CS 35L

Software Construction Laboratory

Lecture 4.2

24th October, 2019

Logistics

- Assignment 4
  - Due on October 28th
- Hardware requirement for Week 8
  - Seeed Studio BeagleBone Green Wireless Development Board
- Assignment 10 Signup Sheet
  - Teams of 2
  - https://docs.google.com/spreadsheets/d/1PVqVMEEsHjmmj9YLyqz5K4wU-k0Dwwm2iO9uS1Zq1wk/edit?usp=sharing
  - Names must be filled by Friday of Week 3
    - Topic can be selected later
    - Ensure that topics are not the same
- Office Hours this week:
  - Friday - 9:30 – 11:30 am – BH3256S

Assignment 10 Rubric

- Presentation (50%):
  - Organization
  - Relevance to topic
  - Technical Details and Subject Knowledge
  - Presentation abilities (Elocution and Eye contact)
  - Content of slides
  - Ability to answer questions and interactivity with audience
- Report (50%)

Review - Previous Lab

- C Programming
  - Pointers, Double Pointers
  - Pass by Value, Pass by Reference
  - Structs
  - Dynamic Memory Allocation

Signed vs. Unsigned

https://stackoverflow.com/questions/3812022/what-is-a-difference-between-unsigned-int-and-signed-int-in-c

```
      two's complement   |  ones' complement   |   sign/magnitude
    +--------------------+--------------------+--------------------+
 5  | 0000 0000 0000 0101 | 0000 0000 0000 0101 | 0000 0000 0000 0101 |
-5  | 1111 1111 1111 1011 | 1111 1111 1111 1010 | 1000 0000 0000 0101 |
    +--------------------+--------------------+--------------------+
```

- In two's complement, you get a negative of a number by inverting all bits then adding 1.
- In ones' complement, you get a negative of a number by inverting all bits.
- In sign/magnitude, the top bit is the sign so you just invert that to get the negative.

C Programming

Pointers to Functions

- A pointer that points to a function
- Declaration
  - double (*func_ptr) (double, double);
  - func_ptr = &pow; // func_ptr points to pow()
- Usage
  - // Call the function referenced by func_ptr
  - double result = (*func_ptr)(1.5, 2.0);
  - // The same function call
  - result = func_ptr(1.5, 2.0);

Function Pointers

- Variable which stores address to a function's executable code in memory.

```
#include <stdio.h>
void fun(int a)
{
      printf("Value of a is %d\n", a);
}
int main()
{
      void (*fun_ptr)(int) = &fun;
      (*fun_ptr)(10);
      return 0;
}
```

Qsort Example

```
#include <stdio.h>
#include <stdlib.h>
int compare (const void * a, const void * b) //qsort wants compare function to return int
{
return ( *(int*)a - *(int*)b ); // typecasts void ptr to int ptr and then dereferences it
}
int main () {
      int values[] = { 40, 10, 100, 90, 20, 25 };
      qsort (values, 6, sizeof(int), compare); //pass compare function
      int n;
      for (n = 0; n < 6; n++)  printf ("%d ",values[n]);
      return 0;
}
```

Formatted I/O

- int fprintf(FILE * fp, const char * format, …);
- int fscanf(FILE * fp, const char * format, …);
- FILE *fp can be either:
  - A file pointer
  - stdin, stdout, or stderr
- The format string
  - int score = 120; char player[] = “Mary”;
  - fp = fopen(“file.txt”, “w+”)
  - fprintf(fp, “%s has %d points.\n”, player, score);

Debugging Process

- Reproduce the bug
- Simplify program input
- Use a debugger to track down the origin of the problem
- Fix the problem

Debugger

- A program that is used to run and debug other (target) programs
- Advantages:
  - Programmer can:
    - step through source code line by line
    - each line is executed on demand
  - interact with and inspect program at run-time
  - If program crashes, the debugger outputs where and why it crashed

GDB – GNU Debugger

- Debugger for several languages
  - C, C++, Java, Objective-C… more
- Allows you to inspect what the program is doing at a certain point during execution
- Semantic errors and segmentation faults are easier to find with the help of gdb

Using GDB

- Compile Program
  - Normally: $ gcc [flags] <source files> -o <output file>
  - Debugging: $ gcc [other flags] –g <source files> -o <output file>
    - enables built-in debugging support
- Specify Program to Debug
  - $ gdb <executable> or
  - $ gdb
  - (gdb) file <executable>

Run-Time Errors

- Segmentation fault
  - Program received signal SIGSEGV, Segmentation fault. 0x0000000000400524 in function (arr=0x7fffc902a270, r1=2, c1=5, r2=4, c2=6) at file.c:12
    - Line number where it crashed and parameters to the function that caused the error
- Semantic Error
  - Program will run and exit successfully
- How do we find bugs?

