Assignment 6

`Answers.txt` Due: 10:00 PM, Wednesday, October 15, 2003
Final Code Due: 10:00 PM, Thursday, October 16, 2003

The purpose of this assignment is to familiarize you with interactive debuggers and debugging techniques.

Preliminaries

This assignment is written for use with the `ddd` front end to the Gnu `gdb` debugger. It is not possible to do the written part of assignment with a different debugger, primarily because minor debugger variations make grading impossible. Also, it is probably not possible to do the assignment by using `Putty` on a Windows machine to connect to turing, because `ddd` requires a display that supports the X-Window system. You can use a VNC client, however, assuming you know how to use `ssh` and `vncserver`.

Before You Begin

Set up your assignment directory on turing by executing

```
cs70checkout cs70ass6
```

This command should check out the following files and directories:

```
cs70ass6/SCHEDULE
cs70ass6/HISTORY
cs70ass6/Answers.txt
cs70ass6/Makefile
cs70ass6/shuffle.cpp
cs70ass6/shuffle-private.hpp
cs70ass6/lineshuffler.hpp
cs70ass6/lineshuffler.cpp
cs70ass6/input.txt
cs70ass6/random.hpp
cs70ass6/random.cpp
cs70ass6/linearray.hpp
cs70ass6/linearray.cpp
```

Overview

In this assignment, we return to the simple shuffling program we saw in Assignment 1. Since you last saw it, the program has been modified to make it more useful. It has been extended so that it can either read from standard input (rather like it used to, except that it produces no prompts) or read its input from files. It also has an option to number the lines it shuffles. Finally, some of the improvements that we noted were

possible in Assignment 1 have been applied (the shuffling process itself is now $O(n)$ and not $O(n^2)$ [assuming that lines are less than 80 columns]).

For example, we can pick a random turing user and find out what they are doing by typing

```
unix% w | tail +2 | ./shuffle | head -1
```

or look at a mixed up version of all our C++ source by typing

```
unix% ./shuffle *.cpp *.hpp | more
```

In theory, at least....

In practice, the program seems to be broken. It always seems to die with a segmentation fault when no filenames are given, and, when it does work, the output doesn't seem quite right. In this assignment you will track down several bugs in the code using a visual debugger, `ddd`.

Written Component

Your answers to this component are to be submitted as a file, `Answers.txt`. This file should be a plain text file (following the usual rules about line length, etc.). When you have completed this portion assignment, submit your `Answers.txt` file by typing `cs70submit Answers.txt`.

This assignment is divided into several sections. You should perform the sections in order. However, each section is independent, so that you can log out and come back later, or go back to the start of a section if you think you have gotten things out of whack. If you log out and come back, you will need to repeat the steps listed in Section B.

There are also tips in the margins.

The procedure also includes comments like this one that provide useful background information, but can be skipped if you are already familiar with Unix and gdb/ddd.

A. Preparation

- Check out the source and compile it by typing `make`.

B. Setup

- Make sure that your preferred editor is set properly by typing

  ```
  echo $EDITOR
  ```

  If you use `bash` instead of `tcsh` as your shell, use `export EDITOR=vi`

  If the response is the name of some other editor, fix it by typing

  ```
  setenv EDITOR emacs
  ```

  (or, if `emacs` isn't your favorite editor, `vi`, pico, or whatever)

- Start the ddd debugger from the command line by typing `ddd shuffle`.

> In addition to the version of ddd installed on `turing`, the versions of gdb and ddd provided in the most Linux distributions are probably okay to use in this assignment. Mac OS X comes with gdb, but it does not provide ddd; instead Project Builder provides graphical interface to gdb, but one that is not compatible with the written portion of this assignment.

C. Basic ddd

- There are several ways to get the program started. We can't use the obvious Run button in this case, because we want to give the program some command-line arguments.Instead, explore the menus to find a way to run the program that will pop up a dialog box asking for arguments. Specify `-d` in the dialog box and let the program run.

> The Run button is in the Command Tool window, a small window with about sixteen buttons for commonly used ddd functions. If you don't see this window, it may be hidden or you may have accidentally closed it. You can make it appear again by choosing the View/Command Tool... menu option.

W1. What happens? (Be sure to include a description of all the output that shows up in the bottom pane following the `Starting program` line. You will probably have to scroll the bottom pane back a bit.)

> In general, the bottom pane is worth watching; it is the interface to the debugger gdb—gdb does all the actual debugging operations, ddd just provides a friendly graphical interface to gdb . When you don't need the graphical features of ddd (for example when you only need a stack backtrace), you can work with gdb directly. Most button clicks in the interface get converted into textual commands that are automatically typed into this bottom window.

W2. What is a "Segmentaton Fault" (usually called a "segfault"), and what generally causes one? (You *can* ask anyone in CS except fellow members of CS 70 for an answer to this question if you do not know.)

