Assignment 4 Description

CMPUT201 Assignment 4: Arrays, Pointers, and Linters

- 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.

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 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 Week 5.
- **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.
  - Program must not use static local mutable variables (`static` in a function without `const`).
    - Program can use static local constant variables (`static const` in a function).
    - Using constants with `const` is highly encouraged.
  - **Exceptions**:
    - Contants declared with `const` are always okay! Good even!
    - assignment4: question2.c can use global variables that are already in question2.c
    - assignment4: code in question2.c can use a static local variable.
- **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.

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-outputr-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

2D arrays and propagating values! We're going to calculate recursive functions on a map.

Make a 2D array to store a map. Then each round we calculate a new map based on the old map.

Each round you calculate a new map. Each cell of the new map is sum of the current cell and adjacent cells (up, down, left, right, and self). The sum is modulus 10 so it always between 0 and 9 inclusively.

For example if you have a cell surrounded by 4 neighbors who are all 0, the next round that cell will still be 1, but its 4 neighbors will have at least +1 in their cells. All coordinates not within the map count as 0. So if you are on the top row and no cell can be above your cell you count that value as 0 and it will always be 0. Do not access memory you didn't allocate.

Do not access memory you didn't allocate.

```
1 0 0
0 0 0
```

The next round will be:

```
1 1 0
1 0 0
```

And the next round will be:

```
3 2 1
2 2 0
```

The input format is in integers:

```
R M N value[0][0] value[0][1] value[0][2] ... value[M-1][N-2] value[M-1][N-1]
```

Where R is rounds (iterations) M is height (rows) and N is width (cols)

So for 2 rounds of 3 3 all 0s we would have

```
2 3 3 0 0 0 0 0 0 0 0 0
```

or you can lay it out nicely

```
2
3 3
0 0 0
0 0 0
0 0 0
```

All the numbers are whitespace seperated (spaces or new lines or end of file).

Invalid inputs (not enough, or bad dimensions) should be met with:

```
Invalid input!
```

- The maximum number of rounds (inclusive) are 32000
- The maximum dimensions of height or width is 16000

For examples, check the tar file.

Additional Requirements

- Put your C code for this question in `question1.c`
- You should compile the program as `./question1`
- You must make a `./question1.sh` to compile and run your question1.c program with `gcc` for 64-bit PCs.
- You must make a `./question1-32.sh` to compile and run your question1.c program with `gcc` for 32-bit PCs.
- You must make a `./question1-clang.sh` to compile and run your question1.c program with `clang` for 64-bit PCs.
- You must make a `./question1-lint.sh` to check your program with the two linters: `oclint` and `clang-tidy`
    - You may only exclude the warning listed on eClass!
- 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`)
- You may not use static local variables (except constants with `const`)

Marking

- **1 Point** Program is well-organized into functions. (See above.)
- **1 Point** `question1.sh` meets the requirements above and program output is correct for a valid input. (Examples: `test-q1a.sh`)
- **1 Point** `question1.sh` meets the requirements above and program output is correct for a invalid input. (Examples: `test-q1b.sh`)
- **1 Point** `question1-32.sh` meets the requirements above, and your program is *architecture*-independent as described above. (Examples: `test-q1a-32.sh` `test-q1b-32.sh`)

- **1 Point** `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

- You can use scanf, and not worry about whitespace.
  - You don't have to check that lines end in the correct places.
- Remember to check the bounds of the array.
- you should consider copying from 1 array to another.
- If you find it repetitive you should make it a function

Question2: Random number generation

Do you remember rand()? Do you remember how it can use srand() for a seed? Did you know that rand uses that seed as the initial state of their random number generator. Did you know that a random number generator is really a deterministic sequence generator?

Yeah! So your next random number might be calculated using a recursive equation like:

rand_n = rand_(n-1) * coeffecient1 + coeffecient2

That is the random number produced is the previous random number times coeffecient1 plus coeffecient2.

Random number generators *rely* on unsigned integer overflow.

For this question, `question2.c` has already been started. Don't change the lines above `// Don't change anything above this line` or below `// Don't change anything below this line`.

Replace `// Your code goes here` with your code.

`question2.c` is in the example tar.

