CS 35L- Software Construction Laboratory

Fall 2018

TA: Guangyu Zhou

Lab 3

C Programming and Debugging

Week 4

Outline

- **GDB – Debugging**
- Issues on C Programming

Debugging

- Goal: Find and eliminate errors from programs
- Steps
  - Reproduce the bug
  - Simplify the program input
  - Use a debugger to track down the origin of the problem
  - Fix the problem

Debug is Annoying!

- We need efficient tools for debugging

Things happened during debugging

Debugger

- A program that is used to run and debug other programs
- Advantages
  - Step through the code line by line, execute on demand
  - Interact with and inspect program at run-time
  - For crashes, the debugger outputs why and where crashes happened

GDB- GNU Debugger

- Debugger for multiple languages
  - C, C++, Java, Objective-C… more
- Allow you to inspect what the program is doing at certain point during execution
- Make it easy to find logical errors and segment faults

GDB usage: Compile the program

- Normal: *$ gcc [flags] <source files> -o <output file>*

```
gcc -Wall -Werror -ansi -pedantic-errors prog1.c -o prog1.x
```

- Debug: *$ gcc [other flags] –g <source files> -o <output file>*
  - *-g option will enable built-in debugging support (which gdb needs)*

```
gcc -Wall -Werror -ansi -pedantic-errors -g prog1.c -o prog1.x
```

GDB: starting

- Specify program to debug
  - $ gdb <executable> or *$ gdb*
  - Just try “gdb” or “gdb prog1.x.” You’ll get a prompt that looks like this:

```
(gdb)
```

- If you didn’t specify a program to debug, you’ll have to load it in now:
  - *(gdb) file <executable>*

```
(gdb) file prog1.x
```

  - prog1.x is the program you want to load, and “file” is the command to load it.

gdb interactive shell

- gdb has an interactive shell, much like the one you use as soon as you log into the Linux machines. It can recall **history** with the arrow keys, **auto-complete** words (most of the time) with the TAB key, and has other nice features.

Tip

If you're ever confused about a command or just want more information, use the "`help`" command, with or without an argument:

```
(gdb) help [command]
```

You should get a nice description and maybe some more useful tidbits. . .

GDB usage:

- Run program
- To run the program, just use:
  - (gdb) run [arguments]
- This runs the program.
  - If it has no serious problems (i.e. the normal program didn't get a segmentation fault, etc.), the program should run fine here too.
  - If the program did have issues, then you (should) get some useful information like the line number where it crashed, and parameters to the function that caused the error:

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400524 in sum_array_region (arr=0x7fffc902a270, r1=2, c1=5,
r2=4, c2=6) at sum-array-region2.c:12
```

GDB usage:

- In GDB interactive shell
  - Tab: autocomplete
  - <UP> / <DOWN> arrow: recall history
  - help [command]: like **man** in linux

- Exit the GDB debugger
  - (gdb) quit

What can GDB do?

- If you run successfully w/o errors, no need for debugger.
- Chances are if this is not the case, you don't want to run the program without any stopping, breaking, etc. Otherwise, you'll just rush past the error and never find the root of the issue. So, you'll want to **step through your code a bit at a time**, until you arrive upon the error.

- => Breakpoints!
- Breakpoints can be used to stop the program run in the middle, at a designated point. The simplest way is the command "break."

Break points

- break [filename] line_number
  - Sets a breakpoint at the specified line in the current source file, or in the source file *filename*, if specified.

```
(gdb) break file1.c:6
```

- break [function]
  - Sets a breakpoint at the first line of the specified function.
- info breakpoints
  - Shows information about all declared breakpoints

Tip

You can set as many breakpoints as you want, and the program should stop execution if it reaches any of them.

Deleting, Disabling, and Ignoring BP

- (gdb) delete [bp_number | range]; d [*bp_number* | *range*]
  - Deletes the specified breakpoint or range of breakpoints
  - Without arguments: delete all break points
- disable [bp_number | range]
  - Temporarily deactivates a breakpoint or a range of breakpoints.
- enable [bp_number | range]
  - Restores disabled breakpoints.
- ignore [bp_number] [iterations]
  - Instructs GDB to pass over a breakpoint without stopping a certain number of times.
  - Arguments: the number of breakpoint, the number of times to be passed over

More about breakpoints

- Breakpoints by themselves may seem too tedious. You have to keep stepping, and stepping, and stepping. . .
- Once we develop an idea for what the error could be, we only care if such an event happens; we don’t want to break at each iteration regardless.
- So ideally, we’d like to condition on a particular requirement (or set of requirements). Using conditional breakpoints allow us to accomplish this goal. . .

Conditional Breakpoints

- break [*position*] if *expression*

