# CMPUT201 Assignment 7: Debugging!

- 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 Debugging!
  - Fix stack issues
  - Fix memory allocation
  - Fix warnings
- 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

- Warnings in non-user code can be ignored. That's not your fault :-)
- 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

Using only the stack make a program that segfaults.

The program should be less than 50 lines. It should pass the linter.

### 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 not use global variables (except constants with `const`).

### Marking

- **1 Point** `question1.c` and `question1.sh` meets the requirements above and program output is expected. (Examples: `test-q1a.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`)
- **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

- Crash using the stack not malloc

## Question 2

### Overview

Using the heap make a program that segfaults via heap access.

The program should be less than 50 lines. It should pass the linter.

### 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 not use global variables (except constants with `const`).

### Marking

- **1 Point** `question2.c` and `question2.sh` meets the requirements above and program output is expected. (Examples: `test-q2a.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`)
- **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

- Crash using the heap not the stack

# Question 3, 4, 5, 6, 7

This is 5 questions written up as 1 question.

### Overview

For each question you are to interpret the program `questionX.c` (`question3.c, question4.c, question5.c, question6.c, question7.c`) where X is a question number, and make it pass the tests. Not only does it have to pass the tests, you have to debug its subtle error that might not crop up in the tests. Each has a subtle error!

Tasks:

- Fix the implementations of questions 3 to 7 to pass tests.
- Fix the implementations of questions 3 to 7 to not leak memory or have subtle memory bugs.

- Write up a comment in the top of `questionX.c` file about what the subtle error was and how you fixed it.
- Provide valgrind output to for each `questionX.c` running the first test (before or after is fine---before is better).

Given the subtley you might have to use gdb or valgrind to solve them.

You can also try compiling with the flag for gcc or clang `-fsanitize=address`. This is in `questionX-sanitize.sh`.

Consider running valgrind with `--tool=exp-sgcheck` as well to check for stack issues.

In the comment at the top of each file describe what the problem was and how you fixed it--specifically what tools you used to find the issues.

Capture the output of valgrind running each question with the question's first test input into the following files respectively:

- `q3a-test1-valgrind.txt`
- `q4a-test1-valgrind.txt`
- `q5a-test1-valgrind.txt`
- `q6a-test1-valgrind.txt`
- `q7a-test1-valgrind.txt`

e.g., capture the output of:

```
valgrind ./question3 < test-q3a-test1.txt
valgrind ./question4 < test-q4a-test1.txt
valgrind ./question5 < test-q5a-test1.txt
valgrind ./question6 < test-q6a-test1.txt
valgrind ./question7 < test-q7a-test1.txt
```

## For examples, check the tar file.

- We provide `question3.sh`, `question3-clang.sh`, and `question3-lint.sh` in the tar file.
- We provide `question4.sh`, `question4-clang.sh`, and `question4-lint.sh` in the tar file.
- We provide `question5.sh`, `question5-clang.sh`, and `question5-lint.sh` in the tar file.
- We provide `question6.sh`, `question6-clang.sh`, and `question6-lint.sh` in the tar file.
- We provide `question7.sh`, `question7-clang.sh`, and `question7-lint.sh` in the tar file.

## Additional Requirements

Where X is 3, 4, 5, 6, or 7:

- Put your C code for this question in `questionX.c`
- You must fix the subtle bug in each example program.
- You should compile the program as `./questionX`
- You must not use global variables or static local variables, unless they are constants declared with `const`.
- You may not use global variables (except constants with `const`).

### Marking FOR EACH QUESTION where X in { 3, 4, 5, 6, 7 }

- **1 Point** `questionX.c` and `questionX.sh` meets the requirements above and program output is expected. (Examples: `test-qXa.sh`)
- **1 Point** `questionX.c` contains a writeup of what the problem is with the example program.
- **1 Point** `questionX.c` and `questionX-clang.sh` meets the requirements above and passed `test-qXa.sh`.
- **1 Point** Valgrind out put for the first test of the question is include in qXa-test1-valgrind.txt program is *compiler*-independent as described above. (Examples: `test-q1a-clang.sh` `test-q1b-clang.sh`)
- **1 Point** `questionX-lint.sh` runs both linters correctly, as above, and your program is lint-free.
- **1 Point** Quality of questionX.c meets all other quality standards, listed above.

### Hints

- Use valgrind!
- Use gdb!
- Read the program!
- Use `questionX-sanitize.sh`
- Use the linter before you change the program!

# 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.

Run the `test-q3a.sh` script for question3. Run the `test-q4a.sh` script for question4. Run the `test-q5a.sh` script for question5. Run the `test-q6a.sh` script for question6. Run the `test-q7a.sh` script for question7.

