COMP 161 - Lecture Notes - 11 - Iterative and Recursive Procedures for Strings

Spring 2016

In these notes we look at developing iterative and recursive implementations of procedures for Strings. These problem solving and program design principles generalize to sequential structures like lists, vectors, and arrays.

The problem: a toupper for std::string objects

The C char type library[1] contains a procedure for converting a lowercase alphabetic *char* to its uppercase counter part. Let's first imagine this problem in two ways and capture our thinking as procedure declarations in C++.

[1] called **cctype**

As a functional problem we'd imagine taking a string object, computing the uppercase version of that string as an entirely new object, and returning that new object. In doing this we can focus our logic purely around string objects as values[2].

[2] r-values really

```
/**
 * strToUpper compute the uppercase version of std::string str
 * @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 we come up with tests both as a way to explore concrete examples of the problem and for testing out implementation when we finally complete it.

```
TEST(strToUpper,all){

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

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

  EXPECT_EQ(std::string("DOG"),
            strToUpper(std::string("dog")));
```

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

Now let's re-imagine this problem as one of mutation. Our goal now is to physically change one particular string object. This constitutes a shift in thinking from r-values to l-values. Notice how the name and purpose of the procedure reflect our new perspective on the problem.

```
/**
 * setStrToUpper changes all the letters of std::string str
 *  to their uppercase counterparts
 * @param str the std::string object getting modified
 * @return none
 * @pre str is composed of alphabetic characters only
 * @post str is now the same letters, in the same order, but uppercase
 */
void setStrToUpper(std::string& str);
```

Once again, the new way of thinking requires a new way of testing.

```
TEST(setStrToUpper,all){

  std::string s{""};

  setStrToUpper(s);
  EXPECT_EQ(std::string(""),s);

  s = std::string("to");
  setStrToUpper(s);
  EXPECT_EQ(std::string("TO"),s);

  s = std::string("DO");
  setStrToUpper(s);
  EXPECT_EQ(std::string("DO"),s);

  s = std::string("hEllO");
  setStrToUpper(s);
  EXPECT_EQ(std::string("HELLO"),s);

}
```

So, how do we implement these procedures? Logically, we know

we could just apply the *cctype* function *toupper* to each character in the string. Doing this first means working with std::string data as a *sequence of char data*. Thus far we've mostly treated it as something that's more struct-like. Once we know how to work with the string as a sequence of data, then we need to know how to setup code repetition for that sequence. The two main[3] tools we have in programming to capture code repetition is a RECURSIVE PROCEDURE[4]. and *Iteration*. A recursive procedure is a procedure that calls itself and is by its very nature repetitive. You used these exclusively in Racket. We'll review some basic strategies and talk about using them in C++. We'll also look at recursion for effect when we implement the mutator setStrToUpper.

[3] only?

[4] sometimes we just say *recursion*

Iteration is a new strategy for you. When we iterate we typically invoke the notion of time and the accumulation of state. The idea is that as our code repeats it incrementally accumulates the solution in such a way that the accumulated solution at the time of termination is the final solution. Iteration works really well in imperative environments where time and state are readily available. It can also be managed via recursive procedures; we'll explore this a bit but not spend much time with it until we really need it.

The thing to keep in mind before we get into the details of recursion and iteration it's important to realize that these things are techniques for solving problems that involve repetition. They're generic strategies that often lead to solutions to problems. We began by imagining our problem as being a functional problem or a mutation problem. We're not shifting our thinking towards the solution, the implementation. This mental separation of problem (declaration) and solution (implementation) is vital to long term success in computing. Poorly specified problems are difficult or impossible to solve and the ability to try different implementations and compare these options leads to the real science of computing.

Strings as Recursive Structures

Recursion is not just a way to solve a problem. We can apply recursive thinking in order to understand the structures inherent in our data. You saw this with lists in Racket. Once you've identified recursive structure in you data, you can often use that a guide in writing a recursive procedure to solve a problem involving that data. We'll work through this as we solve our std::string toupper problem. You should also reflect back on all the recursive Racket code you saw and see how the same logic plays out in that environment.

Racket lists were your introduction to recursively structured data. Let's review them by looking at a list of numbers:

