Arrays, Structures, Functions and Errors

1. Array

Array in C can be defined as a method of clubbing multiple entities of similar type into a larger group. These entities or elements can be of int, float, char, or double data type or can be of user-defined data types too like structures. However, in order to be stored together in a single array, all the elements should be of the same data type. The elements are stored from left to right with the left-most index being the 0th index and the rightmost index being the (n-1) index.

Array in C are of two types;

* Single dimensional arrays
* Multidimensional arrays.
* Single Dimensional Arrays:

Single dimensional array or 1-D array is the simplest form of arrays that can be found in C. This type of array consists of elements of similar types and these elements can be accessed through their indices.

* Multi-dimensional Arrays:

The most common type of multi-dimensional array that is used in the C language is a 2-D array. However, the number of dimensions can be more than 2 depending upon the compiler of the user’s system. These arrays consist of elements that are array themselves.

2.Structure in C

Structures are the user-defined data type, which allow us to collect the group of different data types. Here, all the individual components or elements of structures are known as a member. ‘Struct’ is a keyword that is used to create a structure.

Example of Structure in C

You can understand the structure in C with the help of the example below:

struct studentname

{

int rollnumber;

char name[20];

Float percentage;

};

Here struct Student is keeping the information of a student which consists of three data fields, roll number, name, and percentage. These fields are known as structure elements or members.

These elements can be of different data types. For example, here roll number is int type, the name is char type, etc.

3.Functions;

Functions in C are the basic building blocks of a C program.

A function is a set of statements enclosed within curly brackets ({}) that take inputs, do the computation, and provide the resultant output. You can call a function multiple times, thereby allowing reusability and modularity in C programming. It means that instead of writing the same code again and again for different arguments, you can simply enclose the code and make it a function and then call it multiple times by merely passing the various arguments.

What Are the Main Types of Functions in C Programming?

By Ravikiran A S

Last updated on Aug 3, 2023100568

An Introduction to Functions in C Programming

Table of Contents

Why Do We Need Functions in C Programming?Basic Syntax of FunctionsAspects of Functions in C ProgrammingTypes of Functions in C ProgrammingHeader Files for Library Functions in C ProgrammingView More

Functions in C are the basic building blocks of a C program. A function is a set of statements enclosed within curly brackets ({}) that take inputs, do the computation, and provide the resultant output. You can call a function multiple times, thereby allowing reusability and modularity in C programming. It means that instead of writing the same code again and again for different arguments, you can simply enclose the code and make it a function and then call it multiple times by merely passing the various arguments.

Why Do We Need Functions in C Programming?

* Enables reusability and reduces redundancy
* Makes a code modular
* Provides abstraction functionality
* The program becomes easy to understand and manage
* Breaks an extensive program into smaller and simpler pieces

Basic Syntax of Functions;

return\_type function\_name(arg1, arg2, … argn){

Body of the function //Statements to be processed

}

In the above syntax:

return\_type: Here, we declare the data type of the value returned by functions. However, not all functions return a value. In such cases, the keyword void indicates to the compiler that the function will not return any value.

function\_name: This is the function’s name that helps the compiler identify it whenever we call it.

arg1, arg2, ...argn: It is the argument or parameter list that contains all the parameters to be passed into the function. The list defines the data type, sequence, and the number of parameters to be passed to the function. A function may or may not require parameters. Hence, the parameter list is optional.

Body: The function’s body contains all the statements to be processed and executed whenever the function is called.

Aspects of Functions in C Programming

1. Function Declaration

The function declaration lets the compiler know the name, number of parameters, data types of parameters, and return type of a function. However, writing parameter names during declaration is optional, as you can do that even while defining the function.

2. Function Call

As the name gives out, a function call is calling a function to be executed by the compiler. You can call the function at any point in the entire program. The only thing to take care of is that you need to pass as many arguments of the same data type as mentioned while declaring the function. If the function parameter does not differ, the compiler will execute the program and give the return value.

3. Function Definition

It is defining the actual statements that the compiler will execute upon calling the function. You can think of it as the body of the function. Function definition must return only one value at the end of the execution.

Here’s an example with all three general aspects of a function.

