

LAB 7, introduction to GDB

CSCI 2270

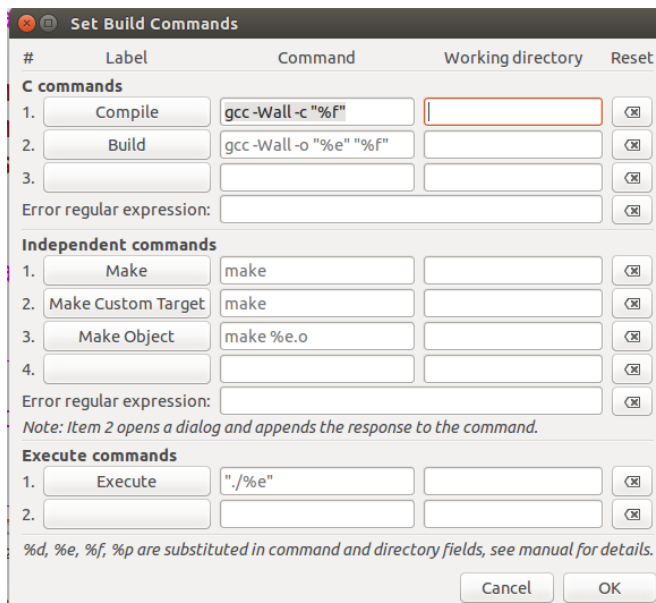
Purpose: to learn how to debug code interactively (instead of by using endless cout statements ☺) and to get practice in hunting for bugs in code, which is a logical skill you can and should learn.

GDB (the gnu debugger) is a debugger used for C and C++ code. It can run your program interactively, so you can see how it branches or recurs, and check for certain kinds of errors. I admit that it's a little clumsy, and therefore kind of a pain to learn at first, but if you get good at using this resource, it will pay off. Also, gdb's basics generalize well to other debugging frameworks.

To use gdb, we have to compile our code with a special flag, -g. I've been using that in our Makefiles for a while. For instance, here are our compiler commands for the code for this lab:

```
make (in directory: /home/user/Dropbox/csci2270-Fall-2014/labs/lab-7/Solved)
make[1]: Entering directory `/home/user/Dropbox/csci2270-Fall-2014/labs/lab-7/Solved'
g++ -std=c++0x -Wall -g -c test_int_array.cpp -o test_int_array.o
g++ -std=c++0x -Wall -g -c terrible_dynamic_size_array_unsorted.cpp -o
terrible_dynamic_size_array_unsorted.o
g++ -std=c++0x -Wall -g test_int_array.o terrible_dynamic_size_array_unsorted.o -o
test_int_array
make[1]: `terrible_dynamic_size_array_unsorted.o' is up to date.
make[1]: Leaving directory `/home/user/Dropbox/csci2270-Fall-2014/labs/lab-7/Solved'
Compilation finished successfully.
```

(Note: You can also set the -g option as a default in Geany for Build and Compile. Go to Build and click Set Build Commands. You'll see a window something like this:



In this window, the name "%f" refers to your cpp file. The name "%e" refers to the executable you get from compiling the cpp file. For this example, "%f" can be either `terrible_dynamic_size_array_unsorted.cpp` or `test_int_array.cpp`, and "%e" is `test_int_array`, the executable we get from building all the files together.

Just so we all use the same defaults, please go ahead and edit the Compile command to be:

```
g++ -std=c++0x -Wall -g -c "%f"
```

The `-std=c++0x` gives the compiler C++ 11 capabilities, like the ability to recognize `nullptr` in your Geany builds.

Please edit the Build command as well to be:

```
g++ -std=c++0x -Wall -g -o "%e" "%f"
```

#	Label	Command	Working directory	Reset
C commands				
1.	Compile	<code>g++ -std=c++0x -Wall -g -c</code>		<input type="button" value="X"/>
2.	Build	<code>g++ -std=c++0x -Wall -g -c</code>		<input type="button" value="X"/>
3.				<input type="button" value="X"/>
Error regular expression: <input type="text"/>				
Independent commands				
1.	Make	<code>make</code>		<input type="button" value="X"/>
2.	Make Custom Target	<code>make</code>		<input type="button" value="X"/>
3.	Make Object	<code>make %e.o</code>		<input type="button" value="X"/>
4.				<input type="button" value="X"/>
Error regular expression: <input type="text"/>				
<i>Note: Item 2 opens a dialog and appends the response to the command.</i>				
Execute commands				
1.	Execute	<code>./%e</code>		<input type="button" value="X"/>
2.				<input type="button" value="X"/>
<i>%d, %e, %f, %p are substituted in command and directory fields, see manual for details.</i>				
<input type="button" value="Cancel"/> <input type="button" value="OK"/>				

Now we should all be consistent. But use Build->Make for this assignment.)

