Module 1: Debugging Basics

Understanding the Problem

Debugging involves identifying and fixing errors in the code. Errors can be:

- **Logical errors**: Code runs but produces incorrect results.
- **Runtime errors**: Code crashes or behaves unexpectedly during execution.

Example Program:

```
#include <stdio.h>

int main() {
    int a = 5, b = 0;
    printf("Result: %d\n", a / b); // Logical error: division by zero
    return 0;
}
```

Here, dividing by zero causes undefined behavior, typically resulting in a runtime error.

Step-by-Step Debugging in GDB

Compile with Debugging Information

```
gcc -g -o debug_example program.c
```

1.
   - The `-g` flag includes debug symbols in the compiled binary, allowing GDB to map binary instructions back to source code.

Launch GDB

```
gdb ./debug_example
```

2.
   - GDB starts with your program loaded, displaying a `(gdb)` prompt for commands.

Set Breakpoints

```
break main
```

3.
    - A breakpoint pauses execution when a specific line or function is reached.
    - Here, we pause at the start of `main()`.

Run the Program

```
run
```

4.
    - Executes the program. If a breakpoint is hit or the program crashes, control is returned to GDB.

Analyze Crashes with Backtrace

```
backtrace
```

5.

Displays the call stack, showing the sequence of function calls leading to the error. For example:
less

```
#0  main at program.c:6
```

    -
    - This indicates the error occurred in `main()` at line 6.

Examine Variable Values

```
print a
print b
```

6.

The `print` command displays the values of variables at the current breakpoint or crash point. Here, you'd see:

```
$1 = 5
$2 = 0
```

    -

7. **Fix the Problem**

Change the code to handle division by zero:

```
if (b != 0) {
    printf("Result: %d\n", a / b);
} else {
    printf("Error: Division by zero.\n");
}
```

○

Module 2: Memory Errors with Valgrind and GDB

Understanding Memory Errors

Memory-related issues include:

1. **Memory leaks**: Allocated memory is not freed.
2. **Invalid memory access**: Accessing memory outside allocated bounds.

Example Program:

```
#include <stdlib.h>

int main() {
    int *arr = malloc(5 * sizeof(int));
    arr[5] = 10; // Out-of-bounds access
    free(arr);
    return 0;
}
```

Here, `arr[5]` accesses memory outside the allocated range, leading to undefined behavior.

Step-by-Step Debugging

Compile with Debugging Symbols

```
gcc -g -o memory_example program.c
```

1. The `-g` flag enables debugging symbols.

Detect Memory Issues with Valgrind

```
valgrind ./memory_example
```

2.

Valgrind analyzes memory usage. Sample output:

```
Invalid write of size 4
  at 0x4005E6: main (program.c:5)
Address 0x520304 is 0 bytes after a block of size 20
```

  -
  - This shows an out-of-bounds write occurred at `program.c:5`.

Run the Program in GDB

```
gdb ./memory_example
```

3.
  - Load the program in GDB for detailed analysis.

Set Breakpoints

```
break main
```

4.
  - Stops execution at the start of `main()`.

Run Execution

```
run
```

5.
  - Executes the program until a breakpoint or crash occurs.

Set Watchpoints

```
watch arr[5]
```

6.

- **Watchpoints** monitor memory locations. Execution stops if `arr[5]` is accessed or modified.

Examine Local Variables

```
info locals
```

7.
   - Displays local variables and their current values.

Inspect the Call Stack

```
backtrace
```

8.
   - Shows the sequence of function calls leading to the error.

Fixing Memory Issues

1. **Adjust Allocation**

Ensure you allocate enough memory:

```
int *arr = malloc(6 * sizeof(int)); // Allocate 6 elements
```

○

2. **Add Bounds Checking**

Avoid out-of-bounds access:

```
if (index >= 0 && index < 5) {
    arr[index] = 10;
} else {
    printf("Index out of bounds.\n");
}
```

○

3. **Free Memory Correctly**

Always free allocated memory:

```
free(arr);
arr = NULL; // Avoid dangling pointers
```

    -

In-Depth Explanation of Commands

GDB Commands

- `break <location>`: Set a breakpoint at a specific function or line number.
- `run`: Start program execution.
- `backtrace`: View the call stack to trace errors.
- `print <variable>`: Display the value of a variable.
- `watch <expression>`: Monitor a variable or memory location for changes.
- `info locals`: View values of all local variables.

Valgrind

- Detects:
    - **Memory leaks**: Memory not freed before program exits.
    - **Invalid memory access**: Accessing unallocated or freed memory.
    - **Uninitialized memory**: Using memory before initializing it.

Summary

Debugging with tools like GDB and Valgrind is a systematic process. GDB provides insight into execution flow and variable states, while Valgrind excels at identifying memory-related issues. Together, these tools help diagnose and resolve logical, runtime, and memory errors effectively.