GNU Debugger (GDB)

1 Sample Program

```cpp
#include <iostream>
#include <string>
#include <vector>

// 1. A global variable to test watchpoints
int global_counter = 0;

// 2. A struct to inspect complex data types
struct Point {
    int x;
    int y;
    const char* label;
};

// 3. A recursive function to demonstrate the call stack
long factorial(int n) {
    global_counter++; // This change can be caught by a watchpoint
    if (n <= 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

// 4. A function to demonstrate arrays and loops
int sum_array(int* arr, int size) {
    int total = 0;
    for (int i = 0; i < size; ++i) {
        total += arr[i]; // A good place for a conditional breakpoint
    }
    return total;
}

// 5. A function designed to cause a segmentation fault
void cause_crash() {
    Point* p_null = nullptr;
    std::cout << "About to dereference a null pointer..." << std::endl;
    p_null->x = 10; // This will cause a SEGFAULT
    std::cout << "This line will never be reached." << std::endl;
}

int main(int argc, char *argv[]) {
    // Basic variables for printing
    int a = 10;
    float b = 3.14f;
    const char* message = "Hello, GDB!";

    // Structs and dynamic memory
    Point p1 = {5, -2, "Point A"};
    Point* p2 = new Point{15, 20, "Point B"};

    // Arrays
    int numbers[] = {10, 20, 30, 40, 50};
    int total = sum_array(numbers, 5);
    std::cout << "Sum of numbers: " << total << std::endl;

    // Recursive call
    int fact_arg = 4;
    long fact_result = factorial(fact_arg);
    std::cout << "Factorial of " << fact_arg << " is " << fact_result << std::endl;
    std::cout << "Factorial function was called " << global_counter << " times." << std::endl;
```

```
    delete p2; // Clean up dynamic memory

    // Check for a command-line argument to trigger the crash
    if (argc > 1 && std::string(argv[1]) == "crash") {
        cause_crash();
    }

    std::cout << "Program finished successfully." << std::endl;
    return 0;
}
```

Compile it:

```
g++ -g -o gdb_test gdb_test.cpp
```

2 Start GDB

```
gdb ./gdb_test
```

3 Breakpoints

```
# Set a breakpoint at the beginning of the main function
(gdb) break main
Breakpoint 1 at 0x1234: file gdb_test.cpp, line 43.

# Set a breakpoint at a specific line number
(gdb) break 33
Breakpoint 2 at 0x5678: file gdb_test.cpp, line 33.

# List all breakpoints
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000004011f0 in main(int, char**) at gdb_test.cpp:43
2       breakpoint     keep y   0x00000000004011bd in sum_array(int*, int) at gdb_test.cpp:33
```

4 Running program

```
(gdb) run
Starting program: /path/to/gdb_test

Breakpoint 1, main (argc=1, argv=0x7fffffffe3b8) at gdb_test.cpp:43
43      int a = 10;
```

5 Running through the code

```
# Execute line 43
(gdb) next
44      float b = 3.14f;

# Step over a few more lines
(gdb) n
45      const char* message = "Hello, GDB!";
(gdb) n
48      Point p1 = {5, -2, "Point A"};
(gdb) n
49      Point* p2 = new Point{15, 20, "Point B"};
(gdb) n
52      int numbers[] = {10, 20, 30, 40, 50};
(gdb) n
53      int total = sum_array(numbers, 5);

# Now, step INTO the sum_array function
(gdb) step
```

```
sum_array (arr=0x7fffffffe2b0, size=5) at gdb_test.cpp:31
31      int total = 0;
```

6 Inspecting Data

```
# In main, before calling sum_array
(gdb) print a
$1 = 10

(gdb) whatis message
type = const char *

# Print the contents of a struct
(gdb) print p1
$2 = {x = 5, y = -2, label = 0x40200e "Point A"}

# Print the contents of a pointer to a struct
(gdb) print *p2
$3 = {x = 15, y = 20, label = 0x402016 "Point B"}

# Print an array. Use @<size> to show its contents.
(gdb) print *numbers@5
$4 = {10, 20, 30, 40, 50}
```