```
A List of Numbers (LoN) is :
 - empty
 - (cons fst rst)
Where fst is a number ans rst is a LoN.
```

In plain English:

> A list of numbers is either the empty list or a list with two parts, the number fst and the list of numbers rst.

The recursion is pretty obvious. Non-empty lists contain other lists[5] It's easy to fixate on the recursion and forget all of the other equaly important stuff. Some lists, empty lists, are non-recursive. We call this variant of the LoN the BASE CASE because all other variants use this form of list must as their basis. *All recursive structures must have non-recursive base cases*. When lists are not-empty, then they're broken down into a struct like form with multiple pieces where each piece is some part of the whole. *All recursive structure must decompose the structure as part of the recursive variant of the structure.*

[5] which can be empty or not, and if not, they have a rest which is either empty or not, etc. etc.

We can apply this exact structure to std::string data. The empty string is *std::string("")*. You can recognize it with the class method *empty()*[6]. You can select the first of a non-empty string *s* with *s[0]* or *s.at(0)*. The rest of the string is *s.substr(1)*. The *cons* function is effectively replaced by the append operator + as *std::string(1,s[0])*[7] + *s.substr(1)* reconstructs the string *s*.

[6] or check for a length of 0

[7] notice we need to use the fill constructor

The *first* and *rest* logic we learned from lists is great. It was necessary with Racket lists because those were the only selectors we had. We are not always so restricted in how we recursively decompose data. We can do *last* and *but last*[8]. We have to be careful now. For this to work *s.substr(0,-1)* must return the empty string. It does not. To fix this we can rethink our base case. If the non-recursive case is the singleton string, the string of length 1, then this works. This limitation is not one of the last/but-last recursive structure but of our std::string selection methods. So, for strings of length at least two, then we can recursive decompose as follows.

[8] everything except the last

```
s = s.substr(0,s.length()-1) + std::string(1,s[s.length()-1])
```

The key observation is you can define recursive structure in many ways and you can actually identify several different base cases. We'll play with alternatives from time to time. Just be on the look out for different ways of capturing recursive structure of data.

Functional, Recursive strToUpper

If you've identified the recursive structure of your data, then you can map a recursive procedure to that structure. Let's build this definition up one piece at a time. First, a stub.

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

We begin by recognizing that strings are itemized: they're either empty or they're not.

```
std::string strToUpper(std::string str){
  if( str.empty() ){
    ...
  }
  else{ //not empty

  }
}
```

When they're not empty, then they have a first character and the rest of the string.

```
std::string strToUpper(std::string str){
  if( str.empty() ){
    ...
  }
  else{ //not empty
    ... str[0] ... str.substr(1) ...
  }
}
```

The rest is recursive, so let's have the procedure be recursive where the data is recursive.

```
std::string strToUpper(std::string str){
  if( str.empty() ){
    ...
  }
  else{ //not empty
    ... str[0] ... strToUpper(str.substr(1)) ...
  }
}
```

The uppercase form of the empty string is still the empty string.

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

```
  }
}
```

The expression *strToUpper(str.substr(1)* gives us all but the first of our string in uppercase form, but *str[0]* is still possibly lowercase. We can use toupper to fix that.

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

Now we just need to glue the two pieces back together by appending. In order to append, we need to convert our new first from a char to a std::string.

```
std::string strToUpper(std::string str){
  if( str.empty() ){
    return std::string("");
  }
  else{ //not empty
    return std::string(1,toupper(str[0])) +
           strToUpper(str.substr(1));
  }
}
```

That's it. Now go back and notice that most of what we did was simply convert the structure of the data to a procedural structure. When that was done, we filled in the blanks with the problem specific logic. Now, what if you wanted to go with the last and but-last decomposition. To keep the empty string we'll have to be a little creative and use two base cases.

