CSCI 262 Data Structures

Spring 2019

Lab 3: Debug

(Back to Labs)

Goals

- Practice using the debugger

Overview

This lab will take you through some simple uses of the CLion debugger, applied to a somewhat silly program based on a previous week's lab. As usual, we ask that you work with a partner or a team. If you end up solo, please raise your hand and we'll pair you with someone else (or make a three-person team if necessary). You are welcome to change partners throughout the semester!

Instructions

Step 0: preliminaries

Download lab03.zip, which contains a CLion project folder for this lab. Open or import the folder in CLion, and it should compile and run as is (although it won't yet work as intended!) The program we are building is called `rwg`, for Random Word Generator.

Note: If you are using the school computers for this lab, make sure you do the run.processes.with.pty fix given on the Help page of the course website. Just so you don't have to look it up, the instructions tell you: on the top menus in CLion, select Help > Find Action. Search for "registry" and hit enter. Uncheck the box that says `run.processes.with.pty`.

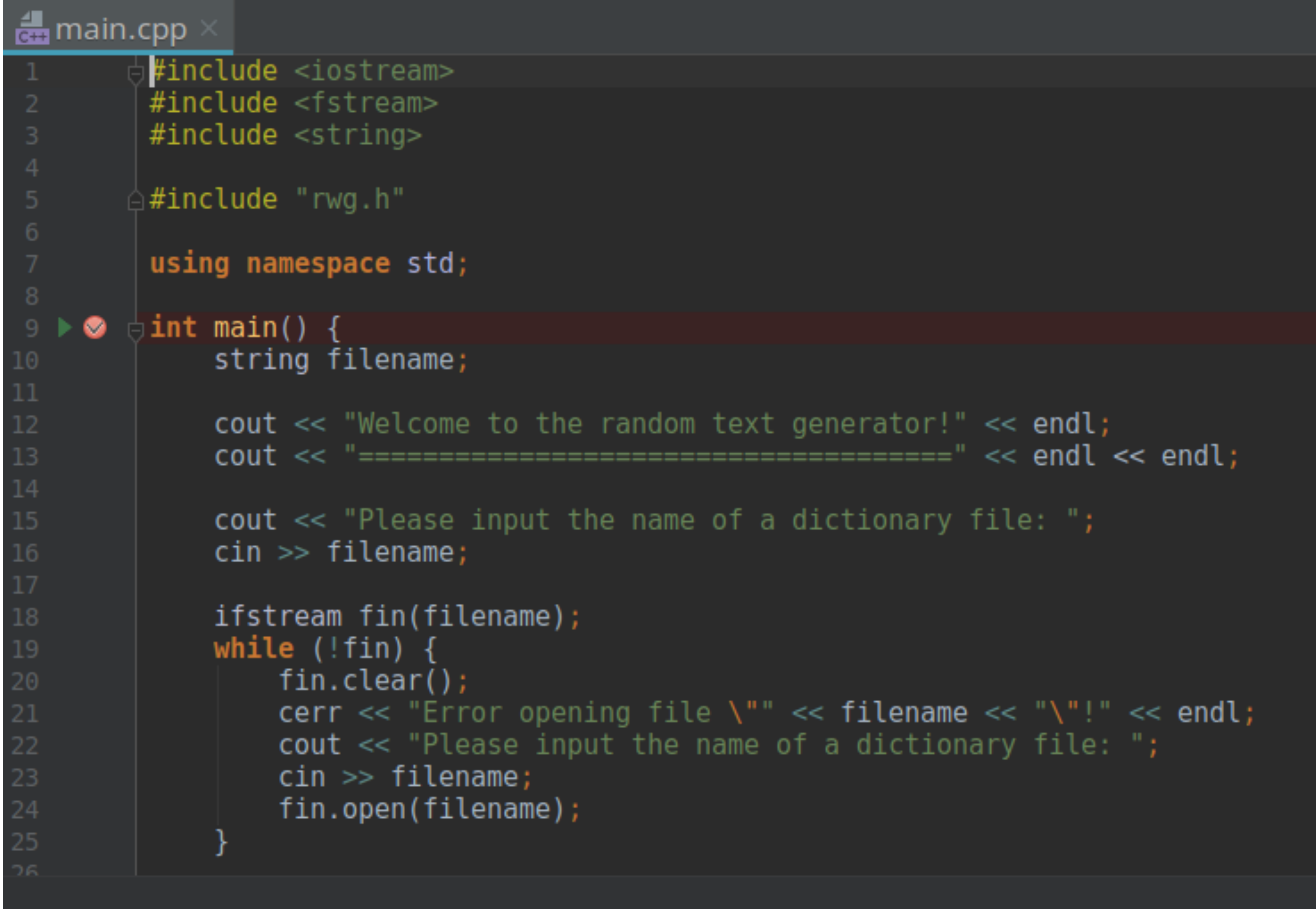
When working the Random Word Generator prompts the user for a word file, reads it in, performs a quick self-test (printing out a word of every length), then generates as many random words as the user requests. Here's a transcript of the program in action (with a very small words file - you'll want to use the provided dictionary.txt):

```
Welcome to the random text generator!
====================================

Please input the name of a dictionary file: somewords.txt
Performing self test:
length 2: aa
length 3: aah
length 4: aahs
length 5: aahed

Self test successful!

Please input number of words to generate: 10
Generating 10 words...
-------------------------------
rhus tut achy taus per haw agon dose pa coke
-------------------------------
Done.
```

