Assignment 5 Description

CMPUT201 Assignment 5: Arrays, Pointers, Dynamic Memory Allocation and Strings

- By: **YOUR_NAME_HERE**
- CCID: **YOUR_CCID_HERE**
- Student Number: **YOUR_STUDENT_NUMBER_HERE**

Sources

Tell us what online resources you used and who you collaborated with:

- **COLLABORATOR_1**
- **StackOverflow_Link**

Reminder: You may not use code from anyone else! Online resources and collaborators are for concepts only. As for all your assignments, this assignment will be checked for plagiarism using sophisticated tools so beware.

Goals

- Demonstrate knowledge of arrays
    - Fixed-Length Arrays
    - C99 Variable-Length Arrays
    - Bounds Checking
    - Memory layout
    - Multi-dimensional arrays
- Demonstrate knowledge of pointers
    - Difference between arrays and pointers
        - When does an array turn into a pointer
    - Unary `&` operator
    - Unary `*` operator
    - Subscripting a pointer `p[i]`
    - Modifying values "by reference"
    - Pointers are values
    - When are pointers valid?
- Demonstrate knowledge of malloc
    - When to allocate memory dynamically
    - Returning pointers pointing to arrays declared with malloc
- Demonstrate use of linters
    - Use linters to improve code quality

Code Quality Standards

Your code must meet the code quality standards. If you've taken CMPUT 174 before these should be familiar to you.

- Use readable indentation.
  - Blocks must be indented (everything between `{` and `}`)
  - One line must not have more than one statement on it. However, a long statement should be split into multiple lines.
- Use only idiomatic for loops.
- Use descriptive variable names. It must be obvious to the person reading (and marking your code) what each variable does.
- Never use complicated switch logic. Each case must fall through immediately to the next without running any code, or it must run some code and then break out of the switch statement.
- Never use goto.
- Never use control flow without curly braces (`if`, `else`, `do`, `while`, `for`, etc.)
- Use `<stdbool.h>`, `bool`, `true`, and `false` to represent boolean values.
  - Never compare with `true`, e.g. never `== true`.
- Do not leave commented-out code in your code.
- Provide comments for anything that's not totally and completely obvious.
- Always check to see if I/O functions were actually successful.
- On an unexpected error, print out a useful error message and exit the program.
  - For invalid input from the user you should handle it by asking the user to try again or by exiting the program with `exit(1)`, `exit(2)`, etc. or returning 1 or 2 etc. from main.
  - For unexpected errors, such as fgets failing to read anything, consider `abort()`.
- Main must only return 0 if the program was successful.
- Do not use magic literals (magic numbers or magic strings).
  - If a value has a particular meaning, give a meaningful name with `#define` or by declaring a constant with `const`.
  - Values other than 0 and 1 with the same meaning must not appear more than once.
  - 0 or 1 with a meaning other than the immediately obvious must also be given a name.
  - String literals must not appear more than once.
  - This includes magic numbers that appear in strings!
- Program must compile without warnings with `gcc -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3`.
- Program must be architecture-independent:
  - Program must not rely on the sizes of int, long, size_t, or pointers.
  - Program must compile without warnings with `gcc -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 -m32`. Note the added **-m32**!
  - The result of this compilation must be an executable program.
  - The 32-bit program must produce the same output as the 64-bit program.

New Code Quality Standards

- **Program must be compiler-independent:**
  - Program must compile without warnings with `clang`.
    - You can use the same options for `clang` that you use for `gcc`!
  - Program compiled with `clang` should produce the same output as when it's compiled with `gcc`.
- **Code must be lint-free:**
  - Program must pass `clang-tidy --checks=*` without warnings, except those which are explicitly allowed.
    - Currently allowed:
      - `cert-err34-c`
      - `cert-msc30-c`
      - `cert-msc50-cpp`
    - More allowed warnings may be added. Check eClass for updates.
    - See instructions on how to run the linters below.
  - Program must pass `oclint` without warnings, except those which are explicitly allowed.
    - Currently allowed:
      - `UselessParentheses`
    - More allowed warnings may be added. Check eClass for updates.
    - See instructions on how to run the linters below.
- **Code must be well-organized into functions:**
  - Each function should do one thing and one thing only.
  - The function's name should indicate what it does.
  - The same code should never appear twice!
  - Functions should be short, simple, and take few parameters.
  - See "Organizing Code into Functions" on eClass under Guides and FAQs.
- **Code must use globals appropariately:**
  - Program must not use global mutable variables (variables without `const` outside of a function).
    - Program can use global constant variables (`const`).
    - Using constants with `const` is highly encouraged.
- **General:**
- Program must use `size_t` variables where appropriate.
- New types must be named in CamelCase (starting with a capital letter) or in all_lower_case_t ending with _t.
- Constants and defines must be named in ALL_CAPS.
- Mutable variables and functions must be named camelCase (starting with a lowercase letter) or in all_lower_case.
- Dynamically allocated memory shall be freed.