#include <stdio.h>

// Function declaration

int max\_Num(int i, int j){

// Function definition

if (i > j)

return i;

else

return j;

}

// The main function. We will discuss about it later

int main(void){

int x = 15, y = 20;

// Calling the function to find the greater number among the two

int m = max\_Num(x, y);

printf("The bigger number is %d", m);

return 0;

}

4. Errors

Errors in C programming can disrupt the intended functionality of a program, causing issues such as failed compilation, program crashes, or incorrect output. There are several common types of errors:

* Syntax Error
* Run-Time Error
* Logical Error
* Semantic Error
* Linker Error

How to Read an Error in C

To address an error in our code, it is crucial to understand its cause and occurrence. When encountering an error, the compiler halts the compilation process if it detects a syntax error. The specific line of code responsible for the error is usually highlighted in such cases. The root cause of the error can often be identified on the highlighted line or in the code above it.

The syntax for error in C is -

#include <stdio.h>

int main() {

printf("Hello, world!")

return 0;

}

In this example, a syntax error occurs due to the missing semicolon (;) at the end of the printf statement. The compiler would highlight the line and display an error message indicating the syntax error.

1. Run-Time Error example :

#include <stdio.h>

int main() {

int a = 5;

int b = 0;

int result;

result = a / b; // Division by zero

printf("Result: %d", result);

return 0;

}

A run-time error occurs in this case because we attempt to divide an integer by zero. During program execution, a run-time error is detected, causing the program to terminate abruptly. The division by zero is not allowed and leads to undefined behaviour.

1. Logical Error example :

#include <stdio.h>

int main() {

int radius = 5;

double pi = 3.14;

double area;

area = pi \* radius \* radius;

printf("Area: %f", area);

return 0;

}

Here, a logical error arises due to an incorrect formula for calculating the area of a circle. The formula should be pi \* radius \* radius, but the programmer mistakenly uses pi \* radius instead. As a result, the program will produce incorrect output for the area.

Types of Errors in C Programming

Here is a list of types of errors in c with examples -

Syntax Error in C

Syntax error in C occurs when the code violates the rules and structure of the programming language. These errors prevent the code from being compiled and executed. The compiler identifies syntax errors and displays error messages pointing to the specific line or lines where the error occurred.

Causes of Syntax Errors:

Missing or incorrect punctuation marks: Forgetting to include semicolons (;) at the end of statements or using parentheses, brackets, or braces incorrectly.

Mismatched parentheses, brackets, or braces: Failing to close or open parentheses, brackets, or braces properly can lead to syntax errors.

Misspelt keywords or identifiers: Incorrectly typing keywords, variable names, or function names can result in syntax errors.

Improper use of operators: Using operators incorrectly or placing them in the wrong order can cause syntax errors.

Examples -

Missing Semicolon -

#include <stdio.h>

int main() {

printf("Hello, world!")

return 0;

}

In this example, a semicolon is missing at the end of the printf statement. The compiler will display an error message indicating the missing semicolon.

Example of semicolon error

Mismatched Bracket:

#include <stdio.h>

int main() {

if (1 > 0) {

printf("True");

else {

printf("False");

}

return 0;

}

In this example, there is a mismatched bracket in the if statement. The closing bracket for the if block is missing before the else statement. The compiler will detect this syntax error and provide an error message.

Missing Brackets Error Example

Runtime Error in C

Runtime error in C, also known as exceptions or execution errors, occur during the execution of a program. Unlike syntax errors, runtime errors do not prevent the code from being compiled, but they cause the program to behave unexpectedly, crash, or produce incorrect output. Runtime errors are typically caused by invalid input, zero division, accessing out-of-bounds memory, or incompatible data types.

Causes of Runtime Errors:

Poor Programming: Runtime errors can occur due to coding mistakes, memory leaks, or inadequate error handling.

Aging/Damaged Hardware: Runtime errors can be caused by deteriorating or faulty hardware components.

Software Interference: Conflicting or poorly performing software can trigger runtime errors in other programs.

Virus/Malware Infections: Viruses and malware can disrupt software execution and lead to runtime errors.

Examples of Runtime Errors:

Division by Zero:

#include <stdio.h>

int main() {

int a = 10;

int b = 0;

int result;

result = a / b; // Division by zero

printf("Result: %d", result);

return 0;

}

In this example, a runtime error occurs because we are attempting to divide an integer by zero. During program execution, a runtime error is detected, and the program terminates abruptly.

Division by Zero Runtime Error

Null Pointer Dereference:

#include <stdio.h>

int main() {

int\* ptr = NULL;

\*ptr = 10; // Null pointer dereference

return 0;

}

Logical Error in C

Logical error in C, also known as semantic errors, occur when the program runs without any syntax or runtime errors but produces incorrect or unexpected results. These errors are caused by flawed logic or incorrect algorithmic implementation in the code.

