CSCI-243 Homework 3 -- Debugging Conway's Game of Life

Due Date

Due: 7/05/2017 at 11:59pm

Overview

This assignment is a practice/tutorial on:

- Using the `git` version control system to track stages of code repair and improvement;
- Debugging with `gdb`;
- Studying existing code not written by you;
- Fixing C language syntax;
- Refactoring code to have better structure; and
- Redesigning code to produce different, *cursor-controlled output*.

`gdb` Resources

This assignment leads you through the use of `gdb` and its commands step by step, and you should be able to finish this by *reading and doing*.

For those who may be interested in learning more, there are a number of online tutorials covering use of `gdb` to debug programs; some are lightweight, and some are comprehensive. See the course resources page for links to several of them. You can also get help on using `gdb` commands while running `gdb` itself, through its built-in `help` command. For example, the following shows the `help` command issued at the `gdb` prompt:

```
    (gdb) help
```

That will display a long menu of command information. For help with an individual `gdb` command, just add the command name to the `help` command; for instance,

```
    (gdb) help where
```

prints a description of how the `where` command works.

Style, Documentation and Version Control

The code we supply to you has very poor style and documentation, and no version control. As you work on this assignment, you will revise the style and add documentation while simultaneously using a version control system to record your evolution of the code.

Your solution should follow the course's style conventions for layout of functions, placement of braces`{}`, and code blocks.

At the end, your solution must have proper function documentation detailing the purpose of the function, its input parameters, and its output.

The documentation should have structure that allows documentation generation using a tool such as Doxygen. That means use of `/// ...` or `/** ... */` style comments for text that you want published. (There are a number of online tutorials that cover basic use of Doxygen, as well as an online manual covering full details of the program.)

For version control, you will use the `git` version control system (VCS). Instructions for using this from the command line are included in the steps you will follow.

START HERE: Supplied Code (`git`, then `get`)

We supply a naïve, buggy implementation that fails to compile cleanly without errors or warnings; it also does not run properly. You will clean up the problems with the code using `gcc` flags and the `gdb` debugger.

Navigate to the *parent directory* for your assignment location, and use a command to create a *git repository* named for this assignment; you could call the folder `hw3` as shown or something like `game-of-life`. The command will *create a new directory* that is set up for `git` to manage.

```
git init hw3
```

Then `cd hw3` and issue this `get` command to get the supplied materials:

```
get csci243 hw3
```

The `get` command creates the directories (folders) `act1`, `act2`, and `act3` and places several files into them. You should work in these sub-directories for each separate activity described below.

Here is what `get` fetches:

- The file `act1/bad-life_c` is a faulty version of code for Conway's Game of Life. This file is the starting point for all the code in this assignment. (The file is named `act1/bad-life_c` just so you don't compile it by accident!)
- The `act1/Makefile` file will compile and link the code for debugging. Do not change this Makefile.
- The `act2/header.mak` file will be used by `gmakemake` to create a `Makefile` for activity 2.
- The `act3/header.mak` file will be used by `gmakemake` to create a `Makefile` for activity 3.
- We also supply a module that controls the cursor position so that it can overwrite output on arbitrary lines of a standard terminal window. The display module source `act3/display.c`, and its `act3/display.h` header file have functions to pre-position the cursor before outputting a character.

This provides cursor control capability.

Instructions: Follow These Steps *Carefully*

The assignment leads you through the use of `gdb` showing the major commands that will help you learn how to apply a debugger to solve coding problems you will encounter later in the course.

You will start with the supplied `bad-life_c`, a buggy version of Conway's game of life which crashes.

One thing you will need to do several times is to copy and paste the output from `gdb` into a text file for submission. An easy way to do this is to use two shell windows:

- In the first window, you will be doing the work for the various activities; this is where the copied text originates.
- In the second window, you will be creating the text files containing the pasted output from `gdb`.

Note: Be sure that you have used `cd` in *both* shell windows to change directory into the appropriate activity directory, or your text files may go somewhere you don't expect to see them!