- You can find out the more about the context of the problem by getting a backtrace. Select the Status/Backtrace... menu item. Click Up, observe how the display changes, and then click Down.

> You can also get a backtrace (without the GUI) by typing `backtrace` or `bt` into the lower pane. Many bugs are found simply by running the program under gdb (i.e., without the ddd graphical interface) and getting a backtrace. A backtrace is often enough information to figure out what went wrong.

W3. What is the backtrace telling you?

Be sure to include both sets of brackets.

- It seems likely that the cause of the problem is the reference to `argv[argNo][0]`. Select this entire expression with your mouse. Click the Display button on the toolbar to display it.

W4. What is shown in the top pane?

- Hmm, that's odd. The bottom pane contains a clue: `0x0` is the value of a `NULL` pointer. Let's try looking at `argv[argNo]`.

W5. What is the value of `argv[argNo]`?

You may need to scroll the code down a little to find the line you were looking at.

- Select `argv[argNo]` in the middle pane and click Display.
- That's not good.

W6. What are the values of `argNo` and `argc`?

> Another, easier, way to display simple variables is to rest the mouse pointer on them.

- It turns out that `argc` is the size of `argv`. As you know, if a C++ array `foo` has 5 elements, it's illegal to refer to `foo[5]`.

W7. Why did the program violate that rule?

- Click on the `Edit` button in `ddd`'s Command Tool to bring up your editor, and fix the bug. Exit your editor and click the Make button. Then click the green Run button to run `shuffle` again with the same arguments. A large number should appear in the bottom window.

Don't worry about what this number means.

You can to use cut-and-paste to make sure you copy the number accurately.

W8. What is the value of this number?

- If you've done everything correctly so far, the program is now waiting for you to type something. Click in any of the three panes in the main `ddd` window, and type Control-D on the keyboard.

> Control-D is the standard Unix method of sending an "end of file" (EOF) to a program when it is reading from the terminal. In this case, EOF will cause the program to exit. (If you wanted to provide some other kind of input, you could have typed it instead.)

- Finally, clean up the data display. Click on either of the displays in the top pane so that it is completely highlighted, and then click the Undisp button on the toolbar. Repeat for the other display. You should now be back to two panes.

C. Breakpoints and Stepping

- Run the program with the arguments `/foo/bar /baz/zap`. Giving these arguments should cause the program to try to open two nonexistent files by those names.

W9. What output do you see in the bottom window? (Be sure to scroll back to make sure you see everything after the `(gdb)` prompt.)

- It seems a bit odd (or at least it *should* seem odd) that only the second missing file is reported. The program's attempt to access the files is done in a function called `doShuffling`. You can find a function quickly by typing its name in the small text box on the toolbar (the one labeled `():`[ ]) and then clicking [Find].
- Without doing anything else, click the little stop sign to set a *breakpoint* at the beginning of the function.

  Watch the lower pane to see the `gdb` command used to set the breakpoint. You can type your own `break` commands to set breakpoints, and doing so is often more convenient than doing so with the point-and-click `ddd` interface.

- Run the program again.

W10. What happened when you ran the program?

- The [Next] button is a good way to work your way through the function one line at a time. Hit [Next] until you get to the statement *ifstream* `nextFile(argv[argNo]);`. The constructor argument is the name of the file to open.

W11. What file is it trying to open?

- That's odd. Hit the [Up] button to see the line that called the function. Type `argv[argNo]` into the box on the toolbar and click on the toolbar's [Display] icon.

W12. What is the value of `argv[argNo]`?

- Hit the [Down] button to get back to where we were.

W13. Why is the value of `argv[argNo]` different?

- The cause of this bug should be fairly obvious. Fix it, recompile, and run the program again.

W14. What happens when you rerun the program?

- Since you're paranoid (or should be), use the [Next] button a few times to get to the place where the bug happened last time. There seems to be a bug in `ddd`, so if the value of `argv[argNo]` isn't automatically displayed, try hitting [Up] and then [Down] to make it show up.

W15. What is the value of `argv[argNo]`?

- Hit the [Cont] ("continue") button to continue running the program at full speed. In the bottom window, you should see two error messages (corresponding to the two files the program could not find).

- At this point, it may be easiest to exit and restart `ddd` to get rid of your breakpoints and display commands. Alternatively, you could use the Source/Breakpoints... and Data/Displays... menu items to achieve the same effect.

E. More on Stepping and Breakpointing

- Use File/Open Source... to bring up a list of the source files that make up the program. Most of them are system files that aren't of interest to us, so find `lineshuffler.cpp` and open it.
- Find the function named LineShuffler::removeLine, by scrolling or searching. Then find the line rnd_.next(count);, which calls Random::next.
- You can set a breakpoint on a line by clicking at the *beginning* of that line and then on the stop sign. Set a breakpoint on the line containing the rnd_.next(count) call.
- Run the program again, this time giving the arguments `-s 0 shuffle.cpp`.
- When the program stops at the breakpoint, click the Next button.