```
std::string strToUpper(std::string str){
  if( str.empty() ){
    return std::string("");
  }
  else if( str.length() == 1){
    return std::string(1,toupper(s[0]);
  }
  else{ //not empty and not size 1
    return strToUpper(0,str.length()-1) +
           std::string(1,toupper(str[str.length()-1]));
```

```
  }
}
```

The beauty of this strategy is that most of the work is done by simply identifying the recursive structure of the data. Once you know what that looks like, then you simply direct your procedure to make a recursive call where the data is itself recursive. From a process perspective, you can imagine you're delegating all but a small fraction of the work to another process, the recursion, and then finishing off that last little bit that's left, toupper the first and append. To see how this plays out, you can trace the computation out using substitution[9].

[9] this trace is not strict. Steps are combined and simplified to highlight the recursive process

```
strToUpper(std::string("dog"))

std::string(1,toupper('d')) + strToUpper(std::string("og")

std::string(1,toupper('d')) + std::string(1,toupper('o')) + strToUpper(std::string("g")

std::string(1,toupper('d')) + std::string(1,toupper('o')) + ...
   std::string(1,toupper('g')) + strToUpper(std::string(""))

std::string(1,toupper('d')) + std::string(1,toupper('o')) + ...
   std::string(1,toupper('g')) + std::string("")

std::string(1,toupper('d')) + std::string(1,toupper('o')) + std::string("G")

std::string(1,toupper('d')) + std::string("OG")

std::string("DOG")
```

Functional, Iterative strToUpper

Our recursive solution used the recursive structure of the data to guide the recursive structure of the procedure. There isn't really such a thing as iterative structure. Iteration is strictly a process description and isn't something we attribute strictly to data. Iteration does, however, have a general structure that we can fit out problem into: the incremental accumulation of state.

To solve our problem iteratively we imagine progressing through each character in our string and as we go we repeat some computation that updates our accumulated state such that when we've *visited* all characters the accumulated state is the solution we seek. Just like with recursion, our repeated computation focuses on just one character at a time. Conventionally, we TRAVERSE the data in first to

last order. So, let's say we're doing *strToUpper* on the string "hello". Imagine we've visited 'h' and 'e' and are about to visit 'l'. Then we should have accumulated "HE" and the next step is use *toupper* to compute 'L' and then add that to the accumulated string "HE" to get "HEL". What we're doing is establishing the repeated computation by defining the next accumulated state value in terms of the current value. If s_t is the state at time t, then we want to find some sequence of operations that computes s_{t+1} given s_t. In C++ programs, s_t and s_{t+1} are the same object and we simply mutate that object. In order for this to work, we must also identify s_0, the initial value for our accumulated state. This is analogous to the base case for recursion.

There are generally two ways to think about s_0: in terms of the first update and as the solution to the empty sequence. Just because we're not writing a recursive procedure doesn't mean we should ignore the recursive structure of our string. Empty strings are still possible. In the context of iteration, an empty string means no iterative updates. It has no first character and therefore no s_1. This means that s_0 *is the solution when an empty string is encountered*. This perspective is extremely helpful if you have yet to identify the accumulative update logic. We already have some sense of how to accumulate state: take the current character, make it uppercase, and add it to the end of the accumulated string. Let's return to the string "hello". After one update we should have an s_1 as "H". The question is then, what string do we add 'H' to the end of to get "H". The answer is, of course, the empty string. It's important to note that no matter how you arrive at the value for s_0, the result should be the same just like we see with *strToUpper*.

Once we've established the accumulative update logic and the initial value for the accumulator, all that's left is to encorporate the traversal logic. If we repeat the update operation on each character starting with the first and proceeding one at a time until the last, then we should end up with the correct accumulated state. This also means that update s_1 accumulates the character at index 0, s_2 accumulates character 1, etc. In general, step t of the iterative process works with character $t - 1$[10].

[10] for non-empty strings of course

Before we express all of this in C++, let's lay out the steps of iterative problem solving:

1. Identify the data that needs to be accumulated and the operations that update the current accumulated state[11] to the next accumulated state[12]

[11] s_t

[12] s_t

2. Identify the initial value for the accumulator variable. This is the solution when dealing with an empty sequence of data and the value needed to correctly perform the first accumulator update.

3. Identify the traversal pattern that fits best with your accumulative update.

You don't have to work these parts in this order, but you do need to give some thought to each part.

Let's begin our C++ implementation of our iterative solution with a stub.

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

The first thing we'll do is work out the accumulative update logic. Setting up the initial accumulator variable is easy enough. While we're at it we can modify the stub to return the accumulator, which is what we want to do anyway. I like to use the name *acc*[13] when I can't think of anything better.

[13] short for accumulator

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

   std::string acc{""};

   return acc;
}
```

Now, just to help figure things out, let's explicitly write out a few updates. We'll get rid of these later, but it helps figure out the code. The std::string method *push_back* pretty much does exactly what we're thinking– add a character to the end of a string.

