One-Week-C-Primer

Day 1: Introduction, Keywords & Data Types

Note : Refer to the Attached Document along with this for the codes

Introduction To C

Welcome to Day 1 of our programming journey! In this tutorial series, we will explore the fundamentals of C programming, one of the most widely used and influential programming languages in the world. Whether you're a complete beginner or already have some programming experience, this series will provide you with a solid foundation in C.

What is C?
C is a high-level, general-purpose programming language developed in the early 1970s by Dennis Ritchie at Bell Labs. It was designed to be simple, efficient, and versatile, making it suitable for a wide range of applications, from system programming to developing applications and games.

Why Learn C?
Learning C is like learning the language of computers. It allows you to communicate directly with the hardware, giving you precise control over the system's resources. C's efficiency and portability have made it the language of choice for building operating systems, compilers, and embedded systems.

Key Features of C:

1. Efficiency: C is known for its fast execution and low-level memory access, making it ideal for performance-critical tasks.
2. Portability: Programs written in C can be easily adapted to run on different platforms with minimal modifications.
3. Extensibility: C provides various features to support modular and scalable programming, making it suitable for large projects.
4. Standard Library: C comes with a rich standard library that provides a wide range of functions for various operations.

5. Wide Adoption: C has a vast and active community, making it easier to find resources, libraries, and support.

Getting Started:
To begin our C programming journey, we'll set up our development environment and write our first "Hello World" program. We'll explore the basic syntax, data types, variables, and more as we progress through the series.

Let's embark on this exciting adventure of learning C programming. Whether your goal is to become a proficient programmer or to deepen your understanding of computer systems, C is the perfect language to master. So, let's dive in and start coding!

Keywords In C

Keywords in C are reserved words that have predefined meanings and specific functionalities in the C programming language. These words cannot be used as identifiers (e.g., variable names, function names) in the program. Here is a list of all keywords in C:

auto	break	case	char	const	co
double	else	enum	extern	float	for
int	long	register	return	short	sig
struct	switch	typedef	union	unsigned	voi

These keywords are fundamental to the C language and serve various purposes, including defining variables, controlling program flow, and declaring user-defined data types. Proper understanding and usage of these keywords are essential for writing correct and efficient C programs.

Identifiers In C

Identifiers play a crucial role as they serve as the names given to various elements such as variables, functions, arrays, and more. An identifier is essentially a user-defined name used to represent a specific entity within the code.

Rules for Naming Identifiers:

1. Identifiers must start with a letter (a-z, A-Z) or an underscore (_). Following the first character, you can use letters, digits (0-9), and underscores in the identifier.
2. C is a case-sensitive language, meaning that uppercase and lowercase letters are treated as distinct characters. For example, "variable" and "Variable" are considered different identifiers.
3. Identifiers cannot be a C keyword or a reserved word. Keywords are special words reserved for specific purposes in the language and cannot be used as identifiers. For instance, "int," "if," "while," and "for" are all keywords.

Naming Conventions and Best Practices:

To ensure consistency and readability in your code, it's essential to follow common naming conventions and best practices for identifiers. Here are some suggestions:

1. Use meaningful and descriptive names: Choose names that clearly represent the purpose of the variable or function. For instance, instead of using "x" or "temp," opt for more descriptive names like "counter" or "temperature."
2. CamelCase or underscore_case: There are two popular naming styles. In CamelCase, words are written together without spaces, and each word's first letter, except the first one, is capitalized (e.g., firstName, calculateTotal). In underscore_case, words are separated by underscores and are usually written in lowercase (e.g., first_name, calculate_total).

Using Keywords in Identifiers:

It's essential to avoid using keywords as identifiers, as it results in compilation errors. For example, attempting to use "int" as an identifier will lead to a conflict with the built-in "int" keyword used for defining integer variables.

Length Restrictions:

While C does not impose strict limits on identifier lengths, it is wise to keep them reasonably short and meaningful. Extremely long identifiers may lead to readability issues, while excessively short ones might not be expressive enough.

