Assignment #1: Pointers and Recursion

Due: Monday, April 10th in class

Most of the assignments in this course are single programs of a substantial size. However, to get you started this first one is a series of problems you will solve in isolation. The first three problems are more practice with the important concepts from CS106A of pointers and dynamic allocation and the remaining five problems are short recursive programs. By short, we mean that none of them will require more than a page or so to complete. That doesn't mean you should put this assignment off until the last minute though—recursive solutions can often be formulated in just a few short, elegant lines but if you don't yet comprehend recursion, they can be extremely dense and complex.

1. Memory Mistakes Hall-of-Fame

The following two fragments exhibit problems that often show up when students are learning how to work with pointers and memory in C. Your job is to look carefully at each fragment, determine the memory error and show how to correct it.

a) The **`ConvertToUpperCase`** function is given a string and returns a new string that is a copy of the original string with all lowercase letters converted to uppercase. Identify the memory error below and explain how to fix the function so that it performs correctly.

```
static string ConvertToUpperCase(string str)
{
   string result;
   int i;

   for (i = 0; i < StringLength(str); i++)
      result[i] = toupper(str[i]);
   result[i] = '\0';
   return result;
}
```

b) **`GetRandomArray`** is supposed to prompt the user for a count and then create an array of random numbers of size count and return the size of the array created. Identify the memory error below and explain how you would fix the function so that it performs correctly.

```
main()
{
   int count, *randomNumbers;

   count = GetRandomArray(randomNumbers);
   printf("Last number is %d\n", randomNumbers[count-1]);
}

static int GetRandomArray(int array[])
{
   int i, count;

   printf("How many random numbers do you need? ");
   count = GetInteger();
   array = NewArray(count, int);
   for (i = 0; i < count; i++)
       array[i] = RandomInteger(1, 100);
   return count;
}
```

2. Freeing Memory

Memory that is dynamically allocated (via **malloc**/**GetBlock**/**New** or indirectly by **strlib.h** functions such as **Concat** that allocate memory for strings they return) is persistent in that it remains allocated until you explicitly free it via the **free** or **FreeBlock** functions. On one hand this persistence is desirable because it allows you to allocate storage that lives beyond the current stack frame, but as a result, the heap can become clogged with unused variables if the memory isn't freed when it is no longer needed. In CS106A we tend to ignore this consequence— the space wasted by not freeing things is often not significant and the problems introduced by not-quite-correct attempts to free the memory is a cure worse than the disease. However, as you move on to CS106B and beyond, it is instructive to think about how to correctly deallocate memory. Unfortunately keeping track of what to free can be tricky. And as important as it is to know what to free, it's also critical to know what not to free. Add code at the end of the following code fragment to free each piece of heap memory once and exactly once.

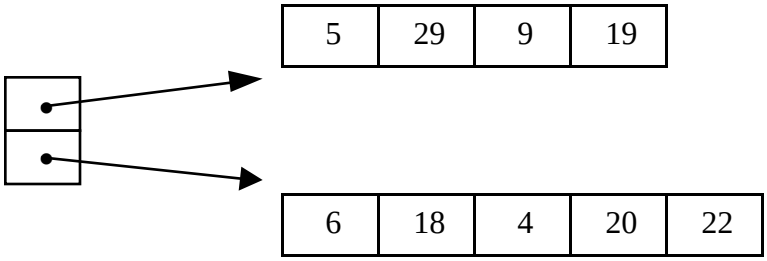
```
string *colors, spectrum;
int **palette;
char rainbow[10];

colors = NewArray(8, string);
colors[0] = "Red";
palette = GetBlock(sizeof(int *));
spectrum = "Purple";
colors[1] = ConvertToUpperCase(colors[0]);
colors[2] = spectrum;
strcpy(rainbow, spectrum);
*palette = New(int *);
colors[3] = rainbow;
colors[4] = colors[1];
colors[5] = GetLine();
colors[6] = GetBlock(5);
```