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

   std::string acc{""};

   //first update..
   acc.push_back(toupper(str[0]));
   //second update
   acc.push_back(toupper(str[1]));
   //third update
   acc.push_back(toupper(str[2]));


   return acc;
}
```

At this point we a pattern. For update *i* we do

```
acc.push_back(toupper(str[i-1]));
```

The reason we're subtracting one in $str[i-1]$ is due to the fact that we index string characters starting at zero but we count our updates starting at one. It's usually better to adjust our thinking to the indexes. From this perspective we're updating the accumulator with respect to the i^{th} character by doing

```
acc.push_back(toupper(str[i]));
```

Making this adjustment sets aside the focus on "updates" in favor of a focus on the data itself, the characters in the string.

Let's setup a bit more logic. We have a generalized update statement that uses a new variable i that needs to somehow work its way incrementally through the interval $[0, str.length())$.

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

   std::string acc{""};
   int i{0};

   // Repeat for all i = 0 to str.length()-1
   acc.push_back(toupper(str[i]));


   return acc;
}
```

Loop statements provide a general means of repeating a sequence of statements. The most basic C++ look is the *while* loop. It looks like this:

```
while( <continuation-condition> ){

    <Loop-body>

}
```

The loop body is a sequence of statements that is repeated for as long as the continuation condition is true[14]. Let's fill in what we know:

[14] this means the continuation condition is a boolean expression

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

   std::string acc{""};
   int i{0};

   while( ... ){

      acc.push_back(toupper(str[i]));
```

```
   }

   return acc;
}
```

Now look closely. If we leave i as is, then we'll never work with any character but the first. This means that part of our loop body should also update the value of i. What we want to do is update the accumulator for character i, then increase i by one. All the the following statements increase i by 1.

```
i = i+1;
i += 1;
i++;
++i;
```

The difference between the last two options is subtle but only important when you use $i++$ or $++i$ as expressions. As statements, they're equivalent. Let's update our code.

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

   std::string acc{""};
   int i{0};

   while( ... ){
      // accumulate the next uppercase letter
      acc.push_back(toupper(str[i]));
  // increase i by 1
      ++i;
   }

   return acc;
}
```

All that's left is the continuation condition. Recall our goal is to repeat this code such that i moves incrementally through the interval $[0, str.length() - 1)$. We start i at 0 and we increase it by 1 each time the loop executes. So, we want to keep repeating as long as i is less then $str.length()$.

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

   std::string acc{""};
   int i{0};
```

```
   while( i < str.length() ){
      // accumulate the next uppercase letter
      acc.push_back(toupper(str[i]));
  // increase i by 1
      ++i;
   }

   return acc;
}
```

This will cause a compiler warning because the string length is an unsigned integer and *i* is a signed integer. We can fix this easily enough by changing the type of *i*.

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

   std::string acc{""};
   unsigned int i{0};

   while( i < str.length() ){
      // accumulate the next uppercase letter
      acc.push_back(toupper(str[i]));
  // increase i by 1
      ++i;
   }

   return acc;
}
```

There's another type of loop called the *for* loop lets us move all of the logic surrounding the variable *i* to one place. The benefit of this style is that it seperates the accumulator logic from the logic that drives the loop.

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

   std::string acc{""};

   for(unsigned int i{0}; i < str.length(); ++i ){
      // accumulate the next uppercase letter
      acc.push_back(toupper(str[i]));
   }

   return acc;
}
```

You should be able to guess the overall structure of the for loop,

but just in case:

```
for( <init> ; <continuation> ; <update> ){
   <body>
}
```

The for loop first runs the $< \ init \ >$ statement. Then, as long as $< \ continuation \ >$ is true, it will execute the $< \ body \ >$ and then the $< \ update \ >$. One important difference between the for loop and the while loop is that any variable declared in the the $< \ init \ >$ statement of a for loop is only usable within the loop statement. Occasionally we'll need to use a loop variable, like *i*, outside of the loop statement. For this you either use a while loop or a for loop with the initialize separated like this one:

```
unsigned int{0};
for( ; i < str.length() ; ++i ){

}
```

There's a problem with our iterative implementations. Starting with the empty string and accumulating by pushing on new characters clearly illustrates the accumulative logic, but its inefficient. Each call to push_back can cause the system to resize the string object to account for the new character. This is unavoidable when we have no idea how big the final string will be. If however, you know how big the final string should be, then it's better to start with a string of that length and modify it. For *strToUpper* we know exactly how long the resultant string should be and can therefore apply this optimization. Let's look at a few ways to enact this with std::strings.

Our first option is probably the way to go. Unfortunately, the logic is not quite so easy to follow. Here we recognize that the string *str* is passed by value and is therefore a copy of the string we're converting to uppercase letters. This means it is exactly the length we want and we can simply overwrite the existing characters with their uppercase counterparts.

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

   for(unsigned int i{0}; i < str.length(); ++i ){
  // set str[i] to the uppercase version of its current contents
      str[i] = toupper(str[i]);
   }

   return str;
}
```

The beauty of this implementation is we avoid extra string variables. We could have avoided the implicit copy[15] by making an explicit copy. Once we do this, we might as well ignore the input and rework the code in terms of the copy.

[15] and sometimes must

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

   std::string acc{str};

   for(unsigned int i{0}; i < acc.length(); ++i ){
      acc[i] = toupper(acc[i]);
   }

   return acc;
}
```

Once we start programming this way, it's easy to imagine different traversal patterns. Here we work from last to first. Notice we no longer need to use unsigned integers. In fact, you'll run into problems if you do[16]. It's also important to note the change in update code[17].

[16] if there are no negative values, then i >= 0 is always true!

[17] –i,i–,i -= 1, i= i -1 all have the same effect– subtract 1 from *i*

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

   std::string acc{str};

   for(int i{str.length()-1}; i >= 0; --i ){
      acc[i] = toupper(acc[i]);
   }

   return acc;
}
```

If you're dead set on keeping the push_back logic around, then it's worth noting that we can change the capacity of a string. In doing so we make more room for future data. If we then use push_back, the computer will use that extra space rather than create more backup space.[18].

[18] compare std::string reserve with resize

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

   std::string acc{""};

   // add enough space for new characters
   acc.reserve(str.length());
```

```
   for(unsigned int i{0}; i < str.length(); ++i ){
  acc.push_back(toupper(str[i]));
   }

   return acc;
}
```

As always, there are lots of ways to capture the details. All of the implementations given about work off the same principle of iteration: incrementally accumulate the solution as you traverse the data.

Mutator-Based, Iterative setStrToUpper

Let's now turn our attention to the mutator version of this problem.

```
/**
 * 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);
```

The iterative version of this is very simple. You've actually already seen it except for the very important change from pass-by-value to pass-by-reference.

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

   for(unsigned int i{0}; i < str.length(); ++i ){
  // set str[i] to the uppercase version of its current contents
      str[i] = toupper(str[i]);
   }

   return;
}
```

All the heavy lifting is done by the computer when it passes the argument to setStrToUpper by reference. Once we have direct access to the string we can use loop-based iteration to modify the entire object right then and there.

Mutator-Based, Recursive setStrtoUpper

The recursive version of the mutator is tricky. The problem is that our selector for the rest of the string is functional. When we select *str.substr(1)*, we get a copy of the rest. So, making the recursive call on this copy doesn't modify the correct object. The immediate solution is to then replace the existing rest of the string with the new uppercase copy.

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

if( str.empty() ){
  return;
}

// upper the first
str[0] = toupper[str[0]];

// upper the rest
std::string rst{str.substr(1)}; //get the rest
setStrToUpper(rst); // uppercase the rest
str.replace(1,str.length()-1,rst); //str's rest to rst


    return;

}
```

This is not an efficient implementation[19]. The heart of the matter is that there's no direct way to pass part of a string by reference. If we could get the rest by reference, then we could pass as the argument to the recursive call and not worry about the replace. So, if we are really set on doing a recursive mutator for this problem, then we have create a helper procedure or overload[20] the current definition. Let's go the second route.

[19] the problem is replace causes repetitive writes of characters. do you see why?

[20] new signature, same name

Pay very close attention to the pre and post conditions. Notice that

```
/*
 Let all the letters from i to the end of str to their upper case counterparts
 @param str the string object getting modified
 @param i the lowest location to be modified
 @return none
 @pre str is all alphabetic letters. 0 <= i <= str.length()
 @post the letters in str at location i to the end have been converted to uppercase
 */
void setStrToUpper(std::string& str, unsigned int i);
```

Now a stub and some tests.

