CS35L Software Construction Laboratory

Lab 5: Sneha Shankar

Week 5; Lecture 1

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 comparator 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 comparator function

        int n;
        for (n = 0; n < 6; n++)  printf ("%d ",values[n]);
        return 0;
}
```

Structs

- No classes in C
- Used to package related data (variables of different types) together
- Single name is convenient

```
struct Student {
        char name[64];
        char UID[10];
        int age;
        int year;
};

struct Student s;
```

```
typedef struct{
        char name[64];
        char UID[10];
            int age;
        int year;
    } Student;

Student s;
```

typedef Declarations

- Easy way to use types with complex names

```
typedef struct { double x, y; } Point_t;

typedef struct
{
   Point_t top_left;
   Point_t bottom_right;
} Rectangle_t;
```

Dynamic Memory

- Memory that is allocated at runtime
- Allocated on the heap

void *malloc (size_t size);

- Allocates *size* bytes and returns a pointer to the allocated memory

void *realloc (void *ptr, size_t size);

- Changes the size of the memory block pointed to by *ptr* to *size* bytes

void free (void *ptr);

- Frees the block of memory pointed to by *ptr*

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
- Logical errors and segmentation faults are easier to find with the help of gdb

Using GDB

1. **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
2. **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
- Logic Error
  - Program will run and exit successfully
- How do we find bugs?

Using GDB

3. Run Program

- `(gdb) run` or
- `(gdb) run [arguments]`

4. In GDB Interactive Shell

- Tab to Autocomplete, up-down arrows to recall history
- `help [command]` to get more info about a command

5. Exit the gdb Debugger

- `(gdb) quit`

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 file1.c:6`
    - Program will pause when it reaches line 6 of file1.c
  - `(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 expression`
    - The debugger stops the program whenever the program reads the value of any object involved in the evaluation of *expression*

Process Memory Layout

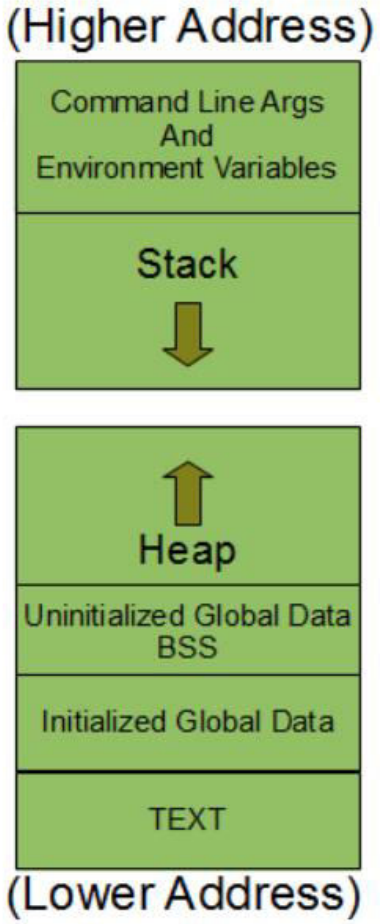


Image source : thegeekstuff.com

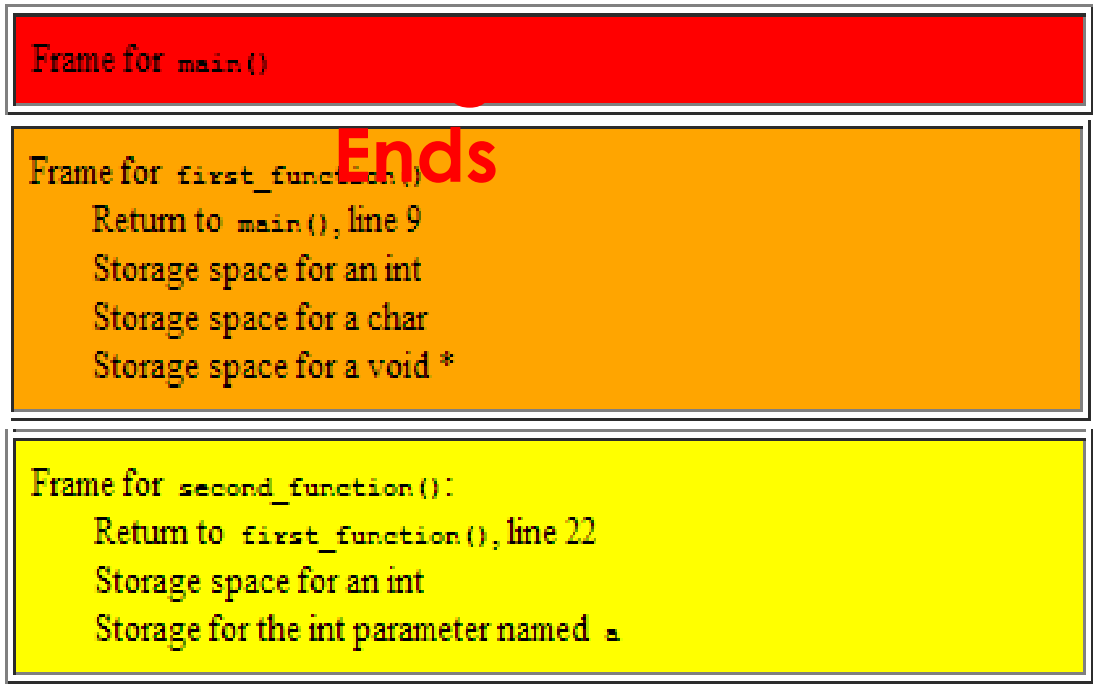
- TEXT segment
  - Contains machine instructions to be executed
- Global Variables
  - Initialized
  - Uninitialized
- Heap segment
  - Dynamic memory allocation
  - malloc, free
- Stack segment
  - Push frame: Function invoked
  - Pop frame: Function returned
  - Stores
    - Local variables
    - Return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