The easiest way to create a file containing copied text on a UNIX® or Linux® system is using the `cat` command. For instance, to create the `debug1.txt` file for the first activity, run the command

```
cat > debug1.txt
```

in the second shell window to begin creating the file. In the first shell window, use the mouse to highlight the text you want to copy, then paste it into the second shell window. This text will all be added to the `debug1.txt` file. Afterwards, press the `Enter` key, followed by a `CONTROL-D` key combination (which tells the system that you're done entering text).

The following are the activities of this assignment:

- Activity 1: Debugging uses `gdb` to investigate some fatal and not so fatal problems, correct them, and commit the corrections.
- Activity 2: Fixing Warnings and Refactoring Code rewrites the code to improves its efficiency, correct poor practices, and correct faulty algorithms, while also adding documentation and better formatting.
- Activity 3: Redesigning to use Cursor-controlled Output changes the code's behavior to use an overlayed display of generations rather than printing a sequence of generations one after the other.

Activity 1: Debugging with `gdb` in 2 Phases

The debugging activity has two phases, cleverly named Phase I and Phase II. It is good practice to run the code without any changes to find out what it does first. The first phase runs `gdb` and then makes a minimal change just to stop the crashes, and The second runs `gdb` after making some changes and recompiling.

While the knowledgable among you may find the problems before debugging and want to repair the problems before using gdb, please do not do so; follow the steps given below.

Phase I

In Phase I, you will create two files, `debug1.txt` and `debug2.txt`, to document your `gdb` investigation of the code problems, fix the first problem found, and commit the changes.

Important note: *do not* use `gmakemake` to create a `Makefile` for this activity - just use the one retrieved by the `get` command.

1. Change to the `act1` directory that `get` created.
2. Issue the command `cp bad-life_c good-life.c` to create your initial source file for the assignment. This is the subject program to debug.
3. Issue the command `make phase1` to compile and link creating the `good-life` executable.
4. Follow this step carefully. You will need to save (i.e. copy and paste into a file) and submit the outputs of this step.

   Run the command:

   ```
   gdb -q good-life
   ```

   and do these things at the `(gdb)` prompt:

   - Issue the command `run` and then enter `123` for the number of organisms at the prompt. The program will crash.
   - Issue the command `where` or `backtrace` to show the program call stack.
   - Issue the command `frame N` or `up` and `down` commands to move up/down the call stack in order to. go to the function frame in *user code* that failed. User code is code in the `good-life.c` file.
   - Issue the `list` command to show surrounding lines.
   - Issue the `break` to put in a breakpoint at the line where the crash stopped execution in user code.
   - Issue the command `info break` to list breakpoint information.

     Notice that the breakpoint's line number is incorrect. The command `break` put a break at line number 174, but the failing line is 176.
   - Put a new *breakpoint* at line 176.
   - To delete the incorrect break, `delete breakpoint 1`
   - Then check with `info break` that the breakpoint is in the correct place, and there is only one.

5. Copy and paste the lines of text beginning at the `gdb` command and continuing through the `backtrace`, `frame/up`, `list`, `break`, and `info break` commands into the file **`debug1.txt`**. You will submit this file at the end of activity 1.

6. Follow this step carefully. You will need to save (i.e. copy and paste into another file) and submit the outputs of this step also.

   - `run` again; `gdb` should stop at the breakpoint.
   - Issue `info break` to show breakpoint information.
   - Issue the command `display row` (`display` can display only one value at a time.)
   - Issue the command `display col`
   - Issue the command `whatis life` to find out the variable's type.
   - Issue `display life[row][col]`
   - Issue `run` again, and
   - Issue the `next` command when you reach the breakpoint.

7. Copy and paste the text from the last step (i.e. the last `run` command) through what happened after `next`) into the file **`debug2.txt`**.

8. Search for the same error since it occurs elsewhere in the program. **Append a short description of the cause and source of the problem** to the end of the file **`debug2.txt`**. This description should identify the problem and the line number or numbers at which the problem occurs.

9. Issue the `quit` command to exit from `gdb`.

10. Edit the file and fix the problem you found. If you found other problems, do not change them yet; just document them. (You'll have opportunity to fix those problems later in the lab.)

11. Issue the command `make phase1` to re-compile and link.

12. Run `good-life` to test your changes. The program should no longer crash; it now runs forever, but the output will be incorrect. To *kill the program* you must enter the `CONTROL-C` key combination. (You may have to enter that a couple times to interrupt and finally terminate the program.)

