COMP 161 - Lecture Notes - 17 - Iteration ... again

Spring 2014

In these notes we take a more general look at using iteration to solve problems involving a vector of data. The key is to understand that iteration is means to solving programming problems, not a a by-product of solved problems.

Generalizing Functional Vector Iteration

So you need to solve some problem that involves a set of data stored in a vector. Iteration might just be the tool for you. We know that stepping through the contents of a vector is easy with a counted loop, but the interesting part is what to do while I'm stepping through the vector. Basic iteration patterns work by building up, or accumulating, a partial solution as you go so that when you're done, you have a complete solution.

Let's start to generalize what this looks like for functional procedures, procedures that take and return values without producing side-effects. We have a vector full of data of type S, i.e. a *vector<S>* and we've determined that from that vector we need to compute some value of type T. In the case of the *sum* problem from our previous notes, S and T are both *int*. If I'm finding the location of something, then S can be just about anything but T is an *int* or *unsigned int* because locations in vectors are integer index values. If on the other hand I'm taking a *vector<double>* and finding all the doubles from the interval $(0, 1)$, then S is clearly *double* and upon examination we find that T must be *vector<double>*. The point here is that S and T can be just about anything, and the following signature captures a very large set of functional procedures we might encounter.

```
T mystery(vector<S> v, ...)
```

With the ... we leave open the possibility of other arguments with the understanding that the key processing will take place with respect to the *vector<S>* named v.

Now, how do we design an iterative process?. We'll work under the assumption that we plan to accumulate from left to right, possibly with the all encompassing 0 to $v.size() - 1$ loop. First, let's recognize that because our desired solution of type T, then our iterative design needs to accumulate a value of type T as well. To represent this we'll assume a variable named *accumu*, short for accumulator, that is of type T. With the variable *accum* fixed in our mind, we use a bit of INDUCTION. Assume we've iterated up to some index i and that i is neither the place we started nor the place we'll end. This means imagining ourselves in the middle of the process. What do we know? Well, under the inductive assumption that iteration works, we assume that *accum* must contain a partial solution to our problem that involves only the data up to but not including the value at location *i*. This is important moment so let's look at a few examples.

If we're summing up the integers $\{1, 2, 3, 4, 5\}$ and $i = 3$, then we assume that *accum*$= 1 + 2 + 3 = 6$, or the sum of all the stuff before *i*. Part of a sum is a sum. If $i = 4$ and we're finding all the doubles in $\{4.3, 0.2, 0.95, 1.0, 3.5, 235.0\}$ that are in the interval $(0, 1)$, then *accum* is the vector containing $\{0.2, 0.95\}$. Part of a *vector<double>* is a *vector<double>*. If $i = 2$ and we're finding the location of the first occurrence of 2 in the vector<double> containing $\{1, 3, 5, 6, 2, 0\}$, then $accum = -1$ because we haven't found 2 yet and that's the standard result of "not there". In all these cases we focused on what should have accumulated if our iterative procedure worked and *not how* such accumulation might have occurred. Figuring that out is the problem we're trying to solve!

Now, back to the iterative process. We've accumulated a partial solution involving all the data up to but excluding what's at $v[i]$, and we've stored that partial solution in the variable *accum*. Our task now is to determine *how to accumulate the data at* $v[i]$ *into accum*. Put another way, we need to identify some operation, function, or procedure that takes a value of type S, the one that's found at $v[i]$, and combines it with the value of type T, the one accumulated in the variable *accum*, and produces a new value of type T, that we can then write to *accum*. If we call this operation *accumulateNext*, then we can express the signature of *accumulateNext* as a mathematical function.

```
accumulateNext : T S -> T
    OR
accumulateNext : S T -> T
```

If we call the current value of *accum* $accum_{i-1}$ and let $accum_i$ be the value it should have after accumulating the value at $v[i]$, then we can express the desired behavior of *accumulateNext* mathematically as well.

$$accum_i \equiv accumulateNext(accum_{i-1}, v[i])$$

We've only written this functionally because we're using the language of mathematics to capture the logic of iteration. If we use C++ functional procedures we might have the following procedure declaration and usage:

```
// declaration
T accumulateNext(T ac, S val);

//usage
accum = accumulateNext(accum,v[i]);
```

We don't have to use a function though. We could write *accumulateNext* as a mutator procedure. In which case the C++ you'd write would be something like this:

```
//declaration
void accumulateNext(T &ac,S val);

//usage
accumulateNext(accum,v[i]);
```

Sometimes we really need a helper procedure for our iteration, but sometimes a few built-in operators will work just as well. In the case of *sum*, we didn't need a function or procedure at all. The operation we need for *accumulateNext* already existed in C++ as a built in operator.

```
sum = sum + v[i];
   OR
sum += v[i];
```

Ok. Let's recap what we have so far. It's important and worth repeating. We start by assuming we've accumulated the part of our desired solution involving everything up to and including the data at location $i - 1$ and that we've stored that partial solution in the variable *accum*. We then determine what computation is needed to combine the value at location i with the value in *accum* in order to produce the partial solution that accounts for everything up to and including the value at location i. Abstractly, we'll call this operation *accumulateNext*. Mathematically, we can think of it as a function mapping a T vale and an S value to another T value such that the resultant T value is the right partial solution to our problem given the contents of our vector found in $[0, i]$. What we've figured out is the body of our loop. We've figured out the core kernel of computation that we need the loop to repeat. Now we turn our attention to the loop and ensure that it repeats this process on all the values we need it to accumulate the complete, final solution.

