Programming Assignment #5: Hashtastic

COP 3502, Spring 2020

Due: Sunday, April 19, *before* 11:59 PM

Abstract

In this program, you will implement hash tables with linear and quadratic probing. Some of the functions in this assignment count the number of collisions that occur over a hash table's lifetime and are designed to help show that the amortized cost of your hash table operations (insertion, search, and deletion) is really low when you have a good hash function.

By completing this assignment, you will also learn a bit about function pointers and enumerated types, and you will gain additional experience with dynamic memory management and the use of *valgrind* to test programs for memory leaks.

When you're finished, you will have a very solid implementation of hash tables that you can use for all kinds of things.

Deliverables

Hashtastic.c

Note! The capitalization and spelling of your filename matter!

Note! Code must be tested on Eustis, but submitted via Webcourses.

1. Overview

In this assignment, you will implement hash tables that use linear and quadratic probing and support insertion, search, and deletion operations. The hash tables will expand dynamically when they start to get too full, and they will keep track of how many operations have been performed on them, as well as how many collisions have occurred over the course of their lifetimes.

By tracking the number of operations performed on each hash table (insert, search, and delete), as well as the total number of collisions encountered while executing those operations, you can gather empirical data to demonstrate whether you have a good hash function that leads to few collisions on average. You can also use this data to convince yourself that, with very few collisions per operation on average, hash tables actually have the ability to yield very good runtimes for all the operations listed.

Within each hash table, you will also store an indication of what probing mechanism is being used (linear or quadratic probing), as well as a pointer to the hash function that drives the hash table. That way, by passing a single hash table pointer to a function, you give that function everything it needs to know in order to perform a variety of operations on the hash table.

To facilitate all these goals, we have defined a powerful *HashTable* struct for you. (See Section 3, "Hashtastic.h," below.)

2. Function Pointers

In this assignment, each of your hash tables will be tightly coupled with a hash function that travels everywhere your hash table goes, ensuring that you never get stuck in a situation where you want to perform an operation (insertion, search, or deletion) but have no idea what hash function was being used with that hash table.

These hash functions will be written outside of your hash table code, in our test case files. However, each of your hash tables will contain a pointer to one of those hash functions.

I know what you're thinking: "Whaattttt?! A pointer to a *function*?!" Yes, this is really happening. And it's not really any more complex than having a pointer to, say, an int or a double. Here's how it works:

First, suppose you have the following function in some source file:

```
char mystery(int m, int n)
{
   return 'a' + (m * n * n * n) % 26;
}
```

Don't worry too much about what that *mystery()* function does, because it's total nonsense. The point is, it takes two integers as its only input parameters, and it returns a single character (which happens to be a character on the range 'a' through 'z').

Inside any other function, you can declare a variable that is capable of pointing to *mystery()* like so:

```
int main(void)
{
   // This just creates a function pointer. It is currently uninitialized.
   char (*foo)(int, int);

   return 0;
}
```

In general, the basic syntax for declaring a function pointer is as follows (without the angled brackets):

```
<return_type> (*<ptr_variable_name>)(<input1_datatype>, <input2_datatype>, ...);
```

Setting up your function pointer to point to a function is super easy. The name of a function is basically already a pointer (just like the name of an array is already a pointer to the base of that array). So, to set up our *foo* pointer to point to our *mystery()* function, we'd do this:

```
int main(void)
{
   // This creates a function pointer and initializes it.
   char (*foo)(int, int) = mystery;

   return 0;
}
```

Voila! That *foo* variable is now a pointer to the *mystery()* function. We can still call *mystery()* directly, or we can call it by dereferencing *foo* – just like we can set the value of an integer variable directly or change its value by dereferencing a pointer to that variable. The syntax for calling a function via a function pointer is as follows (without the angled brackets):

```
(*<ptr_variable_name>)(<input1_value>, <input2_value>, ...);
```

For example:

```
int main(void)
{
   // This creates a function pointer and initializes it.
   char (*foo)(int, int) = mystery;

   printf("Calling mystery directly: %c\n", mystery(5, 99));
   printf("Calling mystery indirectly: %c\n", (*foo)(5, 99));

   return 0;
}
```

You're probably asking, "Why would I ever need a function pointer?" or "Why is this happening to me?" This assignment provides one compelling example of the utility of function pointers: any time you need to strongly associate a function with a particular instance of some data structure (such as a hash table being strongly coupled with the hash function you've been using to insert keys into it), tucking a function pointer inside that data structure can help you achieve that.