13. Issue the command

    ```
    git add good-life.c
    ```

    to add the file to the commit set.

14. Issue the command

```
git commit
```

to put the repaired code in the repository. You will have to enter a comment for the commit; usually, these comments are short phrases describing the changes made.

The default editor for `git` is `vim` on Linux. If you wanted to switch to `nano` in your `bash` environment however, you could do this to set the editor:

```
EDITOR=nano
```

Phase II

In Phase II, you investigate the new problem, and document your second investigation in `debug3.txt`.

1. Issue the command `make phase2` to compile and link again. There should be no warnings from this step.
2. Follow this step carefully. You will need to save (i.e. copy and paste into another file) and submit the outputs of this step.

   Re-run the `gdb` command, and do the following things:

   - `break 176`
   - `run` and enter 123 for the number of organisms.
   - `display row`
   - `display col`
   - `display life[row][col]`
   - `continue` through 17 breakpoints. You can `continue N` to do N cycles through a breakpoint.
   - `continue` through 4 more breakpoints and observe the program output interspersed with your debugging commands and `display` output.
3. Copy-paste the lines from the setting of the first breakpoint to where gdb stopped after `continue 4` into the file **`debug3.txt`**.
4. Identify the source of the problem and search the code for the same error since it occurs in *several places*. **Append a short description of the cause and sources of this problem** to the end of the file **`debug3.txt`**. This description should identify the problem and list the line numbers at which the problem occurs.
5. Quit `gdb`, correct those errors you found, re-`make phase2` and re-test using `gdb` as appropriate.
6. Issue the command `git add good-life.c` to add these changes to your commit set.

7. Issue the command `git commit` to save the repaired code to the repository.

8. You need to save the version log for your submission because you have finished this activity. Issue the command

   ```
   git log > revisions.txt
   ```

   to save the log to a file called `revisions.txt`.

You will know you have finished phase II when you can `make phase2` and run the program, which now runs forever until you kill it with the `CONTROL-C` combination. (The code still has algorithmic and other problems, but it does run without crashing.)

Activity 1 Submission

Submit your results for this activity with the `try command`:

```
try grd-243 hw3-1 debug1.txt debug2.txt debug3.txt revisions.txt
```

(You submit the revision information but not the code; it's not ready yet!)

Activity 2: Fixing Warnings and Refactoring Code

Fixing warnings to clean up the code

First clean up the code based on the compiler's detailed warning messages.

1. Change to the `act2` directory created by `get`, and copy the `good-life.c` file from the previous activity into `act2`. This code should compile (with warnings) and run without crashing.

2. Issue the command `gmakemake > Makefile` to create a `Makefile` for use by `make` for this activity.

3. Issue the command `make` to compile and link. You will see quite a few warning messages that it is time to fix.

4. Fix the warnings about "statement with no effect".

5. Also fix the warnings about "comparison with string literal".

6. Lastly fix the "unused..." warnings in this activity.

7. Now modify the while loop to make the loop end after 100 generations.

8. Issue the command `make` to re-compile and link. Find and fix any errors and warnings, editing and making until the program no longer produces any compile and link warnings.

9. As the code now compiles and links cleanly with no warnings, it is time to issue the commands `git`

`add good-life.c` and `git commit` to save the cleaned-up code to the repository.

At this point of activity 2, the program should run and produce outputs of '*' and ' '(space) characters, but it works incorrectly due to coding design errors.

10. Run the program like this to save output to a file for study:

    ```
    echo 321 | ./good-life > output.txt
    ```

11. Open `output.txt` and study several of its cases. Observe that the program does not implement Conway's rules correctly.

    One way to investigate the flaws in the algorithm is to look at sequential generations where there are several neighboring organisms. For example, the rules state that too many neighbors cause an organism to die, but the program does not correctly make them die.

Now it is time to redesign the code.

Background: What is Refactoring?

As you work on an implementation and get it running, you should consider how you can *refactor* the code to reduce the amount of repetition of similar code segments. This turns repeated code into proper functions that can be called and reused when appropriate.

