CS 246 Spring 2019 - GDB Tutorial

Josh Rampersad

June 14, 2019

GDB is the GNU Project Debugger. It is a tool that allows us to see what is going on inside a program as it runs. This makes identifying and fixing bugs much easier and can save us a lot of time that would have been spent scratching our heads at why our program does not work. GDB is a very powerful and versatile tool, so we will be exploring a subset of its functionalities which will come in handy while writing programs for this course.

1 Summary

- Compiling and Running in GDB
- Breakpoints and Stepping Through Code
- Navigating the Stack
- Watchpoints
- Running in Reverse and Modifying Behaviour
- Approaches to Debugging

2 Compiling and Running in GDB

- Before we start using GDB, we need to compile our program such that GDB has all of the information it needs to debug our code. We do this by running the compiler with the `-g` flag. This would appear as follows.

  ```
  g++ -std=c++14 -g file1.cc file2.cc ...
  ```

  The `-g` flag tells the compiler to do two things:

  - Disable optimizations
  - Ship with a copy of the source code

  Disabling optimizations stops the compiler from messing around with our code so that we can debug it as written. Shipping with the source gives GDB a ton of information that it can use to help us find and fix bugs.

- Now that we have compiled an executable of our program, we can run it with GDB as follows

  ```
  gdb ./myprogram
  ```

  This will run GDB on our executable file, meaning that we are now ready to start debugging. The first GDB command we will learn is the `run` command. This tells GDB to start running our program.

  ```
  (gdb) run
  ```

  Once we hit Enter, the program will execute as if we had run it outside of GDB. What is handy about the run command is that it can take arguments and input redirection. So, if I wanted to run `./myprogram` with arguments and input from a file, it would look something like this.

  ```
  (gdb) run arg1 arg2 <infile
  ```

  This is fine and all, but so far, GDB has not shown any functionality past what we are already able to do. Now, we will see just what GDB has to offer.

3 Breakpoints and Stepping Through Code

- Let's say that I have a program `basic1.cc` that is written as follows

  ```
  #include <iostream>

  int main(){
    int j = 3;
    int k = 7;

    j += k;
    k = j * 2;
    std::cout << "Hello there" << std::endl;
  }
  ```

  Now, I can use my `run` command to run this program in GDB, but if we want to debug our code, we will want a way to stop the program during execution to see what it is doing. For this, GDB gives us something called a **breakpoint**. Breakpoints are points in our program where we want GDB to stop running. We can set a breakpoint at any line in our program and GDB will stop executing the program when it arrives at that line. To set a breakpoint, we can use the `break` command.

  ```
  (gdb) break main
  Breakpoint 1 at 0x938: file basic1.cc, line 4.
  (gdb)
  ```

This will set a breakpoint at the main function of our program. Now when we run our program in GDB, it will stop when it reaches the main function and then wait for us to give it more commands. It looks something like this.

```
(gdb) run
Starting program: /u/userid/cs246/1195/tutorials/gdb/basic1

Breakpoint 1, main () at basic1.cc:4
4   int j = 3;
(gdb)
```

So the program stops at breakpoint 1, which is our main function, and GDB prints out the next line of code to be run. In this case, it is the definition of our variable `j`.

- Now that we are at this point, let's say that I want to step through my program one command at a time. In other words, I want to run this next command `int j = 3;` and then stop at the following command. The command to step over one line of code is `next`. Typing in the command and hitting Enter will give us the following output.

```
(gdb) next
5   int k = 7;
(gdb)
```

The next command has just told GDB to run the command at line 4 (`int j = 3;`) and then to output the next line to run. This output is telling us that GDB has stopped at line 5. The line we see is the next piece of code to be executed.