For convenience, go ahead and click on the blank button under Execute (circled here in red):

Set Build Commands

#	Label	Command	Working directory	Reset
C commands				
1.	Compile	g++ -std=c++0x -Wall -g -c		
2.	Build	g++ -std=c++0x -Wall -g -o		
3.				
Error regular expression:				
Independent commands				
1.	Make	make		
2.	Make Custom Target	make		
3.	Make Object	make %e.o		
4.				
Error regular expression:				
<i>Note: Item 2 opens a dialog and appends the response to the command.</i>				
Execute commands				
1.	Execute	"/%e"		
2.				
<i>%d, %e, %f, %p are substituted in command and directory fields, see manual for details.</i>				
		Cancel	OK	

A window pops up:

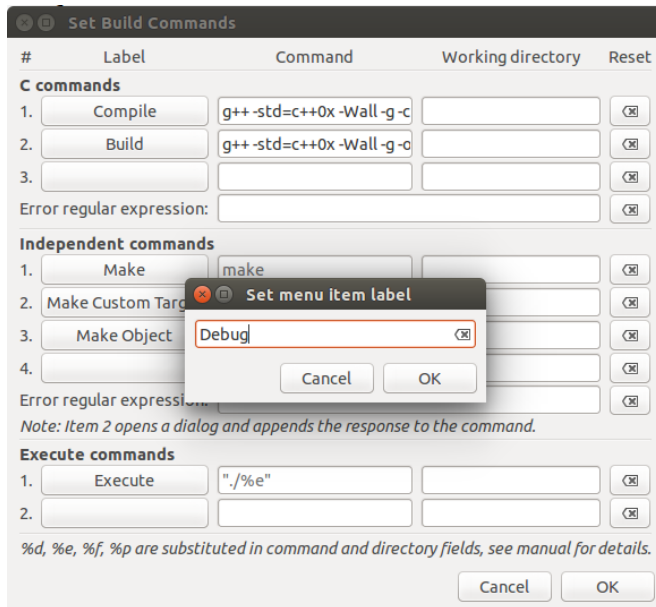
Set Build Commands

#	Label	Command	Working directory	Reset
C commands				
1.	Compile	g++ -std=c++0x -Wall -g -c		
2.	Build	g++ -std=c++0x -Wall -g -o		
3.				
Error regular expression:				
Independent commands				
1.	Make	make		
2.	Make Custom Target			
3.	Make Object			
4.				
Error regular expression:				
<i>Note: Item 2 opens a dialog and appends the response to the command.</i>				
Execute commands				
1.	Execute	"/%e"		
2.				
<i>%d, %e, %f, %p are substituted in command and directory fields, see manual for details.</i>				
		Cancel	OK	

Set menu item label

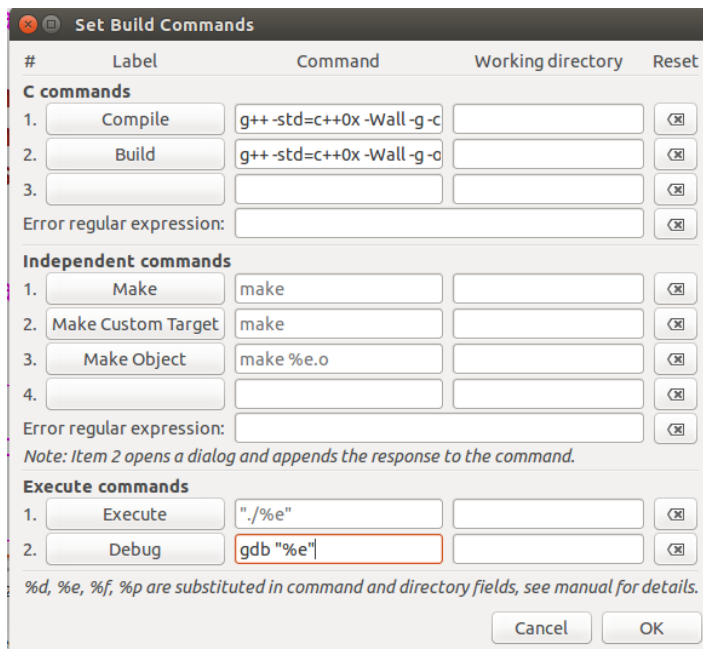
Cancel OK

Name this menu item Debug:



Click OK, and add to the Command for this Debug item:

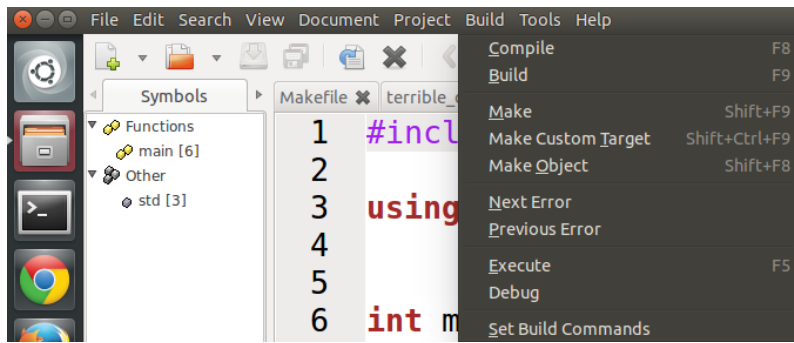
`gdb "%e"`



Click OK again.

Now, activate your `test_int_array.cpp` tab, and use Build->Make to compile your code using Make.

To debug your code, you can again activate the `test_int_array.cpp` file tab, go to Build, and click Debug, and gdb will launch and be ready to run the code.



Click this Debug option, and you should see a terminal window pop up. The important thing here is to make sure gdb reports that it has read the symbols from the test_int_array code. (If it doesn't, you might have forgotten the -g switch in the compilation steps.)

```
Terminal
GNU gdb (Ubuntu 7.7-0ubuntu3.2) 7.7
Copyright (C) 2014 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".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from test_int_array...done.
(gdb) 
```

