COMP 161 - Lecture Notes - 12 - Iterative and Recursive Procedures for Strings

Spring 2014

In these notes we look at developing iterative and recursive procedures for Strings. These principles generalize for array structures and recursively structured data.

Note: From char to std::string

In what follows, we'll regularly need to turn a *char* value into a *string* value. To do this we'll use a special string constructor designed to fill a string with a single character. Here's a few tests to illustrate how it works[1].

[1] see the "fill" constructor here `http://www.cplusplus.com/reference/string/string/string/`

```
TEST(charToString,all){

  EXPECT_EQ("a",string(1,'a'));
  EXPECT_EQ("aa",string(2,'a'));
  EXPECT_EQ("aaaaa",string(5,'a'));

  std::string word("dog");
  EXPECT_EQ("---",string(word.length(),'-'));

}
```

The problem: strToUpper

The C char type library[2] contains a procedure for converting a lowercase alphabetic *char* to its uppercase counter part. What if we wanted to convert a string? First let's state this problem as a C++ procedure declaration.

[2] called **cctype**

```
/**
 * strToUpper converts all the letters of
 *  a string to uppercase
 * @param str the string
 * @return str in all uppercase
 * @pre str is composed of alphabetic characters only
 * @post none
 */
 std::string strToUpper(std::string str);
```

Now some gUnit tests.

```
TEST(strToUpper,all){

  // empty case
  EXPECT_EQ(string(""),strToUpper(string("")));

  // other cases
  EXPECT_EQ(string("A"),strToUpper(string("a")));

  EXPECT_EQ(string("DOG"),
            strToUpper(string("dog")));

  EXPECT_EQ(string("CAT"),
            strToUpper(string("Cat")));
}
```

The problem should now be clear. Let's consider two classic strategies for solving it: recursion and iteration.

Recursive Functions on Strings

Strings can be viewed as recursive structures. They are either empty, or if they're not, they are a single *char* value followed by another string value. We can select the first char value using *operator[]* or the class method *at*. The rest of the string can be retrieved using the *substr* method. We saw the basic recursion recipe in COMP160, but let's review it here.

Recursive structures invoke most of the basic structures found in data: atomic, compound, itemization, and self-reference. To work on a recursive structure we begin with the *itemization*. A string has a non-recursive case, the empty string, and a recursive case, non-empty strings. This means we need to use a conditional to identify the type VARIANT[3]. Let's stub out *strToUpper* to account for this part of the template. We'll use the empty string as a stub return value.

[3] template rule: use conditionals to identify variants of itemization data

```
std::string strToUpper(std::string str){
  if( str.isEmpty() ){
    // empty case
    return string("");
  }
  else{ //no empty
    // non-empty case;
    return string("");
  }
}
```

Now we proceed with the templates by case. When a string is

empty it is effectively *atomic data*, so we just need to figure out what computation should occur here[4]. We'll leave this case stubbed out for now and look at the non-empty case. The non-empty string is *compound data*; it has the first char and the rest of the string. So, we must select each part[5]. Because the rest of the string is the recursive part of the structure, we should recurse there[6]. Let's add some notes to the stub.

[4] template rule: compute with atomic data

[5] template rule: deconstruct compound data by field

[6] template rule: recurse on recursive data

```
std::string strToUpper(std::string str){
  if( str.isEmpty() ){
    // empty case
    return string("");
  }
  else{ //not empty
    // non-empty case;
    // str[0]
    // strToUpper(str.substr(1) )

    return string("");
  }
}
```

Now, it's time to be clever. First, let's look at the empty case. What string should result from turning all the characters in the empty string to upper case letters? Why the empty string of course. So, it turns out the stub value was the correct answer. Now on to the non-empty string case. If I give you first character and the rest of the string in all upper case letters, what do you need to do? First things first we should *toupper* the first character. Once that's done, we need to append the uppercase version of the rest to the uppercase first. Thankfully, the *append* string class method can take care of this for us. We can now finish our procedure.

```
std::string strToUpper(std::string str){
  if( str.isEmpty() ){
    // empty string case
    return string("");
  }
  else{
    // non-empty string case;
    return string(1,toupper(str[0])).append( strToUpper(str.substr(1)) );
    }
}
```