Step 1: First bug

The `rwg` program, when working, should generate some random words from the provided dictionary. Give the program a try now, giving it the dictionary.txt file as input when prompted (don't forget to copy or move dictionary.txt to the cmake-build-debug directory).

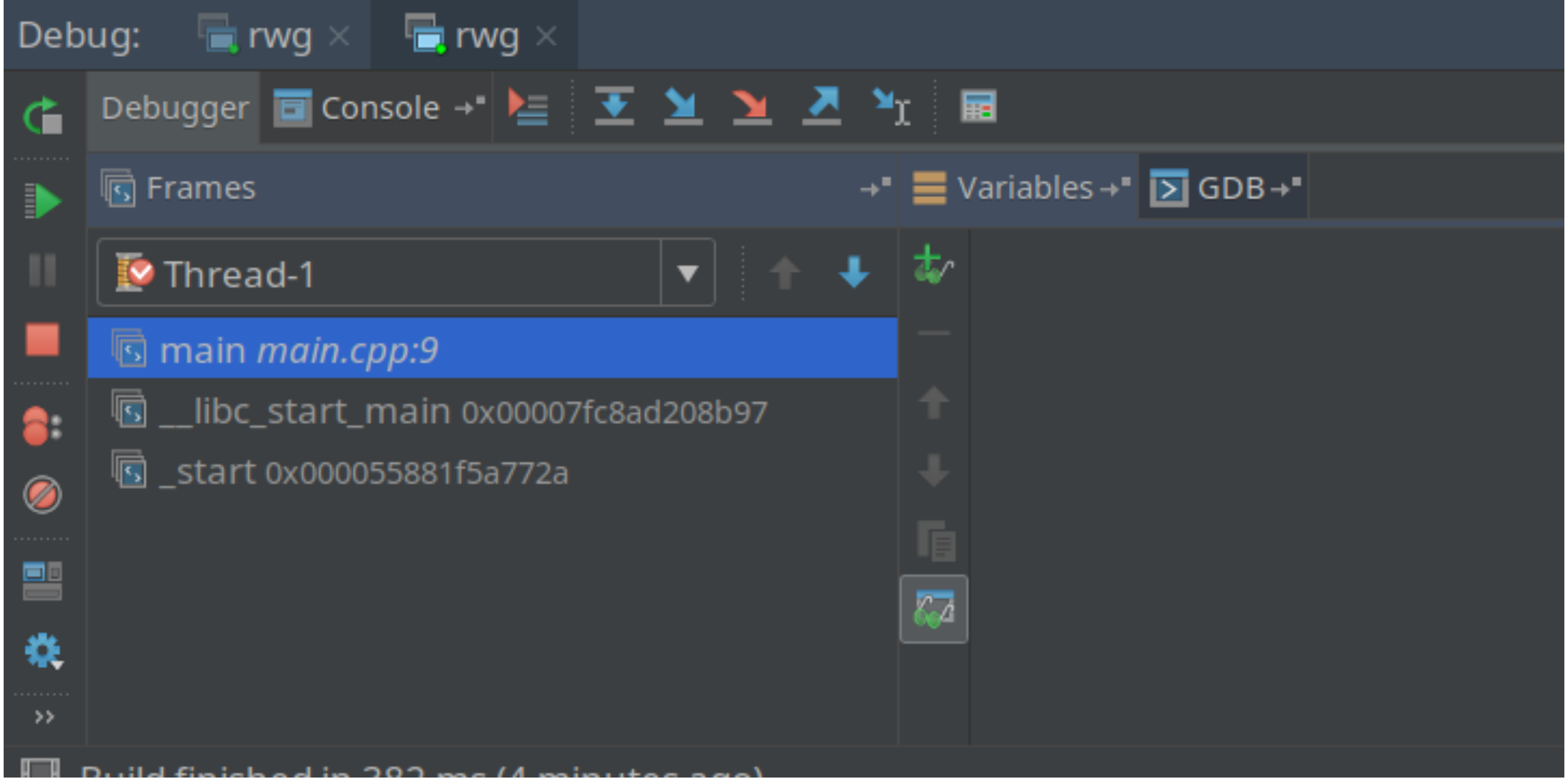
Okay, it obviously didn't work. You can probably look at the code and find the problem in a few minutes, but this lab is intended to help you with using **the debugger**, so let's find the bug that way. To start getting familiar with the debugger, let's set a *breakpoint* on `main()`, and start single-stepping through lines of code until the program exits. A breakpoint is simply some location in the code at which the running program will pause and give control to the debugger. To set the breakpoint, open main.cpp in the CLion editor and put your mouse above the bar to the left of the code window (between the actual code and the line numbers). If you click on this bar, you should see a red dot appear on the bar, indicating a breakpoint has been set at this location in the program. (See image below for what you should see in CLion).



