COSC-320
Notes on Recursion

Introduction

Recursion is a technique in which we break down a problem into one or more subproblems that are similar in form to the original problem. For example, suppose we need to add up all of the numbers in an array. We'll write a function called `add_array` that takes as arguments an array of numbers and a count of how many of the numbers in the array we would like to add; it will return the sum of that many numbers.

To develop the `add_array` function, we observe that if we had a function that would add up all but the very last number in the array, then we would simply have to add the last number to that sum and we would be done. But, the `add_array` function is just what we need for adding up all but the last number (as long as the array contains at least one number). After all, `add_array` takes an array and a count, and adds up that many array elements. If there are no numbers in the array, then zero is the desired answer. These observations suggest the following definition for the function:

```
int add_array(int arr[], unsigned count)
{
  if (count == 0)
    return 0;
  return arr[count - 1] + add_array(arr, count - 1);
}
```

Notice that the function has two components:

1. a *base case*, represented by the `if` and the `return 0`, in which the function does *not* call itself. This handles the case where there are no numbers to add.
2. a *recursive case* that breaks the problem down into a smaller version of the original problem together with an addition. In the recursive case, `add_array` is used to add together `count-1` items; the `count`-th item is then added to this result (remember that the `n`-th item of an array is stored at position `n-1`).

The call to `add_array` from inside `add_array` is called a *recursive call*.

Example: The Factorial Function

One of the classic textbook examples of recursion is the factorial function. Although factorial is not best done recursively, it provides us with many useful observations.

Recall that factorial, which is written `n!`, has the following definition:

```
        n! = 1 * 2 * 3 * .... * (n-2) * (n-1) * n
```

We can use this definition to write an iterative C++ function that implements factorial:

```
unsigned fact(unsigned n)
{
  unsigned i;
  unsigned result;

  result = 1;
  for (i = 1; i <= n; i++)
    result = result * i;
  return result;
}
```

We can develop a recursive definition for factorial by noticing that `n! = n * (n-1)!` and `1! = 1`. For example, `4! = 4 * 3!`. Notice that we need to specify a value for `1!` because our definition does not apply when `n=1`. This kind of definition is known as an *inductive definition*, because it defines a function in terms of itself.

The inductive definition of factorial immediately suggests the following recursive function:

```
unsigned fact(unsigned n)
{
  if (n == 1)
    return 1;
  return n * fact(n - 1);
}
```

Notice that this function precisely follows our new definition of factorial. It is recursive, because it contains a call to itself.

Let's compare the two versions:

- The iterative version has two local variables; the recursive version has none.
- The iterative version has three statements; the recursive version has one.
- The iterative version must save the solution in an intermediate variable before it can be returned; the recursive version calculates and returns its result as a single expression.

Recursion simplifies the definition of the `fact` function, making it more readable.

Writing Recursive Functions

To successfully apply recursion to a problem, you must be able to break the problem down into subparts, at least one of which is similar in form to the original problem. For example, suppose we want to count the number of occurrences of the number `k` in an array of `n` integers. The first thing we should do is write the header for our function; this will ensure that we know what the function is supposed to do and how it is called:

```
// Return the number of occurrences of k in the first
// n elements of array
// Precondition: the array contains at least n elements
unsigned count_ks(int array[], unsigned n, int k);
```

To use recursion on this problem, we must find a way to break the problem down into subproblems, at least one of which is similar in form to the original problem. If we know that the array contains `n` numbers, we might break our task into the subproblems of:

1. counting the number of times that `k` appears in the first `n-1` elements of the array (this is a subproblem that is similar in form to the original problem);
2. counting the number of times that `k` appears in the `n`-th element of the array (*i.e.* determining whether the `n`-th element is `k`); and
3. adding these two sums together and returning the result.

Part 1 of this decomposition suggests the following recursive call

```
count_ks(array, n-1, k);
```

If successful, this recursive call will return the number of occurrences of the number `k` in the first `n-1` positions of the array (we will discuss the conditions that must hold for a recursive call to be successful in Section 5; until then, we will assume that all recursive calls work properly). We must now determine how to use this result. If the last element in the array is not `k`, then the number of `k`s in the entire array is the same as the number of `k`s in all but the last element of the array. If the last number in the array is `k`, then the number of `k`s in the entire array is one more than the number found in the subarray. This suggests the following code:

```
if (array[n-1] != k)
  return count_ks(array, n-1, k);
return 1 + count_ks(array, n-1, k);
```