Using GDB

- Run Program
  - (gdb) run or
  - (gdb) run [arguments]
- In GDB Interactive Shell
  - Tab to Autocomplete, up-down arrows to recall history
  - help [command] to get more info about a command
- Exit the gdb Debugger
  - (gdb) quit

Factorial Program – to be Debugged

```
# include <stdio.h>

int main()
{
	int i, num, j;
	printf ("Enter the number: ");
	scanf ("%d", &num );

	for (i=1; i<num; i++)
		j=j*i;
	printf("The factorial of %d is %d\n",num,j);
}
```

Setting Breakpoints

- Breakpoints
  - used to stop the running program at a specific point
  - If the program reaches that location when running, it will pause and prompt you for another command
- Example:
  - (gdb) break 6
    - Program will pause when it reaches line 6
  - (gdb) break my_function
    - Program will pause at the first line of my_function every time it is called
  - (gdb) break [position] if expression
    - Program will pause at specified position only when the expression evaluates to true

Breakpoints

- Setting a breakpoint and running the program will stop program where you tell it to

- You can set as many breakpoints as you want
  - (gdb) info breakpoints|break|br|b shows a list of all breakpoints

Basic Commands

- (gdb)step - Step to next line of code. Will step into a function.
- (gdb)next - Execute next line of code. Will not enter functions.
- (gdb)print <var> - Print value stored in variable.
- (gdb)continue - Continue execution to next break point.
- (gdb)set var <name>=<value> - Executes rest of program with new value of variable.

Deleting, Disabling and Ignoring BPs

- (gdb) delete [bp_number | range]
  - Deletes the specified breakpoint or range of breakpoints
- (gdb) disable [bp_number | range]
  - Temporarily deactivates a breakpoint or a range of breakpoints
- (gdb) enable [bp_number | range]
  - Restores disabled breakpoints
- If no arguments are provided to the above commands, all breakpoints are affected!!
- (gdb) ignore bp_number iterations
  - Instructs GDB to pass over a breakpoint without stopping a certain number of times.
    - bp_number: the number of a breakpoint
    - Iterations: the number of times you want it to be passed over

Displaying Data

- Why would we want to interrupt execution?
  - to see data of interest at run-time:
  - (gdb) print [/format] expression
    - Prints the value of the specified expression in the specified format
  - Formats:
    - d: Decimal notation (default format for integers)
    - x: Hexadecimal notation
    - o: Octal notation
    - t: Binary notation

Resuming Execution After a Break

- When a program stops at a breakpoint
  - 4 possible kinds of gdb operations:
  - c or continue: debugger will continue executing until next breakpoint
    - s or step: debugger will continue to next source line
    - n or next: debugger will continue to next source line in the current (innermost) stack frame
    - f or finish: debugger will resume execution until the current function returns. Execution stops immediately after the program flow returns to the function's caller
    - the function's return value and the line containing the next statement are displayed

Watchpoints

- Watch/observe changes to variables
  - (gdb) watch my_var
    - sets a watchpoint on my_var
    - the debugger will stop the program when the value of my_var changes
    - old and new values will be printed
- (gdb) rwatch my_var
  - The debugger stops the program whenever the program reads the value of my_var

