COMP 161 - Lecture Notes - 16 - Vectors and Arrays

Spring 2014

In these notes we look at the C++ Vector class and talk further about Array-like containers.

The Vector

If all we ever needed we to manage collections of characters, then the *string* class would probably suit us just fine. But what if I 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 *char*, 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.

Whenever you set out to learn about a new container type, or any type for that matter, you always want to figure out four things: how to construct[2] an instance of the type, how to recognize different variants of the type, how to select elements out of the collection, and how to modify the collection. We know these core functions as CONSTRUCTORS, VARIANT PREDICATES, SELECTORS, and *mutators*.

[2] declare and initialize

Declaring and Initializing Vectors

Declaring a vector variable and filling it with data is a bit more involved than the types we've encountered so far. Vector types require the use of templates to declare the type contained in the vector. Up until now, we've been able to use literal values to initialize variables, but there is no literal syntax for vectors. So, for situations where are vectors contain anything other than the repeated instance of a single value, we'll need to write special INITIALIZER procedures.

Vector Types

The question you must always answer is, "a vector of what?" Strings always contain char data, so the contained type isn't a variable and we don't have to declare it in our code. With vectors, the contained type is a variable and must be clearly declared for the compiler. Vectors can contain any previously defined C++ type. Here are a few examples:

Table 1: Some vector types

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

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

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

```
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.

```
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
```

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(15,0.0);
EXPECT_FALSE(w.empty());
EXPECT_EQ(15,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());
for(int i(0); i < v.size() ; i++){
   // this loop shouldn't execute the loop body
   // if it does, we'll send of an explicit test fail
   FAIL();
}

vector<double> w(15,0.0);
EXPECT_FALSE(w.empty());
EXPECT_EQ(15,w.size());

for(int i(0); i<v.size() ; i ++){
```

```
EXPECT_FLOAT_EQ(0.0,w[i]);
EXPECT_FLOAT_EQ(0.0,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(15,2);

vector<int> w(15,4-2);

EXPECT_EQ(v,w);
```

Mutators

The vector selectors also provide us with a references to vector locations. When the selector is used in *l-value* position[4], then we can use it to mutate individual elements of the vector.

[4] 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);
```

Vector Initializers

It's often the case that you want to fill the vector with a set of initial values that consists of more than one value. In the newer C++11 standard we can use array initializer lists as follows:

```
vector<int> v{0,1,2,3,4,5};

vector<int> w = {0,1,2,3,4,5};

EXPECT_EQ(v,w);

for(int i(0); i < 6; i++){
   EXPECT_EQ(i,v[i]);
   EXPECT_EQ(i,w[i]);
}
```

First things first, the expression $\{0,1,2,3,4,5\}$ is not a *vector<int>* literal. It's just convenient syntax for specifying the sequence of values needed in this case. Next, for this syntax to work with vectors, we need to compile with the option *-std=c++11*.

If we can't use C++11 or the initial values aren't something we can type out by hand, then we need to write an initializer procedures. This is simply a vector mutator used to initialize the vector. If we're serious about using it for initialization only, then we might document that as a precondition. Here we see an initializer that sets a vector of doubles to random values from $[0,1]$.

```
/**
 * setToRand initializes a vector to random values
 *  from [0,1].
 * @param vRef is a reference to the vector
 * @return none
 * @pre PRNG has been seeded. Vector at vRef is uninitialized.
 * @post Vector at vRef contains values in [0,1]
 */
void setToRand(vector<double> &vRef);


TEST(setToRand,all){

   vector<double> w;

   EXPECT_TRUE(w.empty());
   setToRand(w);
   EXPECT_TRUE(w.empty());

   vector<double> v(10);
   for(int c(0); c < 100; c++){
      EXPECT_EQ(10,v.size());
      setToRand(v);
      EXPECT_EQ(10,v.size());

      for(int i(0); i<v.size(); i++){
        EXPECT_TRUE( v[i] >= 0 & v[i] <= 1 );
      }
   }
}

void setToRand(vector<double> &vRef){

for(int i(0); i < vRef.size(); i++){
   vRef[i] = double(rand())/double(RAND_MAX);
}
return;
}
```

There's nothing stopping us from using *setToRand* for general purpose mutation, but our initial intent was to initialize so we call it an *initializer*. It's also hard to imagine how completely replacing one set of data with another unrelated set of data would be anything other than initializing. So, perhaps the true sign of an initializer is setting the contents of a vector to values that are independent of its current contents.

Vectors and Iteration

Vectors are well suited to iteration because we can easily traverse over an index range with a counted loop. If you want to step through all the elements in the index range $[f, l)$, then we can use the following loop:

```
for(int i(f); i < l ; i++){
  //iterative update
}
```

We can also easily work our iteration from l down to f instead.

