## Use Case 1: Debugging Logical Errors in Loops with Reverse Debugging

### Overview

Logical errors in loops can produce incorrect results, such as exceeding bounds or miscalculating outputs. **Reverse debugging** in GDB allows you to step backward to trace how the loop state evolved.

---

### Example Program

```
#include <stdio.h>

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int size = sizeof(arr) / sizeof(arr[0]);
    int sum = 0;

    // Incorrect loop condition: accessing out-of-bounds index
    for (int i = 0; i <= size; i++) {
        sum += arr[i];
    }

    printf("Sum: %d\n", sum); // Undefined behavior due to
out-of-bounds access
    return 0;
}
```

---

**Step-by-Step Debugging**

**Compile with Debug Symbols**

```
gcc -g -o loop_debug example.c
```

1. The `-g` flag includes debugging information.

**Start GDB**

```
gdb ./loop_debug
```

2. This launches GDB with your program loaded.

**Set a Breakpoint**

```
break main
```

3. Stops execution at the start of the `main()` function.
4. **Enable Reverse Debugging**

To enable reverse debugging, GDB needs process record/replay mode:

```
target record-full
```

    -
    - This mode allows backward stepping during execution.

**Start Execution**

```
run
```

5. The program runs until the breakpoint at `main()`.

**Step Forward Until the Error** Use `next` to step line by line:

```
next
```

6. Stop at the line causing out-of-bounds access.

**Step Backward** Use reverse commands to trace back:

```
reverse-next
```

7.
    - Reverse through loop iterations to identify the condition causing the error.

For this case, the incorrect condition is:

```
for (int i = 0; i <= size; i++) {
```

    -

**Fix the Loop** Update the loop condition to:

```
for (int i = 0; i < size; i++) {
```

8.

**Retest** Recompile and rerun the program:

```
gcc -g -o loop_debug_fixed example.c

./loop_debug_fixed
```

9.

---

## Use Case 2: Debugging Null Pointer Dereference with Reverse Debugging

### Overview

A **null pointer dereference** occurs when a program tries to access memory using a null pointer, causing a segmentation fault. Reverse debugging helps identify where the pointer was assigned a null value.

---

### Example Program

```
#include <stdio.h>

#include <stdlib.h>
```

```c
int main() {
    int *ptr = NULL;

    // Incorrectly assigning NULL
    if (1) { // Simulating a condition
        ptr = NULL;
    }

    // Dereferencing NULL pointer
    printf("Value: %d\n", *ptr); // Segmentation fault
    return 0;
}
```

---

**Step-by-Step Debugging**

**Compile with Debug Symbols**

```
gcc -g -o null_debug example.c
```

1.

**Start GDB**

```
gdb ./null_debug
```

2.

**Set a Breakpoint**

```
break main
```

3.

**Run the Program**

```
run
```

4.
   - The program stops at the breakpoint in `main()`.

A segmentation fault occurs at:

```
printf("Value: %d\n", *ptr);
```

   -

**Enable Reverse Debugging** Use process record/replay:

```
target record-full
```

5.

**Step Backward** Use:

```
reverse-step
```

6.
   - Go back through the execution flow to see when `ptr` was assigned `NULL`.

In this case, it happens due to:
c
CopyEdit

```
if (1) {

    ptr = NULL;

}
```

   -

**Fix the Issue** Update the conditional logic to avoid assigning `NULL`:

```
if (some_condition) {

    ptr = malloc(sizeof(int));

    *ptr = 42; // Assign a valid value

} else {

    // Handle the case where ptr is not initialized

    printf("Pointer not initialized.\n");

    return -1;

}
```

7.

**Free Allocated Memory** Ensure memory allocated to the pointer is freed:

```
free(ptr);
```

8.

**Retest** Recompile and rerun the program:

```
gcc -g -o null_debug_fixed example.c

./null_debug_fixed
```

9.

---

## Key GDB Commands for Reverse Debugging

1. **Enable Reverse Debugging**:
    - `target record-full`: Enables reverse debugging mode.
2. **Step Backward**:
    - `reverse-next`: Steps backward through the code without entering functions.
    - `reverse-step`: Steps backward and enters functions.
3. **View Backtrace**:

- `backtrace`: Displays the call stack at the current point.
- `reverse-finish`: Steps out of the current function in reverse.

---

### Benefits of Reverse Debugging

- Helps locate errors by tracing execution backward.
- Ideal for identifying when variables take incorrect values.
- Prevents repeated forward executions, saving debugging time.