What this means is that you find places where the same code is repeated; pull that code out and convert the code into a function which you then call at the places where the duplicate code was originally found. When you refactor code, you have to identify and define the parameters and types which the new function must have to be able to serve the needs where the new function will be called.

Refactoring also reorganizes code for greater execution efficiency, space consumption, or both.

Refactoring to fix the algorithm

Your refactoring will fix the algorithm and the matrix-passing syntax, and finish reformatting and documenting the code along the way.

1. Implement Conway's rules correctly.

   Search online to learn the rules, and implement them properly. As you repair the implementation, refactor the code to reduce the amount of repetition of similar code. Turn repeated code into proper functions that can be called when appropriate.

   Consider merging functions that have a lot of duplication. Notice that the three `...Rule()` functions are very similar and are called sequentially in the while loop. These operations are not sequential and should be combined into one function that handles the cases of birth, death and survival. You can also factor out the neighbor counting into a function to simplify the rule computation. (See the *Redesigning the neighbors* section below.)

2. As you refactor, document the functions you revise and the ones you create.

3. Reformat the code for readability. Use either *space* (' ') or *TAB* ('\t') indentation characters consistently and do not mix them.

4. Declare parameters properly for passing a matrix. The compiler needs to know the number of columns in the matrix passed into the function; this means that you must specify the size of the second dimension, because the compiler is processing the parameter list left-to-right. This example shows the proper way to declare a matrix parameter for a function in C:

   ```
   void foo( int size, char matrix[][size] ) { ... }
   ```

   You will need to change your revised rule function to have proper parameter declarations, and update the calling code so that it calls the function properly.

   As part of this refactoring, you should replace all the *magic numbers* with appropriate symbols so that it is easy to change the grid size by changing one symbol's value. For example, `19` and `20` appear in *magically* in many places and should be replaced by a well-named symbol.

   If the matrix were non-square, you would need a rowsize and a columnsize parameter to specify the matrix dimensions before the matrix parameter itself.

5. Remake (compile and relink using `make`) and be sure there are no errors or warnings.

6. Add the file to the repository change list using `git add good-life.c` and issue `git commit` to save the code to the repository.

7. Test the code to see that the game rule implementation is correct.

   (You could temporarily change the size of the grid and the number of cycles the code runs to make testing easier. Be sure however, to change that back to the required size 20 and 100 cycles before you submit.)

   Continue your refactoring *edit, recompile and retest cycle* until the code properly implements the algorithm.

8. After the refactoring is finished, tested, documented, and version controlled (i.e. committed), issue the command `git log >revisions.txt` to save the version log. (This would overwrite any previous `revisions.txt` file you may have created in the current folder, but that's ok.)

9. It is now time to submit this activity.

Details: Redesigning the neighbors

The supplied code calculates the neighbors over and over, and it is incorrect at the boundaries of the life grid. What's needed is a common way to count neighbors of an arbitrary grid location, *and wrap around at the horizontal and vertical margins*. That way the game of life can have its life forms flow around from bottom to top and left to right and vice versa.

Given a location with (row, col) indices, you calculate their neighboring indices using row +/- 1 and col +/- 1 to get the 8 neighbors' locations. For example, if the grid location is (row 3, col 0), then the neighbors on the row above there are:

```
    (row 2, col -1) (row 2, col 0) and (row 2, col 1)
```

Clearly the negative index will go out of bounds, and this needs to wrap around to:

```
    (row 2, col 19) (row 2, col 0) and (row 2, col 1)
```

because the size of the grid is `20 X 20`.

Whenever a neighbor's row index is -1 or 20 (the width of the life grid), you know that the neighbor is on the top or bottom of the grid. In the case that the row is -1, then you can adjust the value to be 19, and when row is 20, you adjust it to 0 to get the row index wrapped around. The situation works similarly for the column index of the life grid.

You should factor this logic into a function that can compute the neighbors and count whether they have a `*` and are organisms or a `(space)` indicating no organism.

Activity 2 Submission

Submit your results for this activity with this `try` command:

```
    try grd-243 hw3-2 good-life.c revisions.txt [readme.txt]
```

The program must run without crashing and print 100 generations of the game of life. The initial generation counts as one generation, numbered as generation 0.

The `readme.txt` is an optional file you can submit to provide additional comments.

