Vectors

Mark Slater

UNIVERSITY OF BIRMINGHAM

Overview

- For the next part of the cipher code we will need to start using arrays or collections of objects
- To deal with collections of objects dynamically we need to be able to:
  - Hold any type
  - Resize the collection based on runtime values
  - Ensure the memory is allocated and de-allocated correctly
  - Add and remove objects from the collection
  - Loop over the collection
  - Get basic information from it (e.g. size)
- There is another extended C++ type very similar to std::string that can do all of these things and more – std::vector

Declaring and Initialising

- In order for a std::vector to store any type you want, you need to specify at compile time what type you want it to hold
- You do this using the angle bracket/template notation with the type you want it to store in the brackets
- You can initialise the contents of the vector ('={}') OR declare it's properties ({}) on creation (not both!)
- This is just a convention due to a quirk of the language but will help to avoid errors.
- Note there is an added complication for numerical vectors and declaring there size – it will actually create a vector of 1 element.

As with std::string, you need the 'vector' header

You can put any type that meets the vector requirements

```
#include <vector>
#include <string>

int main()
{
    std::vector<int> vec_int{};
    std::vector<int> vec_int2 = {1, 2, 3, 4};
    std::vector<double> vec_dbl = {1.2, 3.4, 4.5};
    std::vector<std::string> vec_str = {"msg1", "msg2"};

    std::vector<std::string> vec_dbl2{5};

    std::vector<double> vec_dbl2{5};
}
```

Create a vector with 5 (uninitialised) elements

Actually creates a vector with a single element (‘5’) in it

Useful Member Functions

- As std::vector is a more complex type than an integer or double type, it also has some functions associated with it that can be used to manipulate and get info from the object
- Some of the most useful are:
    - size() - return the number of elements in the vector
    - empty() - returns true or false depending on if the vector has zero elements
    - push_back(‹object›) - Increase the size of the vector by one and add an object to the end
    - pop_back() - Remove the last object in the vectors
    - at(‹index›) / [‹index›] operator – Access element ‹index›
    - emplace_back(‹args to construct object›) - a more efficient version of push_back that creates the object in place. See Day 6!
- To call these functions, you use the '.' operator on the object
- We'll learn a lot more about this when we deal with classes!

std::vector Example 1: Manipulation

```
#include <vector>
#include <string>
#include <iostream>

int main()
{
    // Construct a vector
    std::vector<double> vec = {1.2, 3.4, 5.6};

    // print out the vector size (3)
    std::cout << vec.size() << std::endl;

    // add a few elements
    vec.push_back(7.8);
    vec.push_back(9.1);

    // vector size (5)
    std::cout << vec.size() << std::endl;

    // remove an element
    vec.pop_back();

     // vector size (4)
    std::cout << vec.size() << std::endl;

    // loop over the vector using an index counter
    for (size_t i{0}; i < vec.size(); i++)
    {
        std::cout << "Index: " << i << " " << vec[i] << std::endl;
    }
}
```

Use the '.' operator to call the member function 'on' the object

To access the elements you can use a for loop and index counter. There is another way but we'll come back to this!