- Now, before we continue to go through the program, I want to check the current value of `j`. The previous command supposedly set `j` to `3`, but I want to make sure that at this point in our program, that the value is what I want it to be. GDB provides us with a handy `print` command which allows us to peek at the values of variables throughout the execution of our program. We just need to tell GDB what we want to print and, if it is able, it will print out the requested value.

```
(gdb) print j
$1 = 3
(gdb)
```

This tells us that `j` is 3, so we can continue with execution. I will continue through my program as follows/

```
(gdb) next
7   j += k;
(gdb) next
8   k = j * 2;
(gdb) print j
$2 = 10
(gdb) print j + k
```

```
$3 = 17
(gdb) print (j + k) * 3
$4 = 51
(gdb)
```

This exhibits one of the really useful and powerful features of GDB, the fact that it understands C/C++ expression syntax. This means that I can ask GDB to print out an expression based on values in my program and it will evaluate it for me.

4 Navigating the Stack

Now, we will take a look at an example of a piece of buggy code and how we can use GDB to isolate where the bug is. We will be using the `example1.cc` program. Here is the program:

```
#include "black_box.h"

void crash(int *i){
  *i = 1;
}

void f(int *i){
  int *j = i;
  j = sophisticated(j);
  j = complicated(j);
  crash(j);
}

int main(){
  int i;
  f(&i);
}
```

In our program, we have two black box functions `sophisticated` and `complicated`. These two functions take `int*`s as parameters and return `int*`s. Let's break at our function `crash` and see what happens.

```
(gdb) break crash
Breakpoint 1 at 0x6f5: file example1.cc, line 4.
(gdb) run
Starting program: /u8/userid/cs246/1195/tutorials/gdb/example1

Breakpoint 1, crash (i=0x0) at example1.cc:4
4   *i = 1;
(gdb)
```

As we can see from the output upon breaking at `crash`, the value of `i` is `nullptr`. This means that when we dereference variable `i` in the next line of code, this will cause a Segmentation Fault. So, how did we get here? Why am I passing `nullptr` to my function?

- GDB provides us with many ways of seeing different information about the current state of our program. Information about the current state of the call stack is readily available at all times in GDB. Using the commands `up` and `down`, we can move up and down the call stack to see how we arrived at the current point in our program.

```
(gdb) up
#1  0x0000555555554742 in f (i=0x7fffffffe8b4) at example1.cc:11
11   crash(j);
(gdb) up
#2  0x0000555555554768 in main () at example1.cc:16
16   f(&i);
(gdb) down
#1  0x0000555555554742 in f (i=0x7fffffffe8b4) at example1.cc:11
11   crash(j);
(gdb) down
#0  crash (i=0x0) at example1.cc:4
4   *i = 1;
(gdb)
```

  As we can see, by moving up and down through the different frames of our call stack, we can see that our `crash` function was called from within function `f`. By further moving up the stack, we can see where in our main function `f` was called.

- Now, while this ability to move up and down the stack is nice, when we have more complicated programs with more function calls, it can become tedious to repeatedly call `up` and `down` to see our stack frames. This is why GDB gives us a handy command to see our entire stack in one command, `backtrace`.

```
(gdb) backtrace
#0  crash (i=0x0) at example1.cc:4
#1  0x0000555555554742 in f (i=0x7fffffffe8b4) at example1.cc:11
#2  0x0000555555554768 in main () at example1.cc:16
(gdb)
```

  Calling `backtrace` gives us a bird's eye view of the whole call stack. This can be very enlightening if we want to understand the context in which a bug occurs. From this, we can see that when function `f` is called, the value of `i` is not nullptr. This means that there is some point between `main`'s call to `f` and `f`'s call to `crash` where the pointer is set to nullptr.

- In order to find where the pointer is changed, lets break at function `f` and then step through while keeping track of our pointer. We can do this using the `display` command. This will tell GDB to print out the value of `j` after each command.

```
(gdb) run
Starting program: /u8/j2ramper/cs246/1195/tutorials/gdb/example1

Breakpoint 1, f (i=0x7fffffffe8b4) at example1.cc:8
8   int *j = i;
```

```
(gdb) display j
1: j = (int *) 0x0
(gdb) next
9   j = sophisticated(j);
1: j = (int *) 0x7fffffffe8b4
(gdb) next
10   j = complicated(j);
1: j = (int *) 0x7fffffffe8b4
(gdb) next
11   crash(j);
1: j = (int *) 0x0
(gdb)
```

Now, we can see that it is function `complicated` which sets `j` to `nullptr`. After this, if we want to stop displaying variable `j`, we can just use command `undisplay j` to tell GDB to stop displaying the value of `j` at every line.