3. Hashtastic.h

You must use the node struct we have specified in *Hashtastic.h* without any modifications. You **must** *#include* the header file in your *Hashtastic.c* source file like so:

```
#include "Hashtastic.h"
```

Note that the capitalization of *Hashtastic.c* matters! Filenames are case sensitive in Linux, and that is of course the operating system we'll be using to test your code.

In addition to some basic functional prototypes, the header file contains the following *#define* directives. These identifiers (`UNUSED`, `DIRTY`, `HASH_ERR`, and `HASH_OK`) will be used throughout your code. You should never hard-code a value like 0 or 1 in place of these identifiers, and you should **not** re-*#define* these constants in your *Hashtastic.c* file. We might change the underlying numeric values of these to something else when writing test cases to grade your program.

```
// Use this to mark a cell unused. Do NOT use INT_MIN directly.
#define UNUSED INT_MIN

// Use this to mark a cell dirty. Do NOT use INT_MAX directly.
#define DIRTY INT_MAX

// Several functions require you to return this in the event of an error.
#define HASH_ERR 0

// Several functions require you to return this upon success.
#define HASH_OK 1

// Default capacity for new hash tables. See makeHashTable() instructions.
#define DEFAULT_CAPACITY 5
```

The header file also contains two struct definitions that you will use in this assignment: Firstly, there's a *HashStats* struct. Each hash table has its own *HashStats* struct that contains two members, as described below:

```
typedef struct HashStats
{
   // This field keeps track of how many insert, search, or delete operations
   // take place of this hash table's lifetime.
   int opCount;

   // This field keeps track of how many collisions occur when performing
   // insert, search, or delete operations on this hash table.
   int collisions;
} HashStats;
```

After several operations have been performed on a hash table, dividing the number of collisions by the number of operations will tell you how many collisions took place per operation on average, which will give you a clearer window into the average runtimes for these operations.

Hashtastic.h contains your *HashTable* struct, as well:

```
typedef struct HashTable
{
   // Your hash table will store integer keys in this array.
   int *array;

   // The current capacity of your hash table (the length of 'array').
   int capacity;

   // The size of your hash table (the number of elements it contains).
   int size;

   // A pointer to the hash function for this hash table (initially NULL).
   unsigned int (*hashFunction)(int);

   // Probing type: LINEAR or QUADRATIC. Initialize to LINEAR by default.
   ProbingType probing;

   // A struct within a struct for maintaining stats on this hash table:
   // number of operations performed and number of collisions encountered.
   HashStats stats;

} HashTable;
```

You might be curious about that *ProbingType probing* field. The header file also contains a *ProbingType* "enumerated type" (or "enum" for short):

```
typedef enum ProbingType { LINEAR, QUADRATIC } ProbingType;
```

Let's break that down a bit. The underlying *enum* definition is as follows:

```
enum ProbingType { LINEAR, QUADRATIC };
```

Loosely speaking, that makes a new data type called *enum ProbingType* (similar to how *struct HashTable* is a data type, given the struct definition above). A variable of type *enum ProbingType* can be set equal to `LINEAR` or `QUADRATIC`. The typedef surrounding that whole line allows us to use *ProbingType* as our data type (instead of having to type out *enum ProbingType*), much in the same way our typedef for the *HashTable* struct allows us to use *HashTable* as a data type (rather than having to use the more verbose *struct HashTable*). For example:

```
int main(void)
{
   ProbingType probey01 = LINEAR;
   ProbingType probey02 = QUADRATIC;
   printf("probey01: %d, probey02: %d\n", probey01, probey02);
   return 0;
}
```

Note that C treats these enumerated types as ints in the background (where `LINEAR` = 0 and `QUADRATIC` = 1). Enumerated types effectively give us a quick list of numeric constants without forcing us to use *#define*. Used correctly, they help eliminate magic numbers and improve code readability. There are more examples of how to set enumerated types in the test cases included with this assignment.

