CS255 - Introduction to GDB

Konstantina Papafragkaki
csdp1339@csd.uoc.gr
Computer Science Department, University of Crete, Greece

Why use GDB?

- ❑ Debugging is one of the most time-consuming tasks in software and hardware development
- ❑ GDB helps us identify the **cause** of errors
  - ❑ One of the most common errors is *segmentation fault* (trying to access memory that we don’t own)
- ❑ GDB allows us to track variables, set breakpoints and step through code to find bugs

What is GDB?

- GNU Debugger (GDB) is an open-source tool that helps us debug our programs faster
- Allows inspection and control of program execution
- Supports multiple languages: C, C++, Assembly, Go, Rust, etc
- Available on multiple platforms:
  - Linux -- pre-installed, Windows, macOS
  - UI-based alternatives:
    - gdbgui, Seer, gdb -tui # Text UI mode

Official Link: GDB Documentation

How does GDB work?

1. Compile the program with debugging symbols:
   1.1. gcc **-g** program_name.xxx -o program_name #xxx = suitable extension (e.g c, cpp)
2. Start debugging:
   2.1. Without arguments:
      2.1.1. gdb program_name
   2.2. With arguments:
      2.2.1. gdb --args program_name arg1 … argN
3. Run the program inside GDB:
   3.1. Without arguments:
      3.1.1. (gdb) **run**
   3.2. With arguments (if not specified using gdb --args):
      3.2.1. (gdb) **run** arg1 … argN

Basic GDB commands

1. (gdb) **break** func_name # Set breakpoint at func_name or variable_name or filename.xxx:line_number - Stops execution when func_name is called
2. (gdb) **next** # Steps over next line
3. (gdb) **step** #Step into a function call
4. (gdb) **continue** # Resume execution after a breakpoint
5. (gdb) **print** variable_name # Prints the value of variable_name
6. (gdb) **info locals** # Show local variables in the current stack frame
7. (gdb) **backtrace** # Show the stack frame

Recap of Stack frames (1/2)

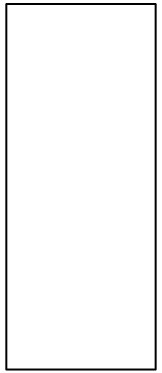
- A Stack Frame is created each time a function is called.
- It contains function arguments, local variables and the return address.
- Caller: the function that **calls** another function
- Callee: the function that **is called by** another
    - In the Call stack: the callee is placed above the caller
    - The most recent function call is at the **top of the stack**

Recap of Stack frames (2/2) - Simple C example

```
#include <stdio.h>

int mul(int x) {
    return x * 2;
}

int main() {
    int a = mul(4);
    printf(“%d\n”, a);
    return 0;
}
```



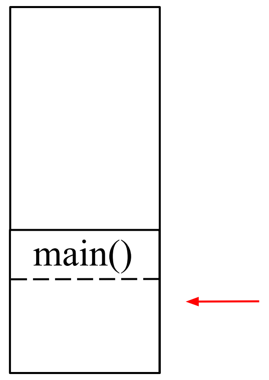
Empty stack frame.

Recap of Stack frames (2/2) - Simple C example

```
#include <stdio.h>

int mul(int x) {
    return x * 2;
}

int main() {
    int a = mul(4);
    printf("%d\n", a);
    return 0;
}
```



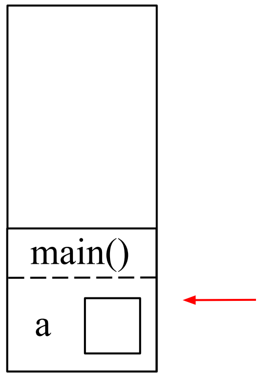
Step 1: At the beginning of the program, **main** is called, creating a new stack frame. Since **main** has no parameters, the frame only contains space for local variables and the return address.