- Let's move on from our `example1.cc` program to see an instance of when viewing the call stack can be useful. Those of you who remember CS135 will recall that many programming problems can be solved in few lines of code, simply by making use of recursion. For example, if I want to calculate the factorial of a number, we can see that there is a simple recursive solution to this problem.

```
#include <iostream>

int factorial(const int &n){
  if(n){
    return n * factorial(n - 1);
  }else{ // Zero Base Case
    return 1;
  }
}

int main(){
  int n;

  std::cout << "Please enter a positive integer: ";
  if(std::cin >> n && n >= 0){
    std::cout << n << "! = " << factorial(n) << std::endl;
  }else{
    std::cout << "That's not a positive integer!" << std::endl;
  }
}
```

We can use GDB to try and understand the behaviour of the program through its recursive calls. Let's set a breakpoint at the recursive function `factorial` and see what happens when we run.

```
(gdb) break factorial
Breakpoint 1 at 0xafd: file recursive.cc, line 3.
(gdb) run
Starting program: /u8/userid/cs246/1195/tutorials/gdb/recursive
Please enter a positive integer: 4

Breakpoint 1, factorial (n=@0x7fffffffe894: 4) at recursive.cc:3
3 int factorial(const int &n){
(gdb)
```

Now that I am here, I know that `factorial` will be called multiple times. Let's step through the program to see how the recursive calls behave.

```
(gdb) next
4   if(n){
(gdb) next
5     return n * factorial(n - 1);
(gdb)
```

Now that I am at line 5, instead of stepping over this line, I want to step into function `factorial` and run through the code in that function call. To step into a line of code instead of stepping over it, we use the `step` command.

```
(gdb) step

Breakpoint 1, factorial (n=@0x7fffffffe864: 3) at recursive.cc:3
3 int factorial(const int &n){
(gdb) next
4   if(n){
(gdb)
```

We can now see that we are in the function factorial when it is called with `n` being `3`. We can step through it as we always have. Note that even though the breakpoint did not stop our execution, GDB still tells us that this line is a breakpoint.

- At this point, we know that this recursive function will call itself many more times. It will take a long time to step through all of these function calls. We can actually make use of the fact that our function is a breakpoint. Instead of stepping through one line at a time, we can tell our program to keep running until it hits a breakpoint. This can be achieved by using the `continue` command.

```
(gdb) continue
Continuing.

Breakpoint 1, factorial (n=@0x7fffffffe824: 2) at recursive.cc:3
3 int factorial(const int &n){
(gdb) continue
Continuing.
```

```
Breakpoint 1, factorial (n=@0x7fffffffe7e4: 1) at recursive.cc:3
3 int factorial(const int &n){
(gdb)
```

When we use command `continue`, GDB will indicate that it is continuing and then stops when it reaches another breakpoint.

- Now, if we take a look at the call stack, we can examine what function calls with what values led to this point.

```
(gdb) bt
#0  factorial (n=@0x7fffffffe7e4: 1) at recursive.cc:3
#1  0x0000555555554b34 in factorial (n=@0x7fffffffe824: 2) at recursive.cc:5
#2  0x0000555555554b34 in factorial (n=@0x7fffffffe864: 3) at recursive.cc:5
#3  0x0000555555554b34 in factorial (n=@0x7fffffffe894: 4) at recursive.cc:5
#4  0x0000555555554bd3 in main () at recursive.cc:16
(gdb)
```

Notice that in recursive calls, the `backtrace` command can bring a lot of clarity as to how our recursive function behaves.