4. Prime Numbers and the *primes.c* Source File

The *primes.c* file included with this assignment has a handy function that you're welcome to copy and paste into your *Hashtastic.c* file. It uses the *sqrt()* function from *math.h*, so you will need to use the -lm flag to compile your source code:

```
gcc Hashtastic.c testcase01.c -lm
```

5. Output

The functions you write for this assignment should not produce any output. If your functions cause anything to print to the screen, it might interfere with our test case evaluation. Be sure to disable or remove any *printf()* statements you have in your code before submitting this assignment.

6. Function Requirements

Function descriptions for this assignment are included on the following several pages. Please do not include a *main()* function in your submission.

```
HashTable *makeHashTable(int capacity);
```

Description: This function creates a hash table. Start by dynamically allocating space for a new hash table, and then properly initializing all the members of that struct. Note that by default, *hashFunction* should be initialized to `NULL`, *probing* should be initialized to `LINEAR`, and *array* should be initialized to a dynamically allocated integer array of length *capacity*. If the *capacity* parameter passed to this function is less than or equal to 0, you should use `DEFAULT_CAPACITY` (defined in *Hashtastic.h*) to set the initial length of the array. All cells within the array should be initialized to `UNUSED` *(*which is – you guessed it! – defined in *Hashtastic.h*).

Within this function, if any call to a memory allocation function fails, you should free any memory that was dynamically allocated within this function up until that point and then return `NULL`.

Returns: A pointer to the new hash table (or `NULL` if any call to a dynamic memory allocation function fails).

```
HashTable *destroyHashTable(HashTable *h);
```

Description: Free all dynamically allocated memory associated with this hash table. Avoid segmentation faults in the event that *h* is `NULL`.

Returns: `NULL`

```
int setProbingMechanism(HashTable *h, ProbingType probing);
```

Description: Within the hash table, set the *probing* field to either `LINEAR` or `QUADRATIC`, depending on

the value of the *probing* parameter passed to this function.

Returns: If the function is successful, return `HASH_OK` (defined in *Hashtastic.h*). If the function fails (either because *h* is `NULL` or because the function received a *probing* value other than `LINEAR` or `QUADRATIC`), return `HASH_ERR` (also defined in *Hashtastic.h*).

```
int setHashFunction(HashTable *h, unsigned int (*hashFunction)(int));
```

Description: Within the hash table, set the *hashFunction* field to the hash function that is passed as the second parameter to this function.

Returns: Return `HASH_ERR` if *h* is `NULL`. Otherwise, return `HASH_OK` (even if *hashFunction* is `NULL`).

```
int isAtLeastHalfEmpty(HashTable *h);
```

Description: Determines whether the hash table is at least half empty.

Runtime Restriction: This must be an O(1) function.

Returns: Return 1 if the hash table is at least half empty. If the hash table is less than half empty (more than half full), or if *h* is `NULL`, return 0. (You may assume that the hash table's *capacity* member will not be zero, although it's good practice to guard against this, just in case.)

```
int expandHashTable(HashTable *h);
```

Description: Within the hash table, replace the array (whose length is currently *capacity*) with a longer array. If the hash table is set to use linear probing, the length of the new array should be (*capacity* * 2 + 1). If the hash table is set to use quadratic probing, the length of the new array should be the first prime number that is greater than or equal to (*capacity* * 2 + 1). (Note that there is a handy function in *primes.c* that will help you generate prime numbers, and you're welcome to copy and paste it into your source code for this assignment.)

