

Lab: Using the Code::Blocks Debugger

Steven Zeil

May 25, 2013

Contents

1	Setup	2
2	Finding the Location of a Crash	2
3	Debugger Commands	4
3.1	Observing Variables	5
3.2	Setting Breakpoints	6
3.3	Stepping Through Code	6

In this lab, we will look at the basic use of an automated debugger. Later lessons will cover overall *strategies* for debugging, but in this lab we will look at the mechanics of using supporting tools.

The program we will work with computes some basic statistics on plain text.

Enter some text, terminated by an empty line:

In this lab, we will look at the basic use of an automated debugger.

The program we will work with computes some basic statistics on plain text.

There are 27 words in 2 sentences.

The average word length is 4.3 characters.

1 Setup

1. Start Code::Blocks. Create a new Console Application project containing the files [words.h](#), [words.cpp](#), and [textstats.cpp](#).
2. Compile (build) the project and run it to be sure it is working.

2 Finding the Location of a Crash

Now, let's pretend that we were in the process of developing this program and had not yet worked out all the bugs.


1. First, we'll need a bug. In the function body for `initializeWords`, change the initialization of the `words` variable to:

```
words = NULL; // new string[MaxWords];
```

Recompile, and run the program again. The program will crash as soon as you finish entering the first line of input. (If you get a Windows pop-up box saying "A problem caused the program to stop working correctly" and offering a choice of buttons "Debug" and "Close Program", choose "Close Program". The "Debug" option it is offering will launch a different debugger that is not the one we want.)

Now, unfortunately this is all too typical a scenario during program development. Our code has crashed and we have no idea why. (OK, this time we know it's because of the change we made to `initializeWords`, but play along with me here. Normally, we would not know where the problem was.

2. Now let's run the program again, but this time do not run it via the blue arrow symbol or the Build->Run menu entry. Instead, go to the Debug menu and select "Start". The program will begin running. Enter some input as before and wait for the crash.

This time, a couple of things have changed. First, the message area of Code::Blocks has a list of Debug messages. At then end is the notification that a segmentation fault occurred. A *segmentation fault* is one of a *handful*  (??) of messages that are associated with inappropriate use of memory and addresses such as trying to fetch data via a NULL pointer or trying to access an array element with an index that is negative or too large for the array size.

Second, a box should have popped up offering to show you a "backtrace". A *backtrace* is a list that tells the address at which the program is stopped and, more importantly for us, the name of the function containing that address (in this case, the function name is "operator>>", a bit odd as function names go, but perfectly legitimate - it's the internal C++ name for the input operator that we normally write as simply >>), then the address (and name of the function) from which that function was called, then the address (and name of the function) from which *that* second function was called, and so on until we reach the function `main` that started off the whole execution.

3. Accept the offer to view the backtrace.

The best part is, double-clicking on one of those lines will take your Code::Blocks editor window straight to that line of code.

4. Try moving up and down the backtrace by double-clicking on those lines.
 - You may feel that you've been a bit cheated here. None of the lines mentioned in the backtrace is the one were we inserted our bug. That's because the location of the an actual bug and the place where we observe the effects of that bug are often far apart.
 - In this case, the backtrace is showing us where it observed the crash. Assuming that the function `operator>>'` from the `std` library is working correctly, we have to assume that we called it with bad data. The backtrace shows that we called it from a line inside `addText`. That line does indeed contain a use of the `>>` operator, so we might immediately start to wonder if there is something wrong either with the input stream we are reading from or with the location that we are trying to read into.

- In essence, we are in the same position as a doctor who sees a patient complaining of a pain in the chest. The pain is a symptom, and the doctor must now work from the symptom to investigate possible causes. So the doctor examines the patient's skin for signs of bruising, listens to the heart for evidence of a recent heart attack, probes the ribs and takes x-rays looking for fractures, etc.
- Debugging a program is much the same. We observe symptoms (an incorrect output or a crash) but have to investigate to find the actual cause. Strategies for conducting such investigations will be discussed later this semester.

You won't be able to go to the code for the function operator<>>, because that's actually part of the C++ std library, but you can see the portions of your own code that were directly involved in the crash. We now know exactly where the crash occurred.

3 Debugger Commands

There's a relatively common set of commands that most automated debuggers will provide:

Running and Stopping Expect a debugger to provide commands to run your program and to stop it. In addition, debuggers allow you to set , locations in the source code where the debugger will pause the program execution. The debugger will have some sort of "continue" command to restart execution that has paused at a breakpoint. (This is usually different from the main "run" command, which restarts the program from its beginning.)

Stepping Once the program is paused, instead of simply using "continue" to start it moving again, you can step through the code a statement or so at a time. There are 3 common variants on this idea:

- Step "in": If the current statement calls any functions, step into body of the function being called. Otherwise step to the next statement in this function body.
- Step "over": Regardless of whether or not the current statement calls any functions, step to the next statement in this function body.
- Finish: run to the end of this function, stopping once it returns back to its caller.

Examining the State Most debuggers have a way that you can type in a variable name (or sometimes an expression involving one or more variables), after which the debugger will show you the value of that variable (or expression) at the current moment in time for a paused program. Then you can step forward, repeat the process, and see if the value has changed.

Many debuggers will present you with a list of all the local variables declared in the current function, allowing you to simply select them to see their values. You may have to fall back on the first, more general, approach to see global or static variable values.

Some debuggers allow you to put variables into a list of "watched" variables, in which case that variable will be printed each time the program pauses.

Besides displaying variables, most debuggers can provide you with other important information about the program state. In particular, they allow you to move up and down in the call stack. We've already seen this with the Call Stack window that popped up in response to our request for a backtrace after a crash.

3.1 Observing Variables

- Using the Call Stack window, go to the line in `addText` associated with our crash.
- In the Debug menu, under "Debugging windows", select "Watches". You can use this window to examine the function arguments and local variables. Alas, this turns out to not be very useful, because our program is actually stopped inside the `operator<<` function, and we don't have access to the local variables inside there. (Hence the message "No symbol table info available.")

- It appears to be zero, which tells us that the crash occurred on the attempt to read the very first word.

In the editor window, though, double-click or drag your mouse to highlight `numWords`. Right-click on that and select "Watch 'numWords'". The current value of `numWords` appears in the Watches window.

- In the editor window, double-click or drag your mouse to highlight `words`. Right-click on that and select "Watch 'words'".
 - OK, the debugger did not like that request at all. Apparently it can't even display that value.
 - This is apparently a bit of a bug in the Code::Blocks debugger. It should have told us that `words` was null. But, even this behavior is enough to tell us that the value of `words` seems suspicious.

- On the Debug menu, select "Stop Debugger".

3.2 Setting Breakpoints

A **breakpoint** is a location in the source code where the debugger will pause the program execution.

9. Set a breakpoint by clicking, just to the right of the line number, on the first line of code in the function `addText`. A small red circle should appear to show that the breakpoint has been set. (You can remove a breakpoint by clicking on this circle).
10. Set another breakpoint on the first `cout <<` line within the function `main`.
11. Start the program running via the Debug menu.
 - Notice that execution pauses at the first breakpoint encountered, which happens to be the second one you set.
12. To get a better feel for where we are in the code, use the Debug menu, "Debugging Windows" entry to open up the "Call Stack" and "Watches" windows. Expand the list of function arguments local variables in the Watches window.
 - If words is still showing in the Watches window, right-click on it and select "Delete watch" so it will stop annoying you.
13. Resume execution of the program. You can do this by selecting "Continue" from the Debug menu, or you may notice that a small debugging toolbar has been added. Hover over the symbols until you find the button "Debug / Continue" and click on that.
 - Note that execution stops, now, at the other breakpoint you had set.

Breakpoints give you an opportunity to decide where in the code you would like to pause execution and to start observing the call stack and program variables.

3.3 Stepping Through Code

Sometimes, after stopping execution at a breakpoint, we want to move forward just a little bit and observe the changes made to our variables.

We could do this by setting another breakpoint a line or two away, but the total number of breakpoints can get unwieldy.

Debuggers provide commands for moving forward by different amounts. We have already used one of these - Continue - which moves us forward to the next breakpoint (or until the program crashes).

The other commands you will want to know, which can be accessed from the Debug menu or via buttons in the debugging toolbar, are "Next Line", "Step Into", and "Step Out".

"Next Line" and "Step Into" are similar in that each tries to move you forward one "step" of code. They differ, however, in what happens if the current line of code contains a function call. "Next Line" executes the function call without stopping (unless it hits a breakpoint) and stops at the next line of code within the same function where you started. "Step Into", on the other hand, starts the function call executing but stops at the first line of code inside the called function.

"Step Out" is the logical opposite of "Step Into". "Step Out" will try to execute the rest of the current function, and will stop in the caller of that function, just after the point at which the current function was called. (One of the most common uses of "Step Out" is to get back to where you were if you do a "Step Into" and then realize that you wish you had done a "Next Line" instead.

14. Make sure your breakpoints are still set. Restart the program from the debugger menu. Have the Call Stack and Watches windows open and showing the local variables.
15. Try moving forward using only "Next Line", until the program crashes.
16. Restart the program. This time, move forward using "Step Into".
 - You will almost immediately see something odd. The call stack will show that you are inside operator<<, but the editor window won't change to show you the source code. It can't show you that source code - it's part of the std library.
 - Whenever you see this sort of thing happen, use "Step out" to get back up to into your own code.
 - You should, however, be able to step into the addText function. Continue stepping from there.
17. Once you have stepped though to the point of the crash, stop the debugger as before.

Debuggers are powerful tools. They can help a lot. But you want to be smart about their use as well. They can become a tremendous time sink if you simply go single-stepping through all of your code hoping to notice when things go wrong. You need to think about likely causes for the symptoms you have observed, and set your breakpoints wisely to investigate possible causes for those symptoms.