```
(gdb) break file1.c:6 if i >= ARRAYSIZE
```

- This command sets a breakpoint at line 6 of file file1.c, which triggers only if the variable i is greater than or equal to the size of the array (which probably is bad if line 6 does something like arr[i]).
- Conditional breakpoints can most likely avoid all the unnecessary stepping, etc.

After breakpoint, then what?

- Once you've set a breakpoint, you can try using the **run** command again. This time, it should stop where you tell it to (unless a fatal error occurs before reaching that point).
- You can proceed onto the next breakpoint by typing "**continue**" (Typing **run** again would restart the program from the beginning, which isn't very useful.)
- You can single-step (execute just the next line of code) by typing "**step**." This gives you really fine-grained control over how the program proceeds. You can do this a lot...

Resuming Execution After a Break

- continue [*passes*] , c [*passes*]
  - Allows the program to run until it reaches another breakpoint, or until it exits if it doesn't encounter any further break points.
- step [*lines*] , s [*lines*]
  - Executes the current line of the program, and stops the program again before executing the line that follows
- next [*lines*] , n [*lines*]
  - Similar to "step," the "next" command single-steps as well, except this one **doesn't execute each line of a sub-routine,** it just treats it as one instruction.
- finish
  - To resume execution until the current function returns, use the finish command

Tip

Typing "`step`" or "`next`" a lot of times can be tedious. If you just press ENTER, `gdb` will repeat the same command you just gave it. You can do this a bunch of times.

Displaying variables

- So far you've learned how to interrupt program flow at fixed, specified points, and how to continue stepping line-by-line.
- What if you want to see things like the values of variables, etc.?
- print [/*format*] [*expression*], p [/*format*] [*expression*]
- Output Formats
  - d: Decimal notation (also default)
  - u: Decimal notation, unsigned value
  - x: Hexadecimal notation
  - o: Octal notation
  - t: Binary notation
  - c: Character, displayed together with the character code in decimal notation/

Watch points

- Whereas breakpoints interrupt the program at a particular line or function, **watchpoints** act on variables. They pause the program whenever a watched variable's value is modified.
- watch [*expr]*
  - The debugger stops the program when the value of [expr] **changes**
- rwatch [expr]
  - The debugger stops the program whenever the program **reads** the value of any object involved in the evaluation of [expr]
- awatch [expr]
  - The debugger stops the program whenever the program **reads or modifies** the value of any object involved in the evaluation of [expr]

Program Process Layout

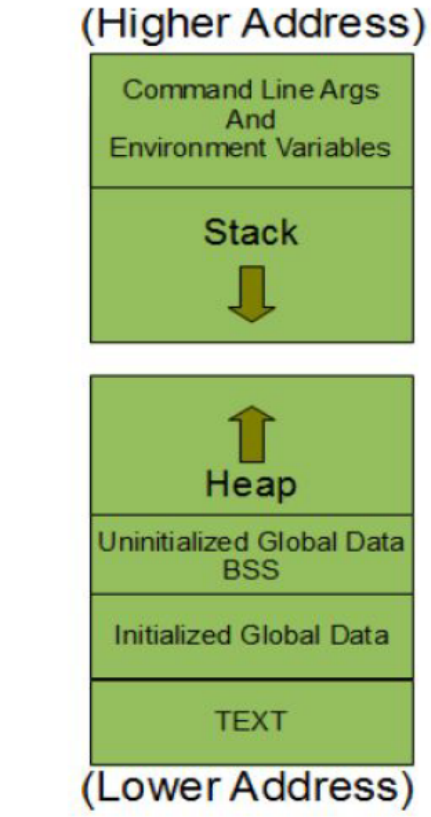

Image source : thegeekstuff.com

- TEXT segment
  - Contains machine instructions to be executed
- Global variables
  - Initialized\uninitialized
- Heap segment
  - Dynamic memory allocation
  - Malloc, free
- Stack segment
  - Push frame: function invoked
  - Pop frame: function returned
  - Stores: local variables; return address, registers
- Command line arguments and environmental variables

Stack Information

- A program is made up of one or more functions which interact by calling each other
- Every time a function is called, an area of memory is set aside for it. This area is called stack frame and holds the following information:
  - Storage space for all local variables
  - Memory address to return after the called function returns
  - Arguments of the called function
- Each function call gets its own stack frame, all stack frames make up the **call stack**

Stack Frames and the Stack Demo

