ERROR HANDLING AND DEBUGGING

Sercan Külcü | 21.06.2023

Contents

1 Common C programming errors

In this chapter, we'll dive into some of the most common errors that beginners (and even experienced programmers) make while writing C code. By understanding these errors, you'll be better equipped to write efficient, bug-free programs.

1.1 Syntax Errors:

Syntax errors are perhaps the most common mistakes in any programming language, and C is no exception. These errors occur when the code violates the language's rules and grammar. Here are a few examples:

a) Missing semicolon: Forgetting to include a semicolon at the end of a statement is a common oversight. Remember, C uses semicolons to separate statements. So, always double-check your code for missing semicolons.

b) Mismatched parentheses: Unequal numbers of opening and closing parentheses can lead to syntax errors. Make sure to match every opening parenthesis with a corresponding closing parenthesis.

c) Unclosed quotes: Forgetting to close a quotation mark can result in syntax errors, especially when dealing with strings. Always ensure that you have closed all the quotes in your code.

1.2 Logical Errors:

Logical errors are trickier to identify than syntax errors because they do not produce any compiler warnings or errors. These errors occur when the program runs without crashing but produces incorrect or unexpected results. Common logical errors include:

a) Incorrect loop conditions: If your loop doesn't terminate when it should or terminates too early, the logic may be flawed. Carefully evaluate the conditions of your loops to ensure they are correct.

b) Off-by-one errors: These errors occur when you incorrectly access an array or iterate through a loop. For example, starting a loop from 1 instead of 0 or accessing an array element beyond its bounds. Pay close attention to index calculations and boundary conditions.

c) Incorrect operator precedence: Operator precedence determines the order in which expressions are evaluated. Failing to understand or misuse operator precedence can lead to incorrect calculations. Use parentheses to make your intentions clear and avoid confusion.

1.3 Runtime Errors:

Runtime errors occur during program execution and often result in program crashes or unexpected behavior. These errors may include:

a) Segmentation faults: A segmentation fault occurs when a program tries to access memory it is not allowed to access. This typically happens due to incorrect pointer usage, such as dereferencing a null pointer or accessing memory after it has been freed.

b) Buffer overflows: Buffer overflows happen when a program writes data beyond the bounds of an allocated buffer. This can lead to memory corruption and security vulnerabilities. Always ensure that you are working within the allocated bounds of your buffers.

c) Uninitialized variables: Using variables before initializing them can result in unpredictable behavior. Always initialize your variables before using them to avoid unexpected results.

2 Debugging techniques and tools

In this chapter, we'll explore the art of debugging, an essential skill for every programmer. We'll discuss various techniques and tools that can help you identify and fix bugs in your C programs efficiently.

Print Statements:

One of the simplest and most effective debugging techniques is the use of print statements. By strategically placing print statements in your code, you can track the flow of execution, observe variable values, and identify potential issues. Print statements allow you to visualize the program's behavior at different stages, making it easier to pinpoint problems.

For example, consider the following code snippet:

```
int sum(int a, int b) {
  int result = a + b;
  printf("The sum of %d and %d is: %d\n", a, b, result);
  return result;
}
```

By printing the values of a, b, and result, you can verify if the function is calculating the sum correctly. This technique can be used throughout your code to track variables and verify the program's logic.

Debuggers:

Debuggers are powerful tools designed to assist in finding and fixing errors. They allow you to step through your code line by line, inspect variables, and examine program state in real-time. Popular debuggers for C programming include gdb (GNU Debugger), Visual Studio Debugger, and Xcode Debugger.

Using a debugger, you can set breakpoints at specific lines of code, pause the program's execution, and examine the values of variables at that point. You can also

single-step through the code, executing one line at a time, to identify the exact location where an error occurs.

Debuggers provide a wealth of features, such as watches (to monitor specific variables), call stack inspection (to trace function calls), and memory analysis (to detect memory-related errors). Learning to use a debugger effectively can significantly speed up the debugging process and enhance your understanding of program flow.

Assert Statements:

Assert statements are useful for validating assumptions and catching errors during program execution. They allow you to specify conditions that must be true at a particular point in the code. If the condition evaluates to false, an error message is displayed, and the program terminates.

Consider the following example:

```
int divide(int a, int b) {
  assert(b != 0);
  return a / b;
}
```

In this case, the assert statement ensures that the divisor (b) is not zero before performing the division operation. If the condition is false, an error message will be displayed, helping you identify the problematic code.

Compiler Warnings and Errors:

Compiler warnings and errors should not be ignored; they are valuable clues that can lead you to potential bugs in your code. Pay attention to these messages and resolve them promptly.

Warnings highlight questionable code practices or potential issues that may not cause immediate problems but can lead to bugs in certain situations. Errors, on the other hand, indicate problems that prevent the code from compiling successfully.

Make it a habit to address warnings and errors systematically, ensuring your code is clean and error-free. Your compiler is a valuable ally in the debugging process.

Code Review and Pair Programming:

Another effective technique for debugging is to involve a fresh pair of eyes. Engaging in code review or pair programming sessions with other developers can help identify issues that you might have missed. Fresh perspectives can provide valuable insights, improving the overall quality of your code.

During code reviews, encourage constructive feedback and discussion. Embrace the opportunity to learn from others and share your knowledge.

3 Exception handling with try-catch blocks

Exceptions are unexpected events or errors that occur during the execution of a program. These events can range from runtime errors, such as division by zero or accessing invalid memory, to exceptional conditions like file I/O failures or network connection issues.

The purpose of exception handling is to anticipate and deal with such events in a controlled manner, rather than allowing them to crash the program or lead to undefined behavior. By catching and handling exceptions, you can provide meaningful feedback to the user, take corrective actions, or gracefully exit the program.

