**Capstone Project: Reverse Debugging with GDB**

**Objective:** The capstone project focuses on exploring **reverse debugging** using **GDB**. Reverse debugging allows developers to step backward through program execution to identify the root cause of bugs more effectively. This project involves practical implementation of reverse debugging on real-world debugging scenarios, highlighting its utility in resolving complex issues.

**Project Outline:**

**1. Project Goals**

* Understand the concept and setup of reverse debugging in GDB.
* Implement and debug C programs with common issues where reverse debugging is beneficial.
* Demonstrate reverse debugging in scenarios like logical errors, memory corruption, and race conditions.
* Document how reverse debugging aids in fixing issues compared to traditional debugging.

**2. Tools and Setup**

* **GDB with Reverse Debugging Support**: Ensure GDB is built with reverse debugging capabilities. Most distributions (e.g., Ubuntu) provide this by default.
* **Compiler**: GCC with -g flag for including debug symbols.
* **Test Environment**: Linux environment or WSL on Windows.
* **Editor/IDE**: Optional (VSCode or CLion with GDB integration).

**3. Capstone Use Cases**

Each use case demonstrates a scenario where reverse debugging is valuable.

**Use Cases for Reverse Debugging**

**1. Logical Errors in Loops**

**Description**: Write c or C++ program to Reverse debugging helps identify where a loop or algorithm produces incorrect results.

**Debugging Tasks**:

1. Compile:
2. Start GDB:
3. Set a breakpoint
4. Enable reverse debugging:
   * Start execution: run.
   * Step forward until the error: next.
   * Step backward: reverse-next to locate where the loop exceeds bounds.
5. Fix the loop condition and retest.

**2. Debugging Null Pointer Dereference**

**Description**: Reverse debugging helps track how a null pointer is introduced into the program.

**Debugging Tasks**:

1. Compile: g
2. Use GDB to step through:
   * run to start execution.
   * Identify where data is assigned null using reverse commands (reverse-step).
3. Fix the conditional logic that assigns null.

**3. Memory Corruption in Dynamic Arrays**

**Description**: Track memory corruption in dynamic arrays by reversing execution.

**Debugging Tasks**:

1. Compile:
2. Use GDB:
   * Set a breakpoint before the loop.
   * Step into the loop and observe memory writes.
   * Reverse-step to identify where the out-of-bounds write occurs.
3. Fix the loop bounds.

**4. Race Conditions in Multithreaded Code**

**Description**: Reverse debugging is used to analyze nondeterministic bugs caused by race conditions.

**Debugging Tasks**:

1. Compile with threads
2. Use GDB:
   * Run the program and observe output.
   * Use record to enable reverse debugging.
   * Reverse-step through thread execution (reverse-next and reverse-step) to locate simultaneous access to counter.
3. Fix the issue using a mutex and retest.

**5. Recursive Function Debugging**

**Description**: Reverse debugging helps analyze incorrect recursion logic.

**Debugging Tasks**:

1. Compile:
2. Use GDB:
   * Start with record.
   * Step forward until the stack overflow occurs.
   * Use reverse-step to identify why the base case is never met.
3. Fix the recursion logic.