- Now, if we continue, we will hit our base case where `n=0`.

```
(gdb) c
Continuing.

Breakpoint 1, factorial (n=@0x7fffffffe7a4: 0) at recursive.cc:3
3 int factorial(const int &n){
(gdb)
```

At this point, if we continue again, the stack will unravel and we will get our final result. If, however, we want to see what each recursive call returns as we pop successive frames off the stack, GDB allows us to run to the end of the current function call and stop once the current stack frame is popped. The `finish` command allows us to run to the end of the current function and to see what it returns.

```
(gdb) finish
Run till exit from #0  factorial (n=@0x7fffffffe7a4: 0) at recursive.cc:3
0x0000555555554b34 in factorial (n=@0x7fffffffe7e4: 1) at recursive.cc:5
5     return n * factorial(n - 1);
Value returned is $1 = 1
(gdb) fin
Run till exit from #0  0x0000555555554b34 in factorial (n=@0x7fffffffe7e4: 1) at recur
0x0000555555554b34 in factorial (n=@0x7fffffffe824: 2) at recursive.cc:5
5     return n * factorial(n - 1);
Value returned is $2 = 1
(gdb)
```

```
Run till exit from #0  0x0000555555554b34 in factorial (n=@0x7fffffffe824: 2) at recur
0x0000555555554b34 in factorial (n=@0x7fffffffe864: 3) at recursive.cc:5
5     return n * factorial(n - 1);
Value returned is $3 = 2
(gdb)
Run till exit from #0  0x0000555555554b34 in factorial (n=@0x7fffffffe864: 3) at recur
0x0000555555554b34 in factorial (n=@0x7fffffffe894: 4) at recursive.cc:5
5     return n * factorial(n - 1);
Value returned is $4 = 6
(gdb)
Run till exit from #0  0x0000555555554b34 in factorial (n=@0x7fffffffe894: 4) at recur
0x0000555555554bd3 in main () at recursive.cc:16
16     std::cout << n << "! = " << factorial(n) << std::endl;
Value returned is $5 = 24
(gdb)
```

When we run `finish`, GDB will tell us the function we are finishing, the function we return to, the line we return to, and the value returned. All this from one command! Wow!

5 Watchpoints

Assume we have a complicated program like `example2.cc`:

```
 1 #include <iostream>
 2 #include "black_box.h"
 3
 4 void small(int &a){
 5   a /= 68;
 6
 7 }
 8
 9 int bar(int &p){
10   p *= 3;
11   return unknown(p);
12 }
13
14 void oof(int &n){
15   n *= n - 20;
16   n = n - bar(n);
17 }
18
19 void foo(int &z){
20   z = bar(z);
21   oof(z);
22   unknown(z);
23   small(z);
24   ++z;
```

```
25 }
26
27 int main(){
28   int x = 10;
29   foo(x);
30
31   if(x == 0){
32     std::cerr << "Error: x set to 0" << std::endl;
33     return 3;
34   }
35 }
```

All we want is to make sure that by the time we reach the `if` statement, `x` is not `0`. However, when we run the program, we get the error message that `x set to 0`. There is a lot of crazy function calling in this program, so let's start by trying to step through the program.

```
(gdb) break main
Breakpoint 1 at 0xa8d: file example2.cc, line 27.
(gdb) run
Starting program: /u8/userid/cs246/1195/tutorials/gdb/example2

Breakpoint 1, main () at example2.cc:27
27 int main(){
(gdb) n
28   int x = 10;
(gdb)
29   foo(x);
(gdb)
```

Now, this is a complicated program and we want to be able to run through the program and keep track of how `x` is being modified as we run.

- GDB provides more than one way to stop during execution. The first way is to set a breakpoint. This will stop the program when a specific line is reached. There is another option where we can tell the program to instead stop whenever a variable is modified. This is called a watchpoint. We can set a watchpoint with the `watch` command and the program will stop whenever the variable we specify is changed.

  ```
  (gdb) watch x
  Hardware watchpoint 2: x
  (gdb) c
  Continuing.

  Hardware watchpoint 2: x

  Old value = 10
  New value = 30
  bar (p=@0x7fffffffe8b4: 30) at example2.cc:11
  11   return unknown(p);
  (gdb)
  ```

As we can see, GDB stops the program when the value of `x` is changed and it tells us the old value and new value of our variable. We can use this to run through our program to follow how a variable changes throughout execution. This can help us pinpoint where it might become an unwanted value.