```
for(int i(l-1); i >= f; i--){
  // iterative update
}
```

In the case where f is zero and l is the vector size, then we get a loop to count through all the index values for our vector.

As we've already seen, solving a problem iteratively means thinking about the accumulation of a solution as you go. This, in turn, means we need to think about the following pieces of the accumulation process:

1. What am I accumulating, and what should the initial value of the accumulation be?

2. Given the accumulation of $i - 1$ things, what must I do to accumulate the next, ith, thing to have the proper accumulation of i things?

Once we've identified these things, we must ensure they work along side our intended traversal pattern. If they do not, then we either rethink our accumulation logic or our traversal logic.

Summing a vector of ints

Let's look at an example. We want to sum all the integers in a vector of integers.

```
/**
 * 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( std::vector<int> v );


TEST(sum, all){
   using namespace std;

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

   vector<int> notMT(5,3);
   EXPECT_EQ(15,sum(notMT))

   for(int i(0) ; i < notMT.size() ; i++){
     notMT[i] = i;
   }
   EXPECT_EQ(10,sum(notMT));
}
```

Let's start with the standard left-to-right, 0 to *size*-1, traversal pattern and see if we can work from there. We're clearly accumulating an *int* value, so we'll expect to use an *int* state variable as a place to accumulate. Now on to the iteration conditions. It's sometimes easier to think about the accumulation process and then the initial condition, but order doesn't matter as long as the two work together in the end. So, let's start with accumulation and look to our final test case as an example. That vector contains $\{0, 1, 2, 3, 4\}$. Let's pick $i = 3$ so that our accumulated sum is 3[5]. What do we need to do to properly incorporate the 3 at i with our partial sum 3. That's easy enough, just add two together and assign the result to the accumulator. Now, let's generalize our example in terms of variables. Let *acc* be our accumulator variable and v our vector. Then the update operation for *acc* is just:

[5] the sum of everything in $[0, i)$

```
acc += v[i]; //same as acc = acc + v[i]
```

Now for the initial value of *acc*. The question to ask yourself is, "What must *acc* be such that the very first update produces the right partial result?" In this case, we plan to start with $v[0]$. The correct partial solution after we've updated with $v[0]$ would be $acc == v[0]$. So, what number makes $v[0] == acc + v[0]$? The answer is, of course, 0 and we have our initial value. The last thing to check is that the combination of traversal pattern and accumulation logic produce the complete desired result. Working through the vector from $[0, v.size())$ in any order will cause us to add all the values in v to *acc* which will result in the complete sum. The rest is just writing this down in C++.

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

  // initialize accumulator(s)
  int acc(0);

  //traverse [0,v.size() ) in least to greatest order
  for(int i(0); i < v.size() ; i++){
     // update the accumulator
     acc += v[i];
  }
  //return the completed accumulation
  return acc;
}
```

Vectors and Recursion

At first glance, vectors are not well setup for recursion ; they lack a method that lets us easily select the "rest". The trick is to ignore the vector itself and instead focus on the set of index values for the vector. This sequence is easily represented by integer values and is easily broken down recursively. The end result is that we'll recurse on the index sequence and while were doing that, work with the vector[6]. Before we dig in to the vector recursion, let's look at the kind of choices recursion gives us.

[6] This is no different than the loop based iteration we just did. We used a counting loop to step through the index sequence and visited vector locations while we were doing that

Recursive Sequences

For a vector with size n, the sequence of index values available is the interval $[0, n)$. We can view this sequence as a recursive structure in many ways. In the classic, first and rest decomposition we have the number 0 and the interval $[1, n)$. If you follow this to the end, we'd find the empty interval $[n, n)$. As an example, consider the the case where $n = 4$.

$$\begin{aligned} [0,4) &= 0,[1,4) \\ &= 0,1,[2,4) \\ &= 0,1,2,[3,4) \\ &= 0,1,2,3,[4,4) \\ &= 0,1,2,3 \end{aligned}$$

Notice this fits with our experience with lists in Racket. We've just substituted sequences of numbers for list of other things.

First and rest, is not the only way to do recursive decomposition. The only real requirements for recursive decomposition are the following:

1. You have one or more non-recursive BASE CASE.
2. You have at least one recursive case that decomposes the structure into one or more smaller structure of the same type.
3. Decomposition via the recursive cases leads to the bases cases.