3. Dynamic allocation

Write the **`Separate`** function that given an array of integers will arrange the numbers into separate evens and odds arrays and returns the two arrays in a multi-dimensioned result. It also reports the number of evens and odds by reference. Let's see an example to understand how this works. Assuming the input array had 9 entries as such:

5	29	6	18	9	4	19	20	22

The **`Separate`** function would create and return an array with two elements, the first points to an array of the odd numbers from the input array, and the second to an array of the even numbers from the input array, like this:



It would also set the reference parameters nOdds to 4 and nEvens to 5.

```
int **Separate(int arr[], int n, int *nOdds, int *nEvens)
```

4. Recursive IntegerToString

Programming exercise 4-10 on p. 193 of the text.

5. Recursive ListSubsets

Programming exercise 5-6 on p. 226 of the text.

6. Fractal trees

Programming exercise 5-13 on p. 232-234 of the text (be sure to read the entire description). This fascinating problem offers many opportunities for extension. You can, for example, add color or redesign the recursive branching pattern in some interesting way so that the structure is not so regular, make the color lighter for new growth, and use filled regions for the trunk and the major branches. Another possibility: make a tree that grows in a prevailing wind from one side or the other or responds to a light source on one side. (Thanks to Maggie Johnson for some of the extension ideas.)

7. Making change

Everyone who's had a dead-end summer job working at Jack-in-the-Box (that is to say, me, at least) has had to deal with the intellectually stimulating task of making change. The customer is owned $.90, do I give them 9 dimes? 3 quarters, a dime, and a nickel? 90 pennies? What if I can't count that high?

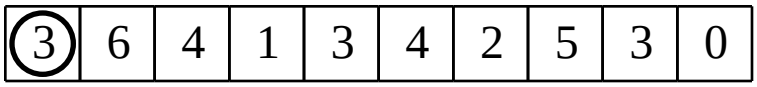
There are clearly many different configurations of coins that will work; let's say that we are interested in the combination that uses the fewest coins. One approach would be to use as many high value coins (i.e. quarters) as possible, then move on to use dimes for what is leftover, then nickels and finally pennies if needed. This type of algorithm is known as a *greedy* algorithm since at any given moment it makes the choice that looks best at the moment, the hope being that the locally optimal choice will lead to a globally optimal solution. However, what if I had no nickels in my change drawer and was trying to make $.31? The greedy solution chooses 1 quarter and 6 pennies, which is worse than the optimal solution of 3 dimes and 1 penny. Clearly we need something more clever to find the truly optimal arrangement, thus, recursion to the rescue!

Write a function **`int MakeChange(int amount, int coinValues[], int n`**) that takes an amount along with the array of available coin values and its effective size. You can assume you have use many of each coin as you want (i.e. if your cash drawer has pennies, it has an infinite supply of them). The function should return the minimum number of coins required that sum to the given amount. If the amount cannot be made (for example if you try to make $.31 and have no pennies), the function should return -1. You do not have to print or return the coin combination, just return the minimum number of coins in the optimal configuration.

```
/* Exmaple usage:
 *    int coins[] = {25, 10,  1};    quarters, dimes, & pennies
 *    min = MakeChange(31, coins, 3);  returns 4
 *    min = MakeChange(31, coins, 2);  returns -1
 */
int MakeChange(int amount, int coinValues[], int n)
```

