

CSCI 2270 Lab

Introduction to gdb (the gnu C++ debugger)

As you look for problems in your code, it's useful to run a debugger. gdb, the debugger for g++, is a powerful if slightly unfriendly tool for tracing the execution of your code interactively. It lets you watch the code run line by line, pausing to quiz the code about what it's thinking, or at least how your variables are changing. If you can learn to get comfortable with a debugging tool early on, you'll have a terrific resource for figuring out what's going wrong in your code, and later, when you are wrestling with memory errors, a tool like gdb will let you know when you have found them (nearly all the time). gdb is available on the lab machines and the VM.

A quick page for looking up gdb commands is at:

<http://web.eecs.umich.edu/~sugih/pointers/summary.html>.

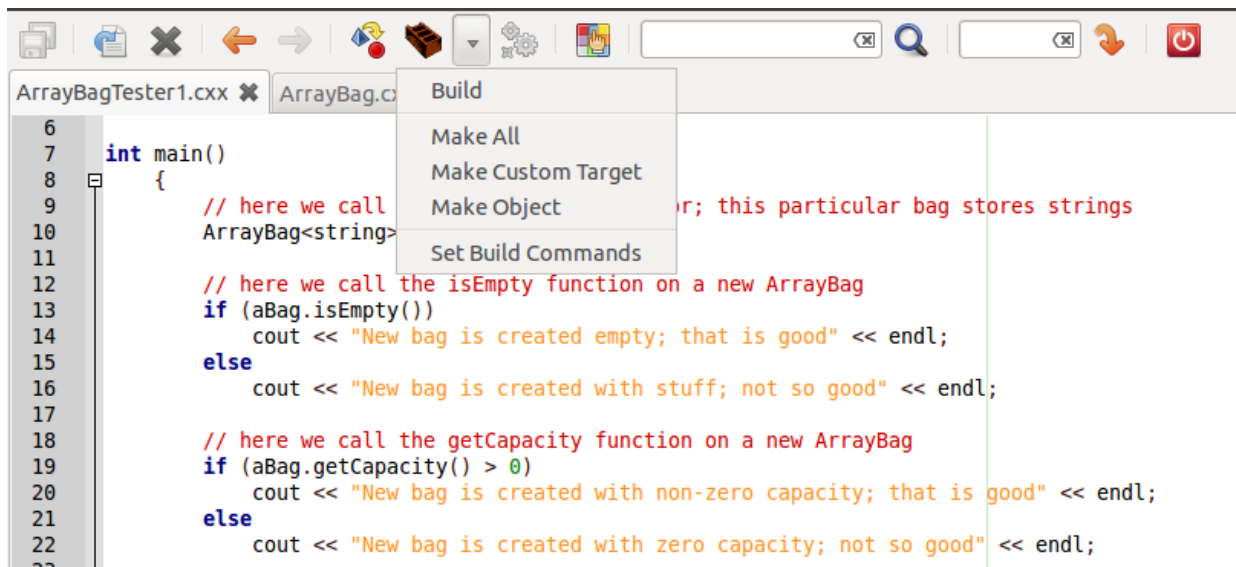
A nice source for more in-depth information on gdb can be found at:

<http://www.cs.umd.edu/~srhuang/teaching/cmsc212/gdb-tutorial-handout.pdf>.




We'll start with just a few commands in gdb this week, and we'll add more in later labs.

Grab the C++ files for this lab from the moodle; it's a set of HW1 bag code with some lovely bugs for you to find using gdb. Open the test code and the ArrayBag.cxx code in Geany. Notice that the code compiles, with no complaints. It does not run well, though.

To use gdb, we need to add one extra parameter to the compilation step. To set this up, click the arrow next to the compile/build icons and choose Set Build Commands from the dropdown menu:



In the Set Build Commands dialog box, you need to make two changes. One is letting GDB hook into your code when you compile it: to do that, add a -g to the command for Build (this command runs when you click that bookshelf), as below. (You can add it to the compile option too, but that has no effect in our case.)

C++ commands			
1.	Compile	<code>g++ -Wall -c "%f"</code>	
2.	Build	<code>g++ -Wall -g -o "%e" "%f"</code>	
3.			