Process Memory Layout

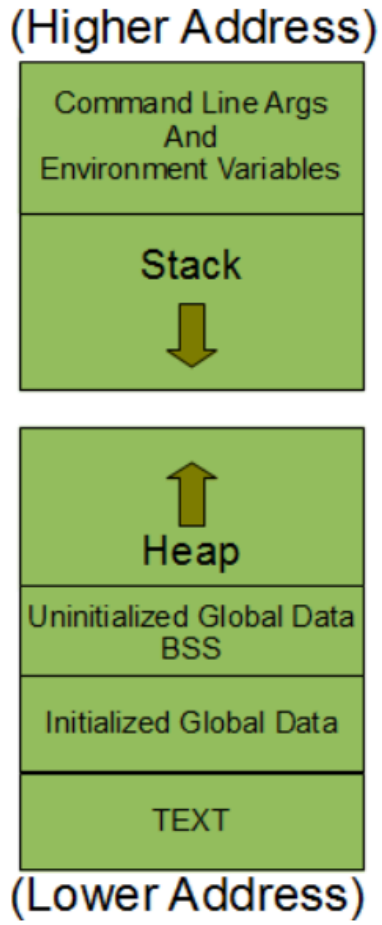


Image source : thegeekstuff.com

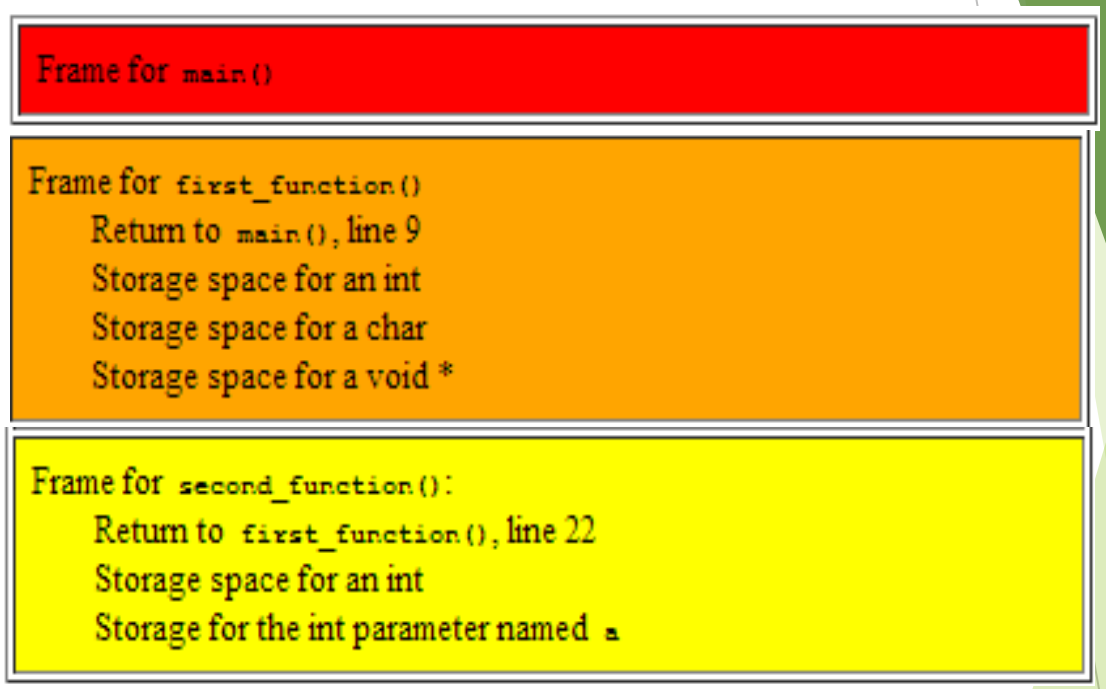
- TEXT segment
  - Contains machine instructions to be executed
- Global Variables
  - Initialized
  - Uninitialized
- Heap segment
  - Dynamic memory allocation
  - malloc, free
- Stack segment
  - Push frame: Function invoked
  - Pop frame: Function returned
  - Stores
    - Local variables
    - Return address, registers, etc
- Command Line arguments and Environment Variables

Stack Info

- 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 of memory is called a stack frame and holds the following crucial info:
  - storage space for all the local variables
  - the memory address to return to when the called function returns
  - the arguments, or parameters, of the called function
- Each function call gets its own stack frame. Collectively, all the stack frames make up the call stack

Stack Frames and the Stack

```
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  }
```



Analyzing the Stack in GDB

- (gdb) backtrace|bt
  - Shows the call trace (the call stack)
  - Without function calls:
    - #0 main () at program.c:10
    - one frame on the stack, numbered 0, and it belongs to main()
  - After call to function display()
    - #0 display (z=5, zptr=0xbffffb34) at program.c:15
      - #1 0x08048455 in main () at program.c:10
  - Two stack frames: frame 1 belonging to main() and frame 0 belonging to display().
  - Each frame listing gives
    - the arguments to that function
    - the line number that's currently being executed within that frame

Analyzing the Stack

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

Other Useful Commands

- (gdb) info functions
  - Lists all functions in the program
- (gdb) list
  - Lists source code lines around the current line

Additional Info on GDB

- Gdb cheat sheet
- Gdb command tutorial and slides
- Running gdb with emacs

Assignment 4 - Laboratory

- Download old version of coreutils with buggy ls program
  - Untar, configure, make
- Bug: ls -t mishandles files whose time stamps are very far in the past. It seems to act as if they are in the future
  - $ tmp=$(mktemp -d)
  - $ cd $tmp
  - $ touch -d '1918-11-11 11:00 GMT' wwi-armistice
  - $ touch now
  - $ sleep 1
  - $ touch now1
  - $ ls -lt wwi-armistice now now1
- Output:
  - -rw-r--r-- 1 eggert eggert 0 Nov 11 1918 wwi-armistice
  - -rw-r--r-- 1 eggert eggert 0 Feb 5 15:57 now1
  - -rw-r--r-- 1 eggert eggert 0 Feb 5 15:57 now
- $ cd
- $ rm -fr $tmp

Fix the Bug!

- Reproduce the Bug
  - Follow steps on lab web page
- Simplify input
  - Run ls with –l and –t options only
- Debug
  - Use gdb to figure out what's wrong
  - $ gdb ./ls
  - (gdb) run –lt /tmp/wwi-armistice /tmp/now /tmp/now1

  (run from the directory where the compiled ls lives)
- Patch
  - Construct a patch "lab4.diff" containing your fix
  - It should contain a ChangeLog entry followed by the output of diff -u

Hints

- Don’t forget to answer all questions! (lab4.txt)
- Make sure not to submit a reverse patch! (lab4.diff)
- “Try to reproduce the problem in your home directory, instead of the $tmp directory. How well does SEASnet do?”
  - Timestamps represented as seconds since Unix Epoch
  - SEASnet NFS filesystem has unsigned 32-bit time stamps
  - Local File System on Linux server has signed 32-bit time stamps
  - If you touch the files on the NFS filesystem it will return timestamp around 2054
  - => files have to be touched on local filesystem (df –l)
- Use “info functions” to look for relevant starting point
- Use “info locals” to check values of local variables

Assignment 10 - Presentations

- Today's Presentation
  - Alvin Nguyen
  - Henry Kou
- Next Class
  - Sheet not filled!

Questions?