Alternatively, we can use some local variables to clear up the return statement a bit.

```
std::string strToUpper(std::string str){
  if( str.isEmpty() ){
    // empty string case
    return string("");
  }
  else{
    // non-empty string case;
    std::string fstUp(1,toupper(str[0]));
    std::string rstUp( strToUpper( str.substr(1) );

    return fstUp.append( rstUp );
  }
}
```

Recursion doesn't have to happen in first to rest order. If you're able to select the last and all but the last, then you can often work the structure the other direction. Consider this version of *strToUpper* that proceeds last to butLast.

```
std::string strToUpper(std::string str){
  if( str.isEmpty() ){
    // empty string case
    return string("");
  }
  else{
    // non-empty string case;

    std::string lastUp(1, toupper(str[str.length()-1]) );
    std::string butLastUp(strToUpper(str.substr(0,str.length()-2));

   return  butLastUp.append(lastUp);
  }
}
```

Let's stop and make sure we understand how to think about these kinds of recursive processes. Recursive processes like our recursive *strToUpper* work based on the principle of INDUCTION[7]. When approaching the procedure, we assumed that the recursive call would present us with all but one of the letters in uppercase and then specified the other computation that needed to be done given that data. We did not initially worry about how the recursion work, we simply proceed as if it would. In addition to dealing with the recursive, non-empty case, we specified a non-recursive base case for the empty string. It's the combination of these two things that allows the recursion to work. Let's see how. Our procedure clearly works for empty strings as we hard-coded the solution. It doesn't take too much to see

[7] http://en.wikipedia.org/wiki/Structural_induction

that it works for strings of length one. We know it works for empty strings and we assume the C++ library code we're using works. The correctness of our procedure on strings of length one is only dependent on that code. So, as long as we glued all of that together correctly, *strToUpper* will work for strings of length one. This same argument works for strings of length 2, then 3, then 4, and so on. We see that if *strToUpper* works on strings of length n, then it must work on strings of length $n+1$ for any $n \geq 0$. The standard metaphor used to explain induction is climbing a ladder. The base case is how to get up to the first rung of the ladder. The recursive case tells you how to get to the next rung. So, if you can get on the first rung and you can climb from one rung to the next, then you can climb as high as you'd like.

Iterative Functions on Strings

Iterative processes work by accumulating a solution as you go. This is subtly different than the recursive processes we just worked with. In recursion we work with the complete picture: the first character and the complete solution of the rest of the string. No part of the string data is unaccounted for. With iteration we have to consider a new picture of the problem. Let's look at our *strToUpper* procedure and generalize from there. The iterative picture of this procedure has three elements: the part of the upper case string we've accumulated so far, the current character we need to compute the uppercase version of and add to our accumulated answer, and the remainder of the string that needs converting. So, we've gone from a two part view of the problem to a three part view of the problem. Not a huge change really. The trick now is how to express the process by which we go character by character, accumulated the uppercase string as we go.

Loops are the standard method for achieving iteration[8]. For *strToUpper*, we'll use a loop to *iterate over the characters in our string*, i.e. to convert them to uppercase and accumulate them in a variable one at a time. Let's look at stub that covers just the general iterative process.

[8] recursive procedures can also carry out iterative processes

```
std::string strToUpper(std::string str){

  // initialize accumulator variable
  std::string accum("");

  // iterate over each character in str
  for(int i=0; i < str.length(); i++){
      //str[i]
      //accum
```

```
  }

  // return accumulated solution
  return accum;
}
```

This stub has the basic iterative stuff but not the *strToUpper* problem specific stuff. First, we see the variable *accum* with which we'll accumulate the solution. It starts as the empty string because that's the logical choice for, "no part of the string has been accumulated". The loop will count over the index values for *str* and accessing *str[i]* inside the loop will guarantee we get each character. The plan is to add a new uppercase letter to *accum* on each step of the iterative process, so we should have that variable in the loop body as well. If we get the accumulating setup right, then when this loop completes, we should have a complete solution. Thus, we return the value stored in *accum* as our result.

