COMP 161 - Lecture Notes - 13 - Vectors

April 5, 2015

In these notes we look at the C++ Vector class.

The Vector

If all we ever needed we to manage collections of characters, then the *std::string* class would probably suit us just fine. But what if we need a collection of numbers? What about a collection of strings? Or a collection of collection of numbers? As soon as the contained type is anything other than the *char* type we need a new container type beside the string. A good general purpose container in C++ is the *vector*[1] class.

[1] http://www.cplusplus.com/reference/vector/vector/

Vectors derive their name from mathematics. At its core, a vector is a fixed size, indexed collection of instances of a single data type. The C++ vector library provides us with means of changing the size of the collection, but this mechanism is not without cost. In what follows we'll look at the core functionality of the vector and leave the remaining functionality for you to explore on your own.

Declaring and Initializing Vectors

Strings always contain characters. Vectors can contain anything and the contained type must be declared as a template argument.

C++ type	Description
std::vector<int>	Vector of ints
std::vector<double>	Vector of doubles
std::vector<std::string>	Vector of strings
std::vector< std::vector<int> >	Vector of Vectors of ints

Table 1: Some vector types

Note that spacing out the > symbols in the vector of vectors is important to avoid confusion with *operator>>*.

Constructors

The vector class provides several convenient constructors[2]. Perhaps the most important attribute we need to establish about a vector when we declare it is the size. While vectors are dynamic structures that can grow to fit our data, resizing vectors is not without cost. So, if we know how much data we'll be dealing with and can create a

[2] http://www.cplusplus.com/reference/vector/vector/vector/

vector to fit that data, then we can avoid hidden resize costs. The most basic constructors let us establish the initialize size.

```
vector<int> v; // v is empty

vector<double> w(25); // w can hold 25 item
```

If you want to set the initial value at each location in the vector to a specific value, you can use a fill constructor similar to the one provided by the string class.

```
vector<int> v(25, 5); // v contains 25 instances of 5

vector<char> w(7, 'a') // w contains 7 instances of 'a'
```

Finally, we have the expected copy constructor to make a copy of an existing vector. In this example we'll use variables just to show that the constructor inputs need not be literals.

```
int size(34);
int init_val(-3);

vector<int> v(size,init_val); // v is 34 instances of -3

vector<int> w(v); // w is a copy v
```

Initializer Lists

The C++11 standard introduced initializer lists for easy initialization of vectors with arbitrary vales.

```
vector<int> v{1,2,3,4,5,6,7};

vector<double> w{0.1, 0.2, 0.3, 0.4, 0.5};

vector< vector<int> > s{ {1,2},{3,4},{5,6} };
```

It is very important to realize that the expression $\{0,1,2,3,4,5\}$ is not seen as a *vector<int>* literal by our compiler. It's just convenient syntax for specifying the sequence of values needed in this case. So, when we start looking at tests, we *cannot* do things like this

```
vector<int> v{1,2,3,4,5,6,7};

EXPECT_EQ( {1,2,3,4,5,6,7}, v);
```

Variant Predicates and Size

The most important characteristic of a vector is its size, which tells us the number of items currently in the vector. Knowing the size let's us iterate over vectors because it gives us access to the maximum index value. You can determine if a vector is empty or not using the *empty* class method. The exact size of a vector can be determined though the use of the *size* method.

```
vector<int> v;
EXPECT_TRUE(v.empty());
EXPECT_EQ(0,v.size());

vector<double> w{0.1, 0.2, 0.3, 0.4, 0.5};
EXPECT_FALSE(w.empty());
EXPECT_EQ(5,w.size());
```

Selectors

The most fundamental action a programmer carries out on a collection of data is to select a single element. Vectors provide the exact same selection mechanism as strings do: *operator[]* or the class method *at*. There are a few other selectors that you're welcome to explore, but we'll stick mainly to *operator[].* Let's expand our previous tests to utilize the selectors.

```
vector<int> v;
EXPECT_TRUE(v.empty());
EXPECT_EQ(0,v.size());


vector<double> w{1,2,3,4,5,6,7};
EXPECT_FALSE(w.empty());
EXPECT_EQ(7,w.size());

for(int i(0); i<v.size() ; i++){
EXPECT_EQ(i+1,w[i]);
EXPECT_EQ(i+1,w.at(i);
}
```

As the above example demonstrates, with the addition of the selectors we can write tests for our vectors that explicitly check the values at each location or select locations. This is, more or less, the same logic carried out by *operator*== as defined by the *vector* class.

When prudent, we can use *EXPECT_EQ* to test for a vector rather than scanning through the vector ourselves.

```
vector<int> v{1,2,3,4,5,6,7};

vector<int> w{1,2,3,4,5,6,7};

EXPECT_EQ(v,w);
```

Mutators

When the selector is used in *l-value* position[3], we can use it to mutate individual elements of the vector.

[3] to the left of the assignment operator =

```
// v is 125 instances of 'a'
vector<char> v(125,'a');

// change all the 'a's to 'A's
for(int i(0); i<v.size() ; i++){
   v[i] = toupper(v[i]);
}

// test for expected change at each location
for(int i(0); i<v.size(); i++){
   EXPECT_EQ('A',v[i]);
}

// or alternatively.
vector<char> w(125,'A');
EXPECT_EQ(w,v);
```

Procedures for Vectors

Like stings, vectors are well suited to iteration because we can easily traverse over the elements of a vector with a counted loop:

```
for(unsigned int i{0}; i < v.size() ; i++){
  ... v[i]...
}
```

As we start designing procedures for containers we need to be aware of the underlying costs of passing these containers between procedures. Pass-by-value means pass a copy. If we're talking about a vector of 10,000 elements, then making a copy isn't something we

want to outright ignore. However, compilers have gotten really good a picking up on essential versus non-essential copies for built in containers like vectors and strings. This means that for the most part, we can pass-by-value when we need to and trust basic optimization tasks to our compiler. This also means we should let the compiler do our copying for us rather than attempt to optimize the process through some kind of explicit copy.