Recap of Stack frames (2/2) - Simple C example

```
#include <stdio.h>

int mul(int x) {
    return x * 2;
}

int main() {
    int a = mul(4);
    printf("%d\n", a);
    return 0;
}
```



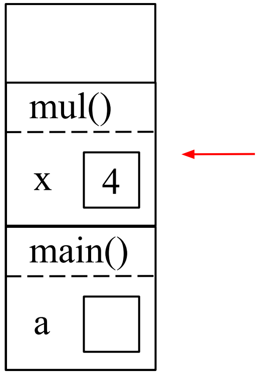
Step 2: The variable a is declared inside **main**, but it is not yet assigned a value. We create an empty box inside the stack frame of **main**, labeled with *a*, to represent this uninitialized value.

Recap of Stack frames (2/2) - Simple C example

```
#include <stdio.h>

int mul(int x) {
    return x * 2;
}

int main() {
    int a = mul(4);
    printf("%d\n", a);
    return 0;
}
```



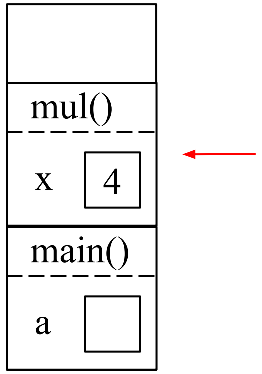
Step 3: We call the function **mul**, which creates a new stack frame above main(). Inside mul, we allocate space for the parameter *x* and store the argument 4 in it.

Recap of Stack frames (2/2) - Simple C example

```
#include <stdio.h>

int mul(int x) {
    return x * 2;
}

int main() {
    int a = mul(4);
    printf("%d\n", a);
    return 0;
}
```



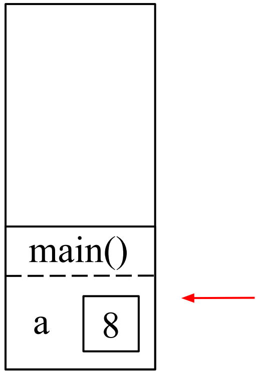
Step 4: The function **mul** computes x * 2, which evaluates to 8 and returns this value to main(), where **mul** was called.

Recap of Stack frames (2/2) - Simple C example

```
#include <stdio.h>

int mul(int x) {
    return x * 2;
}

int main() {
    int a = mul(4);
    printf("%d\n", a);
    return 0;
}
```



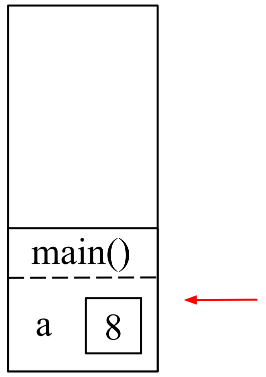
Step 5: The stack frame of **mul** is removed, as the function has completed execution. The variable *a* in main() is updated to store the returned value 8.

Recap of Stack frames (2/2) - Simple C example

```
#include <stdio.h>

int mul(int x) {
    return x * 2;
}

int main() {
    int a = mul(4);
    printf("%d\n", a);
    return 0;
}
```



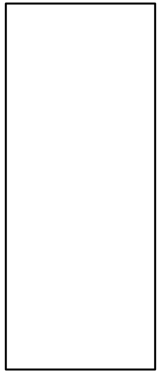
Step 6: The content of the variable *a* (8) is printed to the console.

Recap of Stack frames (2/2) - Simple C example

```
#include <stdio.h>

int mul(int x) {
    return x * 2;
}

int main() {
    int a = mul(4);
    printf(“%d\n”, a);
    return 0;
}
```