main.cpp

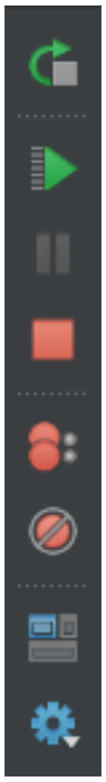
```cpp
#include <iostream>
#include <fstream>
#include <string>

#include "rwg.h"

using namespace std;

int main() {
    string filename;

    cout << "Welcome to the random text generator!" << endl;
    cout << "=====================================" << endl << endl;

    cout << "Please input the name of a dictionary file: ";
    cin >> filename;

    ifstream fin(filename);
    while (!fin) {
        fin.clear();
        cerr << "Error opening file \"" << filename << "\"!" << endl;
        cout << "Please input the name of a dictionary file: ";
        cin >> filename;
        fin.open(filename);
    }
```

Now we are ready to start running the program under the debugger. To do this, click the "bug" icon next to the usual run icon (or using the "Debug 'rwg'" command from the Run menu). You should see some new panes open up at the bottom of CLion in place of the usual console. (The console is still there, in a tab named "Console", next to the "Debugger" tab.) Here's what the new pane looks like on my screen:



main main.cpp:9
__libc_start_main 0x00007fc8ad208b97
_start 0x000055881f5a772a

Let's explore the debugger interface just a bit. On the left hand side there are buttons to let you restart the program, resume running the paused program, pause a running program (disabled at the moment, because we are paused at the first breakpoint), and do some other tasks like managing all of your breakpoints. Here's what those buttons look like on my screen:

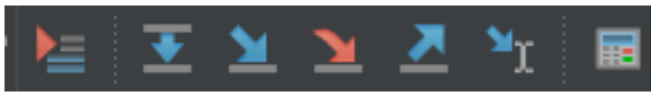


Along the top of the debugging pane is another set of buttons. We're going to use these buttons a lot, as they will let us step through the code in various ways and inspect the values of variables. Left to right, these buttons let you:

- Show the current execution point in the code editor
- Step to the next line in the current function (stepping *over* any function calls)
- Step to the next executed line of code (stepping *into* functions, if a function is called)
- Another step into command, this one just skips over some less interesting stuff

- Complete the current function call and stop at the next line in the calling function
- Resume running the code, but stop at the line in the editor where you have put the cursor
- Evaluate variables or expressions

Here's what those buttons look like on my screen:



If you hover over any of the buttons (on the left or on the top), a help text should appear to tell you what they do, and a longer explanation will appear at the bottom of the screen. You can also find out the keyboard shortcuts for each button this way.

Using the "Step Over" function (the downward pointing arrow icon), start stepping through `main()`. Note that each time you step, a blue bar in the code editor shows you where you are. Some additional info is shown in the panes below the stepping buttons. In the left pane, you can see what function and file (with line number) you are in. In the right pane, you can see the values of any local variables.

If you keep stepping, at some point you'll cross over the code that requests a filename from the user. At this point, you'll need to click on the "Console" tab to interact with the program - type in "dictionary.txt" and hit enter in the console to keep going. Click the "Debugger" tab to return to the debugger view of things.

Keep on stepping until you get to line 31 of main.cpp:

```
if (generator.is_empty()) {
```

If you look at the next line of code, you can see that the current line of code is critical. If the `generator.is_empty()` call returns `true`, the program will exit with the message you got previously when you ran the code. Let's see why `generator.is_empty()` is returning `true`, by stepping into the function, rather than over the next line. Either of the "Step Into" functions should work here.

The editor should now open up in the file rwg.cpp, with the blue bar on this line of code:

```
return _max_length == 0;
```

You should also note that the pane information on the left now shows you that you are in the method `rwg::is_empty`. Below that is the function `main`. You might now be able to tell that what you are seeing in the left pane is the function call stack (with the most recent stack frame shown on top)!

Notice also that the right pane is showing you different variables - now it is showing you what is available in the current frame of reference. At this time, all you can see is `this`, the pointer to the current `rwg` object. You can click on the triangle button next to `this` to dive into the variables held in the object.

Dive into `this`, and take a look at the value of `_max_length` - this the variable being tested in the current line of code in the editor. (You can alternately use the expression evaluator to find this out, but that is overkill for this situation. Save the expression evaluator for when you want to evaluate something more complex than a simple variable.)

Okay, clearly something is wrong in our code, because `_max_length` is zero, which means we're about to return `true`, which means our program is going to quit even though we supplied a valid dictionary file. You might want to spend some time at this point looking at `rwg.h` to see if you can figure out the broad outlines of how the `rwg` class is supposed to work. Or, I can just tell you that the `_max_length` member variable is supposed to be storing the length of the longest word stored in the map `_words`. If you try diving into the `_words` collection inside of `this`, you'll see that, in fact, we have lots of words of varying lengths stored. So somehow, the `_max_length` variable got set incorrectly. (Ignore the fact that this is probably the wrong way to check to see if we have words - just go with it.)

There are a lot of ways to proceed at this point, including ways to use the debugger, but time is short, so let's cut to the chase. The problem is in the `initialize` method of `rwg`. For practice, let's use the debugger to find the offending line of code. You can kill the currently running instance of the program using the red square button on the left of the debugger pane. Now, remove the breakpoint on `main()` (just click on the red dot to make it go away), and instead set a breakpoint on `rwg::initialize` in rwg.cpp (line 21).

Run the code again with this new breakpoint, then stepping through the lines in `initialize()` using the "Step Over" function. Be sure not to use the "Step Into" function, unless you want to step into some gnarly system code! (If you do so accidentally, you can jump back up using the "Step Out" function.) Single step until you hit line 38, and you should see the bug.

Fix the bug by editing line 38 of rwg.cpp **only** (don't fix any other bugs you happen to notice). You are now ready for the next step!