After creating an expanded array, you should re-hash all the values from the old array (starting from index 0 and working your way up through index *capacity* - 1) into the new array. In doing so, you should increment the hash table's *collisions* counter for each new collision that takes place as you re-hash the old values into the new array. However, you should *not* increase the hash table's *opCount* counter for each of those insertion operations. (If our goal is to count the average number of collisions per operation performed on this hash table, then to be fair, if inserting an element causes the table to expand, all the collisions that result should really be attributed to that *one* insertion operation – not all of the insertion operations that occur as a side effect when the hash table is expanded.) So, this function should affect the *collisions* count, but it should not affect the *opCount* of the hash table at all.

After the array is expanded, all of its empty cells should be initialized to `UNUSED`, and it should not have any cells marked as `DIRTY`.

Within this function, be sure to free memory as needed in order to avoid memory leaks.

Returns: Return `HASH_OK` if the expansion of the hash table is successful. If it fails for any reason (such as the failure of a memory allocation function), return `HASH_ERR`.

```
int insert(HashTable *h, int key);
```

Description: First and foremost, before even inserting the new element into the hash table, check that the hash table is still at least half empty – regardless of whether the table is using linear or quadratic probing. (You have a handy function that can do that for you.) If not, expand the hash table as described above. (You have a handy function to do that for you, as well!)

Using linear or quadratic probing (depending on the value of the hash table's *probing* field) in conjunction with the hash table's *hashFunction*, insert *key* into the hash table's array. Be aware that *hashFunction* might return very large values. You are responsible for modding by the table length to prevent array index out-of-bounds errors with the hash values produced by the hash function. One nice thing about using a hash function that returns an *unsigned int* is that we never have to worry that we'll get a negative index when modding by the table length.

While probing for a spot to insert *key*, increment the hash table's *collisions* counter every time a collision occurs. (Note that the last comparison you perform – right when you find an empty spot for the key – should *not* count as a collision.)

After you've inserted the key, increment the *size* and *opCount* for this hash table. (Each of those should get incremented by 1.)

Returns: Return `HASH_OK` if the operation is successful. Otherwise, return `HASH_ERR`. (Some potential causes of failure include: *h* is `NULL`; within *h*, the *hashFunction* is `NULL`; or expanding the hash table fails. As a failsafe measure, I also coded up my version of this function to return `HASH_ERR` if it goes through *capacity* number of iterations without finding an open spot to insert the key. That should not be possible if the table is expanding properly, but it's a solid backup plan to help avoid an infinite loop just in case there's a bug in the hash table expansion function.)

```
int search(HashTable *h, int key);
```

Description: Using linear or quadratic probing (depending on the value of the hash table's *probing* field) in conjunction with the hash table's *hashFunction*, search for *key* in the hash table's array. Be aware that *hashFunction* might return very large values. You are responsible for modding by the table length to prevent array index out-of-bounds errors with the hash values produced by the hash function.

While probing for the *key*, increment the hash table's *collisions* counter every time a collision occurs with an element that is not the key. (Note that hitting an `UNUSED` cell should *not* count as a collision, and should allow us to return from the function right away. However, finding a `DIRTY` cell should count as a collision.) Note: The number of cells probed by this function should not exceed *capacity*. If you perform *capacity* iterations of your probing loop without finding the key, break out and return.

Be sure to increment the *opCount* for this hash table by 1 regardless of whether you find the key or not.

Returns: If the key is found, return the index where it resides in the hash table. Otherwise, return -1. If *h* is `NULL` or the *hashFunction* member of the hash table is `NULL`, return -1. If there are multiple instances of *key* in the hash table, simply return the index of the first one you encounter while probing.

```
int delete(HashTable *h, int key);
```

Description: Using linear or quadratic probing (depending on the value of the hash table's *probing* field) in conjunction with the hash table's *hashFunction*, search for *key* in the hash table's array. If the key is found, delete it by setting the cell's value to `DIRTY`.

