lab_gdb Gruesome GDB

Due: Sunday, September 27 at 11:59 PM

Doxygen for lab_gdb

Assignment Description

Alice is writing a card game program. She's working on the representation of a deck of cards, and she's decided to use a singly linked list. She's built two functions to manipulate the deck, reverse and shuffle. After struggling through compiler errors for hours, everything finally compiles! Alice rushes to test her program with some simple numbers.

```
Segmentation Fault
```

Ouch. What does she do now? She has to debug her code.

Alice has a few options to debug her code. She could open it up and see if she can spot the errors, but that's really difficult to do, especially with segmentation faults. She could try inserting `cout` statements, but that requires adding and removing lots of lines of code for each bug. She could try using ASAN or Valgrind, but that's most helpful with memory errors and doesn't give much information about logical bugs. Her last option is you! She's called on your excellent debugging skills for help, and she's asked you to use a new debugger that she's heard of: GDB.

Getting Started with GDB

To launch your program using gdb, run the following command:

```
gdb [program name]
```
TERMINAL

To run your program with optional command line arguments:

```
(gdb) run [arguments]
```

Note

Throughout the lab, we'll use the notation

```
(gdb) command...
```

to indicate that the `command` should be run from within GDB.

Tip

GDB will provide several helpful features. First, it will output similar debugging information as Valgrind upon errors such as segmentation faults. Second, and more important, it allows you to stop the program execution, move around, and view the state of the running program at any point in time.

To do that, we will use the following common commands (see more details in the slides). We'll also define the abbreviations of these commands, so you don't have to type the full names of these commands when you want to use them.

- **Walking through your code.**
    - `break [file:line number]`
        - *Example usage*: `break list.cpp:40`
        - Create a breakpoint at the specified line. This will stop the program's execution when it is being ran. (See `run`).
        - When your program is stopped (by a previous use of `break`) in a certain file, `break n` will create a breakpoint at line `n` in that same file.
        - **Note**: There are other variations on how to use `break` here. One variation is breaking at a function belonging to a class. *Example*: `break List<int>::print`. We use this variation in this lab.
        - Abbreviation: `b`. *Example usage*: `b list.cpp:40`
    - `clear [file:line number]`
        - Removes a breakpoint designated by `break`.
    - `run (arguments)`
        - Runs the program, starting from the main function.
        - Abbreviation: `r`.
    - `list`
        - Shows the next few lines where the program is stopped.
    - `next`
        - Continues to the next line executed. This does not enter any functions. (See `step` for this).
        - Abbreviation: `n`.
    - `step`
        - Continues to the next line executed. Unlike `next`, this will *step* into any proceeding functions
        - Abbreviation: `s`.

    - `finish`
        - Steps out of a function.
        - Abbreviation: `fin`.
    - `continue`
        - Continues the execution of the program after it's already started to run. `continue` is usually used after you hit a breakpoint.
        - Abbreviation: `c`.

- **Viewing the state of your code.**
    - `info args`
        - Shows the current arguments to the function.
        - If you are stopped within a class's function, the `this` variable will appear.
    - `info locals`
        - Shows the local variables in the current function.
    - `print [variable]`
        - Prints the value of a variable or expression. *Example*: `print foo(5)`
        - The functionality of `print` is usually superseded by `info locals` if you are looking to print local variables. But if you want to view object member variables, `print` is the way to go.
        - *Example*: `print list->head`. Or `print *integer_ptr`.
        - Abbreviation: `p`.

- **Other useful commands.**
    - `backtrace` (Mentioned below.)
    - `frame [n]`
    - `ctrl-l` (clears the screen)
    - `ctrl-a` (moves cursor to beginning of prompt)
    - `ctrl-e` (moves cursor to end of prompt)

Checking Out the Code

To check out your files for this lab, use the following command:

```
svn up
```
TERMINAL

The slides are available here.

For the list of files and their descriptions, see Doxygen for this lab. Doxygen is a nice way to view the documentation for the lab.

Linked Lists: First Tasks

In this lab, we will be using singly linked lists, very similar to the doubly linked lists in MP 3. Two

of the functions in this lab will be coming from your implementation of MP 3.1: `insertFront` and `clear`. If you have not yet completed these functions, take some time now to finish them.

`insertFront`

Copy over your implementation of `List<T>::insertFront` from MP 3.1 into the appropriate location in `list.cpp` for `lab_gdb`. Make sure your insert code works for a singly linked list!

To test this function, run these commands:

```
make
./lab_gdb front
```
TERMINAL

`clear`

Copy over your implementation of the private helper function `List<T>::clear` from MP 3.1 into `list.cpp` for this lab. Make sure your clear works for a singly linked list!