Recall that this command becomes



```
g++ -Wall -g -o "ArrayBagTester1" "ArrayBagTester1.cxx"
```

when you run it, because Geany substitutes the name of your cxx file (ArrayBagTester1.cxx), for %f and substitutes the name of the executable (ArrayBagTester1, always the same as the file, except with no .cxx) for %e.


Second, add the following code to the Debug window at the bottom of the dialog to run your code in gdb when you want to. Remember that this command becomes:

```
gdb "ArrayBagTester1"
```

at runtime, because (again) it substitutes the name of your compiled ArrayBagTester1 code for %e.


Execute commands			
1.	Execute	<code>./"%e"</code>	
2.	Debug	<code>gdb "%e"</code>	



Now, if you hit the Build button (), the bottom of your window should be showing the correct command, as below, with the little -g you added.

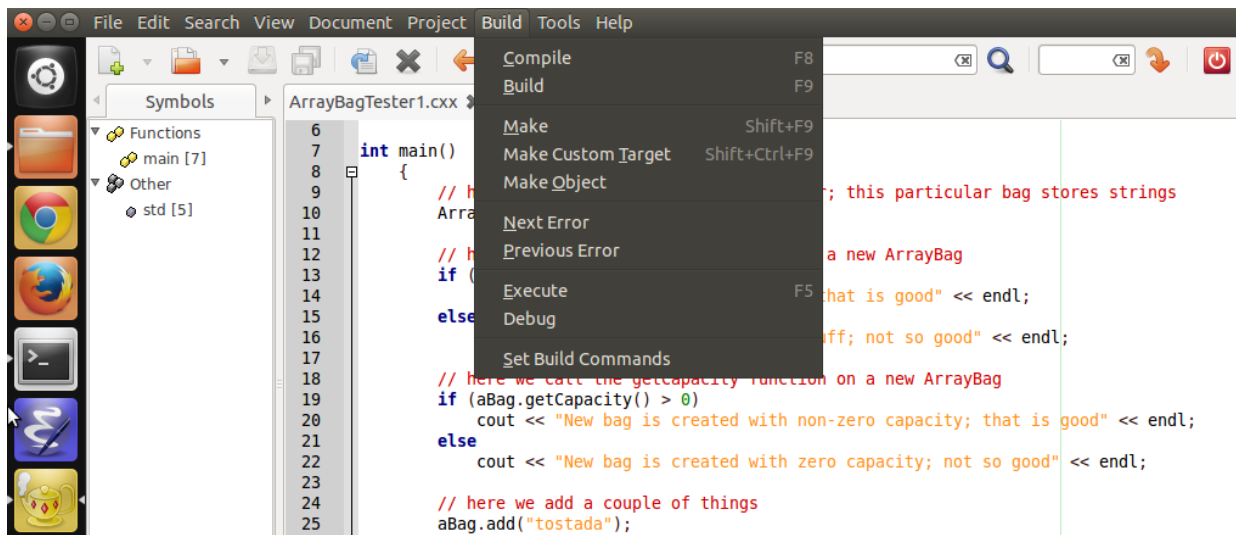
Status	<code>g++ -Wall -g -o "ArrayBagTester1" "ArrayBagTester1.cxx" (in directory: /home/user/Dropbox/CSCI2270Fall2013/Labs/Lab2)</code>
Compiler	Compilation finished successfully.
Messages	
Scribble	
Terminal	
Debug	



