Week 2: Executing a Program

2.1 What Else Can GDB Do?

As your projects get larger, print statements such as `Reached line 134` will be insufficient for debugging. In this article, we'll learn about how to set breakpoints, view the values of variables as your program runs, and start/stop program execution.

2.2 Useful Commands

Here are some useful commands to know about GDB. This will be at the top of every article, and we'll highlight the new commands. The highlighted commands will be the focus of the article.

- `help (h) [optional topic or command]`
  - Learn more about certain topics or commands
- `file [program name]`
  - Load a program to gdb.
- `run (r)`
  - Run the program
- `quit (q)`
  - Quit gdb

We also went over `break` and `print`, but we will review them one more time.

break (b)

```
(gdb) break [file]:[line number]
```

A breakpoint tells gdb to pause your program on `line number` in `file`.

When you reach a breakpoint, you can set another breakpoint in the **same** file by just specifying the `line number`.

```
(gdb) break [line number]
```

You can even set a breakpoint by function name. For example:

```
(gdb) b calcator.cpp:add
```

clear

```
(gdb) clear [file]:[line number]
```

Removes a breakpoint set by `break`. This can be invoked similary to `break`.

For example:

```
(gdb) clear calculator.cpp:add
```

continue (c)

```
(gdb) continue
```

Continues program being debugged, after a signal or breakpoint.

Optionally, you can add an argument `n`, where `n` is a positive integer. This tells gdb to continue ignoring whichever breakpoint you stopped at `n-1` times. This is helpful when you have loops.

```
(gdb) c 10
```

next (n)

```
(gdb) next [N]
```

After a signal or breakpoint, you can execute the next line of code in the function by calling `next`. This will *not* step into subroutines. Instead, it will execute the entire subroutine (unless stopped by a signal or another breakpoint) and continue to the next line in the function.

Optionally, you can pass in an integer `N` to step next `N` times.

step (s)

```
(gdb) step [N]
```

Similar to `next`, except you can *step* into subroutines. Optionally, you can pass in an integer `N` to step `N` times.

finish

```
(gdb) finish
```

Continue execution until the current routine returns.

print

```
(gdb) print [variable | expression]
```

Print the value of the variable or expression. For example:

```
(gdb) print x
$1 = 10
```

```
(gdb) print foo(5)
$2 = 25
```

Note: When you print a value, gdb outputs the information in the format:

"$N = (value)".

N is the count of the variables you examined while debugging. This let's you also view values by calling:

```
(gdb) print $1
$3 = 10
```

set

```
(gdb) set [variable]
```

Modify the value of the variable. For example:

```
(gdb) set x = 3
```

2.3 Task 2 - A Strange Family

In this task, we'll be including the concepts of classes and inheritance.

Download gotw-02.sh

Take a look a the header and cpp files to see what the `pet` and `rock` classes should be doing. Now run `make` and try to run the program.

You should see something like this:

```
$ ./main
Moo I'm a duck.
I am consuming something edible.
What are you doing? I'm a rock.
.
.
.
Rock has evolved into Boulder!
Boulder cannot evolve into anything further. Boulder can only devolve now.
We can, however, list all of it's previous forms:
1: Pebble
2: Rock
3: Boulder
4: Dust
Segmentation fault (core dumped)
```

Oh no, it's the dreaded `Segmentation fault`!

While it looks like the output of the code isn't very useful, the error message - `Segmentation fault` - actually reveals what kind of problem we have. Specifically, we run into segfaults when we try to use memory that is not ours.

When does that happen? Maybe when we:

- write to read-only memory
- attempt to access freed memory
- attempt to access NULL pointers

There are other instances which can cause segfaults. Can you think of any?

Submission:

The files you should be modifying and submitting are:

- `pet.cpp`
- `rock.cpp`
- `rock.h`

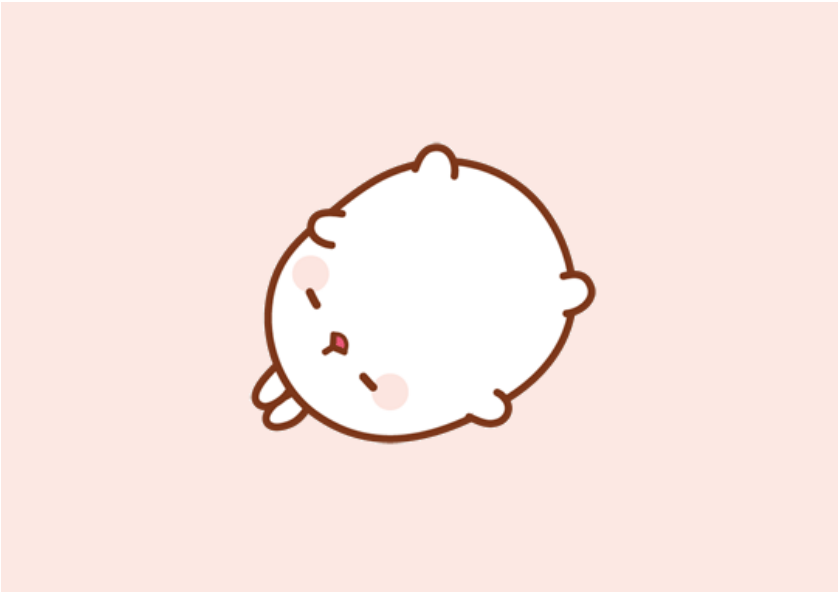
*Note: You should not be modifying any constructors.

2.4 Conclusion

Great job! In this lesson, we learned:

- how to evaluate variables in gdb
- how to control program execution in gdb

And that's all for today, folks!



Upload Solution

Drop files here or click to upload.

Only the files listed below will be accepted—others will be ignored.

Files

- pet.cpp
  not uploaded
- rock.cpp
  not uploaded
- rock.h
  not uploaded

Save & Grade | Save only