Be aware that *hashFunction* might return very large values. You are responsible for modding by the table length to prevent array index out-of-bounds errors with the hash values produced by the hash function.

While probing for the *key*, increment the hash table's *collisions* counter every time a collision occurs with an element that is not the key. (Note that hitting an `UNUSED` cell should *not* count as a collision, and should allow us to return from the function right away. However, finding a `DIRTY` cell should count as a collision and also requires that we continue searching through the table.) Note: The number of cells probed by this function should not exceed *capacity*. If you perform *capacity* iterations of your probing loop without finding the key, break out and return.

Be sure to increment the *opCount* for this hash table by 1 regardless of whether you find the key or not, and decrement the *size* member if the key is successfully deleted.

If you get a bit crafty, you can outsource most of the work for this function to your *search()* function. However, this is not strictly necessary.

Side Note: A solid implementation of hash tables might shrink the array if it starts to get too sparse (in order to cut back on wasted memory), but we will avoid doing that in this assignment (simply to make our lives a bit easier).

Returns: If *key* is deleted successfully, return the index where it resided in the hash table. Otherwise, return -1. If *h* is `NULL` or the *hashFunction* member of the hash table is `NULL`, return -1. If there are multiple instances of *key* in the hash table, simply delete the first one you encounter while probing.

```
double difficultyRating(void);
```

Returns: A double indicating how difficult you found this assignment on a scale of 1.0 (ridiculously easy) through 5.0 (insanely difficult).

```
double hoursSpent(void);
```

Returns: A reasonable and realistic estimate (greater than zero) of the number of hours you spent on this assignment.

7. Running All Test Cases on Eustis (*test-all.sh*)

The test cases included with this assignment are designed to show you some ways in which we might test your code and to shed light on the expected functionality of your code. We've also included a script, *test-all.sh*, that will compile and run all test cases for you.

Super Important: Using the *test-all.sh* script to test your code on Eustis is the safest, most sure-fire way to make sure your code is working properly before submitting.

To run *test-all.sh* on Eustis, first transfer it to Eustis in a folder with *Hashtastic.c*, *Hashtastic.h*, all the test case files, and the *sample_output* directory. Transferring all your files to Eustis with MobaXTerm is fairly straightforward, but if you want to transfer them from a Linux or Mac command line, here's how you do it:

1. At your command line on your own system, use *cd* to go to the folder that contains all your files for this project (*Hashtastic.c*, *Hashtastic.h*, *test-all.sh*, the test case files, and the *sample_output* folder).
2. From that directory, type the following command (replacing `YOUR_NID` with your actual NID) to transfer that whole folder to Eustis:

   ```
   scp -r $(pwd) YOUR_NID@eustis.eecs.ucf.edu:~
   ```

   Warning: Note that the `$(pwd)` in the command above refers to your current directory when you're at the command line in Linux or Mac OS. The command above transfers the *entire contents* of your current directory to Eustis. That will include all subdirectories, so for the love of all that is good, please don't run that command from your desktop folder if you have a ton of files on your desktop!

Once you have all your files on Eustis, you can run *test-all.sh* by connecting to Eustis and typing the following:

```
bash test-all.sh
```

If you put those files in their own folder on Eustis, you will first have to *cd* into that directory. For example:

```
cd HashtasticSpiritsProject
```

That command (*bash test-all.sh*) will also work on Linux systems and with the bash shell for Windows. It will not work at the Windows Command Prompt, and it might have limited functionality in Mac OS.

Warning: When working at the command line, any spaces in file names or directory names either need to be escaped in the commands you type (`cd project\ 5`), or the entire name needs to be wrapped in double quotes.

