Task 1F: Observing Stack Location and Stack Frames using GDB

Objective:
To observe how the stack behaves during function calls in C by:
• Creating two-level function calls
• Passing parameters between functions
• Observing stack frames, local variables, return addresses
• Understanding how SP (Stack Pointer) and BP (Base Pointer / Frame Pointer) are manipulated

This task uses GDB to inspect runtime stack behavior.

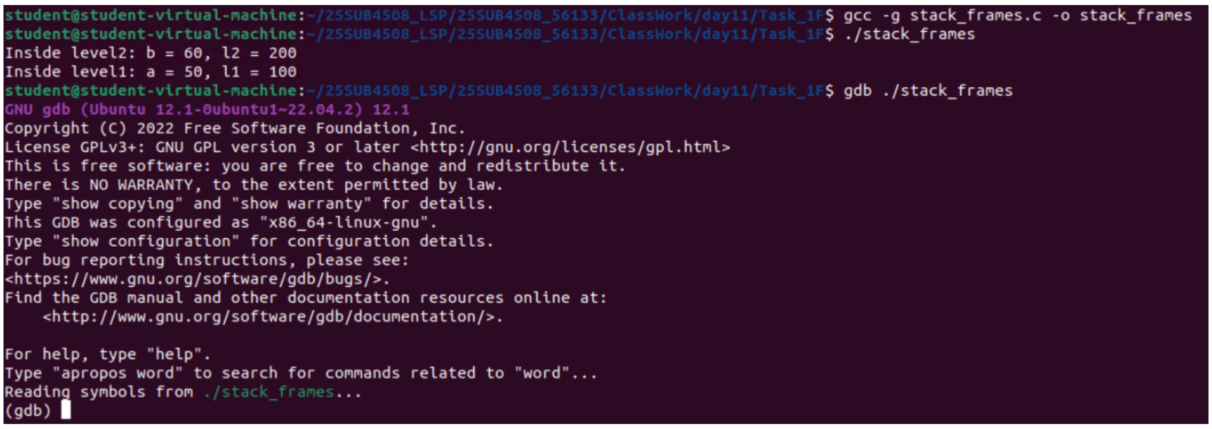
1. Program Code

```
#include <stdio.h>

void level2(int b) {
        int l2 = 200;
        printf("Inside level2: b = %d, l2 = %d\n", b, l2);
}

void level1(int a) {
        int l1 = 100;
        level2(a + 10);
        printf("Inside level1: a = %d, l1 = %d\n", a, l1);
}

int main() {
        int m = 50;
        level1(m);
        return 0;
}
```

2. Compilation Instructions

Compile the program with debugging symbols enabled:

```
gcc -g stack_frames.c -o stack_frames
```

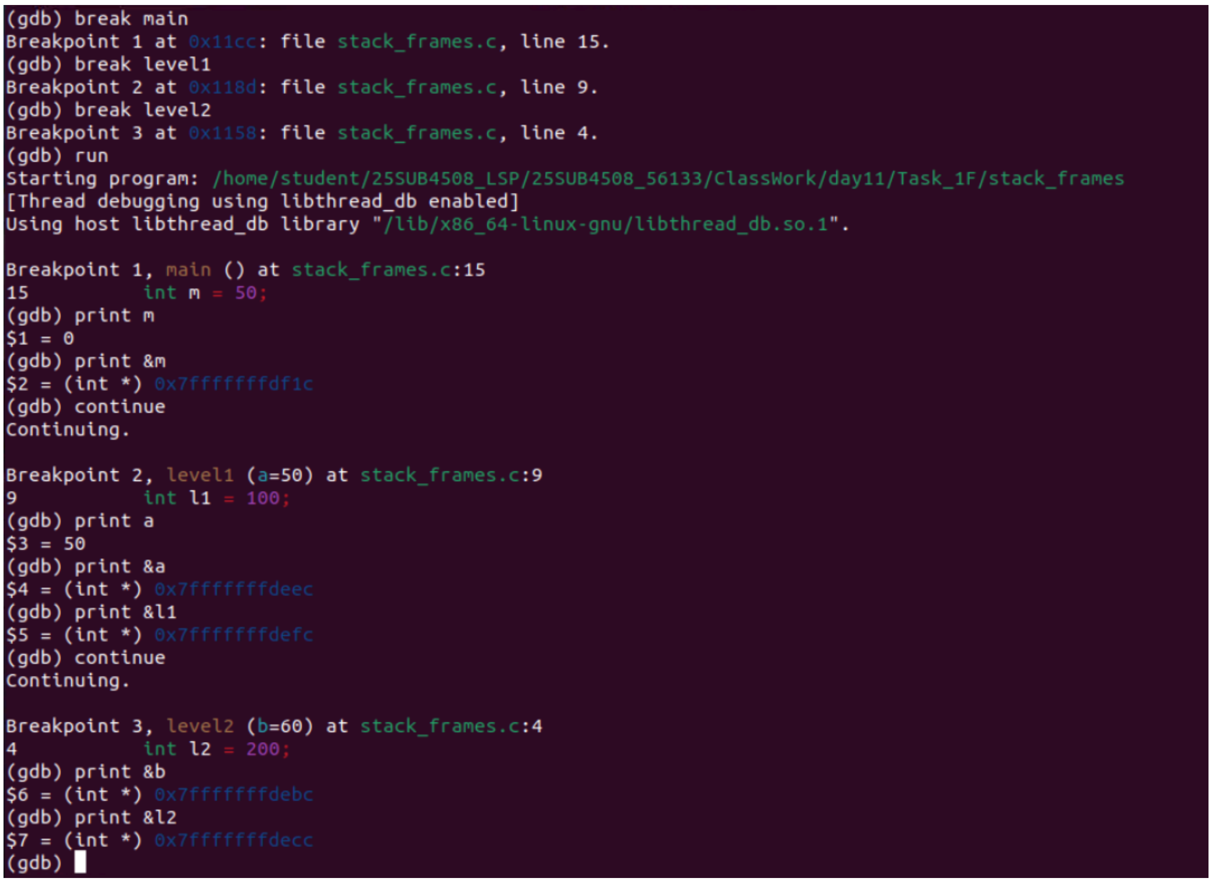


```
student@student-virtual-machine:~/25SUB4508_LSP/25SUB4508_56133/ClassWork/day11/Task_1F$ gcc -g stack_frames.c -o stack_frames
student@student-virtual-machine:~/25SUB4508_LSP/25SUB4508_56133/ClassWork/day11/Task_1F$ ./stack_frames
Inside level2: b = 60, l2 = 200
Inside level1: a = 50, l1 = 100
student@student-virtual-machine:~/25SUB4508_LSP/25SUB4508_56133/ClassWork/day11/Task_1F$ gdb ./stack_frames
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04.2) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./stack_frames...
(gdb) 
```