Testing your Program

Correct input-output examples are provided. For example, `q1a-test1-input.txt` is the input to your `./question1` program. If your program is correct, its output will match `q1a-test1-expected-output.txt`.

You can tell if your output matches exactly by saving the output of your program to a file with bash's > redirection operator. For example, `./question1 >my-output-1.txt` will save the output of your question1 program into the file named `my-output-1.txt` instead of showing it on the screen. Be warned! It will overwrite the file, deleting anything that used to be in `my-output-1.txt`.

Similarly, you can give input to your program from a file instead of typing it by using bash's < redirection operator. For example, `./question1 <q1a-test1-input.txt` will run your program with the contents of `q1a-test1-input.txt` instead of being typed out.

These two can be combined. For example,

```
./question1 <q1a-test1-input.txt >my-output-1.txt
```

will use the contents of `q1a-test1-input.txt` as input and save the output of your program in `my-output-1.txt`.

When you want to check if your output is correct, you can then use the `diff` command from bash to compare two files. For example,

```
diff -b my-output-1.txt q1a-test1-expected-output.txt
```

will compare the two files `my-output-1.txt` and `q1a-test1-expected-output.txt` and show you any differences. `-b` tells diff to ignore extra spaces and tabs.

`diff` will only show you something if there's a difference between the two files. If `diff` doesn't show you anything, that means the two files were the same!

So, putting it all together, to check if your program handles one example input correctly, you can run:

```
./question1 <q1a-test1-input.txt >my-output-1.txt
diff -b my-output-1.txt q1a-test1-expected-output.txt
```

If `diff` doesn't show you anything, that means the two files were the same, so your output is correct.

This is what the included scripts (`test-q1a.sh`, etc.) do.

However, the examples are just that: examples. If your code doesn't produce the correct output for other inputs it will still be marked wrong.

Linting Your Program

The two linters `clang-tidy` and `oclint` will examine your code for a HUGE number of problems.

For example:

- Long lines must be broken into short lines.
  - No line can be longer than 100 chars.
- Functions must be short.
  - No more than than 30 statements. (Check this with oclint, it will warn about "ncss" aka "non-commenting source statements").
- Functions must be simple.
  - Check this with oclint, it will warn about "complexity".
- All variables must be used.
- Don't leave any dead code.
  - Dead code is code that can never run.

Those are just a few of the things `clang-tidy` and `oclint` can check for. There are too many to list here. Because they check for so many things, we may find things that the linters think are problems that we don't think are really problems or that we don't have the tools to fix yet.

We will add them to the eClass list as we find them.

Running the Linters

Both linters take your C filename, some options, then a `--` followed by the exact way you would compile your code with `clang`.

The options for `clang-tidy` are currently `--checks=*,-cert-err34-c,-cert-msc30-c,-cert-msc50-cpp`, which tells `clang-tidy` to look for every problem, except the problems named `cert-err34-c`, `cert-msc30-c`, and `cert-msc50-cpp`.

The options for `oclint` are currently `--disable-rule=UselessParentheses`.

If we find more things that are allowed we will add them to these options.

For example, if you would compile your program with:

```
gcc -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 -o myprogram myprogram.c
```

then you could compile it with clang with:

```
clang -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 -o myprogram
myprogram.c
```

The only that changed was the name of the compiler. So you would run clang-tidy and oclint like:

```
clang-tidy --checks=*,-cert-err34-c myprogram.c -- -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 -o myprogram myprogram.c
oclint --disable-rule=UselessParentheses myprogram.c -- -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 -o myprogram myprogram.c
```

Notice that you have to specify `myprogram.c` twice. This is because `oclint` and `clang-tidy` need to know both what file you want them to look at and exactly how you would compile it.

clang, clang-tidy, and oclint aren't on the lab machines :(This is due to Campus IT (IST) not keeping the lab machine's OS up to date. Please use the VM if at all possible. If absolutely can't run the VM, check back and we will have a way for you to run them soon.