W16. What happened when you clicked Next?

- Rerun the program with the same arguments. This time, when the breakpoint is hit, click the Step button.

W17. What happened when you clicked Step?

- Use Source/Breakpoints... to disable or delete the breakpoint in removeLine.

F. More on Data Display

- Run the program with the arguments `-s 1 input.txt`.

W18. Is the output random? (You may wish to compare the output to the contents of the input file.)

- Set a breakpoint in the LineShuffler::removeLine function and run the program again.
- When the program stops at the breakpoint, display lines_.

Yes, it really is very ugly.

W19. What is the value of this variable?

- Double click on the value of lines_.

W20. Describe what happened.

- If you prefer, drag the display boxes around to rearrange them. Then select the new box and choose Show All from the pulldown Show menu on the toolbar.

W21. What is the value of _M_start?

> As we look into an object belonging to the *vector* class, we see a negative side to data abstraction (at least with the tools we are using). We know a little about how *vector*s work in theory, but we have no details about their internal implementation. The debugger will allow us to delve into the private data members of these classes (because they too were compiled with debugging information), but it doesn't tell us anything about what these private date members *mean*. Thankfully, it is fairly easy to work out what the private data members of the *vector* mean, but it isn't nearly as easy for other STL types, or other classes in general. In fact, the code for this assignment uses a *vector* of C-style strings (i.e., NUL-terminated arrays of characters) rather than C++ *string*s simply because the former is so much easier to "see" in a debugger.

- Select _M_start and choose Display/Other... on the toolbar (you get to this menu by clicking the Display button and holding it down). Add an asterisk at the beginning of the expression in the dialog box, and add `@16` at the end of the expression (so that it reads `*this->lines_._M_start @16`) before you click the Display button in the dialog box.[1]

W22. What is displayed?

You may have to scroll right a bit.

- Use the Next button to step through the function until it returns.

W23. How do the values in the array change?

W24. What is wrong with the code and how should it be fixed? (Hint: No *new* code needs to be written.)

- Clear all the data displays again.

G. Finishing Up

- Fix the bug you identified in question W24.
- Make sure you have a breakpoint at the start of LineShuffler::removeLine, and nowhere else. If you're starting fresh, you'll have to add it. Run the program again (with the arguments `-s 1 input.txt`).

W25. Single step until after the index variable has been initialized. What is its value?

- Display the array again, this time by typing the command

```
graph display *lines_._M_start @ (lines_._M_finish - lines_._M_start)
```

into the bottom pane (the one with the `(gdb)` prompt).

1. If you don't mind the extra typing, you get slightly better display if you use the expression `*this->lines_._M_start @ (lines_._M_finish - lines_._M_start)` in the dialog box. This expression adjusts the size of the display to the size of the vector, rather than fixing it at 16.

- Make sure that you have all the elements of line_ vector displayed. Resize your display window so that you can see all the contents of the box.
- Use Next to step until you have gone past the call to the swap function.

W26. What has changed in the data directory?

- Use Next to step until you have gone past the call to pop_back.

W27. What has changed?

- Delete all breakpoints and run the program one last time, with the single argument `input.txt`. You will see 16 words scroll by in the window at the bottom of the screen.

W28. In order, what are they? (You may use cut-and-paste, and you don't have to list them one per line.)

- Submit your `Answers.txt` file.

Coding Component

In this portion of the assignment, we will begin with the debugged program from the written section.

The *LineShuffler* class currently uses a *vector<const char*>* to store C-style NUL-terminated strings representing the lines of the file. The author intended to replace this vector with a custom class, *LineArray*.

C1. Edit the code for `lineshuffler.hpp` and change line 77 from *vector<const char*>* lines_ to *LineArray* lines_, and then run `make`.

> You will find that when you run the program, it now crashes. If you run it under a debugger, you can get a backtrace but it may not be as useful as you would like. When code crashes inside system library code (especially the memory allocator), it may be that this is a symptom of earlier "damage", rather than the root cause of the bug.

C2. All the bugs are in `linearray.cpp`. Use any and all debugging techniques at your disposal to find the bugs and fix them. Your changes to the code should as non-invasive as possible—it is not acceptable to simply reimplement the functionality from scratch. (You can add more comments to the code, however.)

> A good approach, at least initially, is to use assert statements. (Hint: What assertions can you make regarding size_ and capacity_?)

C3. Describe your debugging experience in a file named `HISTORY`. State what the bugs in the *LineArray* class were, what debugging techniques you used to find them, and how you fixed them. This file can be fairly brief.

C4. Submit your revised program and your `HISTORY` file.