Vectors

Every language needs a flexible "container" that can hold any number of things, can grow as needed and provides quick access to any item given a number, its "index" or "subscript". In C++, this is done using the vector.

How do we use vectors? There's a lot we could cover here, but we just want the basics that make our lives easier. First, vectors are defined in a library so we have to include it:

```
#include <vector>
```

To define a variable as a vector, we have to say what *kind* of vector it is. Is it a vector of ints? A vector of Elephants?

```
vector<int> vectorOfInts;
```

Note that the *type* of the variable is `vector<int>`.

Similarly, to define a variable called "vectorOfElephants" to be a vector of Elephants (assuming that someone has defined an Elephant type):

```
vector<Elephant> vectorOfElephants;
```

To access an element of the vector, we usually use square brackets to specify which element. Inside the brackets goes the index. The first element is always numbered zero. Therefore, *if* our vector of ints had at least 17 items in it, and we want to print out the 17th element to standard output (usually the screen), then we write:

```
cout << vectorOfInts[16]; // The 17th element
```

But how many elements are there in the vector to begin with? *None!* At least there are none if we didn't say how many to start out with. (And we didn't say anything about the size when we defined vectorOfInts.) How do we add elements to a vector that doesn't have any? The easiest way to add elements to a vector is to add them to the end, i.e. after the last element in the vector. To do that then we use the *method* "push_back". Just pass the value that you want to put at the end of the vector and the vector will grow long enough to hold it. Here we will add two values to the end of our empty:

```
vectorOfInts.push_back(5);
vectorOfInts.push_back(8);
```

Now we have two items in the vector. We could print them out with:

```
cout << vectorOfInts[0] << ' ' << vectorOfInts[1];
```

Looping over a vector

What if there were more than just those two elements in our vector and we want to print them out? Obviously we should use a loop. Using the for loop discussed above, the code would look like:

```
for (size_t i = 0; i < vectorOfInts.size(); ++i) {
    cout << vectorOfInts[i] << endl;
}
```

In this code, we introduced two new features: `vector`'s `size` method and the type, `size_t`. The method's purpose is obvious, it returns the count of items that the vector is holding.

What is the purpose to the type? Why not just use `int`? The official answer is that int might not be the right type for some compiler on some operating system on some computer. Maybe int won't be "big enough" to hold the size of a vector. To allow your code to be "portable", the type size_t was defined. It will always match up with whatever the size method returns. You may not be concerned with portabilty right now (though you should be!), however your compiler is. It will give you a warning when you compile your program if you use int instead of size_t, complaining of a "type mismatch". Avoid warnings, use `size_t`.

Ranged For

Back to looping. There is an easier way to loop over a vector! It is known as the *ranged for*. But as mentioned above, this is a new feature provided by the C++11 standard, so be sure to have a current version of your compiler. (The current versions of Visual Studio and g++ support the ranged for.)

```
for (int x : vectorOfInts) {
    cout << x << endl;
}
```

We didn't have to introduce an index variable or even discuss size_t or size. There's less to write, so you are more likely to type it correctly. And there's less to read, so anyone reading your code will have an easier time knowing what you are saying. Definitely a win-win.

While our ranged for accomplishes the same thing as the traditional for loop I showed first, there is an important difference that goes beyond appearances. The ranged for is actually equivalent to the following variation on our loop:

```
for (size_t i = 0; i < vectorOfInts.size(); ++i) {
    int x = vectorOfInts[i];
    cout << x << endl;
}
```

So, what's the important difference? Notice that `x` is a *copy* of the element in the vector. Why does that matter? It doesn't in our example of printing the elements in a vector of ints. But what if we wanted to *modify* the contents of the vector? Below is a loop that *looks like* it modifies all of the elements in the vector to be 17. In fact, it doesn't change the contents of the vector at all.

```
for (int x : vectorOfInts) {
    x = 17;  // Does not modify the vector
}
```

If we want to change the contents, then we need to use a feature we introduced in the discussion of parameter passing, the "reference"

```
for (int& x : vectorOfInts) {
    x = 17;  //  Does modify the vector
}
```

See how we specified `x`'s type to be a *reference* to an `int`, by using the "and-sign"? Now there won't be any copying. Instead `x` will be an alias for each element of the vector as it steps through the loop. (If you have a Java background, you should notice that this is different and more flexible that Java's "foreach" loop.)

Initializing a vector, specifying it's size

Suppose we want to have the vector start out with, e.g. 28 elements, all with the value 17, we could have defined our vector as:

```
vector<int> vectorOfInts(28, 17); // initialised to hold 28 seventeens.
```

Or, alternatively if we wanted 28 elelments but were willing to settle for the "default" value for the type, which for numbers is zero, we could have defined it as:

```
vector<int> vectorOfInts(28); // initialized to 28 zeros.
```

Note that if we had provided the size 28, as shown above in either example, then doing a subsequent push_back would, of course, add a *29th* element.

