C++ Week 6

Vectors

A one-dimensional **array** is a row of data items, all of the same type. The word "array" is used in programming in general to describe this type of structure, but, in C++, the word "array" is used to refer rather more specifically to the implementation of this structure in C. Since C++ subsumes all of C, you can have C-style arrays in a C++ program. But C-style arrays are inflexible and in some ways awkward to use, so we will use the C++ implementation of the structure, which is a **vector**.

Just as a C++ `string` is more than a mere row of characters - it can also do things for you, such as telling you how long it is or providing a substring - so a `vector` is more than a mere row of, say, integers or doubles. Like a `string`, a `vector` is an object and has a number of member functions. To use vectors in your code, you need the appropriate library - `#include <vector>`. Vectors are declared as in this example:

```
vector<int> ivec(4);
```

which declares a vector called *ivec*, containing four integers. In memory it looks like this:

0	0	0	0

By default the data items are initialised to zero (if they are numbers, or empty strings if they are strings). If you want to initialize the vector to a different value, add an argument to the declaration as follows:

```
vector<int> ivec(4, 3);
```

This would create a vector of four elements, each one initialized to 3. `ivec` looks like this:

3	3	3	3

The elements of a vector are numbered, starting from zero. So the first element of `ivec` would be `ivec[0]`, the second element would be `ivec[1]`, and so on. These numbers (the 0 and the 1) are called *subscripts*. To refer to individual elements of the vector you use the element's subscript inside square brackets. Note that the first element of the vector has subscript zero. In other respects, the elements of an integer vector behave just like ordinary integer variables, as in these examples:

```
ivec[2] = 95;
ivec[0] = ivec[2] + ivec[3];
```

After these operations, *ivec* looks like this:

98	3	95	3

You can use the elements of the vector in the same way as you would any other variable of the same type, for example:

```
x = ivec[0] - ivec[2];
if (ivec[0] < ivec[3]) return;
```

The thing that goes into the square brackets is actually an expression (integer constants and single variables are simple expressions). So, as well as having things like `ivec[2]`, you can also have `ivec[x]`, where `x` is an integer variable, or `ivec[x-2]` or `ivec[x+y-z]`

Array bounds checking

When you use vectors you must take care that you do not try to reference an element beyond the bounds of the vector by using a subscript outside the valid values for that vector - for example, trying to access `ivec[-1]` or `ivec[4]` or `ivec[99]` given `ivec` as defined above. The operating system will not check this for you and you may find yourself using values from undefined portions of memory or, worse, overwriting data. At best the o/s might issue a segmentation fault error if you try to use a subscript that is way out of range. You might also consider using the `at` function in preference to the square brackets. `ivec.at(92)` is the same as `ivec[92]` except that it will terminate the program if `ivec[92]` does not exist. The square bracket notation is widely used in other programming languages and in writing about programming generally.

Manipulating vectors

Let us create a vector *v* of eight integers and initialize it so that each element's value is the same as its index.

```
vector<int> v(8);
for (int i = 0; i < 8; i++) v[i] = i;
```

`v` looks like this:

0	1	2	3	4	5	6	7

The following code causes any negative elements to be set to zero:

```
for (int i = 0; i < 8; i++)
   if (v[i] < 0)
      v[i] = 0;
```

The `size()` member function

The `size()` member function for vectors, like the `length()` member function for strings, returns the number of elements in the vector. Using `v` from the previous example, `v.size()` returns a value of 8. Note that, because subscripts begin at zero, `v.size()` is always one more than the highest subscript of `v`.

The = operator

You can use the = (assignment) operator to assign one vector to another, for example `v1 = v2`, so long as they are vectors of the same type (eg both `vector<int>` or `vector<double>`). In this case the contents of *v1* are overwritten with the contents of *v2* and *v1* is truncated or extended to be the same length as *v2*.

The `push_back()` member function

This function allows you to add elements to the end of a vector. Suppose you create an uninitialised vector as follows:

```
vector<int> evec;
```

so that `evec.size()` is zero. *evec* is effectively useless until you add some elements to it. The following statement will add an element to the end (the back) of *evec* and initialise it with the value 21.

```
evec.push_back(21);
```

At this point, `evec[0]` is 21 and `evec.size()` is one.

Now we can add another element and initialise it with the value 33:

```
evec.push_back(33);
```

Now `evec[0]` is 21, `evec[1]` is 33 and `evec.size()` is two.

The `pop_back()` member function

This function removes the last element from the vector. A typical call looks like this:

```
evec.pop_back();
```

Unfortunately `pop_back()` doesn't return the value of the popped element. If you need it, you have to copy it to somewhere else before you pop, for example:

```
to_be_popped = evec[evec.size()-1];
evec.pop_back();
```

Using vectors as parameters

Let us consider a function that returns the sum of the elements of a vector of integers:

```
int total(vector<int> v)
{  int  total = 0;
   for (int i = 0; i < v.size(); i++)
      total += v[i];
   return total;
}
```

Note that the vector here is passed by value, meaning that the vector's data will be copied for use locally. If the vector is small, there is no problem, but if it is large, then the parameter-passing itself could consume a lot of resources. For this reason it is a good habit to pass vectors by reference, and use the `const` keyword to ensure that the data doesn't get inadvertently modified.