8. Running the Provided Test Cases Individually

If the *test-all.sh* script is telling you that some of your test cases are failing, you'll want to compile and run those test cases individually to inspect their output. Here's how to do that:

1. Place all the test case files released with this assignment in one folder, along with your *Hashtastic.c* file.
2. At the command line, *cd* to the directory with all your files for this assignment, and compile your source file with one of our test cases (such as *testcase01.c*) like so:

   ```
   gcc Hashtastic.c testcase01.c
   ```

3. To run your program and redirect the output to *output.txt*, execute the following command:

   ```
   ./a.out > output.txt
   ```

4. Use *diff* to compare your output to the expected (correct) output for the program:

```
diff output.txt sample_output/testcase01-output.txt
```

   If the contents of *output.txt* and *testcase01-output.txt* are exactly the same, *diff* won't have any output:

```
seansz@eustis:~$ diff output.txt sample_output/testcase01-output.txt
seansz@eustis:~$ _
```

   If the files differ, *diff* will spit out some information about the lines that aren't the same. For example:

```
seansz@eustis:~$ diff output.txt sample_output/testcase01-output.txt
1c1
< fail whale :(
---
> Hooray!
seansz@eustis:~$ _
```

Super Important: *Remember, using the test-all.sh script to test your code on Eustis is the safest, most sure-fire way to make sure your code is working properly before submitting.*

9. Testing for Memory Leaks with Valgrind

Part of the credit for this assignment will be awarded based on your ability to implement the program without any memory leaks. To test for memory leaks, you can use a program called *valgrind*, which is installed on Eustis. (This was covered in labs this semester: Valgrind Workshop and Pointer Tracing Mayhem.)

Valgrind will **not** guarantee that your code is completely free of memory leaks. It will only detect whether any memory leaks occur when you run your program. So, if you have a function called *foo()* that has a nasty memory leak, but you run your program in such a way that *foo()* never gets called, *valgrind* won't be able to find that potential memory leak.

The *test-all.sh* script will automatically run your program through all test cases and use *valgrind* to check whether any of them result in memory leaks. If you want to run *valgrind* manually, simply compile your program with the *-g* flag, and then run it through *valgrind*, like so:

```
gcc Hashtastic.c testcase01.c -g
valgrind --leak-check=yes ./a.out
```

In the output of *valgrind*, the magic phrase you're looking for to indicate that no memory leaks were detected is:

```
All heap blocks were freed -- no leaks are possible
```

For more information about *valgrind*'s output, see: http://valgrind.org/docs/manual/quick-start.html

10. Style Restrictions (*Super Important!*)

These are the same as in the previous assignment. Please conform as closely as possible to the style I use while coding in class. To encourage everyone to develop a commitment to writing consistent and readable code, the following restrictions will be strictly enforced:

- ★ Any time you open a curly brace, that curly brace should start on a new line.
- ★ Any time you open a new code block, indent all the code within that code block one level deeper than you were already indenting.
- ★ Be consistent with the amount of indentation you're using, and be consistent in using either spaces or tabs for indentation throughout your source file. If you're using spaces for indentation, please use at least two spaces for each new level of indentation, because trying to read code that uses just a single space for each level of indentation is downright painful.
- ★ Please avoid block-style comments: /* *comment* */
- ★ Instead, please use inline-style comments: // *comment*
- ★ Always include a space after the "//" in your comments: "// *comment*" instead of "//*comment*"
- ★ The header comments introducing your source file (including the comment(s) with your name, course number, semester, NID, and so on), should always be placed *above* your *#include* statements.
- ★ Use end-of-line comments sparingly. Comments longer than three words should always be placed *above* the lines of code to which they refer. Furthermore, such comments should be indented to properly align with the code to which they refer. For example, if line 16 of your code is indented with two tabs, and line 15 contains a comment referring to line 16, then line 15 should also be intended with two tabs.
- ★ Please do not write excessively long lines of code. Lines must be no longer than 100 characters wide.
- ★ Avoid excessive consecutive blank lines. In general, you should never have more than one or two consecutive blank lines.
- ★ When defining a function that doesn't take any arguments, always put *void* in its parentheses. For example, define a function using *int do_something(void)* instead of *int do_something()*.
- ★ When defining or calling a function, do not leave a space before its opening parenthesis. For example: use *int main(void)* instead of *int main (void)*. Similarly, use *printf("...")* instead of *printf ("...")*.
- ★ Do leave a space before the opening parenthesis in an *if* statement or a loop. For example, use use *for (i = 0; i < n; i++)* instead of *for(i = 0; i < n; i++)*, and use *if (condition)* instead of *if(condition)* or *if(condition)*.
- ★ Please leave a space on both sides of any binary operators you use in your code (i.e., operators that take two operands). For example, use *(a + b) - c* instead of *(a+b)-c*. (The only place you do *not* have to follow this restriction is within the square brackets used to access an array index, as in: *array[i+j]*.)
- ★ Use meaningful variable names that convey the purpose of your variables. (The exceptions here are when using variables like *i*, *j*, and *k* for looping variables or *m* and *n* for the sizes of some inputs.)