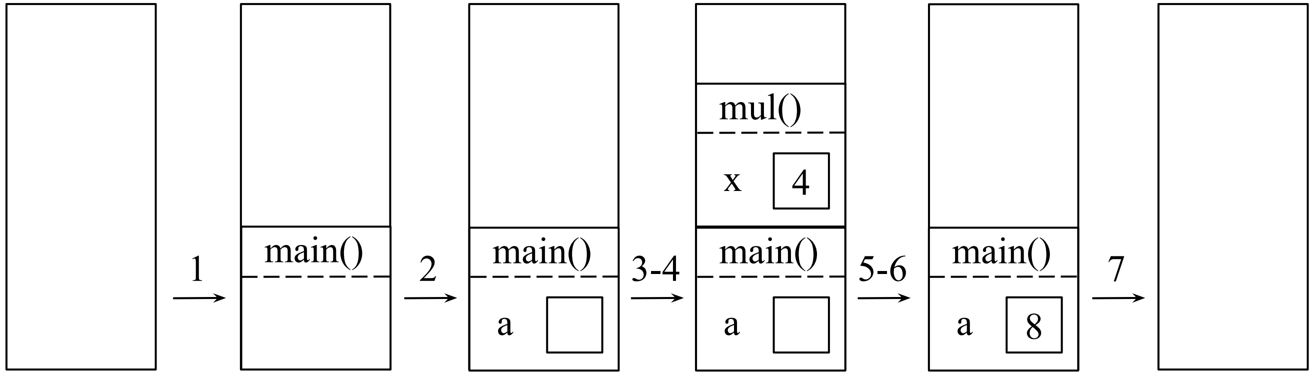
Step 7: The **main** function completes execution and its stack frame is removed. The program terminates.

Recap of Stack frames (2/2) - Simple C example

```
#include <stdio.h>

int mul(int x) {
    return x * 2;
}

int main() {
    int a = mul(4);
    printf("%d\n", a);
    return 0;
}
```



The Stack frame for the code in the left (Steps 1 - 7).

Examples

Segmentation Fault

After compiling and running the program in GDB, we encounter a segmentation fault. GDB provides debugging information, including:

- ❑ The exact line where the crash occurs
- ❑ The fault instruction

To fix the issue, we analyze the error, modify the code accordingly and rerun the program. If the error persists, we repeat the debugging process until the bug is resolved.

Once the issue is fixed, we exit the GDB using:

```
(gdb) q
```

Finally, we can recompile the program without debugging symbols.

Infinite loops

Sometimes a program may enter an infinite loop, causing it to run indefinitely without terminating. In such cases we can use GDB to inspect the issue and debug it.

Solution 1(After compiling and running the program in GDB):

1. Press **Ctrl + C** to pause the program and return the control to GDB.
2. Use the backtrace command to examine the function calls leading to loop to identify where the program is stuck:
   2.1. (gdb) **backtrace** # or (gdb) **bt**
      2.1.1. This generates an output listing stack frames (#X),showing where the program is executing.
3. To observe details of stack frames, use:
   3.1. (gdb) **frame** frame_number # or (gdb) **f** frame_number
4. Analyze the loop conditions, fix the code and rerun the program to check if the issue is resolved.

Infinite loops

Solution 2 (After compiling and running the program in GDB):

Instead of interrupting execution manually, we can use breakpoints to stop the program at specific points and analyze the issue.

1. Set a breakpoint when a function is called, a variable is accessed or a specific line is reached:
   1.1. (gdb) **break** func_name # or (gdb) **break** variable_name or (gdb) **break** filename.xxx:line_number or (gdb) **b** something
2. To list all set breakpoints, use:
   2.1. (gdb) **info breakpoints** # or (gdb) **info b**
3. If no longer need a breakpoint, we can disable it:
   3.1. (gdb) **disable** breakpoint_number
4. Once stop at a breakpoint, we can resume execution until the next one:
   4.1. (gdb) **continue** # or (gdb) **c**

Experimenting with GDB

- Debugging is a skill best learnt by experimenting with different problems
- The solution provided are not the only ones -- GDB offers numerous commands and techniques
- Explore by:
  - Setting watchpoints for variable changes
  - Inspecting memory
  - Automating debugging with **.gdbinit** scripts, etc
- Try debugging different segmentation faults, infinite loops and logical errors to gain experience.

Thank you!