Scope and Visibility:

The scope of an identifier refers to the part of the code where the identifier is accessible. C has block scope and function scope for identifiers. Variables declared inside a block (e.g., within loops or if-else statements) have block scope, and they are only accessible within that block. On the other hand, variables declared outside any function have global scope and can be accessed throughout the program.

Declaring and Defining Identifiers:

To use an identifier, it must first be declared or defined. Declaring an identifier informs the compiler about the identifier's existence and data type. Defining an identifier assigns a value to it. For instance, consider the following examples:

```
// Declaration of a variable
int age;

// Definition of a function
void greetUser() {
    printf("Hello, User!\\n");
}
```

Identifier Naming for Constants and Macros:

For constants, it's a good practice to use the "const" keyword to indicate that the value is not intended to be modified. For macros, which are usually defined using the "#define" preprocessor directive, use all capital letters with underscores to separate words.

```
// Constant variable
const float PI = 3.1415;

// Macro
#define MAX_LENGTH 100
```

Legal and Illegal Identifiers:

Here are some examples of valid and invalid identifiers:

Valid identifiers:

```
score
_player
MAX_SIZE
user123
```

Invalid identifiers:

```
123abc (starts with a digit)
float (matches a keyword)
my-variable (contains a hyphen, not allowed)
```

Predefined Identifiers:

C provides several predefined identifiers, also known as macros, that begin and end with double underscores. These identifiers offer valuable information about the code, such as the current file name, line number, and function name. Some common predefined identifiers include `__FILE__`, `__LINE__`, and `__func__`.

Common Mistakes and Troubleshooting:

When dealing with identifiers, beginners may encounter some common mistakes, such as misspelling identifiers, using keywords unknowingly, or forgetting to declare a variable before using it. To troubleshoot such issues, it is essential to double-check identifier names and ensure proper declaration and scope

---

Variables and Constants in C Language

Variables and constants are essential concepts in the C programming language, serving as the building blocks for data manipulation and storage. In this topic, we'll explore the fundamentals of variables and constants, how to declare and use them, and their significance in C programming.

Variables:

A variable is a named location in memory that stores a value of a specific data type. The value stored in a variable can be changed during the program's execution, allowing programmers to work with dynamic data.

Declaring Variables:

In C, variables must be declared before they are used. The declaration includes the variable's data type and a unique identifier (variable name). The syntax for declaring a variable is as follows:

```
data_type variable_name;
```

Here, `data_type` represents the type of data the variable can hold, such as integers (`int`), floating-point numbers (`float`), characters (`char`), or custom-defined structures.

Initializing Variables:

After declaring a variable, it is good practice to initialize it with an initial value. Variables that are not explicitly initialized may contain garbage values, leading to unexpected behavior in the program.

```
int age;         // Variable declared without initialization
int score = 100; // Variable declared and initialized with value 100
```

Constants:

A constant, as the name suggests, is a value that remains fixed during the program's execution. Constants are used when you want to assign a specific value to a variable that should not change throughout the program's lifespan.

Constants can be of various data types, just like variables, including integer constants, floating-point constants, character constants, and string constants.

Integer Constants:

```
const int MAX_SCORE = 100;
```

Floating-point Constants:

```
const float PI = 3.1415;
```

Character Constants:

```
const char GRADE = 'A';
```

String Constants:

```
const char GREETING[] = "Hello, World!";
```

Constants vs. Macros:

In addition to using the `const` keyword to define constants, C also allows the use of preprocessor macros with `#define`. However, it is generally recommended to use `const` for defining constants, as it provides better type checking and is more expressive.

```
// Constants defined using const
const int MAX_VALUE = 100;

// Constants defined using macros
#define PI 3.1415
```

Modifying Variables and Constants:

Variables can be modified during the program's execution by assigning new values to them. On the other hand, constants, by definition, cannot be changed after they are assigned a value. Attempting to modify a constant will result in a compilation error.

```
int age = 25;          // Variable
const int MIN_AGE = 18; // Constant

age = 30;              // Valid - Modifying the variable
MIN_AGE = 20;          // Invalid - Attempting to modify the constant
```

Basic Data Types in C:

Data types in C specify the type of data that a variable can hold. C provides several fundamental data types, each with its size and range of values. Understanding these basic data types is crucial for writing efficient and accurate C programs.