```
1    #include <stdio.h>
2    void first_function(void);
3    void second_function(int);
4
5    int main(void)
6    {
7        printf("hello world\n");
8        first_function();
9        printf("goodbye goodbye\n");
10
11       return 0;
12   }
13
14
15   void first_function(void)
16   {
17       int imidate = 3;
18       char broiled = 'c';
19       void *where_prohibited = NULL;
20
21       second_function(imidate);
22       imidate = 10;
23   }
24
25
26   void second_function(int a)
27   {
28       int b = a;
29   }
```

Frame for `main()`

Frame for `first_function()`
- Return to `main()`, line 9
- Storage space for an int
- Storage space for a char
- Storage space for a void *

Frame for `second_function()`:
- Return to `first_function()`, line 22
- Storage space for an int
- Storage for the int parameter named `a`

execution within second_function()

Analyzing the Stack

- backtrace, bt
  - Shows the call trace
- info frame
  - Displays information about the current stack frame, including its return address and saved register values.
- info locals
  - Lists the local variables of the function corresponding to the stack frame, with their current values.
- info args
  - List the argument values of the corresponding function call

Running

`# gdb <program> [core dump]`
Start GDB (with optional core dump).

`# gdb --args <program> <args…>`
Start GDB and pass arguments

`# gdb --pid <pid>`
Start GDB and attach to process.

`set args <args...>`
Set arguments to pass to program to be debugged.

`run`
Run the program to be debugged.

`kill`
Kill the running program.

Breakpoints

`break <where>`
Set a new breakpoint.

`delete <breakpoint#>`
Remove a breakpoint.

`clear`
Delete all breakpoints.

`enable <breakpoint#>`
Enable a disabled breakpoint.

`disable <breakpoint#>`
Disable a breakpoint.

Watchpoints

`watch <where>`
Set a new watchpoint.

`delete/enable/disable <watchpoint#>`
Like breakpoints.

<where>

function_name
Break/watch the named function.

line_number
Break/watch the line number in the current source file.

file:line_number
Break/watch the line number in the named source file.

Conditions

`break/watch <where> if <condition>`
Break/watch at the given location if the condition is met.
Conditions may be almost any C expression that evaluate to true or false.

`condition <breakpoint#> <condition>`
Set/change the condition of an existing break- or watchpoint.

Examining the stack

`backtrace`
`where`
Show call stack.

`backtrace full`
`where full`
Show call stack, also print the local variables in each frame.

`frame <frame#>`
Select the stack frame to operate on.

Stepping

`step`
Go to next instruction (source line), diving into function.

`next`
Go to next instruction (source line) but don't dive into functions.

`finish`
Continue until the current function returns.

`continue`
Continue normal execution.

Variables and memory

`print/format <what>`
Print content of variable/memory location/register.

`display/format <what>`
Like „print“, but print the information after each stepping instruction.

`undisplay <display#>`
Remove the „display“ with the given number.

`enable display <display#>`
`disable display <display#>`
En- or disable the „display“ with the given number.

`x/nfu <address>`
Print memory.
n: How many units to print (default 1).
f: Format character (like „print“).
u: Unit.

Unit is one of:

b: Byte,
h: Half-word (two bytes)
w: Word (four bytes)
g: Giant word (eight bytes)).

GDB resources

- Tutorial of GNU GDB Debugger Command:
  - http://www.yolinux.com/TUTORIALS/GDB-Commands.html
- Using GDB Under Emacs:
  - http://tedlab.mit.edu/~dr/gdbintro.html