7 Conditional Breakpoints and Call Stack

```
# Let's delete our old breakpoints and set a new one
(gdb) delete
Delete all breakpoints? (y or n) y
(gdb) break factorial
Breakpoint 3 at 0x40118a: file gdb_test.cpp, line 22.

# Run the program again from the beginning
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
...
Breakpoint 3, factorial (n=4) at gdb_test.cpp:22
22      if (n <= 1) {

# Continue to hit the breakpoint on each recursive call
(gdb) continue
Continuing.
Breakpoint 3, factorial (n=3) at gdb_test.cpp:22
22      if (n <= 1) {

# See the call stack
(gdb) backtrace
#0  factorial (n=3) at gdb_test.cpp:22
#1  0x0000000000401182 in factorial (n=4) at gdb_test.cpp:25
#2  0x00000000004012ce in main (argc=1, argv=0x7fffffffe3b8) at gdb_test.cpp:57

# You can navigate the stack with `up` and `down`
(gdb) up
#1  0x0000000000401182 in factorial (n=4) at gdb_test.cpp:25
25          return n * factorial(n - 1);
(gdb) print n
$5 = 4
```

8 Watchpoints

```
# Restart and stop at main
(gdb) run
...
Breakpoint 1, main ...

# Set a watchpoint on our global variable
```

```
(gdb) watch global_counter
Hardware watchpoint 4: global_counter

# Continue execution. It will stop every time factorial() increments the counter.
(gdb) continue
Continuing.
Hardware watchpoint 4: global_counter

Old value = 0
New value = 1
factorial (n=4) at gdb_test.cpp:22
22      if (n <= 1) {
```

9 Diagnosing Crash, Segmentation fault

```
# Quit the current session and restart GDB
(gdb) quit

$ gdb ./gdb_test

# Run the program with the "crash" argument
(gdb) run crash
Starting program: /path/to/gdb_test crash
Sum of numbers: 150
Factorial of 4 is 24
Factorial function was called 4 times.
About to dereference a null pointer...

Program received signal SIGSEGV, Segmentation fault.
0x00000000004011e9 in cause_crash () at gdb_test.cpp:38
38      p_null->x = 10; // This will cause a SEGFAULT

# The program crashed! Where did it happen? Use backtrace.
(gdb) backtrace
#0  0x00000000004011e9 in cause_crash () at gdb_test.cpp:38
#1  0x0000000000401314 in main (argc=2, argv=0x7fffffffe3a8) at gdb_test.cpp:63

# Let's examine the variables in the frame where it crashed (frame 0)
(gdb) print p_null
$1 = (Point *) 0x0
```

10 Generating Core Dumps

Core dumps are critical for debugging applications in production environments. You can't just attach a debugger to a live server that is serving customers. Instead, if an application crashes, the system can automatically save a core dump. A developer can then copy that core file to their own machine and analyze the crash safely, with the full context, as if they were there when it happened.

10.1 Check the limits for coredump

```
ulimit -c
```

10.2 Enable coredump

```
ulimit -c unlimited
```

10.3 Crash the code using the argument

```
./gdb_test crash
```

Upon crashing, the directory containing the object file will have a new file named **core** or **core.<some_process_id>**.

11 Analysing Core Dumps with GDB

11.1 Execute the code with core file attached

```
gdb ./gdb_test core
```

11.2 Post-Mortem Debugging

11.2.1 Call stack with backtrace/bt

```
(gdb) backtrace
#0  0x00000000004011e9 in cause_crash () at gdb_test.cpp:38
#1  0x0000000000401314 in main (argc=2, argv=0x7fffffffe3a8) at gdb_test.cpp:63
```

11.2.2 Inspect variables with print/p

```
(gdb) print p_null
$1 = (Point *) 0x0
```

11.3 Navigate the stack with frame for a broader view

```
(gdb) frame 1
#1  0x0000000000401314 in main (argc=2, argv=0x7fffffffe3a8) at gdb_test.cpp:63
63          cause_crash();
(gdb) print argc
$2 = 2
(gdb) print argv[1]
$3 = 0x7fffffffe6a5 "crash"
```

12 Symbol Table

Generating a separate symbol file allows you to strip the debugging information from your main executable, making it much smaller. Also, it makes it a bit harder to reverse engineer.

12.1 Generate a separate symbol table

12.1.1 Extract the symbols

```
# Syntax: objcopy --only-keep-debug <executable> <symbol_file_name>
objcopy --only-keep-debug gdb_test gdb_test.debug
```

12.1.2 Strip the object file

```
# Syntax: strip --strip-debug <executable>
strip --strip-debug gdb_test
```

12.1.3 Link executable to symbol file (optional)

```
# Syntax: objcopy --add-gnu-debuglink=<symbol_file_name> <executable>
objcopy --add-gnu-debuglink=gdb_test.debug gdb_test
```

12.2 Running with separate symbol file

If the symbol file was linked, you can directly execute it and the symbols will be loaded automatically. If it was not linked, you will have to manually provide the symbol file in GDB prompt:

```
(gdb) symbol-file gdb_test.debug
Reading symbols from gdb_test.debug...
(gdb)
```