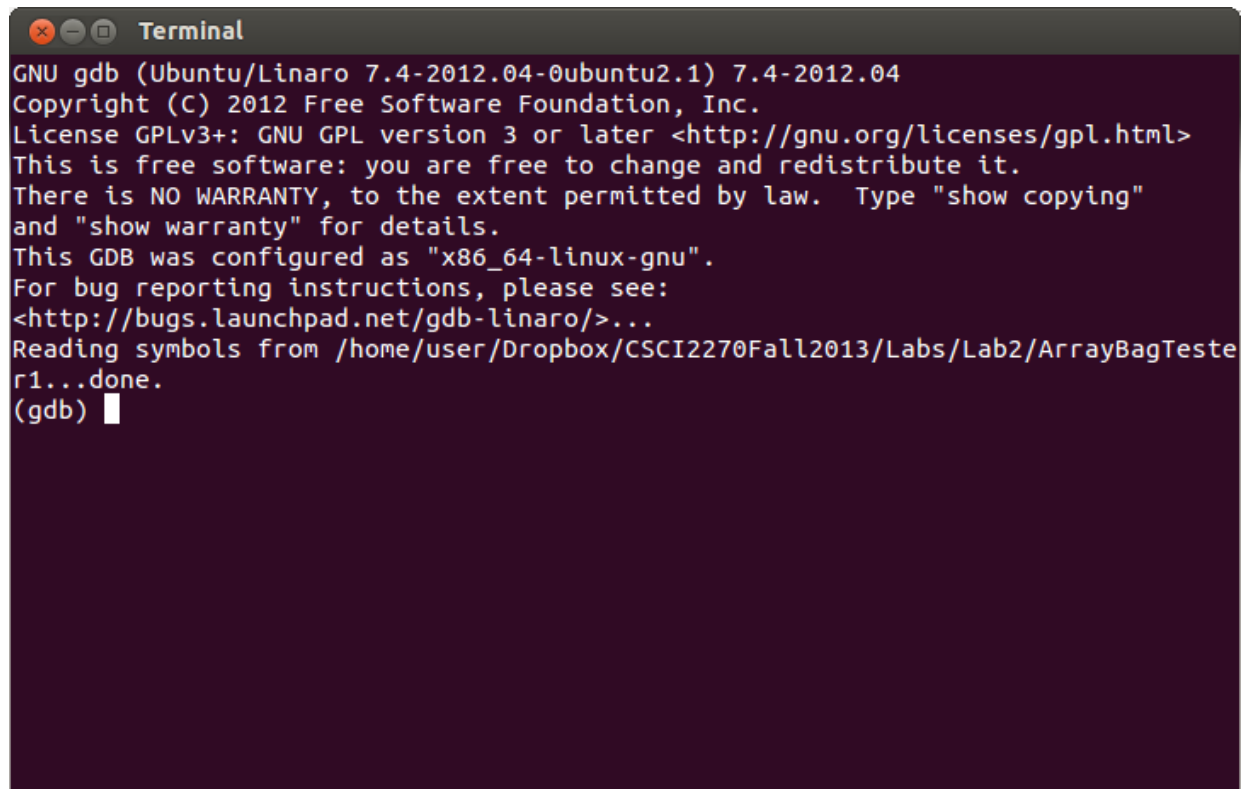
If you run this using the regular gear button (), you'll see that the test code encounters a segmentation fault fairly quickly, which is because the ArrayBag.cxx code is seeded with

sneaky errors that you must find and identify. You should definitely team up and talk to other people in your lab section while you look for these, but you need to show, in your comments, that you understand the errors you have found. It is possible that all of you won't find all of the bugs, and that's ok. It's more important to me that you start thinking about how you can use gdb to hunt them down.

To debug, you need to use the top-level menu in Geany, so mouse over that black line to get the top menu, and click Build:



The Debug option is second from the bottom. When you click that, a terminal window pops up, with gdb running your code. If your -g option worked out, then it should tell you that it's read the symbols from the ArrayBag code you made.

A terminal window titled "Terminal" with a dark background and light-colored text. The text shows the GNU gdb startup sequence, including version information, copyright notice, license information, and the path to the symbols file. The prompt is "(gdb)".

```
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/user/Dropbox/CSCI2270Fall2013/Labs/Lab2/ArrayBagTeste
r1...done.
(gdb) █
```

At the gdb prompt in that little window, if you type


run

you will see the code beheading the chicken, so to speak. (You can type 'r' instead of 'run', for short).

```
Terminal
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/user/Dropbox/CSCI2270Fall2013/Labs/Lab2/ArrayBagTester1...done.
(gdb) run
Starting program: /home/user/Dropbox/CSCI2270Fall2013/Labs/Lab2/ArrayBagTester1
New bag is created empty; that is good
New bag is created with non-zero capacity; that is good

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7b79f8b in std::string::assign(std::string const&) ()
    from /usr/lib/x86_64-linux-gnu/libstdc++.so.6
(gdb) █
```

So far, this is not very impressive; everything you have seen so far is just like hitting the

Execute button (). But gdb is smarter than that. It lets us set a *breakpoint* in the code so that we can run to a certain line, and then gdb will pause and let us interact with the code as it chases chickens.