The Try-Catch Block Structure:

In C, exception handling is typically achieved using libraries or frameworks that provide support for try-catch blocks. Although C does not have built-in language constructs for exception handling like some other languages, you can implement a similar mechanism using libraries like setjmp.h and longjmp.h.

A try-catch block structure in C typically looks like this:

```
#include <setjmp.h>


jmp_buf exception_buffer;


void tryFunction() {
  if (setjmp(exception_buffer) == 0) {
    // Code that may raise an exception
  } else {
    // Code to handle the exception
  }
}
```

```
void throwException() {
  longjmp(exception_buffer, 1);
}
```

In this example, the tryFunction() encapsulates the code that may raise an exception. If an exception occurs, the setjmp() function stores the program's execution state in the exception_buffer and returns zero. However, if an exception is thrown using throwException(), the program jumps to the corresponding catch block using longjmp(). The catch block contains the code to handle the exception.

To handle exceptions in a meaningful way, you can define different catch blocks to handle specific types of exceptions. Each catch block corresponds to a particular type of exception and includes the code to handle that exception. By catching exceptions and providing appropriate error messages or recovery mechanisms, you can enhance the robustness of your program.

Consider the following example:

```
#include <stdio.h>
#include <setjmp.h>

jmp_buf exception_buffer;

void tryFunction() {
  if (setjmp(exception_buffer) == 0) {
    int numerator = 10;
    int denominator = 0;

    if (denominator == 0) {
      printf("Exception: Division by zero!\n");
```

```
        throwException();
    } else {
        int result = numerator / denominator;
        printf("Result: %d\n", result);
    }
  } else {
    printf("Exception handled!\n");
  }
}

void throwException() {
  longjmp(exception_buffer, 1);
}
```

In this example, we attempt to divide the numerator by zero, which raises a division by zero exception. By throwing the exception using throwException(), we transfer control to the corresponding catch block. In this case, we simply print an error message, but you can add more elaborate error handling or recovery logic as needed.

In addition to handling exceptions, some exception handling frameworks provide a finally block that allows you to execute cleanup actions, regardless of whether an exception was thrown or not. However, in C, you can achieve similar behavior by placing cleanup code immediately after the catch block. This ensures that the cleanup actions are executed before leaving the try-catch block.

Exception handling is particularly useful when it comes to resource deallocation. By catching exceptions and ensuring proper cleanup of resources (such as freeing memory, closing files, or releasing locks), you can avoid resource leaks and maintain the integrity of your program.

4 Defensive programming and error handling strategies

In this chapter, we'll explore the importance of defensive programming and effective error handling strategies. By adopting defensive programming techniques and implementing robust error handling mechanisms, you can ensure the reliability and stability of your C programs.

4.1 Defensive Programming Principles:

Defensive programming is an approach that aims to anticipate and prevent problems in software development. By following these principles, you can create code that is resilient and less prone to errors:

a) Validate Inputs:

Always validate user inputs and function parameters. Check for out-of-range values, invalid data types, or unexpected input patterns. By validating inputs, you can catch potential problems early on and provide appropriate feedback to the user.

b) Handle Unexpected Conditions:

Identify and handle exceptional or unexpected conditions gracefully. Rather than allowing the program to crash or produce incorrect results, implement error handling mechanisms to provide feedback and recover from errors. Consider scenarios such as invalid file paths, network timeouts, or unavailable system resources.

c) Avoid Assumptions:

Minimize assumptions about the state of your program, the environment, or the data you're working with. Instead, verify conditions explicitly before proceeding. For example, if you expect a file to be present, check for its existence before attempting to read or write.

d) Use Defensive Coding Techniques:

Employ techniques like data validation, input sanitization, and boundary checks to prevent buffer overflows, memory leaks, and other security vulnerabilities. Avoid using deprecated functions or unsafe practices and favor safer alternatives whenever possible.

4.2 Error Handling Strategies:

Effective error handling is a crucial aspect of defensive programming. It allows you to handle exceptional situations and communicate errors to the user or other parts of the program. Consider the following strategies for robust error handling:

a) Return Error Codes:

Instead of relying solely on return values to indicate success or failure, use error codes to provide more detailed information about the outcome of a function. By defining a set of error codes and returning them when appropriate, you enable callers to handle errors explicitly.

b) Error Reporting:

Implement mechanisms to report errors to users or log files. Provide clear and informative error messages that help users understand what went wrong and how to resolve the issue. Log important details about the error, such as timestamps, function names, and relevant data.

c) Resource Cleanup:

Ensure proper cleanup of resources, such as freeing allocated memory, closing files, or releasing acquired locks. Implement appropriate error handling mechanisms to guarantee that resources are properly cleaned up, even in the event of an error.

d) Graceful Termination:

When encountering critical errors that prevent the program from continuing, ensure graceful termination. Free allocated resources, close open files, and release acquired locks before exiting the program. Inform the user or logging system about the error and the reason for termination.

4.3 Logging and Debugging:

Logging and debugging tools are invaluable when it comes to identifying and diagnosing errors in your programs. By strategically placing log statements and utilizing debugging tools, you can gain insights into program flow, variable values, and error conditions.

Use logging statements to record important information during program execution, such as function entry/exit points, critical variable values, and error details. This can help trace the sequence of events leading to an error and provide useful debugging information.

Utilize debugging tools, such as debuggers or logging frameworks, to step through your code, inspect variables, and track program state. These tools offer features like breakpoints, watches, and call stack inspection to assist in pinpointing errors and understanding program behavior.