This allows for things like last and all-but-the-last,

$$\begin{aligned} [0,4) &= [0,3),3 \\ &= [0,2),2,3 \\ &= [0,1),1,2,3 \\ &= [0,0),1,2,3 \\ &= 0,1,2,3 \end{aligned}$$

It also let's us consider non-empty base cases like the singleton[7] base

[7] just one item

case $[a, a+1) = a$. Here we do first and rest with a singleton sequence as a base case. The difference with first to last and an empty base case is subtle but important.

$$\begin{aligned}[0,4) &= 0,[1,4)\\ &= 0,1,[2,4)\\ &= 0,1,2,[3,4)\\ &= 0,1,2,3,4\end{aligned}$$

Another interesting option is left-half, right-half. Here we use a singleton base case and look at $n = 5$.

$$\begin{aligned}[0,5) &= [0,2),[2,5)\\ &= [0,1),[1,2),[2,3),[3,5)\\ &= 0,1,2,[3,4),[4,5)\\ &= 0,1,2,3,4\end{aligned}$$

The take away here is that recursive decomposition is pretty darn flexible once you know the rules.

Summing recursively

Now that we know our index sequences can be recursively broken down in many different ways. We can consider how to build this in to recursive procedures for vectors. The key problem here will be that while the vector's size implies the initial sequence, $[0, n)$, the vector itself does not carry with it enough information to allow recursion over a subset of the vector. Put another way, if the vector isn't changing from one recursive procedure to the next, then we need new information , information that we change, to drive the recursion towards the base case. This means a new variable or two, to keep track of the sequence, and a new signature that includes the new variable.

Let's go back to the *sum* problem. The problem hasn't changed so our procedure signature shouldn't change either. New implementations should not effect the way we view a problem. So, we still need this procedure:

```
/**
 * 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( std::vector<int> v );
```

```
TEST(sum, all){
   using namespace std;

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

   vector<int> notMT(5,3);
   EXPECT_EQ(15,sum(notMT))

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

   EXPECT_EQ(10,sum(notMT));
}
```

In what follows we'll use namespaces to differentiate different recursive implementations. The easiest recursion to manage is still probably first-rest recursion. This is due to the fact that the size of the vector is always the upper boundary of the sequence and all we need to track is the current index of the first. Thus we arrive at the following procedure. Notice the very different purpose and conditions we place on this procedure.

```
/**
 * sumHelp sums the contents of the vector argument v found in
 *   the index range [first,v.size() )
 * @param v is the vector of integers
 * @param first is the index of the first item to be summed
 * @return is the sum of everything in the [first,v.size() ) range of v
 * @pre  0 <= first <= v.size()
 * @post none
 */
namespace fstRst{
 int sumHelp(vector<int> v, int first);
}

TEST(sumHelp,fstRst){

   vector<int> v(5,0);
   for(int i(0); i<v.size() ; i++){
      v[i] = i;
   }
   // v now contains {0,1,2,3,4}
   EXPECT_EQ(0,fstRst::sumHelp(v,5));
   EXPECT_EQ(4,fstRst::sumHelp(v,4));
   EXPECT_EQ(10,fstRst::sumHelp(v,0));

}
```

Now we implement *sumHelp*.

```
int fstRst::sumHelp(vector<int> v, int first){

  //empty
  if( first == v.size() ){
     return 0;
  }
  else{ // first < v.size() and it's not empty
     return v[first] + fstRst::sumHelp(v,first+1);
  }
}
```

Finally, we can go back to the *sum* procedure and make the appropriate initial call to *sumHelp*.

```
int sum(vector<int> v){
   return fstRst::sumHelp(v,0);
}
```

It turns out last-butLast Recursion is still pretty straight forward because the lower boundary is always 0. We'll put all the pieces together at once now.

```
/**
 * sumHelp sums the contents of the vector argument v found in
 *   the index range [0, last)
 * @param v is the vector of integers
 * @param last is the index of the exclusive upper bound of the region
 *    to be summed
 * @return is the sum of everything in the [0,last ) range of v
 * @pre  0 <= last <= v.size()
 * @post none
 */
namespace lstBLst{
 int sumHelp(vector<int> v, int last);
}

TEST(sumHelp,lstBLst){
   vector<int> v(5,0);
   for(int i(0); i<v.size() ; i++){
      v[i] = i;
   }
   // v now contains {0,1,2,3,4}

   EXPECT_EQ(0,lstBLst::sumHelp(v,0));
   EXPECT_EQ(0,lstBLst::sumHelp(v,1));
   EXPECT_EQ(1,lstBLst::sumHelp(v,2));
   EXPECT_EQ(10,fstRst::sumHelp(v,v.size()));

}

int lstBLst::sumHelp(vector<int> v, int last){

  //empty
  if( last == 0 ){
     return 0;
  }
  else{ // last > 0 and it's not empty
     return v[last] + lstBLst::sumHelp(v,last-1);
  }
}

int sum(vector<int> v){
   return lstBLst::sumHelp(v,v.size());
}
```