Based on the action of *accumulateNext*, we must determine where to start and stop i as well as how to update or increment i. In many cases our intent is to apply *accumulateNext* at every location in the vector. If that's the case then starting i at 0, incrementing by 1, and continuing as long as $i < v.size()$ is what we want. Be careful though. If *acccumulateNext* makes uses of the value at $i - 1$, then starting at 0 will cause the program to access outside the bounds of the vector and crash. Similarly, continuing while $i < v.size()$ but accessing $i + 1$ when we accumulate will cause the final iteration to access outside the bounds and crash. Anytime your *accumulateNext* computation is accessing somewhere other than i you need to check and double check the traversal pattern carried out by your loop, especially at the beginning and end. Determine which locations you need to visit given the problem and the behavior of *accumulateNext*, ensure that those and only those locations are visited, and verify that any other vector accesses needed are within the bounds of the vector.

One critical step remains. We must determine the initial value of *accum*. Ask, "what might we return if the vector is empty?" and "What would we accumulate with the very first value to get the first partial result?". The initial value is the answer you should arrive at for both of these. When summing, 0 is an appropriate sum of an empty vector and $v[0] + 0 = v[0]$ so it gives us the sum of the value at location 0. If we're trying to *find* something then -1 is what I'd return if the vector is empty. We'd also want either -1 or 0 in the accumulator after looking at $v[0]$ so starting with -1 certainly seems reasonable. Occasionally, it's helpful to initialize to the first vector value or some function of that value. In that case, you might need to reconsider the counting pattern because you've effectively kick-started the iterative process.

We've laid down out a general iterative design process. Let's set down the steps:

1. Assume that you have a partial solution for all the data in the location range $[0, i)$ and that the value is stored in a variable *accum*. We imagine that i is in the middle of the vector, not one of the ends.

2. Determine the operation of *accumluateNext* such that *acumulateNext(accum,v[i])* gives you the partial solution for all the values found at locations $[0, i]$.

3. Ensure that our loop starts at the right place, quits at the right place, steps through the necessary locations, and that at no point do we access outside of the vector bounds.

4. Determine the appropriate initial value for *accum*

Our design process works very well if you give yourself a concrete example to work with. With a specific vector in mind you can see an exact index interval, you can pick a specific i and then determine the *accum* value for $i - 1$. From there you can more concretely think about what *accumulateNext* needs to do given a specific $v[i]$. You can then repeat for different i values and see if your logic is sound or if any conditions pop up. So, once again, having test cases prior to attempting to code will pay off when designing an unknown iterative function.

Let's wrap this section up with a couple of templates. First, a very generalized template for iteration based on the following key design elements: What's important are the different components.

- *accum* the accumulator variable
- *init-val* the initial accumulator value
- *first* the initial vector index for our target range $[first, last)$
- *last* the last index for our target range
- *next* the update for the index counter *i*. Often $++$ but can be other things
- *accumulateNext* the accumulator update operation

```
T mystery(vector<S> v, ...){

  T accum(init-val);
  for(int i(first) ; i < last ; i = next(i)){
     accum = accumulateNext(accum,v[i]);
  }
  return accum;

}
```

This template uses functional procedures, but we could also use mutators for things like *next* and *accumulateNext*.

Let's do one more template. This one is for basic iteration where our loop counts over $[0, v.size())$ in increments of one. This means *first* is 0, *last* is *v.size()*, and *next* is ++ We've seen this variation of iteration several times now and can expect to get a lot of mileage out of it.

```
T mystery(vector<S> v, ...){

  T accum(init-val);
  for(int i(0) ; i < v.size() ; i++){
     accum = accumulateNext(accum,v[i]);
  }
  return accum;

}
```

Iterative Predicates and Short-Circuit iteration

Iterative predicates accumulate boolean values. The nature of *boolean arithmetic* often allows us to stop the accumulation of boolean values early. Let's adapt our template for predicate procedures:

```
bool mystery(vector<S> v, ...){

  bool accum(init-val);
  for(int i(first) ; i < last ; i = next(i)){
     accum = accumulateNext(accum,v[i]);
  }
  return accum;

}
```