To finish *strToUpper* we simply need to get the uppercase value of *str[i]* and accumulate it. In this case, we'll want to *append*[9].

[9] the string class provides other ways of adding to our partial solution, see if you can find them

```
std::string strToUpper(std::string str){

  std::string accum("");

  for(int i=0; i < str.length(); i++){
     //use append as mutator
     accum.append(string(1,toupper(str[i])));
  }

  return accum;
}
```

We don't have to iterate through our string in first to last order. Just like with recursion, you usually have options and in this case we could solve the problem working from last to first instead. This time, we'll need to use *append* like a function, not a mutator. Do you see why?

```
std::string strToUpper(std::string str){

  std::string accum("");

  for(int i=str.length()-1; i >= 0; i--){
accum = string(1,toupper(str[i])).append(accum);
  }
```

```
  return accum;
}
```

Iteration and Recursion for Effect

So far we've only considered functional procedures. Let's rethink our problem not as a function but as a mutator.

```
/**
 * setStrToUpper modifies a string so that all the contained letters
 *  are now uppercase
 * @param strRef reference to the string
 * @return none
 * @pre strRef is composed of alphabetic characters only
 * @post string variable referenced by strRef has been modified
 */
 void setStrToUpper(std::string &strRef);
```

Now our gUnit tests.

```
TEST(setStrToUpper,all){
   string S("");

  // empty case
  S = "";
  EXPECT_EQ(string(""),S);
  setStrToUpper(S);
  EXPECT_EQ(string(""),S);

  // other cases
  S = "a";
  EXPECT_EQ(string("a"),S);
  setStrToUpper(S);
  EXPECT_EQ(string("A"),S);

  S = "dog";
  EXPECT_EQ(string("dog"),S);
  setStrToUpper(S);
  EXPECT_EQ(string("DOG"),S);

  S = "cAt";
  EXPECT_EQ(string("cAt"),S);
  setStrToUpper(S);
  EXPECT_EQ(string("CAT"),S);
```

```
}
```

Now, let's see how we can use recursive and iterative strategies to create the desired effect.

Iteration for effect

Loop based iteration goes well with effect-based procedures. Let's start with the basic stub.

```
void setStrToUpper(std::string &strRef){

  //iterate over each character
  for(int i=0; i < str.length(); i++){
//str[i]
  }
  return;
}
```

This time there's no accumulator variable because *strRef* itself is acting as the accumulator. We're effectively planning to accumulate the answer in-place[10]. So, the only thing we need to be prepared to do is iterate over the characters in the string; this is what we see in our stub. Now all that's left is mutate the character inside the loop. Notice this version doesn't require converting chars to strings because our mutation occurs with respect to char values and doesn't require string class methods.

[10] in the memory we already have allocated

```
void setStrToUpper(std::string &strRef){

  //iterate over each character
  for(int i=0; i < str.length(); i++){
// change current char to uppercase version
str[i] = toupper(str[i]);
  }
  return;
}
```

Once again, we could do this going from index *str.length()-1* down to 0 if we wanted to.

Recursion for Effect

Recursing for effect can be tricky when we don't have reference based accessors. When we select the rest of the string *str* we use *str.substr(1)*. This method returns a copy of the rest. If we then mutate that copy, we still haven't mutated the original. We'll have to pay

close attention to details in order to work around this. Let's start with the stub. In this stub we'll go ahead and save our copy of the rest to a local variable and make the mutation based recursive call on that variable.

```
void setStrToUpper(std::string &strRef){

  if( strRef.isEmpty() ){
    return;
  }
  else{
    // strRef[0]
  std::string rst( strRef.substr(1) );
    setStrToUpper(rst);
    // rst
    return;
  }
}
```

It's clear that the empty string case is, once again, complete. If the string is empty, there's nothing to change, so we're free to return. Now, let's stop and think about what's going on with the local variable *rst* and the recursive call *setStrToUpper(rst)*. Why did we non just pass *strRef.substr(1)* to the recursive procedure call? Well, *setStrToUpper* requires a string by reference, and that means we must use a string variable. So, we must first create that variable, give it the appropriate initial value, the "rest", and then use it to make the recursive call. None of this is commented out in the stub because there's nothing left to add to these statements. What is left in the procedure is figuring out what to do with the mutated *rst*. Thus, we leave a comment about *rst* to remind us to do something with it *after* we mutate it with recursion.