Step 2: Second bug

Try running the program again (normally, not under the debugger). You should get a little farther this time - it will start showing you words of different lengths as part of its "self test" routine. However, the program stops before finishing the test. You should get a message like

```
    terminate called after throwing an instance of 'std::out_of_range'
```

along with some less intelligible message text.

To the debugger! This time, we'll run the program under the debugger and let it stop the code for us, by stopping when the "out_of_range" exception occurs. However, we need to do one more step, first, to make sure the debugger stops when exceptions occur. Do Ctrl-Shift-F8 (or choose View Breakpoints from the Run menu) to bring up the Breakpoints dialog. Check the Enabled checkbox in the right-hand pane, if unchecked.

Now you can go ahead and remove the breakpoint on `rwg::initialize`, and start the debugger.

After putting in the dictionary filename, you should be taken into the debugger. If you aren't on the Debugger tab, click on it to see what is going on. In the function call stack view, you should now see a whole bunch of stuff. Most of that stuff isn't useful to us (maybe if you were a deep system programmer it would be). The interesting stuff is near the bottom, where we see our files mentioned: rwg.cpp and main.cpp.

CLion makes it easy to move around the function call stack by just clicking on the stack frame we are interested in - cool! Click on the stack frame that is labeled with the `rwg::self_test` method name. Check out the variables in the right pane - what is the value of the variable `i`, which is the size of word we are currently testing? Hm, do we have any words of this length? Looking at the code (you should be now looking at line 75 in rwg.cpp in the code editor), it seems likely that this is our problem. We are asking for `_words.at(i).at(0)` - either `i` is an invalid index in `_words`, or `0` is an invalid index into the vector of strings at `i`. (The documentation for the vector `at()` method states the reason for the exception: http://www.cplusplus.com/reference/vector/vector/at/.)

In the variables view, dive down into this (the current rwg object), then into _words, and see if you can figure out which of the indices is bad. Is there an entry for index 23? How many words are in the vector stored at index 23? (You should see that if you expand 22, you have some words, but not if you expand 23.) you How did this happen?

Fixing the `self_help` method may or may not be obvious, so I'll go ahead and give you the fix. Inside the `for` loop starting at line 73, you need to put an `if` block around the contents of the `for` loop. Here's the new loop (replacing lines 73 - 76):

```
        for (int i = _min_length; i <= _max_length; i++) {
                if (_words.at(i).size() > 0) {
                        cout << "length " << i << ": ";
                        cout << _words.at(i).at(0) << endl;
                }
        }
```

Step 3: Third bug

If you've fixed the bug properly and retry the program (normally), you should finally get to specify how many words you want to randomly generate. Try asking for 1000 words. Unless you get really lucky, you should get another error. The error you get may vary depending on what version your compiler is - you may get a "std::logic_error" exception, or you might get a "segmentation fault". Either way, the debugger is adept at halting the program when the problem occurs. So fire up the debugger once more! Run your program, and let the debugger tell you where the error occurs. At this point you know enough about the debugger to find the bug yourself - you don't need to do much other than examine the stack trace and examine variables and look at the code around the point where the error occurred in our code. See if you can find and fix the bug yourself - if not, a TA or classmate can help.

Wrapping Up

Obviously there are a lot of things that the debugger can do that we didn't cover here. More advanced/conditional breakpoints, display variables, probes, and much more. You'll have to explore some of this on your own, using whatever CLion documentation you can find, the internet, and your own buggy code :) Also, note there are other debuggers in other IDEs, all of which should have nearly identical features, but different ways of accessing them. If you are a command line user, take a look at the program `gdb` (if using g++ as your compiler), or `lldb` (if using clang). Hopefully we've at least given you the tools to get a running start.

To complete this assignment, please complete the quiz for this lab in Canvas. The quiz asks you to explain, in a few sentences, the nature and cause of the third (and hopefully final) bug in the random word generator.

NOTE: Just because we fixed all of them and the program runs, we can't be certain we got all the bugs!!! As Edsger W. Dijkstra once said, "Program testing can be used to show the presence of bugs, but never to show their absence!"