1. Integer Data Types:

   - `int`: The most common integer data type in C. Typically, it uses 4 bytes and can represent whole numbers in the range from -2,147,483,648 to 2,147,483,647.
   - `short`: Uses 2 bytes and represents a smaller range from -32,768 to 32,767.
   - `long`: Typically uses 4 bytes (may vary across platforms) and extends the range of `int` from -2,147,483,648 to 2,147,483,647.

    - `long long`: Uses 8 bytes and offers an even larger range for very large integers.

2. Floating-Point Data Types:
    - `float`: Represents single-precision floating-point numbers and typically uses 4 bytes. It can store up to 7 significant digits.
    - `double`: Represents double-precision floating-point numbers and usually uses 8 bytes. It can store up to 15 significant digits.
    - `long double`: Represents extended-precision floating-point numbers and may use 10 or 12 bytes on some systems.
3. Character Data Type:
    - `char`: Represents a single character and uses 1 byte of memory. It can store characters from the ASCII character set (0 to 127) or extended ASCII (0 to 255).
4. Void Data Type:
    - `void`: It is a special data type that indicates an empty or no value. It is often used as the return type for functions that do not return any value.

Sizeof Operator:

To determine the size of a data type in bytes, C provides the `sizeof` operator. It allows you to obtain the memory size of variables or data types at compile time.

```
#include <stdio.h>
int main() {
    printf("Size of int: %zu bytes\n", sizeof(int));
    printf("Size of float: %zu bytes\n", sizeof(float));
    printf("Size of char: %zu bytes\n", sizeof(char));
    return 0;
}
```

Output:

```
Size of int: 4 bytes
Size of float: 4 bytes
Size of char: 1 byte
```

Signed vs. Unsigned:

For integer data types (`int`, `short`, `long`, `long long`), you can use the `signed` or `unsigned` keywords to specify whether the numbers should represent both positive and negative values (signed) or only non-negative values (unsigned). By default, `int` is considered signed.

```
signed int num1 = -10;
unsigned int num2 = 20;
```

C Input and output

Input and output operations are vital for any programming language. In C, the standard input/output functions `printf` and `scanf` are used to interact with the user and display results on the screen. Understanding how to use these functions is essential for creating interactive and informative C programs.

1. **printf Function:**

The `printf` function is used to output data to the console (standard output). It allows you to display text and variable values on the screen.

Syntax:

```
int printf(const char* format, ...);
```

- The `format` argument is a string containing the text to be displayed. It may include format specifiers (placeholders) that are replaced by the values of variables.
- Format specifiers begin with the percent symbol `%`, followed by a character that specifies the data type to be displayed.

Example:

```
#include <stdio.h>

int main() {
    int age = 25;
    printf("My age is %d years.\n", age);
    printf("The value of PI is %.2f.\n", 3.14159);
    printf("Welcome, %s!\n", "Winter Soldier");
    return 0;
}
```

Output:

```
My age is 25 years.
The value of PI is 3.14.
Welcome, Winter Soldier!
```

1. **scanf Function:**

The `scanf` function is used to read data from the user (standard input). It allows you to input values into variables based on specified format specifiers.

Syntax:

```
int scanf(const char* format, ...);
```

- The `format` argument is a string that contains format specifiers corresponding to the variables where the input values will be stored.
- You need to pass the addresses of variables to store the input values by using the `&` (address-of) operator.

Example:

```
#include <stdio.h>

int main() {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);
    printf("You entered: %d\n", num);
    return 0;
}
```

Output:

```
Enter a number: 42
You entered: 42
```

Formatting Input with scanf:

To read multiple values from the user, you can use multiple format specifiers in the `scanf` function. Make sure to match the order of format specifiers with the variables' order to which the input values should be assigned.

Example:

```
#include <stdio.h>

int main() {
    int num1, num2;
    printf("Enter two numbers separated by a space: ");
    scanf("%d %d", &num1, &num2);
    printf("Sum: %d\n", num1 + num2);
    return 0;
}
```

Output:

```
Enter two numbers separated by a space: 10 20
Sum: 30
```