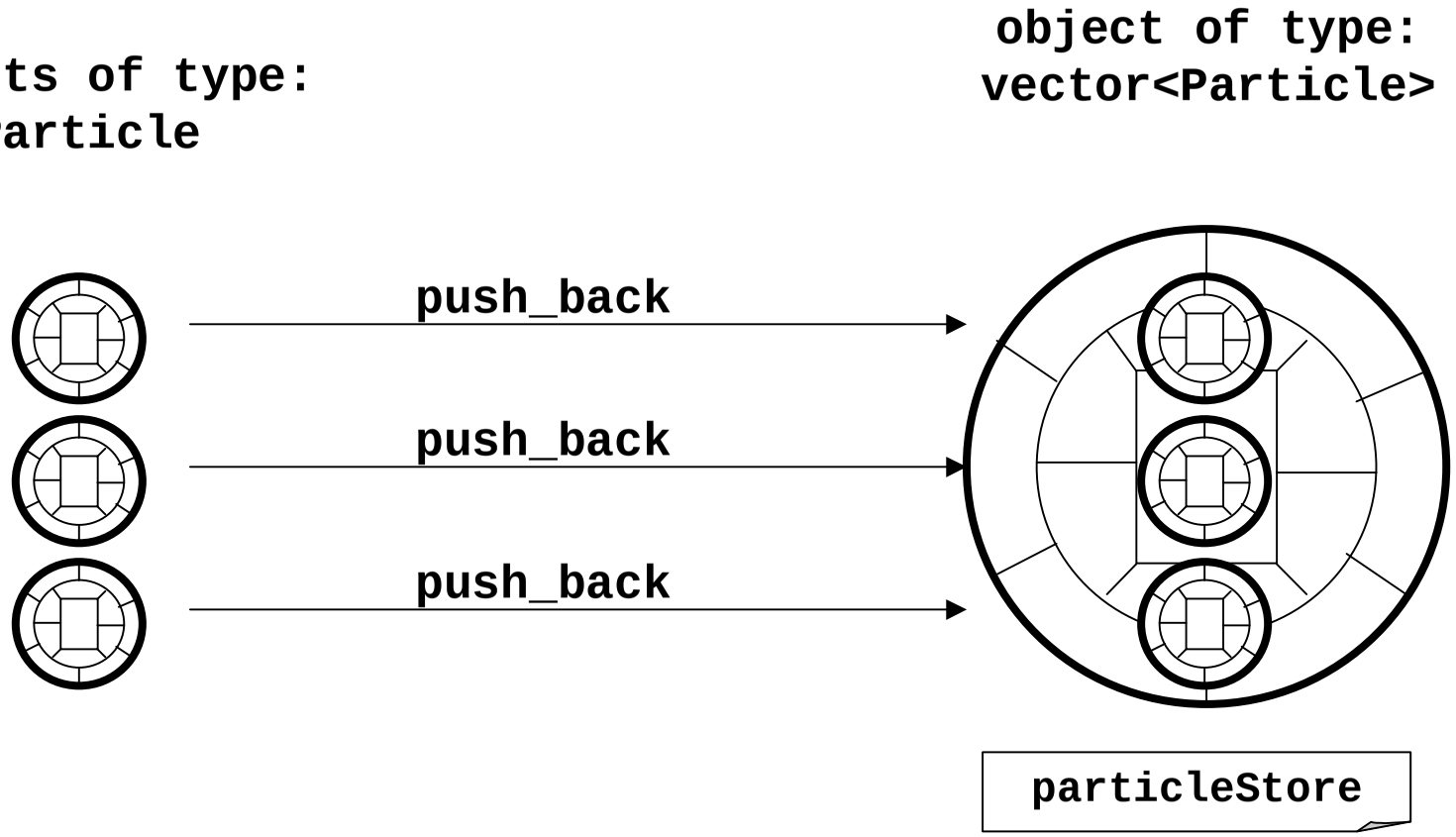
```
// Example of Use of the vector class
#include <iostream>
#include <vector>
#include "Particle.h"

int main()
{
  // Make a vector object to store Particles

  vector<Particle> particleStore ;

  // Set up a loop to find and store 10 particles
  for( int n=0; n < 10 ; n++ )
  {
    // Get parameters for next particle
    float px, py, pz ;
    std::cin >> px >> py >> pz ;

    // Make a particle and initialise it
    Particle p ;
    p.initialise( px, py, pz )

    // Now store it in the vector
    particleStore.push_back( p ) ;
  }
}
```

Now finally we see how the Particle is stored in the particleStore object

The vector class has a method called:
push_back( )
which does the obvious.

Note that a copy of the thing you push back is made internally to particleStore.

Pictorially:



5.3 Basic methods/operators of the vector class

The vector class has lots of useful methods and operators you can use

They are documented in many of the standard reference books

A good web reference is:

```
www.sgi.com/Technology/STL/stl_index.html
```

We are not going to go into these here, as this is a topic of a more advanced module later. At this point we will only look at the minimal set you need to do useful things in programs.

For example, to know how many things have been stored in it you use the size( ) method:

```
// Example of finding how many things in a vector

vector<Particle> particleStore ;
  .... particleStore filled as before....


// Interrogate it to see how many items have been put in it

int numberIn ;

numberIn = particleStore.size() ;

std::cout << "The number of items in the list is " << numberIn ;
```

This is useful if you are filling a vector from a file or the keyboard where the number of items is a priori unknown.