11. Special Restrictions (*Super Important!*)

1. As always, you must avoid the use of global variables, mid-function variable declarations, and system calls (such as `system("pause")`).
2. Do not read from or write to any files. File I/O is forbidden in this assignment.
3. Be sure you don't write anything in *Hashtastic.c* that conflicts with what's given in *Hashtastic.h*, and do not re-*#define* any of the constants from *Hashtastic.h* in your *Hashtastic.c* file.
4. Your *Hashtastic.c* file **must not** include a *main()* function. If it does, your code will fail to compile during testing, and you will not receive credit for this assignment.
5. Be sure to include your name and NID as a comment at the top of your source file.
6. No shenanigans.

12. Deliverable (Submitted via Webcourses, not Eustis)

Submit a single source file, named *Hashtastic.c*, via Webcourses. The source file must contain definitions for all the required functions listed above. Be sure to include your name and NID as a comment at the top of your source file. Don't forget `#include "Hashtastic.h"` in your source code (with correct capitalization). Your source file must work on Eustis with the *test-all.sh* script, and it must also compile on Eustis with both of the following:

```
gcc -c Hashtastic.c
gcc Hashtastic.c testcase01.c -lm
```

13. Grading

Important Note: When grading your programs, we will use different test cases from the ones we've released with this assignment, to ensure that no one can game the system and earn credit by simply hard-coding the expected output for the test cases we've released to you. You should create additional test cases of your own in order to thoroughly test your code. In creating your own test cases, you should always ask yourself, "What kinds of inputs could be passed to this program that don't violate any of the input specifications, but which haven't already been covered in the test cases included with the assignment?"

The *tentative* scoring breakdown (not set in stone) for this programming assignment is:

90%	Correct output for test cases used in grading.
10%	Passes memory leak checks (*valgrind*).
(*)	Adherence to various restrictions (see note below).

Important Note! Significant point deductions may be imposed for poor commenting and whitespace. Significant point deductions may also be imposed for violating the style restrictions listed above. Good code hygiene is the expectation at this point in the semester and won't be rewarded with "gimme points" on this assignment, despite the fact that poor code hygiene might result in deductions. You should also still include your name and NID in your source code.

Note! Your program must be submitted via Webcourses, and it must compile and run on Eustis to receive credit. Programs that do not compile will receive an automatic zero.

Your grade will be based primarily on your program's ability to compile and produce the *exact* results expected. Even minor deviations will cause your program's output to be marked as incorrect, resulting in severe point deductions. The same is true of how you name your functions and their parameters. Please be sure to follow all requirements carefully and test your program thoroughly. Your best bet is to submit your program in advance of the deadline, then download the source code from Webcourses, re-compile, and re-test your code in order to ensure that you uploaded the correct version of your source code.

Note also that your functions should not print anything to the screen. If they do, it will interfere with the output we generate while testing, resulting in incorrect test case results and an unfortunate loss of points.

Start early. Work hard. Ask questions. Good luck!