Hints

- Solve "complexity" warnings by splitting your code into more functions.
  - Instead of putting a bunch of code inside of an loop, just call a function.
  - Instead of putting a bunch of code inside of an `if`, call a function.
- Don't use isdigit, etc. (man 3 isdigit) They cause linter warnings.
- Breaking up long lines.
  - Remember, C doesn't care too much about whitespace, so you can spread your statement over multiple lines.
  - Just be sure to use indentation to make it clear what you are doing.

```
\\ Example 1:
r = a + b * c + d * e * f;
\\ You can rewite it as...
r = a
    + b * c
    + d * e * f;

\\ Example 2:
if (a == b && c == d && e == f) { }
\\ You can rewrite it as...
if (
    a == b
    && c == d
    && e == f
) {
}
```

Questions

Question 1

Overview

In this question you are required to implement the following array manipulation functions that allocate new arrays on the heap.

The array is declared in main and will be shared by the functions. Each function takes array and its length as input. Apart from that some functions need array index and a value to operate (as shown in prototypes). Below you will find what each function is expected to do and what values it returns.

You need to read our question1.c as it is the canoncial definition of the specification. The examples and descriptions below are solely to explain what you will do in the quesiton.

setAt

Implement the `setAt` function in question1.c as described below in the function header comment. `setAt` is meant to safely set entries in an array if the positon is valid.

```
/*  setAt: set an entry in an integer array
 *          given `anArray` of `arrLen` length
 *          set anArray[pos] to value if position is valid.
 *          Upon successful setting of the array, `pos` is returned
 *          If pos is < 0 or pos >= arrLen -1 is returned.
 *
 *  @param anArray  an array of integers
 *  @param arrLen   length of anArray
 *  @param position position in the array to set
 *  @param value    to set in anArray[position] if position is value
 *
 *  @return -1 if position is invalid, and position if position is valid.
 *
 *  setAt( arr, 10, 5,  -2 ) ==  5; // set arr[5] = -2
 *  setAt( arr, 10, 10, -2 ) == -1; // Unsuccessful setting
 *  setAt( arr, 10, -1, -2 ) == -1; // Unsuccessful setting
 */
int setAt(int anArray[], int arrLen, int position, int value) {
  // complete the function body
  // if arguments are valid, set the val at idx in an_array.
  ...
```

subArrPointer

Implement the `subArrPointer` function in question1.c as described below in the function header comment. `subArrPointer` is meant to copy out subArrays from an existing array onto the heap using malloc. It is like asking for a pointer inside of an array except we safely allocate new memory and copy the input.

```
/*  subArrPointer: returns a copy of the array elements
 *          given `anArray` of `arrLen` length
 *          return a malloc'd copy of the sub sequence from `position`
 *          to `position`+`size` exclusive
 *          if the range in too long or invalid or positon is invalid
 *          NULL is returned.
```

```
*           Otherwise a pointer to heap allocated memory where the sub array
*           has been copied to will be returned.
*
*  @param anArray  an array of integers
*  @param arrLen   length of anArray
*  @param position position (index) in anArray of the subarray
*  @param size     length of the sub array to be returned
*
*  @return NULL if position is invalid, NULL if position+size is not
*  a valid subarray, and a pointer to a copy of the subarray if valid.
*
*  @warning memory is allocated with malloc and needs to be free'd later.
*
*  subArrPointer( arr, 10, 5,  -2 ) ==  NULL; // invalid size
*  subArrPointer( arr, 10, 9,  3 ) ==  NULL;  // range does not fit with
arrLen
*  free(subArrPointer( arr, 10, 3,  3 ));  // return a subarray of length 3
and free it
*
*/
int *subArrPointer(const int anArray[], int arrLen, int position, int size) {
  // complete the function body
  // check if arguments are valid
  // you may create a new array using malloc
  // return the sub-array of siz starting at idx
  ...
```