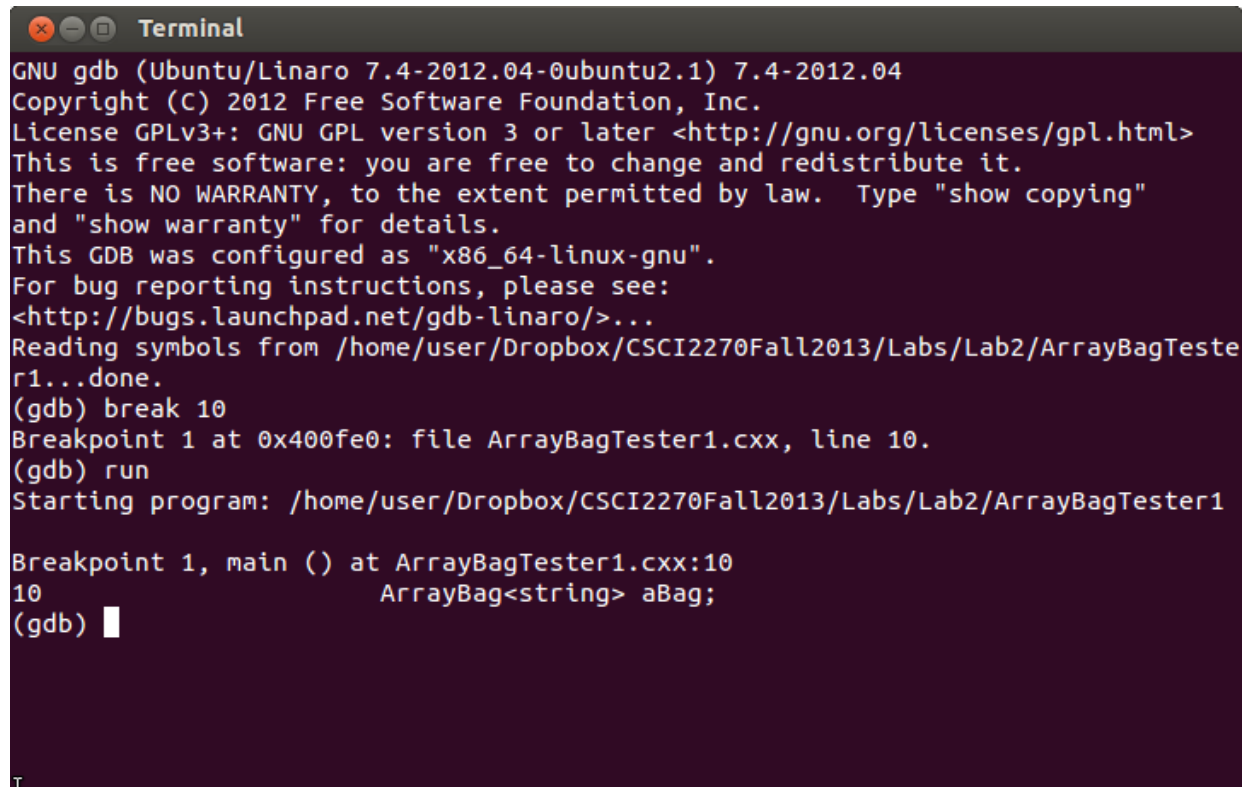
Do not assume, necessarily, that the first error is happening right before the first segfault. We'll start checking at the first line of the test code. That's the line highlighted in grey below.

```
ArrayBagTester1.cxx x ArrayBag.cxx x
6
7 int main()
8 {
9     // here we call the ArrayBag constructor; this particular bag stores strings
10    ArrayBag<string> aBag;
11
12    // here we call the isEmpty function on a new ArrayBag
13    if (aBag.isEmpty())
14        cout << "New bag is created empty; that is good" << endl;
15    else
16        cout << "New bag is created with stuff; not so good" << endl;
17
```

We'll pause the code right there to start. Geany tells you that this is line 10, so we'll tell gdb to pause on that line, using the break command:

```
break 10
```

(You can abbreviate 'break' to 'b'.) So if you set a breakpoint at line 10 in gdb and then tell it to run the code, you'll see this:

A terminal window titled "Terminal" with a dark background and light text. It shows the output of a GDB session. The text is as follows:

```
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/user/Dropbox/CSCI2270Fall2013/Labs/Lab2/ArrayBagTester1...done.
(gdb) break 10
Breakpoint 1 at 0x400fe0: file ArrayBagTester1.cxx, line 10.
(gdb) run
Starting program: /home/user/Dropbox/CSCI2270Fall2013/Labs/Lab2/ArrayBagTester1

Breakpoint 1, main () at ArrayBagTester1.cxx:10
10             ArrayBag<string> aBag;
(gdb) █
```

It's paused at line 10, and has printed that out for you. Now, if you type

step

back to gdb at the prompt, gdb will step into the code that runs on this line. (You can type 's' instead of 'step', for short.) When that happens, you'll see this:

```
Terminal
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/user/Dropbox/CSCI2270Fall2013/Labs/Lab2/ArrayBagTester1...done.
(gdb) break 10
Breakpoint 1 at 0x400fe0: file ArrayBagTester1.cxx, line 10.
(gdb) run
Starting program: /home/user/Dropbox/CSCI2270Fall2013/Labs/Lab2/ArrayBagTester1

Breakpoint 1, main () at ArrayBagTester1.cxx:10
10         ArrayBag<string> aBag;
(gdb) step
ArrayBag<std::basic_string<char, std::char_traits<char>, std::allocator<char> > >::ArrayBag (this=0x7fffffffe410) at ArrayBag.cxx:42
42         ArrayBag<ItemType>::ArrayBag()
(gdb) █
```

Now it's telling you (just above) that it's looking at line 42 of ArrayBag.cxx. That's your constructor. You want to proceed through the next bit of code, so type

```
next
```

(or 'n') to see what this constructor is up to. (Hint: it is VERY NAUGHTY.) Again, type

```
next
```

and notice that you are now back on line 13 of the ArrayBagTester1.cxx code. Let's see what aBag knows about itself. We'll use the disp command to display some of aBag's member variables. Type

```
disp aBag.DEFAULT_CAPACITY
```

and see if you like the answer. Then, type

```
disp aBag.numberOfItems
```

and see about that. Perhaps this will indicate a problem. Notice that each variable you display has a number; aBag.DEFAULT_CAPACITY might be 1, for example. If you keep typing next, these variables of aBag will keep displaying over and over until you make them stop. To

stop displaying a variable, use the undisplay command undisp, plus its number according to gdb:

```
undisp 1
```

When you find a mistake, and I hope that by now you have, you will need to change the ArrayBag.cxx code in Geany and recompile. Quit gdb by typing

```
quit
```

(You can abbreviate 'quit' to 'q' in gdb.) Fix the error, add a comment explaining what you fixed, recompile, and rerun gdb, setting the same breakpoint, to see if you are doing any better.

It's a little ugly to inspect items in the array, but gdb will do this. When the code is debugged enough to add an item like "tostada", you can say

```
disp aBag.itemArray[0]
```

and in return, you'll see a bunch of awful text, with the item you want ("tostada") at the end.

```
(gdb) disp aBag.itemArray[0]
5: aBag.itemArray[0] = {static npos = <optimized out>,
  _M_dataplus = {<std::allocator<char>> = {<__gnu_cxx::new_allocator<char>> = {<
No data fields>}, <No data fields>}, _M_p = 0x606028 "tostada"}}
```

When you are on a line from ArrayBagTester1.cxx, you need to refer to member variables like numberOfItems by making it clear that you are asking about aBag's variables; that's why you have to type the 'aBag.' part in the line below:

```
disp aBag.DEFAULT_CAPACITY
```

When you are debugging a line from ArrayBag.cxx, you can skip the 'aBag.' part; there gdb understands that you mean the bag whose items are being added or removed or counted (which is still aBag). In the Bag's member functions in ArrayBag.cxx, you can type:

```
disp DEFAULT_CAPACITY
```

to get the same information about aBag's default capacity. There are about 11-12 problems in the ArrayBag.cxx file, depending on how you count problems. Your TA will know where they are, but your job is to track many of them down via debugging with gdb; it's more important that you practice with gdb and explain what's wrong with the mistakes that you found than that you find all the bugs. When you finish, upload your fixed ArrayBag.cxx code to the moodle and go do something restful. We're happy to help you catch more bugs in office/help hours this week.