To test this function, run these commands:

```
make
valgrind ./lab_gdb front
```
TERMINAL

Note: In order to test this function, insertFront must be working correctly!

GDB: First Tasks

These tasks exist to help you learn how to navigate through code execution with GDB. Do these AFTER completing the `insertFront` function.

Print/Display

One of the most useful aspects of GDB is the ability to view variable values. In order to do that, you must stop code execution; here, we will use a breakpoint. The goal of this exercise is to find the address of the node with value 3 in a singly linked list.

1. Compile the code using make.

   ```
   make
   ```
   TERMINAL

2. Start gdb with the executable.

   ```
   gdb lab_gdb
   ```
   TERMINAL

3. Insert a breakpoint in the `print` function in `list_given.cpp`.

```
(gdb) break List<int>::print
```

4. Run the command with the argument `front`.

```
(gdb) run front
```

5. Display both the value of `curr` and the data held in that node.

```
(gdb) display curr->data
(gdb) display curr
```

6. Step through the code until you see the node with a value of 3, and note the address given by the value of `curr`.

```
(gdb) next
```

Backtrace

Another very valuable command in GDB is `backtrace`. This shows you the function stack at the current execution time. We will use this to find out how many recursive calls it takes to get to a node with value 42 in the reverse function.

Note: This task may be done before fixing the bugs in `reverse`.

1. Compile the code using make:

```
make
```
TERMINAL

2. Start gdb with the executable.

```
gdb lab_gdb
```
TERMINAL

3. Insert a breakpoint in the `reverse` helper function in `list.cpp`:

```
(gdb) break List<RGBAPixel>::reverse(List<RGBAPixel>::ListNode*, L
```

(all in one line).

4. Run the command with the argument `reverse`.

```
(gdb) run reverse
```

5. Condition the breakpoint to only stop if the current node has an `alpha` value of 42.

```
(gdb) condition 1 curr->data.alpha == 42
```

6. Continue execution. The code should stop again at your breakpoint.

```
(gdb) continue
```

7. Use `backtrace` to find the stack frame number of the first call to the `reverse` helper function!

```
(gdb) backtrace
```

Debugging Alice's Code:

To make and run Alice's code, type the following into your terminal:

```
make
./lab_gdb
```
TERMINAL

Alice's First Bug

As you can see, Alice's code does not work! The first error is logical, the `insertBack` function does not seem to work properly. `insertBack` should walk to the end of the list and then add the new node there.

Try running the code with GDB to find out what's going wrong. Use the following commands to set up a helpful breakpoint.

```
make
gdb lab_gdb
```
TERMINAL

```
(gdb) break List<int>::insertBack
(gdb) run back
(gdb) display this->length
(gdb) display this->head

// If head is not NULL
(gdb) print this->head->data
```

You should see the code stop just before it starts to insert new nodes. Step through the code and watch how the `length` and `head->data` change. Once you have a good idea of what's going wrong, change a few lines of code and run the program again. If you've successfully fixed the bug, you can move on to bug number 2!

To test the `insertBack` function, run the following commands:

```
make
./lab_gdb back
```
TERMINAL

Bug 2

Once you've fixed the first bug, you'll get a segfault. Running the code through GDB will identify the faulting line as in the `reverse` helper function. Usually, the first thing to do when you hit a segfault is to use `backtrace`. However, this is one example of a misleading stack trace. If you look closely, you'll see that this is not an instance of infinite recursion!

Walk through the code again with breakpoints, and watch how different variables change to fix the recursive `reverse` function.

Test the `reverse` function with the following commands:

```
make
./lab_gdb reverse
```
TERMINAL

Bug 3

Only one more bug to go! This time, we've got another strange logical bug. Alice's shuffle function should cut the deck (or list) in half, and then interleave the cards (or nodes). (See here for more information on perfect shuffles.) Again, this function is not working properly. Use GDB to step through the code and see what is broken.

Test the shuffle function with the following commands:

```
make
./lab_gdb shuffle
```
TERMINAL

Checking Your Output

Once you think everything is working, you can run monad with the provided tests:

```
./monad lab_gdb
```
TERMINAL

Committing Your Code

Commit your changes by using the following command:

```
svn commit -m "lab_gdb submission"
```
TERMINAL

Grading Information

The following files are used in grading:

- `list.cpp`

All other files including any testing files you have added will not be used for grading.

Additional Resources

- Quick guide to GDB: http://beej.us/guide/bggdb/.