Here is a function that takes as its arguments a vector of integers and an integer. It returns `true` if the integer is in the vector.

```
bool isin(const vector<int>& v, int a)
{  for (int i = 0; i < v.size(); i++)
      if (v[i] == a) return true;
   return false;
}
```

Let us look at the following code fragment for loading words into a vector:

```
string s;
vector<string> v;
while (cin >> s)
   v.push_back(s);
```

The system will continue to populate the vector v until the input stream fails.

The following procedure also adds new words to a vector but does not store duplicates:

```
void add_word(vector<string>& vs, string s) // the procedure modifies the vector,
                                            // so we pass by reference (without const)
{  for (int i = 0; i < vs.size(); i++ )
      if (s == vs[i]) return;               // don't add the word as it's
                                            // already in the vector
   vs.push_back(s);
```

```
}
```

We could call this procedure thus:

```
string s;
vector<string> v;
while (cin >> s)
   add_word(v,s);
```

C++ strings behave (to some extent) like vectors of char

In C++ a string behaves in many ways like a vector of `char` elements; in particular, you can use the square-bracket operator to access a single character. `char` is a datatype; a `char` is a single character. For example:

```
string s = "eyrie";
s[1] = s[0];      // s is now "eerie"

string t = "cat";
char ch = t[0];   // ch is the letter 'c'; note that ch is of type char, not type string
t[0] = t[1];
t[1] = ch;        // t is now "act"
```

`string` literals are enclosed in double quotes – `"Z"` is a string of length one – whereas `char` literals are enclosed in single quotes – `'Z'` is a single character value.

You can also define strings using the same sort of constructor function that you use for vectors. For example:

```
string s(20, '*'); // creates a string of length 20, initialised to asterisks; note that the second argument is a char, not a string
```

The library `cctype` contains a number of useful predicate functions for returning information about `char` data, for example:

`isupper(ch)`	returns true if `ch` is upper case
`islower(ch)`	returns true if `ch` is lower case
`isdigit(ch)`	returns true if `ch` is a digit
`isalpha(ch)`	returns true if `ch` is a letter

Many languages draw a clear distinction between the types `char` and `int` but C and C++, to a large extent, treat them as interchangeable, with implicit type conversions from one to the other. For example, you could do the following:

```
int     n;
char    ch;
```

```
n = 'A' + '!';
ch = n;
cout << n << " " << ch << endl;
```

and the output would be 98 b. That is, you get the ASCII value of 'A' added to the ASCII value of '!', giving 98, and ch takes the value of the char with ASCII value 98, which is 'b'.

This kind of thing is generally confusing and best avoided, but it can occasionally be useful. If, for example, you had a char variable with a digit as its value, you could convert the digit to the corresponding integer value simply by subtracting '0'. For example:

```
char    ch = '9';
int     n = ch - '0';   // 57 - 48, so n is initialized to the value 9
```

or vice-versa:

```
int     n = 9;
char    ch = n + '0';   // ch is initialized to the value '9'
```

C++ strings are different from C strings. C strings are **arrays** of `char`, not vectors (more about arrays and C strings later in the course). If you needed to cast a C++ string into a C string, you would use `s.c_str()` (where `s` is a string).

You can use the `at` function with strings, as you can with vectors, as an alternative to square brackets so as to get the benefit of array bounds checking. It can be used as follows:

```
string s = "hello, world!";
cout << s.at(7) << endl; // returns letter w
s.at(12) =  '?';          // changes ! to ?
cout << s.at(12) << endl;
cout << s.at(s.length()) << endl;   // aborts the program - there is no s[13]
```

However, a `string` is not exactly the same as a `vector <char>`. For example, a `string` has `push_back` (you could have `s.push_back(ch);` to add a char to a string) but it does not have a `pop_back`.

But don't overuse vectors

Finally a word of warning. Because vectors in C++ are so flexible and versatile, novice programmers are inclined to overuse them. Use a vector where it's needed or where it will really help. Don't use one just because you can.

Say you are writing a program that reads in a file of numbers and outputs the largest. Some programmers, especially those who have just discovered vectors, will read the entire file of numbers into a vector and will then search through the vector looking for the highest. But a moment's thought will show that the vector here is not serving any purpose. You only need to inspect the numbers one at a time, and you can do that when you read them from the file in the first place.

Unnecessary vectors, apart from being a waste of computer storage, tend to clutter a program and, often, to confuse the programmer. The vector is a very useful tool, but don't use it always and for everything.

Notes on R. Mitton's lectures by S.P. Connolly, edited by R. Mitton, 2000