Let's design two vector based procedures. The first is a map-style procedure that applies a function to every element in a vector and produces a new vector. In this case, we'll mutate the compiler's copy of our original vector to produce the result vector. Our second procedure is a fold-style procedure. For this procedure we really only need to read data from our input vector. Whenever this is the case, we should use a mechanism called PASS BY CONST REFERENCE where we pass by reference, but tell the compiler not to allow mutation on the parameter.

Squaring the elements of a Vector

Consider the problem of squaring a collection of doubles. We can capture this as a function with the following procedure.

```
/**
  * Compute the vector containing the squares of
  * all the elements of v.
  * @param v a vector of doubles
  * @return the squares of v
  * @pre none
  * @post none
  **/
 std::vector<double> squareAll(std::vector<double> v);
```

Testing this procedure is complicated a bit by the fact that our contained type is doubles. This means using EXPECT_EQ to compare two vectors is dangerous because it means the use of the == operator to compare double values. This forces us to check each vector element with EXPECT_DOUBLE_EQ. The exception to this is checking the empty case[4].

[4] no double, no problem

```
std::vector<double> mt;
// no double checks. just checking for empty
EXPECT_EQ(mt,ln13::squareAll(mt));

std::vector<double> v{0.1,0.2,0.3,0.4,0.5};
std::vector<double> expected{0.01,0.04,0.09,0.16,0.25};
```

```
std::vector<double> actual{ln13::squareAll(v)};

for( unsigned int i{0}; i < v.size() ; ++i){
   EXPECT_DOUBLE_EQ(expected[i],actual[i]);
}
```

When implementing this function we recognize that we need a vector of the same size as the input vector. The best practice is to go ahead and pass by value and use the copied parameter as that vector. We then modify that vector. We've seen this strategy before. From the accumulation of information perspective, we're actually accumulating effect, not data. After step k of the iteration we've squared v from $[0, k)$, i.e. we've accumulated k mutation effects in the vector v.

```
std::vector<double> squareAll(std::vector<double> v){

  //modify the v. it's a copy of the argument

  for(unsigned int i{0} ; i < v.size() ; i++ ){
    v[i] = v[i]*v[i];
  }

  return v;
}
```

The key here was that we leveraged the compiler generated copy of our vector rather than make a copy of our own. By working with the compiler we give it more opportunities for automated program optimization.

Summing a vector of ints

Let's look at another example. We want to sum all the integers in a vector of integers. We could go ahead and proceed as usual with this problem. In which case we'd pass the vector by value. However, given a little thought, it becomes clear that we only need to read data from our input vector. We never need to write. We also do not need a copy of the vector for any reason. Whenever this is the case, you should use *pass by const reference*. This is a pass-by-reference parameter with the *const*[5] keyword in front of it. By declaring the reference parameter constant, we've told the compiler to ensure that the parameter does not undergo mutation. As a reference, we avoid the copy. As a constant, we ensure the integrity of the now shared data.

[5] const

Like pass-by-reference, pass by const reference is carried out by a small change to the procedure's signature.

```
/**
 * sum will return the sum of contents of a vector of ints
 * @param v is a vector of ints
 * @return the sum of the contents of v
 * @pre none
 * @post none
 */
 int sum(const std::vector<int>& v );


TEST(sum, all){
   using namespace std;

   vector<int> mt;
   EXPECT_EQ(0,sum(mt));

   vector<int> notMT{1,2,3,4,5};
   EXPECT_EQ(15,sum(notMT));

}
```

Let's step through[6] through the design of our iterative procedure for practice. We can start our implementation with the basic loop template.

[6] iterate!

```
int sum(const std::vector<int>& v){


  for(int i(0); i < v.size() ; i++){
     ... v[i] ...
  }

  return 0;
}
```

Now we think about the information that we're accumulating. The solution to the problem is a sum of integers, so the accumulator variable should be an integer. Standard practice is to combine the accumulator with the current vector element within the loop to get the next accumulator value and then return the accumulator. Let's plug that in.

```
int sum(const std::vector<int>& v){
```

```
  int total{...};

  for(int i(0); i < v.size() ; i++){
     ... total ... v[i] ...
  }

  return total;
}
```

We can nail down the initial accumulator value by determining the solution when the vector is empty, when no iteration occurs. The sum of nothing is nothing, aka 0.

```
int sum(const std::vector<int>& v){


  int total{0};

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

  return total;
}
```

Now the iterative update. Let's think about our notMT test case. If $i = 2$ and we're about to execute the loop body, then *total* is the accumulated sum so far, i.e. the sum of everything in v with the index in $[0, 2)$, which for our data is 3. The element $v[i]$ is 3. We want the update operation to update *total* to be the sum of everything v with indexes from $[0, 3)$/ This is just *total* + *v[2]*. Generalizing in terms of $v[i]$ leads to the loop update operation[7]

[7] which we simplify using $+=$

```
int sum(const std::vector<int>& v){


  int total{0};

  for(int i(0); i < v.size() ; i++){
     //total = total + v[i]
     total += v[i];
  }

  return total;
}
```