# Assignment - 2
# Software Engineering Laboratory
# Group - 19


| Members | Enrollment No. |
| --- | --- |
| Alok Ranjan | 2022CSB091 |
| Ranveer Kumar | 2022CSB092 |
| Karan Kumar | 2022CSB093 |
| Rajesh Kumar | 2022CSB094 |

## GNU Debugger (GDB) Command in Linux

The GNU Debugger (GDB) is a powerful command-line tool in Linux used for debugging programs written in languages like C, C++, and Fortran. It allows developers to track down errors, understand program flow, and inspect variables during runtime. GDB helps ensure code correctness by providing detailed insights into the program's execution.

**Key Features of GDB:**

1. **Breakpoint Management**: Set, delete, or list breakpoints to pause program execution at specific lines or functions.
2. **Run Control**: Start, stop, and step through programs line-by-line or instruction-by-instruction.
3. **Variable Inspection**: Examine and modify variable values during runtime.
4. **Backtracing**: View the function call stack to identify the sequence of function calls leading to an error.
5. **Core Dump Analysis**: Analyze core dump files to debug crashes post-execution.

### 1. Running a Program

This involves starting the execution of the program inside GDB. You can specify command-line arguments and environment variables. The GDB command `run` (or `r`) is available for this purpose.

### 2. Loading Symbol Table

The symbol table contains information about variables, functions, and their memory addresses. It is loaded automatically when you start debugging a program using GDB. No additional command is required for loading it, but you can use the `file` command to load a different symbol table explicitly.

### 3. Setting a Breakpoint

A breakpoint pauses the program execution at a specific line or function, allowing you to inspect the state. The GDB command `break` (or `b`) is available for this.

### 4. Listing variables and examining their values

When debugging, checking variable values at different execution points is essential to ensure correct logic and expected outcomes. This helps detect uninitialized, incorrect, or unexpected values that might be causing program failures. This is an essential feature for debugging programs involving complex computations or dynamic memory.

### 5. Printing content of an array or contiguous memory

GDB provides the ability to print arrays or a block of contiguous memory, helping developers visualize data stored at specific locations.This is particularly useful when debugging issues like buffer overflows, incorrect indexing, or memory corruption. By inspecting memory contents directly, we can confirm whether data is correctly stored and retrieved, ensuring proper program behavior.

### 6. Printing function arguments

When debugging function calls, it is crucial to verify whether the correct arguments are being passed. GDB allows developers to print function arguments, helping them confirm the expected parameter values. This is particularly useful for debugging recursive functions, function pointer calls, or complex parameter-passing scenarios.

### 7. Next, Continue, Set command

The **`next`**, **`continue`** and **`set`** commands are fundamental in controlling execution flow within GDB. The **`next`** command allows stepping over function calls without entering them, enabling faster debugging when function internals are not relevant. The **`continue`** command resumes program execution until another breakpoint is hit. The **`set`** command modifies GDB settings and variable values at runtime.

## 8. Single stepping into function

Single stepping allows developers to execute a program one statement at a time, making it easier to analyze control flow and variable changes. This line-by-line execution of any program making it easier to debug and identify the part of code causing errors.

## 9. Listing all break point

In complex debugging sessions, multiple breakpoints may be set to pause execution at different program locations. Listing all breakpoints provides an overview of active breakpoints, including their locations and conditions. It becomes easier to enable, disable, or delete unnecessary breakpoints which in turn makes the debugging process much easier.

## 10. Ignoring a break-point for N occurrence

In many cases, a breakpoint is useful only after a certain number of iterations have taken place. Ignoring a breakpoint for N occurrences allows skipping it a specified number of times before it halts execution.Instead of stopping at every iteration, we can let the program run until the breakpoint condition is met, improving debugging efficiency.

### 11. Enable/disable a breakpoint

A breakpoint temporarily halts a program at a specific line or function. GDB allows enabling or disabling breakpoints without deleting them, which is useful for conditional debugging without re-entering commands.

### 12. Break condition and Command

Breakpoints can be made conditional using expressions that evaluate at runtime. This feature stops execution only when a specific condition is met, reducing unnecessary interruptions. Commands can also be associated with breakpoints to automate debugging tasks.

### 13. Examining stack trace

A stack trace shows the sequence of function calls that led to the current execution point. It helps identify where the program crashed or encountered errors by displaying active function calls and their arguments.

### 14. Examining stack trace for multi-threaded program

GDB allows multi-threaded programs to be debugged by providing stack traces for different threads. It lets developers switch between threads and examine their respective function call stacks. This is essential for debugging concurrency issues.

### 15. Core File Debugging

Core fle debugging in GDB allows **analyzing a program's state** at the time of a crash using a core dump fle. It involves **loading the core** fle with `gdb <executable> <core>` to **inspect** the program's **memory, variables, and registers**. Key commands include `backtrace` for stack traces, `info registers` **for CPU state**, and `print` for variable values. Core debugging requires the executable to be **compiled** with debugging symbols (`-g` fag). It **supports** multi-threaded programs, **enabling thread-specifc analysis**. This technique is essential for post-mortem debugging to identify crashes, invalid memory accesses, or other runtime issues without re-executing the program.

### 16. Debugging of an already running program

Debugging an already running program in GDB involves **attaching the debugge**r to a live process to analyze its runtime behavior. Use `gdb attach <process_id>` to connect GDB to the **target process**. This allows **inspection** of the program state, including variables, memory, and registers, without restarting it. Key commands include `info threads` for thread details, `backtrace` for stack traces, and `print` to examine variable values. The process **must have** debugging symbols (`-g` fag during compilation). This method is **vital** for diagnosing issues in **long-running** or **production processes** without disrupting their execution. **Detach** with `detach` to resume normal execution

### 17. Watchpoint

A watchpoint in GDB is a **special type of breakpoint** that pauses program execution when a **specifed memory location or variable changes.** It is useful for tracking **unexpected modifcations** to critical variables. Key concepts include:

1. **Setting a watchpoint:** Use `watch <variable>` to monitor changes.
2. **Conditional watchpoints:** Add conditions using `watch <expression>` for more targeted monitoring.
3. **Hardware vs. software watchpoints**: Hardware watchpoints rely on processor support, while software watchpoints use GDB's internal mechanisms.
4. **Commands**: Use `info watchpoints` to list active watchpoints and `delete <watchpoint_number>` to remove them.