insert

Implement the `insert` function in question1.c as described below in the function header
comment. `insert` is meant to insert a new value within an existing array by allocating new space
for both all the old entries of the array and the new element. The new element should be inserted
at the requested position. If the end of the array is the position the new element should be
attached to the end.

```
/*  insert: insert a single value `value` into an array and returns a
 *  malloc'd copy of that array. The values after `position` will be
 *  shifted 1 index to the right. The original array, `anArray` is
 *  untouched.
 *
 *          given `anArray` of `arrLen` length return a malloc'd copy
 *          of `anArray` with `value` inserted into `position`. The
 *          resulting array will be of size arrLen+1.
 *          If position is invalid or memory cannot be malloc'd NULL is
 *          returned.
 *          If position == arrLen, value will be placed at the end of
 *          the new array.
 *          Otherwise a pointer to heap allocated memory where the array
 *          has been copied to will be returned. The new array will be
 *          arrLen + 1 length iff position was valid.
 *
 *  @param anArray  an array of integers
 *  @param arrLen   length of anArray
 *  @param position position (index) in anArray of the subarray
 *  @param value    value to be inserted at anArray[position]
```

```
 *
 *  @return NULL if position is invalid, and a pointer to a copy of
 *  the array with value in position if valid.
 *
 *  @warning memory is allocated with malloc and needs to be free'd later.
 *
 *  insert( arr, 10, 5,  -2 )[5] ==  -2; // -2 was inserted!
 *  insert( arr, 10, 10,  -2 )[10] ==  -2; // -2 was inserted at the end!
 *  arr[0] = 99; insert( arr, 10, 0,  0 )[1] == 99; // 99 was pushed over to
index 1
 *  insert( arr, 10, 11,  -2 ) == NULL; // 11 was an invalid position
 *  insert( arr, 10, -1,  -2 ) == NULL; // -1 was an invalid position
 */

int* insert(const int anArray[], int arrLen, int position, int value) {
  // complete the function body
  // if arguments are valid insert val at idx
  // you may have to create a new array using malloc
  // return the new array
  /* inserts the value value at position.
     Shift the elements after position to the right.*/
  ...
```

erase

Implement the `erase` function in question1.c as described below in the function header comment. `erase` is meant to remove an entry (specified by position) within an existing array by allocating new space for all of the old entries of the array minus one. The specified position should be removed. If the position is 0 the first element of the array should be removed.

```
/*  erase: remove a single value at `postion` from `anArray` and returns a
 *  malloc'd copy of modified array. The values after `position` will be
 *  shifted 1 index to the left. The original array, `anArray` is
 *  untouched.
 *
 *          given `anArray` of `arrLen` length return a malloc'd copy
 *          of `anArray` with the value at `position` removed and the
 *          rest of the values shifted to the left. The resulting array
 *          will be of size arrLen-1.
 *          If position is invalid or memory cannot be malloc'd NULL is
 *          returned.
 *          If position == arrLen NULL will be returned.
 *          If the array is size 0 NULL will be returned.
 *          Otherwise a pointer to heap allocated memory where the array
 *          has been copied to will be returned. The new array will be
 *          arrLen - 1 length iff position was valid.
 *
 *  @param anArray  an array of integers
 *  @param arrLen   length of anArray
 *  @param position position (index) in anArray of the subarray
 *
 *  @return NULL if position is invalid, and a pointer to a copy of
 *  the array with value in position if valid.
 *
 *  @warning memory is allocated with malloc and needs to be free'd later.
```

```
 *
 *  arr[6] = 7;  erase( arr, 10, 5)[5] ==  7; // arr[5] was removed
 *  arr[8] = -2; erase( arr, 10, 9)[7] == -2; // we still have the end of arr
 *  arr[1] = 99; erase( arr, 10, 0)[0] == 99; // 99 shifted to index 0
 *  erase( arr, 10, -1  ) == NULL; // -1 was an invalid position
 *  erase( arr, 10, 10 ) == NULL; // 10 was an invalid position
 */

int* erase(const int anArray[], int arrLen, int position) {
  ...
```

More details

For this question, `question1.c` a driver function has been provided. Don't change it, only replace the body of the requested functions with your code.

The `question1.c` is in the example tar.

Based on what's already in `question1.c` provide the function implementations and call them appropriately in the given driver (main) function.

The main function should run an interactive test driver to exercise your functions as well as a small unit test suite.

For examples, check the tar file.

We provide `question1.sh`, `question1-clang.sh`, and `question1-lint.sh` in the tar file.

Additional Requirements

- Put your C code for this question in `question1.c`
- You should compile the program as `./question1`
- You must demonstrate the proper use of functions calls and defining functions.
- You must not use global variables or static local variables, unless they are constants declared with `const`.
- You may ignore extra input.
- You may abort on any invalid input.
- You may use scanf for input.
- You may not use global variables (except constants with `const`).