Based on what's already in `question2.c` provide the functions `seedMyRand` and `myRand`.

`question2.sh`, `question2-32.sh`, `question2-clang.sh`, or `question2-lint.sh`, are also aready written for you. **Do not modify them!** Just use the ones from the example tar. But please, take a look at them! They are a good example for what you need to do on question #1.

For examples, check the tar file.

Additional Requirements

- Put your C code where it says "// Your code goes here"
- Don't change anything that's already there
- Don't use any additional global mutable (changeable) variables
- You should compile the program as `./question2`
- You must use `./question2.sh` provided to compile and run your program
- You must demonstrate the proper use of functions calls and defining functions
- Do not define new global variables (but use the ones that are already there!)
- Only add functions (and stuff inside functions)!

Marking

- **1 Point** Program output is correct for a valid input. (Examples: `test-q2a.sh`)
- **1 Point** Program defines and uses functions.
- **1 Point** Quality of your code in `question2.c` meets the quality standards, listed above.
- **1 Point** Static variables are used appropriately and no new globals are defined in your code.
- **1 Point** The program has no warnings when `question2-lint.sh` is run on the VM.
- **1 Point** Your program compiled with `question2.sh`, `question2-32.sh`, and `question2-clang.sh` all work the same (produce the same output for the same input) (Examples: `test-q2a.sh` `test-q2a-32.sh` `test-q2a-clang.sh`)

- Lose all marks if you modified the code above `// Don't change anything above this line` or below `// Don't change anything below this line`

Hints

- Think about what coeffecients make for better random numbers.
- Review what static means.
- Define at least two functions in `myrand.c`: `seedMyRand` and `myRand`
- Use the two global variables provided in `question2.c` to store the two coeffecients.
- Look at how `main` calls `seedMyRand` and `myRand`. That will tell you what paremeters they need to take and what they should return.

Submission

Test your program!

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

Make a 1 line (excluding the comments and header) shell script for question 1 that will compile and run the 64-bit C program for that question with `gcc`. Name the script `question1.sh`. Make a 1 line (excluding the comments and header) shell script for question 1 that will compile and run the 32-bit C program for that question with `gcc`. Name the script `question1-32.sh`. Make sure the program successfully compiles the program and then runs it. Take a look at the shell scripts for question 2, which are provided for you in the example tar: `question2.sh` and `question2-32.sh`. If the program doesn't compile it should not run the executable. The shell program should use 1 operator to achieve this and it should all fit on the same line. 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-q1a-32.sh` script for question1. Run the `test-q1b-32.sh` script for question1. Run the `test-q2a.sh` script for question2. Run the `test-q2a-32.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 a 1 line (excluding the comments and header) shell script for question 1 that will compile and run the 64-bit C program for that question with `clang`. Name the script `question1-clang.sh`. 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.

Lint your program!

Make a 2 line (excluding the comments and header) shell script for question 1 that will check your program with both linters.

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 that you wrote for question1.

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__-assignment4.tar`

the tar ball should contain:

- `__YOUR__CCID__-assignment4/` # the directory
- `__YOUR__CCID__-assignment4/README.md` # this README filled out with your name, CCID, ID #, collaborators and sources.
- `__YOUR__CCID__-assignment4/question1.c` # C program
- `__YOUR__CCID__-assignment4/question2.c` # C program
- `__YOUR__CCID__-assignment4/question1` # executable
- `__YOUR__CCID__-assignment4/question2` # executable
- `__YOUR__CCID__-assignment4/question1.sh` # shell script
- `__YOUR__CCID__-assignment4/question1-32.sh` # shell script
- `__YOUR__CCID__-assignment4/question1-clang.sh` # shell script
- `__YOUR__CCID__-assignment4/question1-lint.sh` # shell script
- `__YOUR__CCID__-assignment4/question2.sh` # shell script -- should be exactly the same as the example
- `__YOUR__CCID__-assignment4/question2-32.sh` # shell script -- should be exactly the same as the example
- `__YOUR__CCID__-assignment4/question2-clang.sh` # shell script -- should be exactly the same as the example
- `__YOUR__CCID__-assignment4/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 13-point assignment. It will be scaled to 4 marks. (4% of your final grade in the course: A 13/13 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: