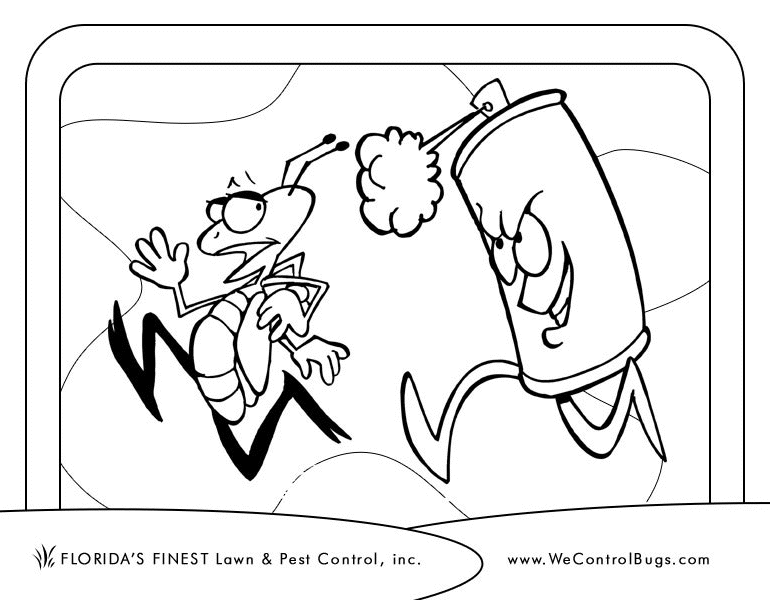
**Lab 7: Debugging with gdb**

[](http://www.csee.umbc.edu/courses/undergraduate/202/fall15_marron/labs/lab07/bug-spray.gif)

Overview

In this lab you will use practice using the gdb debugger.

Introduction: Why you should use a debugger

When you run your program within a debugger, you can stop the program at critical points and examine the values of variables and objects. The debugger provides you with much more capability and flexibility than debugging your program using print statements. For example:

* The debugger allows you to examine the value of a variable **after** your program has crashed. Using print statements you can only see values before a program crashes.
* If you need to examine the value of a variable that you previously had not suspected, you don't have to edit, recompile and re-run your program.
* The debugger allows you to examine values of variables in the function call stack. I.e., while you are debugging a functionfoo(), not only can you print out the local variables of foo(), you can also print out the local variables of the function that called foo(), and the function that called that function, and so on and so forth.

In this lab, we will use gdb, a debugger with a command line interface. Although the user interface is a bit clunky, you will find that gdb has many useful features. It "understands" C and C++ types and syntax. It works well with source code that is distributed across multiple files.

You might ask: *If gdb is so great, why don't more people use it?*

The answer is that most people don't take the time to learn gdb **before** they have a bad bug. Trying to learn gdb when you have a bad bug and a project due date hanging over your head does not leave you with warm fuzzies about the debugger.

Assignment

We will take a tour of gdb's many features in this lab using two simple examples of buggy programs. You should compile and run these programs on a Linux system on an Intel platform (e.g., the UMBC GL system). The bug in the first example might not manifest itself in the same way on other Unix systems.

Step 1: Get the files

If you've ssh’ed into the GL servers, issue the following commands at the command prompt while you are inside your directory for labs:

cp /afs/umbc.edu/users/c/h/chang/pub/cs202stuff/lab7.zip .

unzip lab7.zip

(Note the '.' at the end of the command — that is very important.)

You should get a directory called lab7 with these files:

* [buggy.cpp](http://www.csee.umbc.edu/courses/undergraduate/202/fall15_marron/labs/lab07/buggy.cpp): a program with some buggy code.
* [List7.h](http://www.csee.umbc.edu/courses/undergraduate/202/fall15_marron/labs/lab07/List7.h): the header file for the List class from Lab 6.
* [List7.cpp](http://www.csee.umbc.edu/courses/undergraduate/202/fall15_marron/labs/lab07/List7.cpp): the implementation file for the List class --- with a bug.
* [List7test.cpp](http://www.csee.umbc.edu/courses/undergraduate/202/fall15_marron/labs/lab07/List7test.cpp): the main program that exercises the List class.

You will not need to modify any of these files.

Step 2: A simple Debugging session

We will use the debugger to check out the buggy code in [buggy.cpp](http://www.csee.umbc.edu/courses/undergraduate/202/fall15_marron/labs/lab07/buggy.cpp).

Follow these steps:

1. Compile buggy.cpp using the -g option. This option tells the compiler and the loader to keep the symbol table around for the debugger:
2. g++ -g -gstabs+ buggy.cpp
3. Run the program by typing "./a.out". It should give you a segmentation fault.
4. Now let's run this program inside the gdb debugger. At the Unix prompt, type:
5. gdb a.out

After gdb starts up, type

run

at the gdb prompt. You should get a segmentation fault again.

1. After the segmentation fault, gdb prints out lots of information. Look at the lines:
2. Program received signal SIGSEGV, Segmentation fault.
3. 0x0000000000400888 in main () at buggy.cpp:12
4. 12 A[i/2] = i\*i ;

This says the program crashed while executing main() on line 12 of the file buggy.cpp. It also helpfully prints out line 12 for you.

Using gdb, you can look at the memory contents of your program just after the segmentation fault. This is convenient if running your program requires several steps involving user I/O. At any time while running gdb, you can type the command "where" to find out where you are in the program.

*If all you remember about gdb is the "where" command, you still have a very useful tool in debugging pointer errors.*

1. Of course, we don't remember our programs. The "list" command shows you the source code. Try these variations of the "list" command:
2. list 12
3. list main
4. list 1,18

If you type "list" right after another "list" command, it shows you another 10 lines below the previously listed lines.

1. The command "help list" will tell you more about list. In general, "help" followed by a topic or a command will give you documentation on that topic. Typing just "help" will give you a list of topics.
2. To see the value of variables and objects in the program, use the "print" command. Try these different print commands (you can also use "p" as a shortcut for print):
3. print i
4. print A[i]
5. print A
6. print &A
7. print &i
8. print &A[2]

Conclusion: somehow the value of i has grown so large that A[i] causes a segmentation fault.

1. Now we will re-run the program to see what happened. In gdb we can set "breakpoints" which stop the program at critical places. Type:
2. break 12

This stops the execution of the program *before* the code on line 12 is executed. Type "run" to start the program running. You should see something like:

Breakpoint 1, main () at buggy.cpp:12

12 A[i] = i\*i ;

Type "print A" and "print i" to see that the values of A and i are as expected.

To execute the code in line 12, type:

step

The "step" command executes 1 line of source code and shows you the next line. Print out A and i again to see the effects of line 12.

1. We can continue to step through the program this way, but it is simpler to issue the "continue" command (or just "c" for short).
2. continue

This runs the program until the next breakpoint.

1. We can keep typing "continue" and "print" commands this way, but all those prints get a bit tedious. Type:
2. display A
3. display i

This tells the compiler to print out A and i whenever it stops at a breakpoint.

1. Type "continue" again. Now you see the next iteration of the while loop. To continue again, you just have to hit the return key. Notice that the value of i eventually grows quite large. How did that happen?
2. If you missed the iteration where the value of i ballooned, type "kill" to stop running the program and "run" to restart it. Your breakpoints and display items are still intact.
3. To make sure you don't blow past the critical iteration of the while loop again, stop at the iteration when i is 20. Then use "step". You don't have to type "step" every time. After the first step, hitting the return key is equivalent to typing "step".
4. Now type "p &A[10]" and "p &i". Notice that these are the same hexadecimal (base 16) values (or should be on GL).
5. Now you can quit, using "quit".

Summary:

* Compile using the -g option, then "gdb a.out".
* Use "list" to see your source code.
* Use "break" to set breakpoints.
* Start running with "run".
* Use "print" to see values.
* "display" prints automatically.
* Use "step" to execute one more line.
* Use "continue" to execute until the next breakpoint.
* The commands "kill" and "quit" do what you think

In general you can have many breakpoints and display items. Sometimes you forget where and what they are. To see a list of your breakpoints and display items, use:

info breakpoints

info display

You can remove all breakpoints using "delete" or individual breakpoints using "delete 1", "delete 2", ... (Use the breakpoint number from the "info" list.) Sometimes you just want to temporary disable a breakpoint. You can do that with "disable 1", "disable 2", ... To enable the breakpoint again, use "enable 1", "enable 2", ...

To list your display items, "info display". Use "undisplay" to remove a display item. Use "disable display 1", "disable display 2",... to temporarily disable a display item.

Step 3: A more complicated example

Now, let's try something more complicated, involving programs with different source files and code that uses objects and pointers. We'll see that gdb actually understands C++.