Here we have two recursive calls (only one of which will actually be used in any given situation). We must now determine whether there are any circumstances under which this code will *not* work. In fact, this code will not work when `n==0`; in such a case it tries to subscript the array with -1, which is not a legal array subscript in C++. (Oh alright, it's legal; it's just that it's almost never what you want, and will often lead to a segmentation fault or worse.) That means that unless we treat specially the case where `n` is zero, our function will not work when asked to count the number of `k`s in an array of zero items. We will therefore add a base case that will test for `n==0` and return zero as its result in that case. This gives the function:

```
unsigned count_ks(int array[], unsigned n, int k)
{
  if (n==0)
    return 0;
  if (array[n-1] != k)
    return count_ks(array, n-1, k);
  return 1 + count_ks(array, n-1, k);
}
```

This is a perfectly good recursive solution to the `count_ks` problem. It is not the only recursive solution though; there are other ways to break the problem into subpieces. For example, we could break the array into two pieces of equal size, count the number of `k`s in each half, then add the two sums. To do this, we will need to hand as arguments to `count_ks` not just the array and the subscript of the highest value in the array, but also the subscript of the lowest value in the array:

```
// Return the number of occurrences of k in array that lie between
// position low and position high, inclusive
unsigned count_ks(int array[], unsigned low, unsigned high, int k);
```

We calculate the midpoint between subscript `low` and subscript `high` with `(high+low)/2`. Thus we can count the number of `k`s in each half of the array and add them together with:

```
count_ks(array, low, (low + high) / 2, k) +
     count_ks(array, (low + high) / 2 + 1, high, k);
```

We now have a recursive case but no base case. When will the recursive case fail? It fails when the array does not contain at least two numbers. If the array contains no items, or it contains one item that is not `k`, then we should return zero. If the array contains exactly one number, and that number is `k`, then we should return one. Putting it all together, we get:

```
unsigned count_ks(int array[], unsigned low, unsigned high, int k)
{
  if ((low > high) || (low == high && array[low] != k))
    return 0;
  if (low == high && array[low] == k)
    return 1;
  return count_ks(array, low, (low + high)/2, k) +
         count_ks(array, 1 + (low + high)/2, high, k));
}
```

Note that the line

```
if (low == high && array[low] == k)
```

might just as well be written `if (low==high)`. The comparison with `k` is included here simply to make each of the relevant conditions explicit in the same expression.

These examples demonstrate that there may be many ways to break a problem down into subproblems such that recursion is useful. It is up to you the programmer to determine which decomposition is best. The general approach to writing a recursive function is to:

1. write the function header so that you are sure what the function will do and how it will be called;
2. decompose the problem into subproblems;
3. write recursive calls to solve those subproblems whose form is similar to that of the original problem;
4. write code to combine, augment, or modify the results of the recursive call(s) if necessary to construct the desired return value or create the desired side--effects; and
5. write base case(s) to handle any situations that are not handled properly by the recursive portion of the function.

Thinking About Recursion

How should you think about a recursive function? **Do not** immediately try to trace through the execution of the recursive calls; doing so is likely to just confuse you. Rather, think of recursion as working via the power of *wishful thinking*. Consider the operation of `fact(4)` using the recursive formulation of `fact`. `4` is not `1`, so the recursive case holds. The recursive case says to multiply `fact(3)` by `4`. Here is where the wishful thinking comes in: wish for `fact(3)` to be calculated. Because you have defined `fact(n)` to compute the factorial of `n`, your wish will be granted. You now know that `fact(3)=6`. So `fact(4)` is equal to `6` times `4`, or `24`.

An analogy you can use to help you think this way is corporate management. When the CEO of a corporation tells a vice-president to perform some task, the CEO doesn't worry about *how* the task is accomplished; he or she relies on the vice-president to get it done. You should think the same way when you are programming recursively. Delegate the subtask to the recursive call; don't worry about how the task actually gets done. Worry instead whether the top-level task will get done properly, given that all the recursive calls work properly.

Another way to think about recursion is to pretend that a recursive call is actually a call to a different function, written by somebody else, that performs the same task that your function performs. For example, suppose we had a library routine called `libfact` that returned the factorial of its argument. We could then write our own version of `fact` as:

```
unsigned fact (unsigned n)
{
   if (n == 1)
     return 1;
   return n * libfact(n - 1);
 }
```

This version of `fact` correctly returns one if its argument is one. If its argument is greater than one, it calls `libfact` to calculate `(n-1)!`, and multiplies the result by `n`. Because `libfact` is a library routine, we may assume that it works properly, in this case calculating the factorial of `n-1`. For example, if `n` is `4`, then `libfact` is called with `3` as its argument; it returns `6`. This is multiplied by `4` to get the desired result of `24`.

