Lab 04 — Part B: Intro To GDB and Valgrind

Assigned: 2019-02-12 20:30:00

Due: 2019-02-13 23:59:00

Instructions:

– Written portions of this assignment are submitted via Canvas. All derivations, images, graphs, and tables are to be included in this document. Handwritten solutions will receive zero credit.

– Code portions of this assignment are submitted via `code.vt.edu`. Source code must be in a private repository, to which the CMDA 3634 instructors must have access.

Deliverables: For this assignment, you are to submit the following:

1. (Canvas) `lab04b_writeup.pdf`: A pdf generated on Overleaf with your writeup
2. (GitLab) Your code solutions.

Collaboration: This assignment is to be completed by yourself, however, you make seek assistance from your classmates. In your submission you must indicate from whom you received assistance.

Honor Code: By submitting this assignment, you acknowledge that you have adhered to the Virginia Tech Honor Code and attest to the following:

> I have neither given nor received unauthorized assistance on this assignment. The work I am presenting is ultimately my own.

Resources

- More on Git:
  - `https://www.taniarascia.com/getting-started-with-git/`
  - `http://rogerdudler.github.io/git-guide/`
  - More detailed instructions on generating SSH keys `https://code.vt.edu/help/ssh/README.md`
  - Basics of Git: `https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control`
  - Some information about working with remotes in Git: `https://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes`
- Simple intro to Makefiles:
  - `https://www.gnu.org/software/make/manual/html_node/Introduction.html`
- More on LaTeX:
  - A gentle introduction to Latex from Overleaf: `https://www.overleaf.com/learn/latex/Free_online_introduction_to_LaTeX_(part_1)`
  - More info on inserting figures: `https://www.overleaf.com/learn/latex/Inserting_Images`
- More on C programming:
  - The C Programming Language by Kernighan and Ritchie `http://hikage.freeshell.org/books/theCprogrammingLanguage.pdf`
  - `https://www.geeksforgeeks.org/c-programming-language/`
- More on GDB:
  - `http://www.cprogramming.com/gdb.html`
  - `https://betterexplained.com/articles/debugging-with-gdb/`
  - `https://www.maketecheasier.com/profile-c-program-linux-using-gprof/`

Tasks

1. **Setup** your coding environment.
   (a) Pull the lab materials from the upstream repository.
   (b) Examine the history to see what new files have been added by the instructors.
   (c) Push your local branch to the origin repository to add the new changes.

2. **GDB Testing:** All parts of this problem will use source code in the `gdb` subfolder.
   (a) `matrixProd.c` contains functions which are used to perform matrix vector multiplication. `matrixTest.c` tests this implementation using a few toy examples.
      - There are 5 bugs in this code, 4 of which materially impact the results of the program.
      - Some of these occur in the test cases themselves!
      - First make sure you understand what each operation is supposed to do and what is being tested.

   (b) Modify the rule `matrixText` in the makefile so we can use `gdb`. Replace `-O3` with `-g`. Use

      ```
      > make matrixTest
      ```

      to build your code and run it using `./bin/matrixTest`.

   (c) The test cases you just ran suggest some problems with `dotProd`. Use `gdb` to debug it.
      - Invoke `gdb`
        ```
        > gdb ./matrixTest
        ```
      - Set a breakpoint on the `dotProd` function
        ```
        > break dotProd
        ```
      - Tell `gdb` to execute the program until it encounters the breakpoint
        ```
        > run
        ```
      - Step through the program one line at a time by running the step command
        ```
        > step
        ```
      - If you can't see the error, set some variables (e.g., `i`) to show their contents after each step, for example,
        ```
        > display i
        ```

   (d) When you find the bug, and fix it, add and commit your change, indicating what the bug was and what you did to fix it.

   (e) Note, after you fix the bug, you may need to remove the existing binary using

      ```
      > make clean
      ```

      before it will let you recompile the code

   (f) Use `gdb` to debug the rest of the code. Make sure the code works correctly, your tests pass, and that they properly test what the user intended.

   (g) Dump the test output to `gdbOutput.txt`

      ```
      > ./matrixTest > gdbOutput.txt
      ```

3. **Valgrind Testing:** All parts of this problem will use source code in the `valgrind` subfolder.

   (a) There are four independent programs in the `valgrind` subfolder, each of which abuses memory in some way. Open and read each program, and then try building and running them. To do this, you will need the build rules

   - `make fibonacci`
   - `make pascal`
   - `make matrix_sum`
   - `make rotate_vector`

   You will not see any output (I removed print statements from each program), but they should all run successfully.

   (b) Next, modify the four build rules listed above to compile with the `-g` flag and run the following command to install `valgrind`

   ```
   > sudo apt install valgrind
   ```

   (c) Run each program using valgrind to look for memory issues. Look at the line numbers to help indicate where the errors might be. Find and fix bugs until valgrind gives a clean report for each program. You may use `gdb` if it helps.

   (d) When you find a bug and fix it, add and commit your change, indicating what the bug was and what you did to fix it.

   (e) Once `valgrind` indicates that there are no errors, uncomment the print statements in each program and save the results to four files `fibOutput.txt`, `pasOutput.txt`, `sumOutput.txt`, `rotOutput.txt`.

4. **Write** your report in LaTeX. Answer the following questions in the report file. A blank document, `lab_report_04_b.tex` is in the `reports/` directory.

   - **Note:** When you are asked to list the buggy code and your fix, the following will be helpful. If you committed the bugfixes with git correctly, you can use the git history to view the changes. Use

     ```
     > git log
     ```

     to find the commit for the bug you want to view. Look through the log (the most recent change is at the top) and find the commit with your fix. Copy the SHA hash (the 40 character random string of numbers and letters; for example `1fa740a5ca73fc4c54eafa5d5180aa3ff24b3216`) for the commit of interest, and run

     ```
     > git show <SHA hash>
     ```

     to see the changes made in that commit. You often do not need the entire hash – often, the first 8-12 characters is sufficient to uniquely identify the commit. For example, you can use

     ```
     > git show 1fa740a5
     ```

     in place of

     ```
     > git show 1fa740a5ca73fc4c54eafa5d5180aa3ff24b3216
     ```

   (a) `matrixTest`
   - Use the `listings` package to include your `matrixTest` output in the pdf.
   - For each bug, use the `listings` package to display the original line of code with the error, as well as the fix. Describe the error.

   (b) `fibonacci`
   - Use the `listings` package to include your `fibonacci` output in the pdf.
   - For each bug, use the `listings` package to display the original line of code with the error, as well as the fix. Describe the error.

   (c) `pascal`
   - Use the `listings` package to include your `pascal` output in the pdf.
   - For each bug, use the `listings` package to display the original line of code with the error, as well as the fix. Describe the error.

   (d) `array_sum`
   - Use the `listings` package to include your `array_sum` output in the pdf.
   - For each bug, use the `listings` package to display the original line of code with the error, as well as the fix. Describe the error.

   (e) `rotate_vector`
   - Use the `listings` package to include your `rotate_vector` output in the pdf.
   - For each bug, use the `listings` package to display the original line of code with the error, as well as the fix. Describe the error.