```
void setStrToUpper(std::string& str, unsigned int i){
   return;
}

TEST(setStrToUpper,recur2){

  std::string s{""};

  setStrToUpper(s,0);
  EXPECT_EQ(std::string(""),s);

  s = "hello";
  setStrToUpper(s,0);
  EXPECT_EQ(std::string("HELLO"),s);

  s = "hello";
  setStrToUpper(s,1);
  EXPECT_EQ(std::string("hELLO"),s);

  s = "hello";
  setStrToUpper(s,3);
  EXPECT_EQ(std::string("helLO"),s);

}
```

Now, notice how this solves our problem by letting us modify *i* instead of *str*.

```
void setStrToUpper(std::string& str, unsigned int i){

   if( i == str.length() ){ // empty!
      return;
   }

   str[i] = toupper(str[i]); // set the "first"
   setStrToUpper(str,i+1); // set the "rest"
   return;
}
```

We can now implement the version with the original signature by just calling to the two argument version with an initial value of *i* as zero..

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

```
   setStrToUpper(str,0);
   return;
}
```

The final version of *setStrToUpper* is strikingly similar to our iterative version: it works on a single string object rather than produce any copies and its logic is oriented around counting through indexes.

Recap

Let's wrap-up this discussion by looking at the preferred implementations. Let's first look at the recursive vs. iterative functional strToUpper.

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

  if( str.empty() ){
    return std::string("");
  }
  else{ //not empty
    return std::string(1,toupper(str[0])) +
           strToUpper(str.substr(1));
  }
}

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

   for(unsigned int i{0}; i < str.length(); ++i ){
  // set str[i] to the uppercase version of its current contents
      str[i] = toupper(str[i]);
   }

   return str;
}
```

It turns out that the iterative version is more efficient than the recursive version, but comparing these two implementations is a bit unfair. The iterative version is fairly optimized: it leverages the implicit copy done by pass-by-value functions to accumulate the answer in the same memory space as the function input. The recursive version actually suffers a performance hit by using *substr* which creates a copy of the rest rather than select the rest directly. This is more or less an unavoidable cost of the C++ std::string class. So, the difference is not really a function of recursion vs iteration in general but recursion vs. iteration for std::strings. Given a way to select the rest without making a copy, these two implementations would be equally efficient.

Now let's look at the mutator setStrToUpper.

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

   for(unsigned int i{0}; i < str.length(); ++i ){
  // set str[i] to the uppercase version of its current contents
      str[i] = toupper(str[i]);
   }

   return;
}

void setStrToUpper(std::string& str){

   setStrToUpper(str,0);
   return;
}

void setStrToUpper(std::string& str, unsigned int i){

   if( i == str.length() ){ // empty!
      return;
   }

   str[i] = toupper(str[i]); // set the "first"
   setStrToUpper(str,i+1); // set the "rest"
   return;
}
```

It turns out that these two implementations are, in terms of efficiency, more or less equivalent. This isn't too surprising given that the recursive version more or less mirrors the logic of the iterative version and avoids the use of *substr*.

What should we take away from this? You should probably favor iterative logic for std::string procedures because it plays well with how the string class is implemented. With that being said, our recursive setStrToUpper implementation shows us that we can effectively implement iterative logic using recursive procedures. So in the end, the ability work recursively and iteratively are tools we must have in our problem solving tool box. The other important thing we've started looking at is efficiency. We'll soon equip ourselves with a way to study and understand program efficiency. Until then, always remember that job number one is to make the code function correctly then optimize it. As you gain experience you'll learn different optimization techniques and know to design your code with those in

mind. Until then, don't start to worry about slow code until you have concrete evidence that your code is slow.