This example points out that a recursive call is just like any other function call. In particular, a recursive call gets its own parameter list and local variables, just as `libfact` would. Furthermore, while the recursive call is executing, the top--level call sits there waiting for the recursive call to terminate. This means that execution doesn't halt when a recursive call finds itself at the base case; once the recursive call returns, the top--level call then continues to execute.

Ensuring that Recursion Will Work

One of the most difficult aspects of programming recursively is the mental process of accepting that the recursive call will do the right thing. The following checklist itemizes the five conditions that must hold for recursion to work. If each of these conditions holds for your recursive function, you know that the recursion will operate correctly:

1. A recursive function must have at least one base case and one recursive case (it's OK to have more than one base case, and more than one recursive case).
2. The test for the base case must execute before the recursive call.
3. The problem must be broken down in such a way that the recursive call is closer to the base case than the top-level call.
4. The recursive call must not skip over the base case.
5. The non-recursive portions of the function must operate correctly.

Let's see whether the recursive `fact` function meets these criteria:

1. The first condition is met, because `if (n==1) return 1` is a base case, while the ``` `` ```else'' part includes a recursive call (`fact(n-1)`).
2. If we reach the recursive call, we must have already evaluated `if (n==1)`; this `if` is the base case test, so criterion 2 is met.
3. The recursive call is `fact(n-1)`. The argument to the recursive call is one less than the argument to the top--level call to `fact`. Our base case occurs when `n` is one. The recursive call is therefore closer to the base case *as long as `n` is positive*. If `n` is not positive, the recursive call does not move toward the base case, so the function will not work properly. Since `n` is unsigned, it cannot be negative.
4. Because `n` is an integer, and the recursive call reduces `n` by just one, it is not possible to skip over the base case.
5. Assuming that the recursive call works properly, we must now verify that the rest of the code works properly. We can do this by comparing the code with our second definition of factorial. This definition says that if `n` is one then `n!` is one. Our function correctly returns `1` when `n` is `1`. If `n` is not one, the definition says that we should return `(n-1)! * n`. The recursive call (which we now assume to work properly) returns the value of `n-1` factorial, which is then multiplied by `n`. Thus, the non--recursive portions of the function behave as required.

Some Definitions

This section describes some of the ways in which recursive functions are characterized. The characterizations are based on:

1. whether the function calls itself or not (direct or indirect recursion).
2. whether there are pending operations at each recursive call (tail-recursive or not).
3. the shape of the calling pattern -- whether pending operations are also recursive (linear or tree-recursive).

Direct Recursion

A C++ function is *directly* recursive if it contains an explicit call to itself. For example, the function

```
int foo(int x)
{
  if (x <= 0)
     return x;
  return foo(x - 1);
}
```

includes a call to itself, so it's directly recursive. The recursive call will occur for positive values of `x`.

Indirect Recursion

A C++ function `foo` is *indirectly* recursive if it contains a call to another function which ultimately calls `foo`.

The following pair of functions is indirectly recursive. Since they call each other, they are also known as *mutually recursive* functions.

```
int foo(int x)
{
  if (x <= 0)
   return x;
  return bar(x);
}



int bar(int y)
{
  return foo(y - 1);
}
```

Tail Recursion

A recursive function is said to be *tail recursive* if there are no pending operations to be performed on return from a recursive call.

Tail recursive functions are often said to ``return the value of the last recursive call as the value of the function."

Tail recursion is very desirable because the amount of information which must be stored during the computation is independent of the number of recursive calls. Some modern computing systems will actually compute tail-recursive functions using an iterative process.

The factorial function `fact` is usually written in a non-tail-recursive manner:

```
unsigned fact (unsigned n)
{
   if (n == 0)
      return 1;
   return n * fact(n - 1);
}
```

Notice that there is a ``pending operation," namely multiplication, to be performed on return from each recursive call. Whenever there is a pending operation, the function is non-tail-recursive. Information about each pending operation must be stored, so the amount of information is not independent of the number of calls.