Unlike syntax or runtime errors, logical errors do not generate error messages or warnings from the compiler or runtime system. Detecting and fixing logical errors requires careful analysis of the code's logic and understanding of the intended behaviour.

Causes of Logical Errors:

Incorrect Sequence: Wrong order of code execution leading to unexpected outcomes.

Wrong Boolean Expression: Incorrect evaluation of conditions or incorrect use of Boolean expressions.

Incorrect Data Type Usage: Using incompatible or incorrect data types for variables or operations.

Missing Logic: Omission of necessary logic or checks in the code.

Examples of Logical Errors -

Incorrect Sequence:

a = 5

b = 3

c = a + b

print(c)

a = b

print(a)

In this example, the sequence of statements is incorrect. The value of a is printed before it is updated to b, resulting in the output being 5 instead of 3.

Wrong Boolean Expression:

x = 10

y = 5

if x < y or x > y: # Incorrect Boolean expression

print("x is not equal to y")

else:

print("x is equal to y")

In this example, the Boolean expression used in the if statement is incorrect. Instead of checking for equality between x and y, it checks for inequality. This leads to the incorrect output "x is not equal to y" even when x and y are equal.

Linker Errors

Linker errors occur during the linking phase of the program's compilation process. The linker resolves external references, combines multiple object files, and generates the final executable file. Linker errors occur when there are issues with linking the object files together to create the executable.

Causes of Linker Errors:

Undefined Symbols: When the linker cannot find the definition of a referenced symbol (function, variable, or class) because it is not defined or implemented.

Duplicate Definitions: When multiple source files have definitions for the same symbol, causing conflicts during the linking process.

Missing Libraries or Dependencies: When required libraries or dependencies are not properly linked or included in the project.

Incompatible Architectures or Platforms: Compatibility issues occur when object files or libraries are compiled for different architectures or platforms, are linked together.

Examples of Linker Errors

Undefined Symbol Error:

main.obj : error LNK2019: unresolved external symbol addNumbers referenced in function main

In this example, the linker error indicates that the symbol addNumbers is referenced in the main function but is not defined or implemented in any object files. It suggests that the function addNumbers needs to be defined or adequately linked.

Duplicate Symbol Error:

first.obj : error LNK2005: \_variableName already defined in second.obj

This linker error occurs when the symbol \_variableName is defined in both first.obj and second.obj. It indicates a conflict due to duplicate definitions and suggests resolving the issue by ensuring unique symbol names.

Preprocessor Errors

Preprocessor errors occur during the preprocessing phase of the compilation process, where the preprocessor directives are processed before the actual compilation begins. These errors are related to the preprocessing directives and can prevent the code from being properly processed and compiled.

Causes of Preprocessor Errors:

Missing/incorrect directives: Errors due to missing or incorrect preprocessor directives like forgetting to include necessary headers or using incorrect syntax.

Circular dependencies: Errors caused by circular dependencies among header files, leading to an infinite preprocessing loop.

Macro-related issues: Errors arising from missing/incorrect macro definitions, improper macro expansions, or conflicts between macros.

Conditional compilation issues: Errors resulting from improperly structured or evaluated conditional compilation directives like #ifdef, #ifndef, #if, etc., and mismatched/unbalanced directives.

Examples of Preprocessor Errors:

Missing Header File:

If a required header file is not included or the file path is incorrect, it can cause a preprocessor error. For example,

#include <stdio.h> // Correct header file inclusion

#include <invalid\_header.h> // Incorrect or missing header file

Strategies for Debugging Errors

1.Using Debugging Tools

Debugging tools are essential for identifying and resolving errors. Integrated Development Environments (IDEs) often provide debugging features such as breakpoints, step-through execution, variable inspection, and call stack tracing. By leveraging these tools, you can

* Track the execution flow,
* Examine variable values,
* Identify the source of the error.

2.Tracing the Code:

Tracing the code involves manually examining the code's execution path to identify the point where the error occurs. This can be done by strategically inserting logs or printing statements at different parts of the code to display intermediate values or checkpoints. Observing the output or logs lets you

* track the program's flow
* identify the error's origin.

3.Using Print Statements

Print statements, also known as "printf" statements in languages like C/C++, are simple yet effective for debugging. By selectively adding print statements at key points in the code, you can output variable values or custom messages to the console or log files. This lets you observe the program's state during execution and locate the error.