To finish *setStrToUpper* we need to change *strRef[o]* to upper case and then *replace* the rest of *strRef* with *rst*. Lucky for us, there's a *replace* method in the string class. If there weren't we'd need to write a helper.

```
void setStrToUpper(std::string &strRef){

  if( strRef.isEmpty() ){
    return;
  }
  else{
      // change the first to uppercase
    strRef[0] = toupper(strRef[0]);
```

```
    // get a copy of the rest
  std::string rst( strRef.substr(1) );
      // change the copy of the rest to all upper case
    setStrToUpper(rst);

    // replace strRef with modified copy
    strRef.replace(1,strRef.length()-1,rst);
    return;
  }
}
```

We can clean this code up a bit. The if case really does nothing. This revision gets rid of that case but keeps the same logic.

```
void setStrToUpper(std::string &strRef){

  if( !strRef.isEmpty() ){
    // change the first to uppercase
    strRef[0] = toupper(strRef[0]);
    // get a copy of the rest
  std::string rst( strRef.substr(1) );
      // change the copy of the rest to all upper case
    setStrToUpper(rst);

    // replace strRef with modified copy
    strRef.replace(1,strRef.length()-1,rst);
  }

  return;
}
```

If the string is empty, then it skips the whole conditional and goes right to the return. If it's not empty, then we'll carry out the mutation and then jump down to the return.

Process vs. Procedure

A procedure that calls itself is recursive. This does not mean that the process carried out by the procedure is inherently recursive. It is possible to write a recursive procedure that carries out an iterative process, or a process that accumulates as we go. Consider this two procedure version of *setStrToUpper*.

```
void setStrToUpper(std::string &strRef){
  iterative_helper(strRef,0);
  return;
```

```
}

void iterative_helper(std::string &strRef, int fst){

  if( fst >= strRef.length() ){
    return;
  }
  else{
      // change the first to uppercase
    strRef[fst] = toupper(strRef[fst]);
  iterative_helper(strRef,fst+1);
    return;
  }
}
```

The procedure *iterative_helper* effectively iterates over the indexes of *strRef* in the same manner as our counting loops. The call to *setStrToUpper* acts to initialize the counter, *fst*. The empty case now acts as the *stop-when* check, similar to a loop's continuation check. The non-empty case carries out the same mutation as the iterative loop version we wrote, increments *fst*, and repeats the process via recursion.

How about an iterative function using a recursive procedure?

```
std::string strToUpper(std::string str){
return iterative_helper(str,string(""));
}

std::string iterative_helper(std::string str, std::string accum){

   if(str.isEmpty()){
      return accum;
   }
   else{
 // locally modify accum
     accum.append(string(1,toupper(str[0])));

 // pass new local value to recursion
 return iterative_helper(str.substr(1),accum)

   }
}
```

In this case we used a little local mutation to make the code a little more clear. Because *accum* is pass by value, the mutation caused by *append* only changes the *accum* for the procedure in which it is called. When we recursively call *iterative_helper* the new value of

accum is selected and passed to the recursive call. We could have accomplished this in one overly long expression.

```
iterative_helper(str.substr(1), accum.append(string(1,toupper(str[0]))))
```

What we're seeing is that recursive procedures seem to be more expressive than loops in that you can write iterative code with recursive procedures, but you can't write recursive code with loops. Sure, loops seem to re-call their own bodies, but that's overtly recursive. What that being said, recursion and iteration are simply tools we use as programmer to express the logic of repetition. We use the tool the suits the job. Recursive procedures might be more fundamental to the process of computation, but that's a distinction thats important to our understanding of computation and not necessarily to our desire to write correct, simple and efficient programs. So, choose the right too for the job and we'll save the recursion/iteration comparison for when we're more well equipped to qualify and quantify "better".