To run the code, you can type

```
run
```

or just

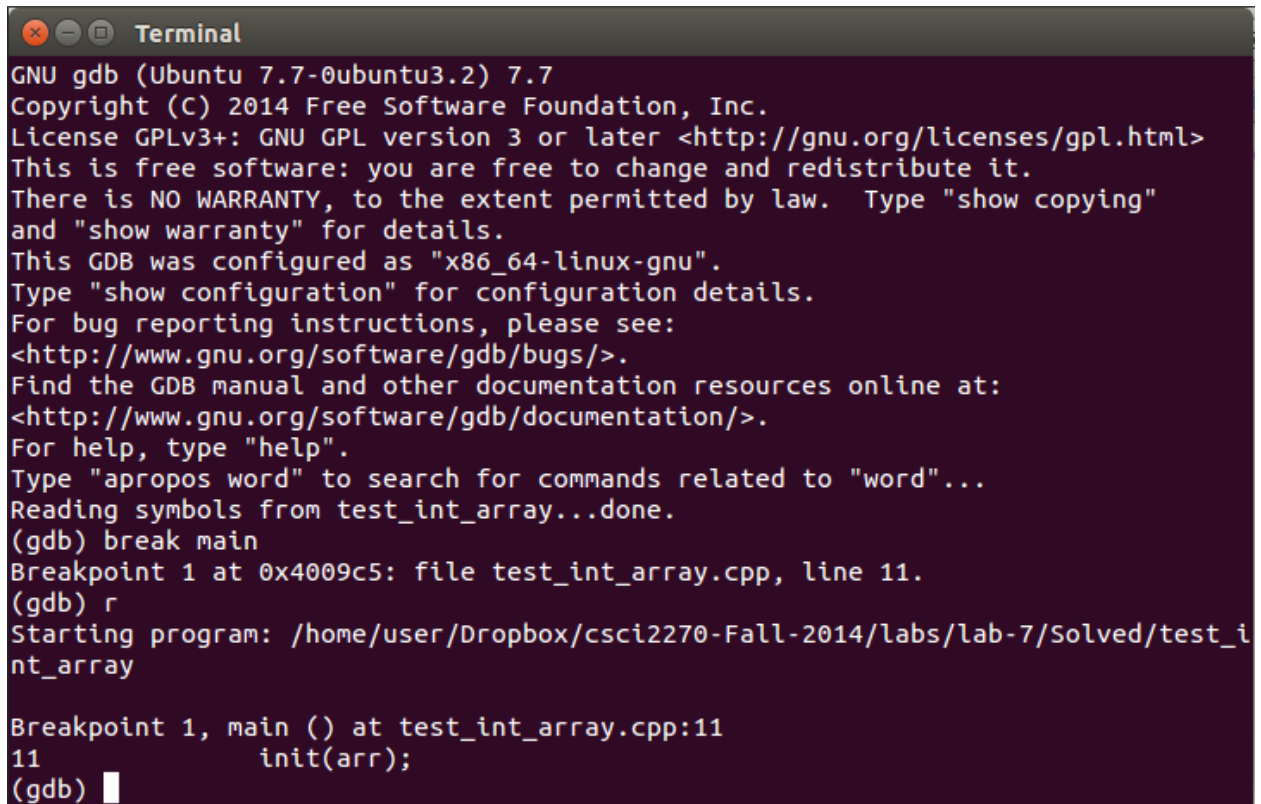
```
r
```

Add a number and see what the code does. It's not pretty. Recall that at this point, only the init and add methods have run. One of them must be causing this problem. Or both of them.

Cancel out of that run of gdb (with control+c), close the window, and we'll step through the code more slowly. Go to Build and click Debug again. This time, we'll begin by telling gdb to set a breakpoint, which pauses the code so we can look at it, in the main function:

```
break main
```

Now, tell it to run and it stops at the first line of main. This is the call to init():



```
GNU gdb (Ubuntu 7.7-0ubuntu3.2) 7.7
Copyright (C) 2014 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".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from test_int_array...done.
(gdb) break main
Breakpoint 1 at 0x4009c5: file test_int_array.cpp, line 11.
(gdb) r
Starting program: /home/user/Dropbox/csci2270-Fall-2014/labs/lab-7/Solved/test_i
nt_array

Breakpoint 1, main () at test_int_array.cpp:11
11         init(arr);
(gdb) |
```

Ideally, we want to see what init is really up to. We can do that by typing

```
step
```

or

```
s
```

to tell the debugger we want to step into (follow the code through) this function:

```
Terminal
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".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from test_int_array...done.
(gdb) break main
Breakpoint 1 at 0x4009c5: file test_int_array.cpp, line 11.
(gdb) r
Starting program: /home/user/Dropbox/csci2270-Fall-2014/labs/lab-7/Solved/test_i
nt_array

Breakpoint 1, main () at test_int_array.cpp:11
11         init(arr);
(gdb) s
init (arr=...) at terrible_dynamic_size_array_unsorted.cpp:8
8         arr.count = 0;
(gdb) |
```

Notice that now we're in the first line of `init`. The line `arr.count` is about to execute.

Let's take a look at some of these variables in the code. We can use `disp` to show them at each step:

```
disp arr.count
disp arr.capacity
disp arr
```

They'll display now with each command as long as we're in `init()`.

To advance to the next line of code, type

```
next
```

or

```
n
```

Notice that you've now set the array's count and are about to set its capacity:

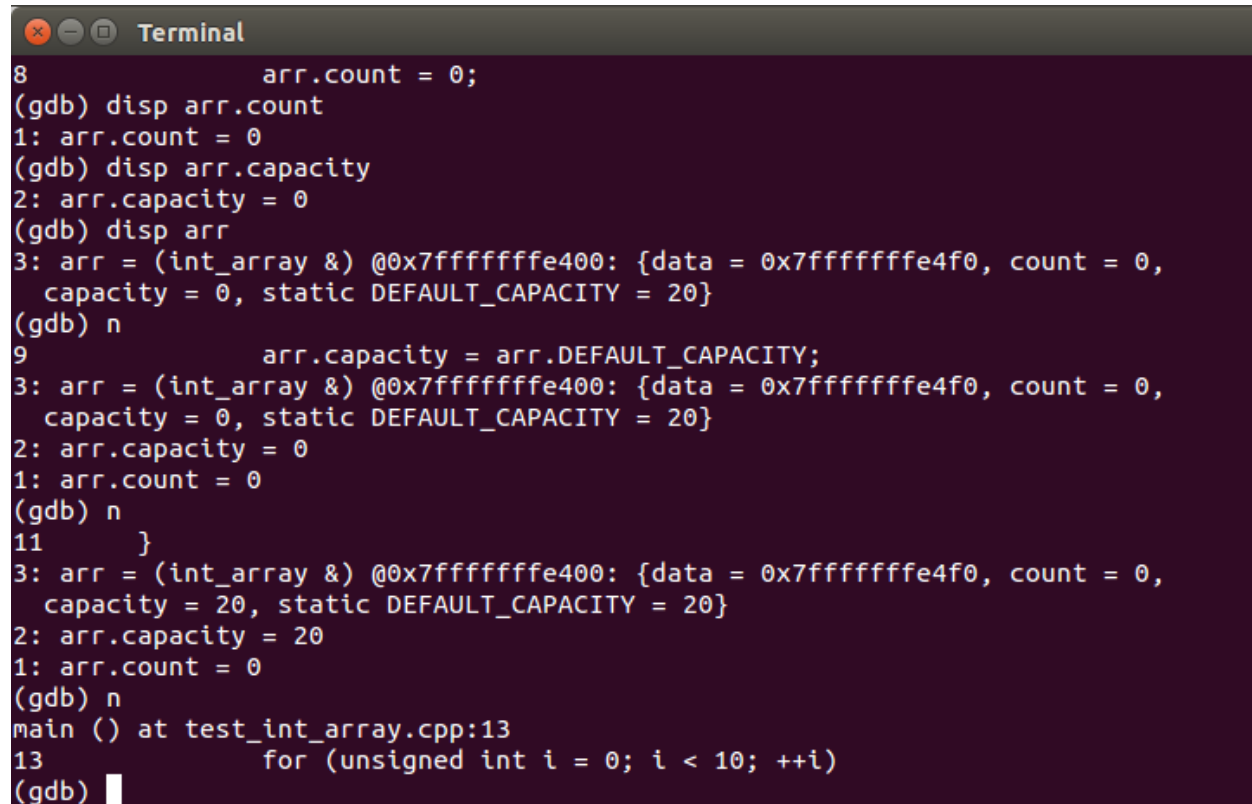
```
Terminal
Breakpoint 1 at 0x4009c5: file test_int_array.cpp, line 11.
(gdb) r
Starting program: /home/user/Dropbox/csci2270-Fall-2014/labs/lab-7/Solved/test_int_array

Breakpoint 1, main () at test_int_array.cpp:11
11         init(arr);
(gdb) step
init (arr=...) at terrible_dynamic_size_array_unsorted.cpp:8
8         arr.count = 0;
(gdb) disp arr.count
1: arr.count = 0
(gdb) disp arr.capacity
2: arr.capacity = 0
(gdb) disp arr
3: arr = (int_array &) @0x7fffffff400: {data = 0x7fffffff4f0, count = 0, capacity = 0, static DEFAULT_CAPACITY = 20}
(gdb) n
9         arr.capacity = arr.DEFAULT_CAPACITY;
3: arr = (int_array &) @0x7fffffff400: {data = 0x7fffffff4f0, count = 0, capacity = 0, static DEFAULT_CAPACITY = 20}
2: arr.capacity = 0
1: arr.count = 0
(gdb)
```

Type another next and see where you are:

```
Terminal
11         init(arr);
(gdb) step
init (arr=...) at terrible_dynamic_size_array_unsorted.cpp:8
8         arr.count = 0;
(gdb) disp arr.count
1: arr.count = 0
(gdb) disp arr.capacity
2: arr.capacity = 0
(gdb) disp arr
3: arr = (int_array &) @0x7fffffff400: {data = 0x7fffffff4f0, count = 0, capacity = 0, static DEFAULT_CAPACITY = 20}
(gdb) n
9         arr.capacity = arr.DEFAULT_CAPACITY;
3: arr = (int_array &) @0x7fffffff400: {data = 0x7fffffff4f0, count = 0, capacity = 0, static DEFAULT_CAPACITY = 20}
2: arr.capacity = 0
1: arr.count = 0
(gdb) n
11     }
3: arr = (int_array &) @0x7fffffff400: {data = 0x7fffffff4f0, count = 0, capacity = 20, static DEFAULT_CAPACITY = 20}
2: arr.capacity = 20
1: arr.count = 0
(gdb)
```


Now we're about to execute the closing bracket of `init()` (!!). When we type the next `n`, we'll be back in the `test_int_array.cpp` code, and these variables will be out of scope, so we won't see them.