Now what if *accumulateNext* just evaluates some property of $v[i]$ using the predicate f and then combines that result with *accum* using the and (&&) operator? Then we could adapt the template as follows:

```
bool mystery(vector<S> v, ...){

  bool accum(true);
  for(int i(first) ; i < last ; i = next(i)){
     accum = accum && f(v[i]);
  }
  return accum;

}
```

First we notice that the initial value for *accum* should be true because *true* && b is b no matter what the value of b. Conversely, if $f(v[i])$ is *false*, then it doesn't matter what *accum* is and what else is in the vector because and'ing with *b && false* is false regardless of the value of b. We can now rewrite our template for this kind of predicate. This new version of the template is optimized to stop the iteration as soon as a false value is encountered.

```
bool mystery(vector<S> v, ...){

  bool accum(true);
  for(int i(first) ; i < last ; i = next(i)){
     if(f(v[i]){
        return false;
     }
     else{
       accum = accum && true;
     }
  }
  return accum;

}
```

In fact, we don't even need *accum* and the explicit accumulation because the accumulation is only accumulating *true*s.

```
bool mystery(vector<S> v, ...){

  for(int i(first) ; i < last ; i = next(i)){
     if(!f(v[i]){
        return false;
     }
  }
  return true;

}
```

The same OPTIMIZATION works if *accumulateNext* is *accum = accum || f(v[i])* except this time we're looking for $f(v[i])$ to be *true* and initializing with $false$.[1]

[1] of course we don't really need to initialize

```
bool mystery(vector<S> v, ...){

  for(int i(first) ; i < last ; i = next(i)){
     if(f(v[i]){
        return true;
     }
  }
  return false;
}
```

This type of OPTIMIZATION process works in many iterative situations. For example, the traditional *find* operation[2] can stop and return a location as soon as you find the first occurrence of the key value. This optimization is called *short-circuiting* because it stops the accumulation process early and in doing so has the potential to save some compute time. We should also note that in the case of predicate procedures, we're also able to get rid of the accumulator variable, which means this optimization saves memory as well as time.

[2] return the location of the first occurrence of a key or return -1

Generalizing Effectful Iteration

Iteration for effect is not altogether different from functional iteration. Rather than accumulate values we accumulate effects. With mutation, we might be mutating the vector on which we're iterating or perhaps using the vector as the basis for the mutation of another variable. In the case of I/O effects, we might be carrying out a series of I/O operations in order to reach some final goal. In both cases, we're incrementally carrying out some part of a total effect and in that way we're still accumulating.

Let's start by re-evaluating our design process in terms of the accumulation of effects not values.

1. Assume that you have carried out a partial effect for all the data in the location range $[0, i)$. We imagine that i is in the middle of the vector, not on the ends.
2. Determine the operation of *affectNext*[3] such that the effect carried out by *affectNext(v[i])* combines with the partial effect already carried out to produce the partial effect desired for all the values found at locations $[0, i]$.
3. Ensure that our loop starts at the right place, quits at the right place, steps through the necessary locations, and that at no point do we access outside of the vector bounds.

[3] new mindset, new name

This leads to the following template:[4]

[4] if *affectNext* carries out mutation, then you need to make v a reference parameter

```
void mystery(vector<S> v, ...){

  for(int i(first) ; i < last ; i = next(i)){
     affectNext(v[i]);
  }
  return;

}
```

Once again, it's nice to see a more concrete variation of the template for the standard one-by-one, left-to-right iteration with which we're most familiar.

```
void mystery(vector<S> v, ...){

  for(int i(0) ; i < last ; i++ ){
     affectNext(v[i]);
  }
  return;

}
```

Printing procedures are classic examples of this design. In that case *affectNext* is simply the streaming output operator $<<$ with respect to an output stream. Similarly, we could initialize a vector from a file using this pattern letting *affectNext* be the streaming input $>>$ with respect to the file stream. Initializers and mutators also fit this pattern, here *affectNext* might really be something like v[i] = f(v[i]) where f is a function giving us new values for the vector relative to the old values.

Iterative Design Recap

At the heart of this design process is an INDUCTIVE HYPOTHESIS. It's that moment where we ask, "If my process makes it through the first i items, then what should I do with item $(i+1)$ to keep going?" The leap that we can make all the way up to i seems like a cheat but it's not. Induction is a tried a true practice in mathematics and it's a problem solving technique that often pays off in programming. What is, perhaps, more important is that by focusing our mental energies on the induction, we tend to avoid over thinking the whole process and worrying about the sum total of the work when we should be focusing on the kernel of code we need to repeat.

All the iterative design process really requires of us is *a clear sense of what we're trying to accomplish*. This is wildly different that how we plan to accomplish it. If we knew how to solve the problem involving a vector, then would we be writing code for it or would we be calling on the library code that carries out the solution?

So, identify and document the problem. Come up with some concrete examples of what should happen when your code solves the problem. Set those examples in the iterative design process and explore the question, "I wonder if iteration will work on this problem?"