Vector

```
vector<type> varName;
```

A vector is like an array except a vector has the ability to grow and shrink in length. In order to use a vector, you need to include the preprocessor directive **#include <vector>**.

If you are not **using namespace std;**, you need to either put **using std::vector;** at the top of your program or prepend vector with **std::**, such as **std::vector<double> myVector;**.

A vector is a sequential container, which means the elements (stored data) may be accessed in order. You can also randomly access an element with the square bracket operator, such as **myVector[5]**. Vectors have elements of only one type. Vectors are also part of the Standard Template Library which means the type you want your vector to be will be in angle brackets. Additionally, there are several member functions that can be used and the most common are on the next few pages.

Here is how you declare a vector of type **double**:

```
std::vector<double> myVector; //Empty vector
```

If you would like to initialize a vector with values you may do:

```
std::vector<double> myVector(3, 1.0); //3 elements all with value 1.0
std::vector<double> myVector = {1.0, 2.0, 3.0} //3 elements
```

The first initializes three elements and every element has the value 1.0. The second initializes three elements and element one has the value 1.0, element two has the value 2.0, and element three has the value 3.0 (much like an array initialization).

In order to insert an element into a vector, you use the member function **push_back(DATA)** where **DATA** is the value you want stored. Here is how to add a **double**:

```
myVector.push_back(4.58); //Puts DATA at the end of the vector
```

Accessing an element can be done a few ways. The first way is exactly like how to access an element in an array, with the square brackets:

```
myVector[0]; //Accesses first element of vector
```

There is a member function **at(INDEX)** that acts just like the square brackets operator (previous example). The difference is that this checks to make sure **INDEX** is a valid index. If not, the program will terminate early with an error message. For example, if **myVector** has one element only with the value of 4.58, **myVector.at(0)** would return 4.58, but **myVector.at(1)** would cause the program to abort early since there is only one element in the vector.

You are able to return the first element of a vector with the member function **first()** and the last element of a vector with **last()**.

The last access method is via the use of an iterator. A vector's iterator is a random access iterator (see iterators page for more information). First, you need to declare and initialize an iterator to your vector, such as:

```
vector<double>::iterator it = myVector.begin();
```

The variable to use as your iterator in this case is **it** (short for iterator). It has been initialized to the beginning of the vector, so the first element. In order to access the first element, you must **dereference** the iterator. **cout << *it;** would display the first element in your vector. Remember that the end for an iterator is **one past the end of the vector**. This means that it does not point to anything of use. The last element of our vector can be accessed when the iterator is set to:

```
it = (myVector.end() - 1); //Move iterator back one position
```

If you prefer to use the reverse iterator to access the last element, you may do so by doing as such:

```
std::vector<double>::reverse_iterator rit = myVector.rbegin();
```

If you want a constant iterator (meaning you can look at each element but not alter it), you would do so as such:

```
std::vector<double>::const_iterator cit = myVector.cbegin();
```

To iterate through your vector you can use a **for** loop a few ways:

```
for(it = myVector.begin(); it != myVector.end(); it++)
for(rit = myVector.rbegin(); rit != myVector.rend(); rit++)
for(cit = myVector.cbegin(); cit != myVector.cend(); cit++)
for(auto e : myVector)
for(auto &e : myVector)
```

The first **for** loop traverses the vector from the first element to the last element. You are able to dereference the iterator and manipulate the contents. The second **for** loop traverses the vector from the last element to the first element. You are also able to dereference the iterator and manipulate the contents. The third **for** loop traverses the vector from the first element to the last, but you are not able to alter the data it points to. You are only able to dereference the iterator to view the contents.

The fourth and fifth **for** loops are much more concise. Here, you can think of **e** as a dereferenced iterator. The **e** without an ampersand can only display the data from the vector and will do so from first element to last element. The **&e** is able to alter the contents of the vector. This also runs through the vector from the first element to the last element. Here is an example of both:

```
for(auto e : myVector)
    cout << e << endl; //Displays each element
for(auto &e : myVector)
    e = 70; //Alters each element to be 70
```

The member functions **size()** and **capacity()** return different values. You can think of as **size()** returning the number of used elements and **capacity()** as the size of the vector as a whole. Once a vector is full, its capacity will double upon the next insertion. For example, if you have if you initialized a vector to have two elements, **myVector.size()** will return 2 and **myVector.capacity()** will also return two. If you **push_back()** a value, then **myVector.size()** will be three (since you now have three elements in the vector), but **myVector.capacity()** will have doubled and return the number 4.

The member function **resize(NUM)** will resize your vector where **NUM** is the new size of your vector. Remember size is different from capacity.

1. If **NUM** is smaller than the current size of your vector, it will be resized and any values after the resize number will cut off and be destroyed. For example, if you have a vector of size 10 and use **myVector.resize(4);** the first four elements will remain in place while the last six elements will be destroyed.
2. If **NUM** is larger than size, it will expand with all previous elements untouched. If you do not specify a value to fill those elements, each element will be set to 0. For example, **myVector.resize(7)** would keep the first four elements untouched (from our previous resize) and add three elements each with the value of 0.
3. If you want a value other than zero, you may specify the value as such: **myVector.resize(7, 2.0);** This will keep the first four elements untouched (from the previous resize in section 1) and add three elements each with the value of 2.0.
4. Do not worry about capacity as that will change automatically if it needs to (if **NUM** is larger than capacity).

To check to see if the vector is empty, you can use the member function **empty()**. This returns a boolean true if empty or false if not empty.

You are able to clear a vector and free it of all data by using the member function **clear()**. This will make the vector empty.

Visit **http://www.cplusplus.com/reference/vector/vector/** to learn more about vectors and see a complete list of member functions.