The most flexible versions takes both first and last as inputs. This signature allows for any type of recursion. Here we'll use it for first-rest again.

```
/**
 * sumHelp sums the contents of the vector argument v found in
 *   the index range [first, last)
 * @param v is the vector of integers
 * @param first is the index of the first number to be included in the
 *    sum
 * @param last is the index of the exclusive upper bound of the region
 *    to be summed
 * @return the sum of everything in the [first,last ) range of v
 * @pre  0 <= first<=last <= v.size()
 * @post none
 */
namespace genFstRst{
 int sumHelp(vector<int> v, int first, int last);
}

TEST(sumHelp,genFstRst){

   vector<int> v(5,0);
   for(int i(0); i<v.size() ; i++){
      v[i] = i;
   }
   // v now contains {0,1,2,3,4}

   EXPECT_EQ(0,genFstRst::sumHelp(v,0,0));
   EXPECT_EQ(0,genFstRst::sumHelp(v,0,1));
   EXPECT_EQ(3,genFstRst::sumHelp(v,0,3));
   EXPECT_EQ(10,genFstRst::sumHelp(v,0,v.size()));
}

int genFstRst::sumHelp(vector<int> v, int first, int last){

  //empty
  if( last == first ){
     return 0;
  }
  else{ // last > 0 and it's not empty
     return v[first] + genFstRst::sumHelp(v,first+1,last);
  }
}

int sum(vector<int> v){
   return genFstRst::sumHelp(v,0,v.size());
}
```

Finally, let's notice a curiosity. What happens if we re-write *fstRst::sumHelp* and *lstBLst::sumHelp* as follows?

```
int fstRst::sumHelp(vector<int> v, int first){

  //empty
  if( first == v.size() ){
     return 0;
  }
  else{ // first < v.size() and it's not empty
     return fstRst::sumHelp(v,first+1) + v[first];
  }
}
```

```
int lstBLst::sumHelp(vector<int> v, int last){

  //empty
  if( last == 0 ){
     return 0;
  }
  else{ // last > 0 and it’s not empty
     return lstBLst::sumHelp(v,last-1) + v[last];
  }
}
```

The specifications for the procedures doesn’t change, but the actual manner in which they work does. The *fstRst* version will now actually sum from last to first where the *lstBLst* version sums from first to last. To see this you might need to step through the computation.

Pass by const reference

All of these recursive versions of *sumHelp* have a real problem. Let’s say that the vector contains $1,000,000$ integers. When we pass this vector by reference, the computer takes us seriously and actually makes a copy of the vector *for each recursive call*. Sum is a function and should not mutate the vector, so pass-by-value was the right call, but this copy costs is a non-trivial overhead for our function. The way to avoid the copy cost is, as we know, to pass by reference and let the recursive calls share access to one vector. The problem is that this opens the door for inadvertent mutation of the vector. However, we really don’t want mutation to occur so we need to tell the compiler that the data found at the reference should not be mutated. What we need is an immutable reference. This is known as a const-reference. Let’s revisit the second version, *lstBLst::sumHelp*

```
/**
 * sumHelp sums the contents of the vector argument v found in
 *   the index range [0, last)
 * @param v is an immutable reference to a vector of integers
 * @param last is the index of the exclusive upper bound of the region
 *    to be summed
 * @return is the sum of everything in the [0,last ) range of v
 * @pre  0 <= last <= v.size()
 * @post none
 */
namespace lstBLst{
 int sumHelp(const vector<int> &v, int last);
}

TEST(sumHelp,lstBLst){

   vector<int> v(5,0);
   for(int i(0); i<v.size() ; i++){
      v[i] = i;
   }
   // v now contains {0,1,2,3,4}

   EXPECT_EQ(0,lstBLst::sumHelp(v,0));
   EXPECT_EQ(0,lstBLst::sumHelp(v,1));
   EXPECT_EQ(1,lstBLst::sumHelp(v,2));
   EXPECT_EQ(10,fstRst::sumHelp(v,v.size()));

}

int lstBLst::sumHelp(const vector<int> &v, int last){

  //empty
  if( last == 0 ){
     return 0;
  }
  else{ // last > 0 and it's not empty
     return v[last] + lstBLst::sumHelp(v,last-1);
  }
}

int sum(vector<int> v){
   return lstBLst::sumHelp(v,v.size());
}
```

Notice how little changed when moving from pass-by-value to pass-by-const-ref.

Wrap-up

We just saw many different ways of summing the contents of a vector. Some were iterative and required loops. Others were recursive and made use of helper functions. The question we now face is simple, "which is the best?" To answer this we must evaluate them all on the standard criteria of correctness, simplicity, and efficiency. As scientists, our evaluations should be based off of objective measures of correctness, simplicity, and efficiency. That is to say, the scale on which we measure correctness should not be based on our own experience and perceptions but on well understood accepted measures. So, go back. Review this code. Run this code. Trace through this code by hand and be certain that you understand how it works because you cannot expect to evaluate “how well” until you understand “how”.