# Debugging/GDB Overview

March 6, 2025

## 1 What is Debugging/GDB?

- **Debugging** is simply finding and removing errors (bugs) from your code
- If you write all your code before testing anything, it can be tough to debug/figure out what went wrong. Especially in a language like C, you can get errors like segmentation faults, which don't narrow down the issue
- GDB (GNU Debugger) is a tool that is used alongside C programs to aid in the debug process
- Through the command line, you can set **breakpoints** and watch variables, and how they behave during line-by-line execution, so you can understand how a program reaches a particular state

## 2 Installation and Setup

We'll focus on GDB. There is another debugger called `LLDB` that you can also try out (it's designed as a modern replacement, and it might be preferred on MacOS).

You probably have gdb installed already, try printing the version to verify.

### 2.1 Linux/WSL

```
# Verify (in terminal)
gdb --version

# If it's not installed, get it with the package manager:
sudo apt install gdb

# Optional, if you want to try out LLDB
sudo apt install lldb
```

### 2.2 MacOS:

If you run into issues with permissions, refer to the GDB documentation, or switch to `lldb`.

```
# Verify (in terminal)
gdb --version

# If not installed, get it with Homebrew
brew install gdb
```

## 3 Debug Process with GDB

### 3.1 Compiling for Debugging

First, compile your C program with the `-g` flag. This tells the compiler to include debugging information. This works across various debuggers (so by including -g during compilation, the file can be debugged using both GDB and LLDB)

```
# Add the -g flag during compilation
gcc -g -o program program.c

# Order of flags doesn't matter, this is equivalent:
gcc program.c -o program -g
```

### 3.2 Starting GDB

Launch GDB with your compiled program:

```
gdb ./program
```

## 3.3 Core GDB Commands

GDB works in the command-line (terminal), by entering specific commands for debugging. There are many more commands which you can find in the GDB documentation, but these are some of the main ones. For example, quit (q) means you can type 'quit' or 'q'.

```
# Starting/stopping execution
quit (q)                    # Exit GDB
run (r)                     # Start the program
continue (c)                # Continue running after a breakpoint

# Breakpoints
break (b) function          # Set breakpoint at function
break filename:line         # Set breakpoint at specific line
info breakpoints            # List all breakpoints
delete (d) 1                # Delete breakpoint number 1

# Stepping through code
next (n)                    # Step over (execute next line)
step (s)                    # Step into (enter function call)
finish                      # Step out (finish current function)

# Examining program state
print (p) expression        # Print value of expression
print/s mystring            # Print as string
info locals                 # Show local variables
backtrace (bt)              # Show stack trace

# Modifying execution
set var = value             # Change variable value
return expression           # Force function to return with value
```

# 4 GDB Example

Let's go through debugging a simple calculator program which has multiple C files (operations.c and main.c) and a header file which contains function templates (calculator.h). Note: if you haven't seen `#ifndef` before, it checks if the provided macro is not already defined, and if not, includes the following code until reaching `#endif`. It's also referred to as an 'include guard'.

```
// calculator.h
#ifndef CALCULATOR_H
#define CALCULATOR_H
// Header content to include:

// Functions that we will implement later (prototypes)
int add(int a, int b);
int subtract(int a, int b);
int multiply(int a, int b);
int divide(int a, int b);

#endif
```

```
// operations.c
#include "calculator.h"

// Implement the functions from the header
int add(int a, int b) {
    return a + b;
}

int subtract(int a, int b) {
    return a - b;
}

int multiply(int a, int b) {
    return a * b;
}

int divide(int a, int b) {
    return a / b;
}
```

Can you spot any bugs in `operations.c`?

```
// main.c
#include <stdio.h>
#include <stdlib.h>
#include "calculator.h"

void print_result(int result) {
    printf("Result: %d\n", result);
}

void main() {
    int n1 = 10;
    int n2 = 5;
    int res[4];

    res[0] = add(n1,n2);
    res[1] = subtract(n1,n2);
    res[2] = multiply(n1,n2);
    res[3] = divide(n1,n2);

    for (int i=0; i<4; i++) {
        print_result(res[i]);
    }
}
```

## 4.1 Compiling the Program

Compile the two C files into an executable program called calculator. Remember, we include the -g flag so we can debug it.

```
gcc -g -o calculator main.c operations.c
```

## 4.2 Finding and Fixing the Bug

We can use GDB to debug our code.

```
gdb ./calculator
```

Next to `(gdb)` are the instructions that I typed

```
(gdb) break main        # looks for main function, sets breakpoint
Breakpoint 1 at 0x11a0: file main.c, line 10.
(gdb) r                 # runs program from breakpoint
Breakpoint 1, main () at main.c:10
10      void main() {
(gdb) n                 # go to next line
11          int n1 = 10;
(gdb) n
12          int n2 = 5;
(gdb) n
15          res[0] = add(n1,n2);
(gdb) s                 # step into the add() function
add (a=10, b=5) at operations.c:6
6           return a + b;
(gdb) p a               # print the value of a
$1 = 10
(gdb) p b
$2 = 5
(gdb) n
7       }
(gdb) n
main () at main.c:15
15          res[0] = add(n1,n2);
(gdb) n
16          res[1] = subtract(n1,n2);
(gdb) p res[0]
$4 = 15                 # result of add() makes sense
(gdb) b main.c:18       # set a breakpoint at divide()
Breakpoint 2 at 0x5555555551f3: file main.c, line 18.
```

```
(gdb) c                    # continue to breakpoint
Continuing.

Breakpoint 2, main () at main.c:18
18          res[3] = divide(n1,n2);
(gdb) s
divide (a=10, b=5) at operations.c:18
18          return a / b;
(gdb) p a
$5 = 10
(gdb) p b
$6 = 5
(gdb) set var b = 0        # set the value of b to 0
(gdb) p b
$8 = 0
(gdb) n

Program received signal SIGFPE, Arithmetic exception.
0x0000555555555296 in divide (a=10, b=0) at operations.c:18
18          return a / b;
(gdb)
```

By debugging, we found a div by 0 error. We can update the code to print an error message and return a placeholder value like -1.

```c
// operations.c - updated segment
int divide(int a, int b) {
    if (b == 0) {
        printf("Error: Division by zero\n");
        return -1;
    }
    return a / b;
}
```

## 4.3 Recompiling and Verifying the Fix

Recompile the updated operations.c and test it out.

```
gcc -g -o calculator main.c operations.c

# Debug new (potentially fixed) program
gdb ./calculator
```

```
Reading symbols from ./calculator...
(gdb) break divide
Breakpoint 1 at 0x12b6: file operations.c, line 19.
(gdb) r

```

```
# in our code, int b = 5
Breakpoint 1, divide (a=10, b=5) at operations.c:19
19          if (b==0) {
(gdb) set var b = 0 # update value to check for div by 0
(gdb) p b
$1 = 0               # updated
(gdb) n
20              printf("Error: Division By 0.\n");
(gdb) n
Error: Division By 0. # error msg printed
21              return -1;
(gdb) n
24      }
(gdb) n
main () at main.c:18
18          res[3] = divide(n1,n2);
(gdb) c
Continuing.
Result: 15
Result: 5
Result: 50
Result: -1
[Inferior 1 (process 9214) exited normally]
```

The program now handles division by zero.