The factorial function can be written in a tail-recursive way:

```
unsigned fact_aux(unsigned n, unsigned result)
{
  if (n == 1)
    return result;
  return fact_aux(n - 1, n * result)
}


unsigned fact(unsigned n)
{
  return fact_aux(n, 1);
}
```

The ``auxiliary" function `fact_aux` is used to keep the syntax of `fact(n)` the same as before. The recursive function is really `fact_aux`, not `fact`. Note that `fact_aux` has no pending operations on return from recursive calls. The value computed by the recursive call is simply returned with no modification. The amount of information which must be stored is constant (the value of `n` and the value of `result`), independent of the number of recursive calls.

Linear and Tree Recursion

Another way to characterize recursive functions is by the way in which the recursion grows. The two basic ways are ``linear'' and ``tree.''

A recursive function is said to be *linearly* recursive when no pending operation involves another recursive call to the function.

For example, the `fact` function is linearly recursive. The pending operation is simply multiplication by a scalar, it does not involve another call to `fact`.

A recursive function is said to be *tree* recursive (or *non-linearly* recursive) when the pending operation does involve another recursive call to the function.

The Fibonacci function `fib` provides a classic example of tree recursion. The Fibonacci numbers can be defined inductively as:

```
  fib(n) = 0  if n is 0,
         = 1  if n is 1,
         = fib(n-1) + fib(n-2) otherwise
```

For example, the first seven Fibonacci numbers are

```
   fib(0) = 0
   fib(1) = 1
   fib(2) = fib(1) + fib(0) = 1
   fib(3) = fib(2) + fib(1) = 2
   fib(4) = fib(3) + fib(2) = 3
   fib(5) = fib(4) + fib(3) = 5
   fib(6) = fib(5) + fib(4) = 8
```

This leads to the following implementation in C++:

```
unsigned fib(unsigned n)
{
   if (n == 0) return 0;
   if (n == 1) return 1;
   return  fib(n - 1) + fib(n - 2);
}
```

Notice that the pending operation for the recursive call is another call to `fib`. Therefore `fib` is tree-recursive.

Converting Recursive Functions to be Tail Recursive

A non-tail recursive function can often be converted to a tail-recursive function by means of an ``auxiliary'' parameter. This parameter is used to form the result. The idea is to attempt to incorporate the pending operation into the auxiliary parameter in such a way that the recursive call no longer has a pending operation. The technique is usually used in conjunction with an ``auxiliary'' function. This is simply to keep the syntax clean and to hide the fact that auxiliary parameters are needed.

For example, a tail-recursive Fibonacci function can be implemented by using two auxiliary parameters for accumulating results. It should not be surprising that the tree-recursive `fib` function requires two auxiliary parameters to collect results; there are two recursive calls. To compute `fib(n)`, call `fib_aux(n 1 0)`

```
unsigned fib_aux(unsigned n, unsigned next, unsigned result)
{
  if (n == 0)
     return result;
  return fib_aux(n - 1, next + result, next);
}
```

Converting Tail Recursive Functions to Iterative

Let's assume that tail recursive functions can be expressed in the general form

```
F(x)
{
  if (P(x))
    return G(x);
  return F(H(x));
}
```

That is, we establish a base case based on the truth value of the function `P(x)` of the parameter. Given that `P(x)` is true, the value of `F(x)` is the value of some other function `G(x)`. Otherwise, the value of `F(x)` is the value of the function `F` on some other value, `H(x)`. Given this formulation, we can immediately write an iterative version as

```
F(x)
{
   int temp_x = x;
   while (P(x) is not true)
    {
      temp_x = x;
      x = H(temp_x);
    }
   return G(x);
}
```

The reason for using the local variable `temp_x` will become clear soon. Actually, we will use one temporary variable for each parameter in the recursive function.

Example - factorial function

In the tail-recursive factorial function (`fact_aux`) given in Section 7,

- the function `F` is `fact_aux`
- `x` is composed of the two parameters, `n` and `result`
- the value of `P(n, result)` is the value of `(n == 1)`
- the value of `G(n, result)` is `result`
- the value of `H(n, result)` is `(n -1, n * result)`

Therefore the iterative version is:

```
unsigned fact_iter(unsigned n, unsigned result)
{
  unsigned temp_n;
  unsigned temp_result;

  while (n != 1)
    {
      temp_n = n;
      temp_result = result;
      n = temp_n - 1;
      result = temp_n * temp_result;
    }
  return result;
}
```

The variable `temp_n` is needed so `result` will be computed on the basis of the unchanged `n`. The variable `temp_result` is not really needed, but is used to be consistent.