Activity 3: Redesigning to use Cursor-controlled Output

Change to the `act3` directory that `get` created, and copy the `good-life.c` file from your previous activity directory to the `act3` directory.

Instead of printing each generation after the previous one, you now must use the supplied `display.c` code to control the cursor's position on the terminal and overwrite the next generation on top of the previous generation.

1. Use cursor functions to make each generation overwrite previous ones.

   Change the code so that it uses the supplied `display.c` code to position the cursor so that it *overwrites the output in the same place on the screen for each generation of the game*. The display of each generation:

   - Draws the game board starting at the upper left of the terminal window.

  - Prints the count of the generation displayed on the line under the last line of the game board. (This is already done in the supplied code, but you might want to revise it.)

2. Revise the code to make the generating loop into an infinite loop that uses cursor control functions (see `display.h`) to overwrite each generation on top of the previous generation.

   The only way to terminate a program with an infinite loop will be by an interrupt (`CONTROL-C`) or a `kill` command (see the `man kill` page for information).

3. Introduce a delay between displaying each generation.

   Because the program is now running with an infinite loop, the program's loop needs to have a *delay* between the display of each displayed generation of the game. The `usleep` function can provide this delay; here is a suitable delay:

   ```
   usleep( 81000 );
   ```

   The system header file `unistd.h` declares this function; however, to get that declaration, you must add

   ```
   #define _BSD_SOURCE
   ```

   at the beginning of your source file before any of your other `#include` statements.

4. Find and fix any remaining warnings.

5. Add proper, publishable function documentation to the code. You should use either the `/// ...` or the `/** ... */` forms.

6. Validate your implementation of Conway's rules in final testing.

7. Add the file to the repository change list using `git add good-life.c` and issue `git commit` to save the code to the repository.

8. After testing and documenting are finished, issue the command `git log > revisions.txt` to save the version log.

Notes on Cursor Control and `display.c` for Activity 3

The display code provides functions to clear the terminal window, position the cursor at a specified row and column, and put a character to the terminal. The module addresses the terminal as a matrix of characters. While the column values are 0-based indices, the row values are 1-based. That means index of the first row of the screen is '1', not '0', but the index of the first column of the screen is '0'.

The `set_cur_pos` function will position the cursor to the location specified by the arguments. The `put` function will output a single character at the location of the cursor, *and move the cursor one position to the right as a side effect*.

Note that the program must **clear the terminal window screen only once, after it receives the number of**

organisms. The supplied module source `display.c` file provides that functionality. It is a design error to clear the terminal within the loop because that defeats the purpose of the cursor-controlled output.

With cursor control in place, here is a sample display of the expected output of one generation: sample-life.txt .

Extra Credit Options for Activity 3

The following are optional enhancements to the solution. Each is worth 3%; do all successfully and earn 9% extra credit. (It is possible to get a score greater than 100% as a result.)

You must supply the `readme.txt` file to document your extra features if you wish to get full credit for the extra work.

1. Override the size of the grid if the user provides a size as a command line argument. For example, entering this command: `good-life 27` would change the game board size from the default *20 by 20* to *27 by 27* characters. Note: this is an **override**; the program must also run using the default value with no supplied command line arguments.
2. Write code to produce one or more *oscillators*, *gliders* or *spaceships* as discussed in Conway's Game of Life.
3. Add code to produce a random birth of a cell.

Activity 3 Submission

Submit your results for this activity with this `try` command:

```
try grd-243 hw3-3 good-life.c revisions.txt [readme.txt]
```

Remember to supply the `readme.txt` if you did the extra credit.

Grading

The distribution for grading the assignment is shown below. The successful, documented implementation of optional, extra credit items would add to your score.

1. 30% Activity 1: **Debugging**
2. 35% Activity 2: **Refactoring Code and Fixing Warnings**

   Code runs the correct algorithm in a fixed size loop and terminates. Style and documentation count for 5% of this activity.
3. 35% Activity 3: **Redesigning to use Cursor-controlled Output**

   Code runs the correct algorithm in an infinite loop and displays generations using cursor-controlled overwrites. Style and documentation count for 5% of this activity.

Revisions:
Mon Feb 6 18:04:45 EST 2017 (original version)