Valgrind

1 Sample Program

```cpp
#include <iostream>
#include <cstdlib> // For malloc, free
#include <thread>  // For threading example

// Global variable for the data race example
int g_shared_counter = 0;

// ============================================================
// 1. Uninitialized Value Use (Detected by Memcheck)
// ============================================================
void use_uninitialized_value() {
    int x; // 'x' is declared but not initialized
    if (x == 0) { // Conditional jump or move depends on uninitialised value(s)
        std::cout << "x is zero." << std::endl;
    }
}

// ============================================================
// 2. Heap Buffer Overflow (Detected by Memcheck)
// ============================================================
void invalid_heap_access() {
    // Allocate an array of 10 integers on the heap
    int* heap_array = new int[10];

    // Invalid write: Writing one element past the end of the block
    heap_array[10] = 5; // ERROR: heap_array has indices 0-9

    // Invalid read: Reading from memory far past the end of the block
    int temp = heap_array[20]; // ERROR: Reading way out of bounds
    (void)temp; // Suppress unused variable warning

    delete[] heap_array;
}

// ============================================================
// 3. Stack Buffer Overflow (Detected by Memcheck)
// ============================================================
void invalid_stack_access() {
    int stack_array[10];
    // Invalid write to the stack, corrupting it
    stack_array[10] = 42; // ERROR: stack_array has indices 0-9
}

// ============================================================
// 4. Memory Leak (Detected by Memcheck)
// ============================================================
void memory_leaks() {
    // This memory is allocated but never freed.
    // Valgrind will report this as "definitely lost".
    int* leaky_ptr = new int(123);

    // This shows that just losing the pointer causes a leak.
    if (leaky_ptr) {
        std::cout << "Leaky pointer value: " << *leaky_ptr << std::endl;
    }
}

// ============================================================
// 5. Invalid Free / Mismatched New/Free (Detected by Memcheck)
// ============================================================
void invalid_free() {
    int* ptr = new int(5);
```

```
    delete ptr;
    // ERROR: Freeing memory that has already been freed
    // delete ptr; // Uncommenting this line will cause a double-delete error

    // ERROR: Mismatched allocation/deallocation
    char* c_ptr = (char*)malloc(10);
    // delete[] c_ptr; // Uncommenting will cause an error: using delete[] on malloc'd memory
    free(c_ptr); // This is the correct way
}

// ============================================================================
// 6. Data Race (Detected by Helgrind/DRD)
// ============================================================================
// A worker function that increments a global counter without locking.
void race_worker() {
    for (int i = 0; i < 100000; ++i) {
        // ERROR: Two threads read and write to this shared variable
        // concurrently without any synchronization (e.g., a mutex).
        g_shared_counter++;
    }
}

void run_data_race_test() {
    std::thread t1(race_worker);
    std::thread t2(race_worker);

    t1.join();
    t2.join();
}

// ============================================================================
// 7. Function for Profiling (Used with Callgrind/Cachegrind)
// ============================================================================
// A simple recursive function to generate some call graph data.
long long fibonacci(int n) {
    if (n <= 1) {
        return n;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}


// ============================================================================
// Main driver
// ============================================================================
int main() {
    std::cout << "--- Valgrind Test Suite ---" << std::endl;

    std::cout << "\n[1] Testing use of uninitialized value..." << std::endl;
    use_uninitialized_value();

    std::cout << "\n[2] Testing invalid heap access..." << std::endl;
    invalid_heap_access();

    std::cout << "\n[3] Testing invalid stack access..." << std::endl;
    invalid_stack_access();

    std::cout << "\n[4] Creating a memory leak..." << std::endl;
    memory_leaks();

    std::cout << "\n[5] Testing invalid free operations..." << std::endl;
    invalid_free();

    std::cout << "\n[6] Testing for data races..." << std::endl;
    run_data_race_test();

    std::cout << "\n[7] Running function for profiling..." << std::endl;
    long long result = fibonacci(20);
    std::cout << "Fibonacci result: " << result << std::endl;

    std::cout << "\n--- Test Suite Finished ---" << std::endl;
    std::cout << "Run this program under Valgrind to see the error reports." << std::endl;

    return 0;
}
```

Compile it:

```
g++ -g -pthread -o valgrind_test valgrind_test.cpp
```

2 Testing with Memcheck (Default Tool)

Memcheck is the default tool and detects memory-related issues like leaks, invalid reads/writes, and the use of uninitialized values.

```
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes ./valgrind_test
```

3 Testing with Helgrind (Threading Errors)

Helgrind is designed to find data races in multithreaded programs.

```
valgrind --tool=helgrind ./valgrind_test
```

4 Testing with Callgrind (Profiling)

Callgrind is a call-graph-generating profiler. It only reports how many instructions are executed and how the program's functions call each other.

```
# Step 1: Run the program under Callgrind to generate a data file
valgrind --tool=callgrind ./valgrind_test

# Step 2: Analyze the output file (e.g., callgrind.out.12345)
# The file will be named callgrind.out.<pid>
callgrind_annotate callgrind.out.*
```