Example - Fibonacci function

In the tail-recursive fibonacci function (`fib_aux`) given in Section 7,

- The function F is `fib_aux`
- `x` is composed of the three parameters `n`, `next`, and `result`
- the value of `P(n, next, result)` is the value of `(n == 0)`
- the value of `G(n, next, result)` is `result`
- the value of `H(n, next, result)` is `(n -1, next + result, next)`

Therefore the iterative version is

```
unsigned fib_iter(unsigned n, unsigned next, unsigned result)
{
  unsigned temp_n;
  unsigned temp_next;
  unsigned temp_result;

  while (n != 0)
   {
     temp_n = n;
     temp_next = next;
     temp_result = result;

     n = temp_n - 1;
     next = temp_next + temp_result;
     result = temp_next;
   }
  return result;
}
```

Converting Iteration to Tail Recursion

Just as it is possible to convert any recursive function to iteration, it is possible to convert any iterative loop into the combination of a recursive function and a call to that function. The ability to convert recursion to iteration is often quite useful, allowing the power of recursive definition with the (often) greater efficiency of iteration.

Converting iteration to recursion is unlikely to be useful in practice, but it is a fine learning tool. This Section gives several examples of the relationship between programs that loop using C++'s built-in iteration constructs, and programs that loop using tail recursion.

Suppose that we want to write a program that will read in a sequence of numbers, and print out both the maximum value and the position of the maximum value in the sequence. For example, if the input to our program is the sequence 2, 5, 6, 4, 1, then the program should tell us that the maximum number is 6, and that it occurs in position 3 of the input. Here is a program that performs this task using C++'s built-in iterative constructs:

```
  #include <stdio.h>
  #include <assert.h>
  #include <stdlib.h>

  #define MAX_NUMS 32
  #define max(num1,num2) ((num1) > (num2) ? (num1) : (num2))

  typedef int array_of_nums[MAX_NUMS];

  int main(void)
  {
    array_of_nums nums;
    int num_count;
    int max_num;
    int pos_of_max;
    int i;

    /* LOOP 1: Read in the numbers */
    num_count = 0;
    while(scanf("%d", &nums[num_count]) != EOF)
      num_count++;
    assert(num_count > 0);

    /* LOOP 2: Find the maximum number */
    max_num = nums[0];
    for (i = 1; i < num_count; i++)
      max_num = max(max_num, nums[i]);

    /* LOOP 3: Find the position of the maximum number */
    pos_of_max = 0;
    while (nums[pos_of_max] != max_num)
       pos_of_max++;

    /* Print the results */
    printf("The largest number, which was %d, occurred in position number %d\n",
           max_num, pos_of_max + 1);
   return EXIT_SUCCESS;
  }
```

You will notice that the above program uses three loops and an array where one loop and an integer would have sufficed. However, the purpose of this program is not to show the best way to find the maximum of a sequence of numbers, but rather to exhibit a program that contains a few loops.

We will convert each of the program's three loops into a tail recursive function. The key to implementing a loop using tail recursion is to determine which variables capture the state of the computation; these variables will then serve as the parameters to the tail recursive function. The first loop is a `while` loop that is responsible for reading numbers into an array. It terminates when `scanf` returns `EOF`. Aside from the `EOF` condition, two variables capture the state of the computation each time through this loop:

1. `num_count`---the number of integers that have been read so far.
2. `nums`---an array that stores those integers.

To translate this loop into a tail recursive function then, we will write a function that takes these two values as arguments. Since we are interested in getting a final value for `num_count`, we will write a function that returns `num_count` as its result. If the termination condition (`EOF`) is not true, the procedure will increment `num_count`, read a number into `nums`, and call itself recursively to input the remainder of the numbers. If the termination condition is true, the procedure will simply return the final value of `num_count`. Here is the code:

```
int read_nums(int num_count, array_of_nums nums)
{
  if (scanf("%d", &nums[num_count]) != EOF)
    return read_nums(num_count + 1, nums);
  return num_count;
}
```

Notice that the recursive call to `read_nums` is closer to the base case (`EOF`) than the top--level call by virtue of the fact that the call to `scanf` consumes input. The base case test happens before the recursive call, and the base case will never be skipped. Finally, if there is no more input, we return the current value of `num_count`, which has accumulated exactly the return value we desire. If there is more input, we add one to `num_count` and recurse. In either case, if we assume that the recursive call works properly, we get exactly the return value we want. Thus this function meets each of our criteria for valid recursive functions.