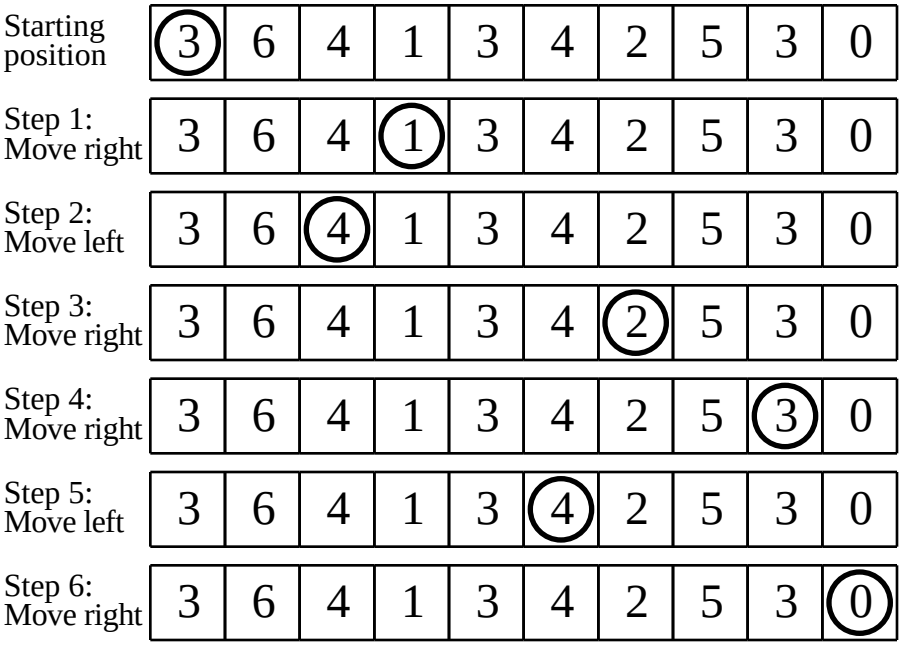
8. A recursive puzzle

You have been given a puzzle consisting of a row of squares each containing an integer, like this:

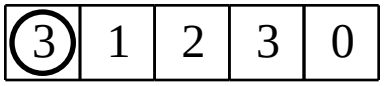


The circle on the initial square is a marker that can move to other squares along the row. At each step in the puzzle, you may move the marker the number of squares indicated by the integer in the square it currently occupies. The marker may move either left or right along the row but may not move past either end. For example, the only legal first move is to move the marker three squares to the right because there is no room to move three spaces to the left.

The goal of the puzzle is to move the marker to the 0 at the far end of the row. In this configuration, you can solve the puzzle by making the following set of moves:



Even though this puzzle is solvable—and indeed has more than one solution—some puzzles of this form may be impossible, such as the following one:



In this puzzle, you can bounce between the two 3's , but you cannot reach any other squares.

Write a function **`bool Solvable(int start, int squares[], int nSquares)`** that takes a starting position of the marker along with the array of squares and its effective size. The function should return **`TRUE`** if it is possible to solve the puzzle from the starting configuration and **`FALSE`** if it is impossible.

You may assume all the integers in the array are non-negative. The values of the elements in the array must be the same after calling your function as they are beforehand.

Thoughts on recursion

Recursion is a tricky topic so don't be dismayed if you can't just sit down and code these perfectly the first time. Take time to figure out how each problem is recursive in nature and how you could formulate the solution to the problem if you already had the solution to a smaller, simpler version of the same problem. Be sure to take care of your base case(s) lest you end up in infinite recursion. The great thing about recursion is that once you learn to think recursively, recursive solutions to problems seem very intuitive. Spend some time on these problems and you'll be much better prepared for the following assignment.

Getting started

There are no starter projects this time, since there is no special modules or starting code we need to give you. Please see the handouts on using CodeWarrior or Visual C++ to see how to create a new project for yourself.

Deliverables

At the beginning of Monday's class, you need to turn in the following:

- Written answers to the first two pointer questions
- Printouts of the .c files for each of the following programming exercises
- All of your .c files on a floppy disk

Place these in a large envelope, clearly marked with your name, CS106B, and your section leader's name. The best kind of envelope is one with a metal clasp closure or some other means of preventing the disk from falling out. Feel free to re-use envelopes, just scratch out any previous info written on it. There is often a box of old envelopes under the handout bins in Gates that can be recycled if you don't have one lying around from 106A.

You are also responsible for keeping a copy of your assignment to ensure that there is a backup in case your submission is lost or the assignment disk is damaged. Never turn in your only copy!

Little-known fact: If you have 3 quarters, 4 dimes, and 4 pennies, you have $1.19, which is the largest amount of money in coins without being able to make change for a dollar.