- One thing that can happen over the course of debugging is that we can accumulate breakpoints and watchpoints. While they are a useful tool, they can become cumbersome as they are always stopping program execution. GDB provides us with a way to track our breakpoints and watchpoints. If we ever want to see which stopping points are set, we can use the `info breakpoints` command.

```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000555555554a8d in main() at example2.cc:27
breakpoint already hit 1 time
2       hw watchpoint  keep y                      x
breakpoint already hit 1 time
(gdb)
```

  This command prints out a table with information on each of the stopping points in the program.

- Now that we have the information on all of our breakpoints, it would be nice to have a way to get rid of them without restarting GDB. Using the `Num` column of the `info breakpoints` output, we can determine the id of each stopping point. In order to remove it, we just need to call the `delete` command.

```
(gdb) delete 1
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
2       hw watchpoint  keep y                      x
breakpoint already hit 1 time
(gdb)
```

  As we can see from the new output of `info breakpoints`, the breakpoint at the main function was deleted. If we want to delete all of the stopping points in the program, we can call `delete` with no id number specified.

6 Running in Reverse and Modifying Behaviour

Let's take a look at another example in `example3.cc`:

```
#include "black_box.h"
#include "fun.h"

int main(){
  int x = 16;
```

```
  mystery(x,1);
  if(x % 2){
    fun(x);
  }else{
    funner(x);
  }
}
```

We do not know what function `mystery` does, just that it takes an `int&` and an `int` as parameters. We will use this example to show some of the more advanced functionalities that GDB has to offer.

- Let's say that I want to be able to run my program in reverse from the point I found a bug in order to find the code which causes the bug. GDB can do this, but only in specific circumstances. In order for this to work on C++ in the student environment, we need to tell GDB to record what a program does in order to be able to rewind. This can be done with the `target record-full` command. Once this command has been called, we can use the `next`, `step`, and `continue` commands in reverse.

  ```
  (gdb) run
  4 int main(){
  (gdb) target record-full
  (gdb) next
  5   int x = 16;
  (gdb) next
  7   mystery(x,1);
  (gdb) next
  8   if(x % 2){
  (gdb) next
  9     fun(x);
  (gdb) reverse-next
  8   if(x % 2){
  (gdb) reverse-next
  7   mystery(x,1);
  (gdb) reverse-next
  5   int x = 16;
  (gdb)
  ```

  This may not always work in certain function and in certain loops, but it can come in handy when you step just a bit too far in your code.

- If we want to change the value of a variable as we are debugging a program, we can use the `set var` command. This will allow us to change the value stored in a variable so that we can alter the behaviour of a program as it runs.

  ```
  (gdb) next
  5   int x = 16;
  (gdb) next
  7   mystery(x,1);
  ```

```
(gdb) print x
$1 = 16
(gdb) set var x=15
(gdb) print x
$2 = 15
(gdb)
```

This can come in handy if we want to see what effect altering our code will have without having to change the source code and recompile.

7 GDB Command Reference

Command	Abbreviation	Description
run	r	Runs the selected program until the next Breakpoint or Watch
break	b	Sets a breakpoint at a line or function (Current line by default)
watch	wa	Sets a watchpoint
next	n	Steps through next line of code (Does not enter function call)
step	s	Steps into next line of code (Enters function call if any)
continue	c	Run from current point to next Breakpoint or Watch
info breakpoints	i b	List out all breakpoints and watchpoints
delete	d	Delete breakpoint or watchpoint with given id
print	p	Prints out the value of the given variable
whatis	what	Prints out the type of the given variable
display	disp	Print out the value of a variable at each step
undisplay	undisp	Undo a "display" command
up	up	Move up the call stack
down	do	Move down the call stack
backtrace	bt	Print a trace of the current call stack
finish	fin	Run to the end of the current function call
set var	set var	Modify the value stored in a variable during execution