..and here is how you access items using the [ ] operator:

```
// Example of accessing items in a vector

vector<Particle> particleStore ;
  .... particleStore filled as before....


// Lets get the Particle stored
// in the 5th location

 Particle p ;

 p = particleStore[4] ;

 p.dump( ) ;
```

We ask the vector to give us the Particle at index 4.

Indices run from 0, so location 5 is index 4

Here we are simply using the dump( ) method of Particle which we wrote earlier.

You have to get used to this : indices start at 0, not 1
This is true in C,C++ and Java

..and here is how you access items using the [ ] operator in a loop:

```
// Example of accessing items in a vector

vector<Particle> particleStore ;
  .... particleStore filled as before....


// Find how many items have been put in it
int numberIn ;
numberIn = particleStore.size() ;

// Loop over all items in vector
for( int ind=0; ind <numberIn ; ind++ )
{

  Particle p ;

  p = particleStore[ind] ;

  p.dump( ) ;

}
```

Here we extract a copy of the [ind]th item in the vector.

We copy it to our local variable called p

..and for the more advanced here is a much shorter version of the same code. Ask about it in surgery if you dont understand it.

```
// Example of accessing items in a vector

vector<Particle> particleStore ;
  .... particleStore filled as before....


// Loop over all items in vector and dump
for( int ind=0; ind <particleStore.size() ;
ind++ )
{
  particleStore[ind].dump() ;
}
```

5.4 An alternative way to make and fill a vector

So far you have been shown the way one will most often put things in a vector if you are doing it sequentially (using `push_back()`).

There is another way which can be used to put an item at a specific location. This uses the [ ] operator in a rather obvious fashion, *eg*:

```
myVector[2] = something ;
```

where `myVector` is a vector you made, and `something` is an item of the type which the vector contains.

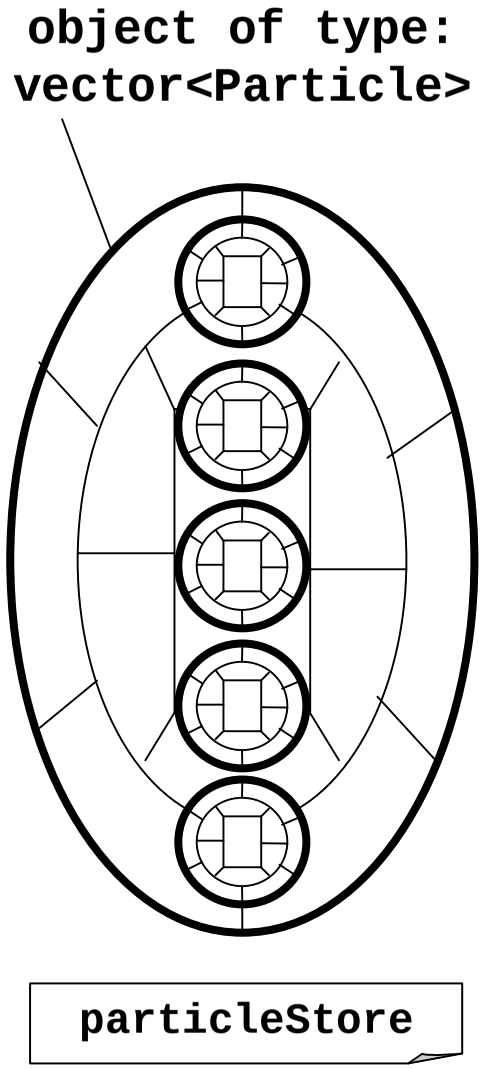
This is clearly just the inverse of accessing an element with [ ]

This is most often used if you are filling an existing vector of a known size, or if you are exchanging items in an existing vector.

This is how you make an empty vector with 5 spaces in it.

```
// Example of making a known size vector

// This makes a vector with space
// for 5 elements
vector<Particle> particleStore(5) ;

....
```

This extra (5) causes a vecotr with empty space for 5 Particle to be created.



This is how you put items at a specific location

```
// Example of filling a known size vector

// This makes a vector with space for 5 elements
vector<Particle> particleStore(5) ;


// Loop over all spaces in vector
for( int ind=0; ind < particleStore.size() ;
ind++ )
{

  // Get a new particle from somewhere
  Particle p ;
  p.initialise( ........ ) ;

  // Put it in the current location

  particleStore[ind]  = p

}
```

This is how we set an element at a specific location given by ind

Later we will see that use of [ ] is rarely used in practice (which is why it wasnt shown here as the primary access)

Instead we will use "iterators" and the begin( ) and end( ) methods which are much more flexible, but have a bit of an overhead to learn so we leave this until later.

Student exercise

- Create a short data file containing several lines, and on each line put three numbers which are the components of a three vector

- write a program to read each line in turn from the file

- From each line it should create a ThreeVector

- it should then store each new ThreeVector in a vector< >

- Finally dump out the contencts of the vector< > by iterating through it and using the dump( ) method of the ThreeVector class.

Summary of Module 5: Handling collections of objects: A simple introduction to the vector class

- The need for collections of things

- The vector class
  - making an collection using vector<SomeType>
  - making an collection of a known size
  - adding items using `push_back( )`
  - finding the number of elements using `size( )`
  - accessing/changing elements using `[ ]`