1. Remove the a.out file from the previous exercise.
2. Compile a buggy implementation of linked lists. (Do not use the files from Lab 6):
3. g++ -g -gstabs+ -ansi -Wall List7test.cpp List7.cpp

Run the program, you should see something like:

Testing Insert...

Should Print: -9 -7 -5 -4 -4 -3 -1 -1 -1 0 1 1 1 2 2 3 5 5 6 7 8 9

Actual Print: -9 -7 -5 -4 -4 -3 -1 -1 -1 0 1 1 1 2 2 3 5 5 6 7 8 9

Testing Remove...

Remove 5...

Should Print: -9 -7 -5 -4 -4 -3 -1 -1 -1 0 1 1 1 2 2 3 5 6 7 8 9

Actual Print: -9 -7 -5 -4 -4 -3 -1 -1 -1 0 1 1 1 2 2 3 5 6 7 8 9

Remove -1...

Should Print: -9 -7 -5 -4 -4 -3 -1 -1 0 1 1 1 2 2 3 5 6 7 8 9

Actual Print: -9 -7 -5 -4 -4 -3 -1 -1 0 1 1 1 2 2 3 5 6 7 8 9

Remove 9...

Segmentation fault (core dumped)

Apparently, the first two removes worked correctly, but the third one did not.

1. Run the program inside gdb, by typing
2. gdb a.out

and then "run" at the gdb prompt. Make sure you are using the a.out from this program and not the previous one. The last few lines of gdb's output should say something like:

Program received signal SIGSEGV, Segmentation fault.

0x0000000000400c4d in List::remove (this=0x7fffffffdf50, data=9) at List7.cpp:96

96 current->next = ptr->next ;

The remove() function is apparently the culprit. Not surprising.

1. We can list the source code from different files this way:
2. list List7.cpp:1,20
3. list List7test.cpp:1,20
4. list main
5. list remove
6. list List::remove

When the source code is spread out over multiple files, listing by function name is much more convenient.

1. To set a break point at a line number in a particular file:
2. break List7.cpp:96

We can also break after entering a function:

break List::remove

1. Run the program using "run". It should break on line 88 after entering the remove function. Type "where". You should see something like:
2. #0 List::remove (this=0x7fffffffdf50, data=5) at List7.cpp:88
3. #1 0x0000000000400fe7 in main () at List7test.cpp:40

This says that the execution of the program is stopped in function List::remove() at line number 88 of List7.cpp. Also, often quite important, it says remove() was called from main() at line 40 of List7test.cpp. This is called the "backtrace of the stack frames".

1. Where we stopped the program, the variable current is not yet initialized. Type:
2. step

to initialize it.

1. We can now print out various local variables in remove. Unfortunately, everything other than data is a pointer, so printing them just gives obscure hexadecimal values. Fortunately, gdb understands C/C++ operators \*, & and ->. Try these print statements:
2. print data
3. print m\_head
4. print current
5. print \*m\_head
6. print \*current
7. print current->next
8. print \*(current->next)

We can also print out values of local variables in main(). Try:

print main::List1

print &main::List1

Sometimes we forget the type of the variables we are using:

ptype current

ptype \*current

ptype current->m\_data

ptype main::List1

Note the \* at the end of the reported type of current. This says current is a Node pointer rather than a Node.

We can even use the "this" pointer:

print this

print &main::List1

The output confirms that the host/calling object is indeed List1 from main().

1. Type "where" and "info breakpoints" to review where we are. There should be a breakpoint on line 96. This is the part of the while loop that is executed when the node to be removed is found. Type:
2. list List7.cpp:96
3. continue
4. Now execution has terminated in the portion of the code that is buggy. See if you can figure out what happened... These print commands should help:
5. print \*current
6. print \*current->next
7. print ptr
8. print \*ptr
9. print \*ptr->next
10. Type "continue" (or use the return key) 4 times. The program should trigger a segmentation fault. Type in these commands:
11. list
12. print data
13. print ptr
14. print current
15. print \*current

You should see why this program crashes.

1. Use "quit" to get out of gdb.

One command we didn't go over here is the "next" command. This command is very much like the "step" command. The difference is when the next line of code is a function call. In this case, "step" will stop at the first line of the function being called, whereas "next" will wait for the function call to finish.

# Step 4: Try it in your own program

If you have time remaining, practice debugging a piece of your own code, say from Project 1 or Project 2. Remember to use the -g option when you compile. Try stepping through your code using "next" versus "step" to see the difference.

With some practice, the next time you have a buggy piece of code and a deadline looming, you can pull out gdb and know how to use it.