However, it is more common that we will simply want to add to the end of an initially empty vector, hence you will most frequently see us definee our vectors without specifying a size:

```
vector<int> vectorOfInts; // initialized to a size of zero.
```

Defining a function that has a vector parameter

When writing a function that will take a vector of ints as an argument, the type is vector<int>. But we will almost certainly want to pass the vector *by reference*! Otherwise when we try to change the values stored in it, the changes will only happen to a *copy*! This is very different from passing arrays. Furthermore, if we are not going to change the constents of the vector then pass it by *constant* reference! Why? So that we don't waste time and space making a *copy* of the vector.

```
void displayVector(const vector<int>& aVector) {
   for (size_t i = 0; i < aVector.size(); ++i) {
      cout << aVector[i] << endl;
   }
}
```

Useful Vector Methods

For most of our uses of vectors, only the two methods that were described above are really needed.

- push_back
    - Takes a single argument, the thing to be *copied* onto the end of the vector.
    - The vector's size increases by one, each time we do a push_back.
- pop_back
    - Takes no arguments and returns *nothing*.
    - Removes the last item from the vector, decreasing the size by one.
- size
    - Takes no arguments.
    - Returns a `size_t` representing how many items are in the vector.
    - Note that the index for the last item in the vector `aVec` would be `aVec.size()-1`.

What other methods are useful?

- clear
    - Takes no arguments.
    - Sets the size of the vector to zero.

- back
    - Takes no arguments.
    - Returns a reference to the last element in the vector.

2D Vectors

There is nothing new in this section as far as C++ features, but I find that students are sometimes unsure how to handle a 2D world.

The following code examples are going to consider how to build up and initialize a two-dimensional world of ints in which all of the cells start out with the same value.

```
int rows = 3;
int cols = 4;
vector<vector<int>> world;  // Our 2d world
for (int r = 0; r < rows; ++r) {
    // The row that we will build up
    vector<int> aRow;
    // Building up a row of 17's
    for (int c = 0; c < cols; ++c) {
        aRow.push_back(17);
    }
    world.push_back(aRow);
}
```

That code is ok. But some students, especially if they are coming from a language like Python, really want a *shorter* way to do the same thing.

That code is perfectly fine, but in a moment we will consider other ways to write it. But first we should make sure we have the right results (test early and often!).

Of course there are the two usual approaches to looping over vectors, either by index or more directly by the elements in the collection. Let's do both!

First the good old-fashioned way, by index:

```
for (size_t rowI = 0; rowI < world.size(); ++rowI) {
    for (size_t colI = 0; colI < world[0].size(); ++colI) {
        cout << world[rowI][colI] << ' ';
    }
    cout << endl;
}
```

And now using the ranged for to directly access the elements. What does the world hold? Rows, represented as vectors.

```
for (const vector& aRow : world) {
        for (int cell : aRow) {
            cout << cell << ' ';
        }
        cout << endl;
    }
```

Now back to modifications to the original code that created the 2D vector. Our first stelp will be to consider an alternative to that internal loop. We did discuss this before. Using one of the vector class's constructors we can simplify, or at least shorten, the above code to:

```
    int rows = 3;
    int cols = 4;
    vector<vector<int>> world;  // Our 2d world
    for (int r = 0; r < rows; ++r) {
        vector<int> aRow(cols, 17);  // Initialize a vector of 17's
        world.push_back(aRow);
    }
```

Many students will be happy to use this since it takes less typing. :-)
But there is nothing wrong with writing the loop! Please don't get too hung up on trying to remember library features.

If we are going to take this approach of using the constructor, we could push it a step further. The original outer loop is again just defining a vector whose length is known in which all of the elements will be the same. What will the length be? `rows`. And what will each element be? A `vector`. Put it all together and we have:

```
    int rows = 3;
    int cols = 4;
    vector<vector<int>> world(rows, vector<int>(cols, 17));  // Our 2d world
```

Please, if you find that version confusing, feel free never to use it! I simply make you aware of it as some students are anxious to know if C++ *can* be written with some of the apparent power that they enjoyed in Python. Yes, but sometimes the expressions will be a little more complicated.

So we are done. But I still want to go back to the original code. We have a nested loop, with the inner loop building up the same vector, over and over again. We could *factor* that out.

```
    int rows = 3;
    int cols = 4;
    vector<int> aRow;
    for (int c = 0; c < cols; ++c) {
        aRow.push_back(17);
    }
    vector<vector<int>> world;
    for (int r = 0; r < rows; ++r) {
        world.push_back(aRow);
    }
```

So, as I said at the beginning, we haven't introduced anything new here, but perhaps seeing it all together will have been helpful.