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. Run the `test-q3a-clang.sh` script for question3. Run the `test-q4a-clang.sh` script for question4. Run the `test-q5a-clang.sh` script for question5. Run the `test-q6a-clang.sh` script for question6. Run the `test-q7a-clang.sh` script for question7.

# Lint your program!

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

- Run the `question1-lint.sh` script for question1. It's in the example tar.
- Run the `question2-lint.sh` script for question2. It's in the example tar.
- Run the `question3-lint.sh` script for question2. It's in the example tar.
- Run the `question4-lint.sh` script for question2. It's in the example tar.
- Run the `question5-lint.sh` script for question2. It's in the example tar.
- Run the `question6-lint.sh` script for question2. It's in the example tar.
- Run the `question7-lint.sh` script for question2. 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 must not be compressed. The tar name is `__YOUR__CCID__-assignment6.tar`

the tar ball should contain:

- `__YOUR__CCID__-assignment7/` # the directory
- `__YOUR__CCID__-assignment7/README.md` # this README filled out with your name, CCID, ID #, collaborators and sources.

- `__YOUR__CCID__-assignment7/question1.c` # C program
- `__YOUR__CCID__-assignment7/question2.c` # C program
- `__YOUR__CCID__-assignment7/question3.c` # C program
- `__YOUR__CCID__-assignment7/question4.c` # C program
- `__YOUR__CCID__-assignment7/question5.c` # C program
- `__YOUR__CCID__-assignment7/question6.c` # C program
- `__YOUR__CCID__-assignment7/question7.c` # C program
- `__YOUR__CCID__-assignment7/question1` # executable
- `__YOUR__CCID__-assignment7/question2` # executable
- `__YOUR__CCID__-assignment7/question3` # executable
- `__YOUR__CCID__-assignment7/question4` # executable
- `__YOUR__CCID__-assignment7/question5` # executable
- `__YOUR__CCID__-assignment7/question6` # executable
- `__YOUR__CCID__-assignment7/question7` # executable
- `__YOUR__CCID__-assignment7/question1.sh` # shell script
- `__YOUR__CCID__-assignment7/question1-clang.sh` # shell script
- `__YOUR__CCID__-assignment7/question1-lint.sh` # shell script
- `__YOUR__CCID__-assignment7/question2.sh` # shell script -- should be exactly the same as the example
- `__YOUR__CCID__-assignment7/question2-clang.sh` # shell script -- should be exactly the same as the example
- `__YOUR__CCID__-assignment7/question2-lint.sh` # shell script -- should be exactly the same as the example
- `__YOUR__CCID__-assignment7/question3.sh` # shell script -- should be exactly the same as the example
- `__YOUR__CCID__-assignment7/question3-clang.sh` # shell script -- should be exactly the same as the example
- `__YOUR__CCID__-assignment7/question3-lint.sh` # shell script -- should be exactly the same as the example
- `__YOUR__CCID__-assignment7/question4.sh` # shell script -- should be exactly the same as the example
- `__YOUR__CCID__-assignment7/question4-clang.sh` # shell script -- should be exactly the same as the example
- `__YOUR__CCID__-assignment7/question4-lint.sh` # shell script -- should be exactly the same as the example
- `__YOUR__CCID__-assignment7/question5.sh` # shell script -- should be exactly the same as the example
- `__YOUR__CCID__-assignment7/question5-clang.sh` # shell script -- should be exactly the same as the example
- `__YOUR__CCID__-assignment7/question5-lint.sh` # shell script -- should be exactly the same as the example
- `__YOUR__CCID__-assignment7/question6.sh` # shell script -- should be exactly the same as the example
- `__YOUR__CCID__-assignment7/question6-clang.sh` # shell script -- should be exactly the same as the example

- `__YOUR__CCID__-assignment7/question6-lint.sh` # shell script -- should be exactly the same as the example
- `__YOUR__CCID__-assignment7/question7.sh` # shell script -- should be exactly the same as the example
- `__YOUR__CCID__-assignment7/question7-clang.sh` # shell script -- should be exactly the same as the example
- `__YOUR__CCID__-assignment7/question7-lint.sh` # shell script -- should be exactly the same as the example
- `__YOUR__CCID__-assignment7/q3a-test1-valgrind.txt`
- `__YOUR__CCID__-assignment7/q4a-test1-valgrind.txt`
- `__YOUR__CCID__-assignment7/q5a-test1-valgrind.txt`
- `__YOUR__CCID__-assignment7/q6a-test1-valgrind.txt`
- `__YOUR__CCID__-assignment7/q7a-test1-valgrind.txt`

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 38-point assignment. It will be scaled to 4 marks. (4% of your final grade in the course: A 38/38 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