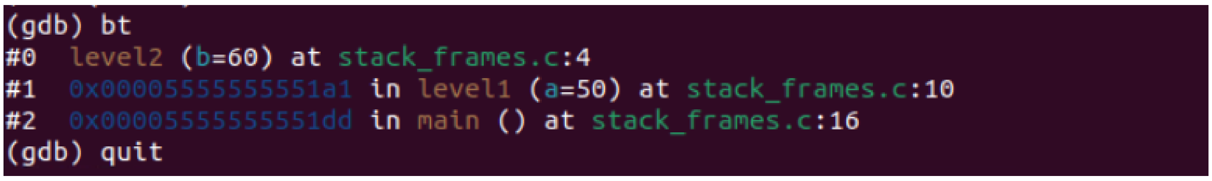
3. GDB Commands Used

```
break main
 break level1
 break level2
 run

 info frame
 info registers sp bp
 print &m
 print &a
 print &l1
 print &b
 print &l2
 bt
```

4. GDB Outputs



```
(gdb) break main
Breakpoint 1 at 0x11cc: file stack_frames.c, line 15.
(gdb) break level1
Breakpoint 2 at 0x118d: file stack_frames.c, line 9.
(gdb) break level2
Breakpoint 3 at 0x1158: file stack_frames.c, line 4.
(gdb) run
Starting program: /home/student/25SUB4508_LSP/25SUB4508_56133/ClassWork/day11/Task_1F/stack_frames
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at stack_frames.c:15
15          int m = 50;
(gdb) print m
$1 = 0
(gdb) print &m
$2 = (int *) 0x7fffffffdf1c
(gdb) continue
Continuing.

Breakpoint 2, level1 (a=50) at stack_frames.c:9
9           int l1 = 100;
(gdb) print a
$3 = 50
(gdb) print &a
$4 = (int *) 0x7fffffffdeec
(gdb) print &l1
$5 = (int *) 0x7fffffffdefc
(gdb) continue
Continuing.

Breakpoint 3, level2 (b=60) at stack_frames.c:4
4           int l2 = 200;
(gdb) print &b
$6 = (int *) 0x7fffffffdebc
(gdb) print &l2
$7 = (int *) 0x7fffffffdecc
(gdb) 
```



```
(gdb) bt
#0  level2 (b=60) at stack_frames.c:4
#1  0x00005555555551a1 in level1 (a=50) at stack_frames.c:10
#2  0x00005555555551dd in main () at stack_frames.c:16
(gdb) quit
```

5. Observations & Explanation

1. Each function call creates a new stack frame.
2. Local variables are stored inside their respective stack frames.
3. Parameters are also stored in the stack frame or passed via registers.
4. The return address is automatically pushed onto the stack during a function call.
5. RSP (Stack Pointer) points to the top of the stack.
6. RBP (Base Pointer) marks the base of the current stack frame.

6. Stack Growth Direction

The stack grows downward (from higher memory addresses to lower addresses).
Each nested function call pushes a new frame onto the stack.

7. Conceptual Stack Layout

```
----------------------------
 | level2 stack frame |
 | return address         |
 | local variables   |
 ----------------------------
 | level1 stack frame |
 | return address         |
 | local variables   |
 ----------------------------
 | main stack frame  |
 ----------------------------
```

8. Conclusion

This task demonstrates how the call stack operates during nested function calls. Using GDB, we observed stack frames, local variable storage, register manipulation, and return address handling. Understanding stack behavior is essential for debugging crashes, stack overflows, and low-level program execution.