A terminal window titled "Terminal" with a dark background and light-colored text. It shows a GDB session. The user enters commands like 'disp arr.count', 'disp arr.capacity', and 'disp arr' to inspect the state of an 'arr' variable. The output shows 'count' and 'capacity' as 0, and 'arr' as a struct with 'data' at 0x7fffffffef0, 'count' as 0, and 'capacity' as 0. The user then enters 'n' to step to the next line, which sets 'arr.capacity' to 'arr.DEFAULT_CAPACITY'. This process is repeated, and the final state shows 'count' as 0 and 'capacity' as 20. The user then enters 'n' to step to the next line, which is the start of a for loop in 'main()' at 'test_int_array.cpp:13'. The user then enters 's' to step into the loop.

```
8         arr.count = 0;
(gdb) disp arr.count
1: arr.count = 0
(gdb) disp arr.capacity
2: arr.capacity = 0
(gdb) disp arr
3: arr = (int_array &) @0x7fffffffef0: {data = 0x7fffffffef0, count = 0,
  capacity = 0, static DEFAULT_CAPACITY = 20}
(gdb) n
9         arr.capacity = arr.DEFAULT_CAPACITY;
3: arr = (int_array &) @0x7fffffffef0: {data = 0x7fffffffef0, count = 0,
  capacity = 0, static DEFAULT_CAPACITY = 20}
2: arr.capacity = 0
1: arr.count = 0
(gdb) n
11     }
3: arr = (int_array &) @0x7fffffffef0: {data = 0x7fffffffef0, count = 0,
  capacity = 20, static DEFAULT_CAPACITY = 20}
2: arr.capacity = 20
1: arr.count = 0
(gdb) n
main () at test_int_array.cpp:13
13     for (unsigned int i = 0; i < 10; ++i)
(gdb) 
```

Back in the test code, we have a loop to add numbers to our buggy `int_array`. Use `n` to advance to the next line of code, and type a number in at the prompt. When you come to the line that is about to execute the `add()` method, type `s` to step into that code.

```
Terminal
capacity = 0, static DEFAULT_CAPACITY = 20}
(gdb) n
9          arr.capacity = arr.DEFAULT_CAPACITY;
3: arr = (int_array &) @0x7fffffff400: {data = 0x7fffffff4f0, count = 0,
  capacity = 0, static DEFAULT_CAPACITY = 20}
2: arr.capacity = 0
1: arr.count = 0
(gdb) n
11      }
3: arr = (int_array &) @0x7fffffff400: {data = 0x7fffffff4f0, count = 0,
  capacity = 20, static DEFAULT_CAPACITY = 20}
2: arr.capacity = 20
1: arr.count = 0
(gdb) n
main () at test_int_array.cpp:13
13      for (unsigned int i = 0; i < 10; ++i)
(gdb) n
15          cout << "Enter a number to add: ";
(gdb) n
16          cin >> num;
(gdb) n
Enter a number to add: 8
17          add(arr, num);
(gdb) s
```

From here, you can step through your first add to the array again. (You will need to re-display the important variables with `disp` to see them.)

For this week, just practice finding bugs with `break`, `disp`, `next`, and `step`. This code contains a bunch of errors that get past the compiler but cause failures. You don't have to find all of them; I tried to add all of the errors I have ever seen students make on this assignment, and that's pretty diabolical. If you can find 5 of the errors in the array code, describe them in your own words in comments, make the needed changes, and describe those in your own words in comments, that will suffice. Upload your cpp file with the right corrections. I'll post the full error list at the end of the week.

For next week's gdb lab, I'll use some more examples from this slideshow, which is a little C-focused but otherwise useful: <http://www.cs.umd.edu/~srhuang/teaching/cmssc212/gdb-tutorial-handout.pdf>.