Lecture 5 Notes

Vectors

The C++ standard library provides a data type template called vector. A vector stores multiple values of the same data type. Consider this example:

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main()
{
   vector<double> v;

   v = { 2.3, 1.0, -420.69, 9001.1, 42.0 };
   cout << v[0] << endl <<
      v[1] << endl <<
      v[2] << endl <<
      v[3] << endl <<
      v[4] << endl;
}
```

This example creates a vector, assigns a list of values to it, then displays each value to the terminal window.

Lets look at each statement one at a time:

1. `vector<double> v;` declares a variable v which is a double vector. v can hold any number of elements which are all of type double.

2. `v = { 2.3, 1.0, -420.69, 9001.1, 42.0 };` assigns 5 elements to v. The brace-enclosed list of values is known as an initializer list.

3. The last statement inserts each element to cout on a new line. The subscript operator, which is a pair of square-brackets, allows us to access an element from a vector via it's index. The first element of v is v[0], second element is v[1], ..., and the last element is v[4]: a total of five elements.

Iterating over a vector

If we wanted to display all the values in a double vector, the ideal way to do it would be to use a ranged-based for loop. Consider this statement:

```
for (double e : v) {
   cout << e << endl;
}
```

This is an alternative form of the for statement that iterates over the elements of a vector. The vector we are iterating through is v and the variable e will contain a copy of the next element during each iteration.

Since e will merely contain a copy of an element, modifying e will not affect an element stored in v. To modify each element in v, we would need to use a reference:

```
for (double& r : v) {
   r *= 3.0;
}
```

This statement multiplies each element in v by 3.0. The variable r is a reference to a double, this means that r will be a reference to the next element in v during each iteration. Modifying the reference will cause the element to be modified as well.

Dynamically adding elements

A variable that uses the vector template contains special functions that can be used to read or manipulate it's elements.

```
int i;
double n;
vector<double> v;
cout << "Enter 5 numbers: "s;
for (i = 0; i < 5; ++i) {
   cin >> n;
   v.push_back(n);
}
```

This snippet of code reads 5 numbers, and appends each of those numbers to v. Appending an element takes place when we use the `push_back` member function.

Member functions can be accessed using the member-of operator, which is represented by a dot

Removing previously added elements

The member function `pop_back()` removes the last element of a vector:

If you want to remove all elements, you can use the `clear()` member function or you can assign an empty initializer list to the vector.

Getting the number of elements in a vector

The member function `size()` returns the number of elements in a vector. This is useful if you're doing something like calculating the average value:

```
double sum = 0.0;
double avg;
for (double e : v) {
   sum += e;
}
avg = sum / v.size();
```

Using vectors as function parameters

This program contains a function that multiples each value in a vector by a scalar value:

```
void scale_vector(vector<double>& v, double s)
{
   for (double& e : v) {
      e *= s;
   }
}

int main()
{
   vector<double> u = { 2.0, 4.0, 6.0 };
   scale_vector(u, 5.0);
   for (double e : u) {
      cout << e << ' ';
   }
   cout << endl;
}
```

The program outputs: "10.0, 20.0, 30.0". The function scale_vector has a reference to a vector as it's first parameter, so when that function is called, the parameter v will refer to some vector variable that exists somewhere else in the program.

Since the main function passes vector u as the first argument into scale_vector, anytime scale_vector modifies it's parameter v, the variable u will also be modified.

Now if we have a function that does not modify it's vector parameter, we can use a "constant reference" instead:

```
double vector_average(const vector<double>& v)
{
   double sum = 0.0;
   for (const double& e : v) {
      sum += e;
   }
   return sum / v.size();
}
```

This function calculates and returns the average value of it's vector parameter. Since v is only read but not modified, we use the keyword "const" to declare that the vector parameter is constant (not modifiable). We also have to use "const" in our ranged-based for loop.

It is possible to not use a reference in a vector parameter, in that case: the program would make a temporary copy of the original vector for the function to work with. Using vector parameters instead of vector reference parameters is usually bad practice because it reduces the speed of the program by performing unnecessary copying.

In general, you should use normal parameters for scalar values (char, int, double, etc) and reference parameters for non-scalar values (vector).

Extracting all data from cin

As you know, the cin variable is an istream linked to the program's standard input. This usually means, that the input is coming from the computer's keyboard.

Like vectors, istreams also have member functions. The expression `cin.good()` returns true if there is more data to read from cin. We can make a program that sums all the numbers read from cin:

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   double sum, n;
   sum = 0.0;
   while (cin.good()) {
      cin >> n;
      sum += n;
   }
   cout << sum << endl;
}
```

When you run this program, enter some numbers delimited by spaces and press CTRL-D to end user input. The program will output the sum of those numbers.

– *Mark Swoope*