We must invoke this procedure with the correct initial values for its arguments; here is the appropriate code from the main program:

```
/* Read in the numbers */
num_count = read_nums(0, nums);
assert(num_count > 0);
```

Notice that this call to `read_nums` causes the initial value of `num_count` in `read_nums` to be zero (which is exactly the initial value it had in the original loop). `Assert` is a statement defined in `<assert.h>`; it flags an error if its argument is false, and does nothing if its argument is true. The `assert` statement is used to ensure that at least one number is read in. Without this statement, subsequent code will bomb if no numbers are entered.

Let's generalize from this example. In general, an iterative loop may be converted to a tail--recursive function by:

- determining what variables are used by the loop;
- writing a tail--recursive function that has one parameter for each of the identified variables;
- using the condition that causes the termination of the iterative loop as the base case test;
- determining how each variable changes from one iteration to the next, and handing the sequence of expressions that represent those changes as arguments to the recursive call; and
- Replacing the original iterative loop with a call to the tail--recursive function, using the initial value of each of the variables as arguments.

The second loop is a `for` loop that is responsible for finding the value of the maximum input number. Before entering the `for` loop, we make the assumption that the zero-th number read in is the largest. We then loop through all but the zero-th element of `nums`, looking for a higher value. Four variables capture the state of the computation during this loop:

1. `max_num`---the value of the largest number examined so far.
2. `i`---the loop control variable.
3. `num_count`---the number of input values.
4. `nums`---the input values.

Therefore, we will write a function that takes these four values as arguments. Since we want to get a single value back---the value of the largest number---we will make our function return an `int`. As with any function, we can give the formal parameters any names we choose. We will rename two of the values to give them more descriptive names. We will use the name `max_so_far` instead of `max_num`, and `pos_to_test` instead of `i`. Here is the code:

```
int find_max(int max_so_far, int pos_to_test, int num_count, array_of_nums nums)
{
  if (pos_to_test < num_count)
     return find_max(max(max_so_far, nums[pos_to_test]),
                     pos_to_test + 1, num_count, nums);
  return max_so_far;
}
```

To invoke this function, we will need to pass the value of the zero-th number read in as its first argument, and 1 (which was the initial value of the loop control variable in the iterative version of the program) as its second argument. Here is the appropriate portion of the main program:

```
/* find the maximum number */
max_num = find_max(nums[0], i, num_count, nums);
```

The third loop looks through the input numbers until it finds the first occurrence of the maximum number, then records the position of the maximum number. Three variables control the state of this computation:

1. `max_num`---the value of the largest number.

2. `pos_of_max`---the position of the number we are currently testing. This variable actually holds the position of the maximum number only when the loop is exited. Before the loop completes, this variable is effectively serving as a loop control variable.
3. `nums`---the array of input numbers.

These three values will serve as the parameters to our tail recursive function. As with the second loop, we will use a function that returns an `int`. If the function finds the maximum value at the location specified by `pos_to_test` (which is the function's name for the variable `pos_of_max` in the original iterative version), it will return that position. If the value at the location specified by `pos_to_test` is not equal to the maximum value, the function will use tail recursion to search the remaining positions:

```
int find_pos_of_max(int max_num, int pos_to_test, array_of_nums nums)
{
  if (nums[pos_to_test] != max_num)
    return find_pos_of_max(max_num, pos_to_test + 1, nums);
  return pos_to_test;
}
```

Note that although the recursive call is not textually the last code of the function, it is the last instruction executed before the return if we have not yet found the position of the maximum number. Therefore, it is a tail recursive call. The invocation of this function in the main program looks as follows:

```
  /* Find the position of the maximum number */
  pos_of_max = find_pos_of_max(max_num, 0, nums);
```

Exercises

1. Write a recursive function that will count the number of occurrences of a given integer in an array. Your function `count_occurrences` should take three arguments: an array of `int`s, the number of elements in the array, and the integer whose occurrences we want to count.

2. Write a recursive function called `count_7s`, which counts the number of occurrences of the digit `7` in the decimal representation of a given integer. For example, if the argument to `count_7s` is 3762797, `count_7s` should return `3` (because there are three occurrences of the digit `7` in 3762797). *Hint:* remember that `n % 10` will give the remainder of `n` divided by ten, while `n/10` will give the integer part of `n` divided by ten.

3. Write a recursive function called `replace_char` that replaces every occurrence of a specified character in a string with another character. Your function, which should be a `void` function, should take three arguments: a string; a character to be replaced, and a character with which to replace it. For example, if `foo` contains the string `"xizzi"`, then the call `replace_char(foo, 'i', 'y')` should modify `foo` so that it now contains the string `"xyzzy"`.

Thomas A. Anastasio
February 1, 2004