Marking

- ☐ **1 Point** Program is well-organized into functions. (See above.)

- ☐ **1 Point** `question1.c` and `question1.sh` meets the requirements above and program output is correct for a valid input. (Examples: `test-q1a.sh`)

- □ **1 Point** `question1.c` and `question1.sh` meets the requirements above and program output is correct for a invalid input. (Examples: `test-q1b.sh`)

- □ **1 Point** `question1.c` and `question1-clang.sh` meets the requirements above, and your program is *compiler*-independent as described above. (Examples: `test-q1a-clang.sh test-q1b-clang.sh`)

- □ **1 Point** `question1-lint.sh` runs both linters correctly, as above, and your program is lint-free.

- □ **1 Point** Quality of question1.c meets all other quality standards, listed above.

Hints

- Initialize your memory when you malloc it
- Remember to check the bounds of the array.
- You should consider creating new arrays dynamically where need be.
- Return appropriate values from the functions to avoid segmentation faults.
- If you find any task repetitive, make it a function.

Question2:

Overview

You have learnt about the string library and its functions in class. In this question you are required to implement two functions similar to those in the string library using char *.

strFind

Implement the `strFind` function that searches for a string within another string. Both the strings are provided by the user as input and then passed to the function as arguments. The function returns the starting index of the searched for string in the original string. It returns -1 if the string is not found.

```
/*  strFind: find a string `str2` within another string `str1` and return an
 *  integer value. If string is found the returned value is the starting
index
 *  of `str2` inside `str1`
 *
 *          given `str1` search for `str2` in it by traversing the `str1`
 *          till the end.
 *          If `str2` is found in `str1` return the starting index of `str2`.
 *          If `str2` is no in `str1` return -1.
 *
 *          you must not use the string.h functions other than mem*
 *          functions like memcpy.
 *
 *  @param `str1`  an array of char to search within
 *  @param `str2`  an array of char to search for within `str1`
```

```
 *
 *   @return -1 if `str2` not found in `str1` , otherwise the index
 *   within the array where `str2` begins in `str1`.
 *   @return -1 if `str1` or `str2` are NULL
 *
 *   int strFind(const char *str1, const char *str2)
 *   str1= "This string has Hello in it";
 *   str2= "Hello;
 *   strFind(str1,str2);                      //returns 16
 *   str2="Happy"; strFind(str1,str2); //returns -1
 */
int strFind(const char* str1, const char* str2) {
```

strCat

Implement `strCat` function that concatenates two strings in a way that the second string 'str2'
appends to the end of the first string 'str1'. Both the strings are provided by the user as input. The
first character of str2 will overwrite the terminating null character of str1. The function then
returns the concatenated string.

```
/*   strCat: append the string `str2` to the end of `str1` and return
 *   the concatenated string.
 *
 *           given `str1` and `str2` are two non-empty strings, you
 *           have to concatenate `str2` to the end of `str1`. The null
 *           char of `str1` is overwritten by the first char of `str2`
 *           If `str2` is empty `str1` remains unchanged.
 *
 *           you must not use the string.h functions other than mem*
 *           functions like memcpy.
 *
 *   @param `str1`  an array of char to be concatenated to
 *   @param `str2`  an array of char to concatenate to the end of `str1`
 *
 *   @return the concatenated string `str1` with `str2` copied on the end
 *            if str1 or str2 is NULL, NULL is returned
 *
 *   @Warning str1 needs to be allocated enough memory to allow for the
 concatenation.
 *
 *   char strCat(char *str1, const char *str2)
 *   str1="Hello World"; str2="Hello";  strCat(str1,str2)    // returns Hello
 WorldHello
 *   //if user enters empty string the program should exit with invalid input
 error.
 */
char *strCat( char *str1, const char *str2) {
```

The main function should run an interactive test driver to exercise your functions as well as a
small unit test suite.

For examples, check the tar file.

We provide `question2.sh`, `question2-clang.sh`, and `question2-lint.sh` in the tar file.

Additional Requirements

- Put your C code for this question in `question2.c`
- You should compile the program as `./question2`
- You must demonstrate the proper use of functions calls and defining functions.
- You must not use global variables or static local variables, unless they are constants declared with `const`.
- You may abort on any invalid input.
- You may use scanf for input.
- You may not use global variables (except constants with `const`).

Marking

- ☐ **1 Point** Program is well-organized into functions. (See above.)

- ☐ **1 Point** `question2.c` and `question2.sh` meets the requirements above and program output is correct for a valid input. (Examples: `test-q2a.sh`)

- ☐ **1 Point** `question2.c` and `question2.sh` meets the requirements above and program output is correct for a invalid input. (Examples: `test-q2b.sh`)

- ☐ **1 Point** `question2.c` and `question2-clang.sh` meets the requirements above, and your program is *compiler*-independent as described above. (Examples: `test-q2a-clang.sh` `test-q2b-clang.sh`)

- ☐ **1 Point** `question2-lint.sh` runs both linters correctly, as above, and your program is lint-free.

- ☐ **1 Point** Quality of question2.c meets all other quality standards, listed above.

Hints

- Remember to check for NULL pointers
- Remember to check the end of array using null character.
- Return appropriate values from the functions to avoid segmentation faults.
- If you find any task repetitive, make it a function.

Submission

Test your program!

Always test your code on the VM or a Lab computer before submitting!

You can assume the shell script is run in the directory that contains both the source code and the executable. Run the `test-q1a.sh` script for question1. Run the `test-q1b.sh` script for question1. Run the `test-q2a.sh` script for question2. Run the `test-q2b.sh` script for question2. The scripts should produce no output.

Test your program with clang and lint your program

Unfortunately `clang` and the linters aren't available on the lab machines, so you need to use the VM for this step. If you aboslutely cannot use the VM, please wait a couple of days and we will have a solution for you.

Make 1 line (excluding the comments and header) shell scripts for question 1 and question 2 that will compile and run the 64-bit C program for that question with `clang`. Name the scripts `question1-clang.sh` and `question2-clang.sh` respectively. Run the `test-q1a-clang.sh` script for question1. Run the `test-q1b-clang.sh` script for question1. Run the `test-q2a-clang.sh` script for question2. Run the `test-q2b-clang.sh` script for question2.

Lint your program!

Hint: check `question2-lint.sh` for an example.

Run the `question2-lint.sh` script for question2. It's in the example tar. Run the `question1-lint.sh` script for question1. It's in the example tar.

To lint and check the code of your questions. If there are warnings, fix the code and try again.

Tar it up!

Make a tar ball of your assignment. It should not be compressed. The tar name is
`__YOUR__CCID__-assignment5.tar`

the tar ball should contain:

- `__YOUR__CCID__-assignment5/` # the directory
- `__YOUR__CCID__-assignment5/README.md` # this README filled out with your name, CCID, ID #, collaborators and sources.
- `__YOUR__CCID__-assignment5/question1.c` # C program
- `__YOUR__CCID__-assignment5/question2.c` # C program
- `__YOUR__CCID__-assignment5/question1` # executable
- `__YOUR__CCID__-assignment5/question2` # executable
- `__YOUR__CCID__-assignment5/question1.sh` # shell script
- `__YOUR__CCID__-assignment5/question1-clang.sh` # shell script
- `__YOUR__CCID__-assignment5/question1-lint.sh` # shell script
- `__YOUR__CCID__-assignment5/question2.sh` # shell script -- should be exactly the same as the example

- `__YOUR__CCID__-assignment5/question2-clang.sh` # shell script -- should be exactly the same as the example
- `__YOUR__CCID__-assignment5/question2-lint.sh` # shell script -- should be exactly the same as the example

Extra files such as the test files are allowed to be in the tar file. Any file we provide you in the release tar is OK to be in your tar file.

Submit it!

Upload to eClass! Be sure to submit it to the correct section.

Marking

This is a 12-point assignment. It will be scaled to 4 marks. (4% of your final grade in the course: A 12/12 is 100% is 4 marks.) Partial marks may be given at the TA's discretion.

- You will lose all marks if not a tar (a `.tar` file that can be unpacked using `tar -xf`)
- You will lose all marks if files not named correctly and inside a correctly named directory (folder)
- You will lose all marks if your C code is not indented. Minor indentation errors will not cost you all your marks.
- You will lose all marks if your code does not compile on the VMs or the lab machines.
- You will lose all marks if `README.md` does not contain the correct information! Use our example README!
  - Markdown format (use `README.md` in the example as a template)
  - Name, CCID, ID #
  - Your sources
  - Who you consulted with
  - The license statement below

License

This software is NOT free software. Any derivatives and relevant shared files are under the following license:

Copyright 2020 Abram Hindle, Hazel Campbell

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, and submit for grading and to permit persons to whom the Software is furnished to do so, subject to the following conditions: