Variables in C :

- A **variable** in C is a named storage location in memory that holds a value. This value can change during the execution of the program. The value that a variable holds depends on its data type, which determines the size of the variable in memory and the kind of data it can store (e.g., integers, floating-point numbers, characters, etc.).
- Variables in C must be declared with a specific data type before they can be used. C supports various data types, such as int, float, char, double, long, etc.

Syntax:

The syntax for defining a variable in C is as follows:

```
data_type variable_name;
```

Here,

- **data_type :** It represents the type of data the variable can hold. Examples of data types in C include ***int (integer), float (a floating-point number), char (character), double (a double-precision floating-point number),***
- **variable_name :** It is the identifier for the variable, i.e., the name you give to the variable to access its ***value*** later in the program. The variable name must follow specific rules, like starting with a ***letter*** or ***underscore*** and consisting of ***letters***, digits, and underscores.

For example, to declare the integer variable ***age*:**

```
int age;
```

It declares an ***integer variable*** named age without assigning it a specific value. Variables can also be initialized at the time of declaration by assigning an ***initial value*** to them. For instance:

```
int count = 0;
```

Here, the ***variable count*** is declared an ***integer*** and initialized with ***0***.

Note: Variables should be defined before they are used within the program. The scope of a variable determines where it is accessible. Variables declared inside a function or block are considered local variables, while declared outside any function are considered global variables.

Syntax:

Let us see the syntax to declare a variable:

```
type variable_list;
```

An example of declaring the variable is given below:

1. **int** a;
2. **float** b;
3. **char** c;

Here, ***a, b,*** and ***c*** are variables. The ***int, float,*** and ***char*** are the data types.

We may also provide values while defining variables, as shown below:

1. **int** a=20,b=30;//declaring 2 variable of integer type
2. **float** f=20.9;
3. **char** c='A';

Variable Naming Rules

In C, there are specific rules and conventions for naming variables. These are crucial to ensure your program compiles and functions correctly.

Rules for Naming Variables:

1. **Start with a Letter or Underscore**: Variable names must begin with a letter (A-Z, a-z) or an underscore (_). They cannot start with a number.

```
int _age;   // Valid

int 2value; // Invalid, cannot start with a number
```

2. **Followed by Letters, Numbers, or Underscore**: After the first character, variable names can contain letters, numbers (0-9), and underscores (_).

```
int age1;   // Valid

int first_name; // Valid
```

3. **No Spaces**: Variable names cannot have spaces between characters.

```
int my age;  // Invalid, spaces are not allowed
```

4. **No Special Characters**: Special characters like @, $, %, etc., are not allowed in variable names.

```
int $value;  // Invalid, special characters are not allowed
```

5. **Case Sensitivity**: C is case-sensitive, meaning that age and Age are considered different variables.

```
int age; // variable 'age'

int Age; // variable 'Age' is different
```

6. **Avoid Reserved Keywords**: Variable names cannot be C keywords (reserved words like int, for, return, if, etc.). Using these words as variable names will lead to errors.

```
int return;  // Invalid, 'return' is a keyword
```

7. **Length of Variable Names**: While the C standard doesn't specify a maximum length, some compilers may impose a limit on the number of characters in a variable name.

4. Variable Scope and Lifetime

The **scope** of a variable refers to the portion of the code in which it is accessible. There are three main types of variable scope in C:

1. **Local Variables**: Declared inside a function, local variables can only be used within that function.

```
void myFunction() {
   int localVar = 5;  // 'localVar' can only be used within myFunction
}
```

2. **Global Variables**: Declared outside of any function, global variables can be accessed by any function in the program.

```
int globalVar = 10;  // Can be accessed by all functions in the program


void myFunction() {
   printf("%d", globalVar);  // Access globalVar inside function
}
```

3. **Static Variables**: Retain their value between function calls. They are initialized only once.

```
void countCalls() {
   static int count = 0;  // 'count' retains its value between function calls
   count++;
   printf("%d\n", count);
}
```

4. **Automatic Variables**: These are local variables created when a function is called and destroyed when the function returns. By default, all local variables are automatic in C.

5. Memory Allocation and Data Storage

- **Automatic Variables**: Local variables are stored in the stack, meaning they are automatically created when the function is called and destroyed when the function ends.
- **Static Variables**: Stored in a fixed memory location (data segment), they retain their value between function calls.
- **Global Variables**: Also stored in the data segment and can be accessed by any function in the program.
- **Dynamic Memory Allocation**: Variables can be allocated memory dynamically at runtime using malloc(), calloc(), or realloc().

6. Example of Variables in C

```
#include <stdio.h>


int globalVar = 10; // Global variable
```

```
void myFunction() {
    int localVar = 5;  // Local variable
    static int staticVar = 1;  // Static variable

    // Modify local and static variables
    localVar += 1;
    staticVar += 1;

    // Output the values of variables
    printf("Local Variable: %d\n", localVar);
    printf("Static Variable: %d\n", staticVar);
    printf("Global Variable: %d\n", globalVar);
}

int main() {
    // Call the function a few times
    myFunction();  // Output: Local Variable: 6, Static Variable: 2, Global Variable: 10
    myFunction();  // Output: Local Variable: 6, Static Variable: 3, Global Variable: 10

    return 0;
}
```

In the above example:

- globalVar is a global variable that retains its value throughout the program.
- localVar is a local variable that resets every time myFunction() is called.
- staticVar is a static variable that retains its value between function calls.

7. Best Practices for Working with Variables

- **Descriptive Names**: Always give variables descriptive names to improve readability and maintainability. For example, int totalAmount is better than int a.
- **Avoid Magic Numbers**: Instead of directly using numbers (like 10 or 100) in your code, use constants with descriptive names.

- **Minimize Global Variables**: Use local variables as much as possible to avoid unwanted side effects and confusion.
- **Initialization**: Always initialize variables when declaring them. This ensures you don't accidentally work with uninitialized values.
- **Limit Scope**: Try to limit the scope of variables to as small a region of code as possible to improve readabilit

Constants in C :

A **constant** in C is a value that cannot be changed during the execution of the program. Once a constant is assigned a value, it cannot be altered, which helps in maintaining fixed values throughout the program. Constants are often used to represent fixed values, such as mathematical constants (e.g., π), system configuration values, or configuration options that should remain unchanged during the program's execution.

1. Types of Constants in C

There are several types of constants in C:

a) Integer Constants

These are used to represent integer values. An integer constant can be written in:

- **Decimal** (base 10): 10, 25, 1000
- **Octal** (base 8, prefixed with 0): 017 (equivalent to decimal 15)
- **Hexadecimal** (base 16, prefixed with 0x or 0X): 0x1A (equivalent to decimal 26)

Example:

```
int a = 25;  // Decimal constant

int b = 017; // Octal constant, equivalent to 15 in decimal

int c = 0x1A; // Hexadecimal constant, equivalent to 26 in decimal
```

b) Floating-Point Constants

These represent real numbers (numbers with decimal points). A floating-point constant can be written in:

- **Decimal form**: 3.14, 0.005
- **Scientific notation**: 1.5e3 (which is 1.5 × 10^3 or 1500.0)

Example:

```
float pi = 3.14159; // Decimal constant
```

```
double e = 2.71828; // Decimal constant
```

c) Character Constants

These represent single characters enclosed in single quotes, such as 'A', '1', or '%'. The constant value is stored as an integer according to the character's ASCII value.

Example:

```
char letter = 'A';  // Character constant
```

d) String Constants

A string constant is a sequence of characters enclosed in double quotes. In C, strings are stored as arrays of characters terminated by the null character ('\0').

Example:

```
char str[] = "Hello, World!";  // String constant
```

e) Enumeration Constants

In C, enumerated types (enum) can define named integer constants. These constants are assigned integer values starting from 0 by default.

Example:

```
enum week { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };

enum week today = Wednesday;  // Enumeration constant
```

f) Constant Variables (using const keyword)

You can define a constant variable using the const keyword. A const variable behaves like a regular variable, but its value cannot be changed after initialization.

Example:

```
const int MAX_SIZE = 100;  // Constant variable, value cannot be changed
```

2. Defining Constants in C

There are two main ways to define constants in C:

a) Using the const Keyword

The const keyword is used to declare constant variables. These variables are treated as read-only, and any attempt to modify their value results in a compile-time error.

Example:

```
const int MAX_LENGTH = 50;  // Constant integer

const float PI = 3.14159;   // Constant floating-point number

const char NEWLINE = '\n';   // Constant character
```

- Constants defined using const must be initialized at the time of declaration because their value cannot be changed afterward.

b) Using #define Preprocessor Directive

The #define directive is often used to define symbolic constants (also known as macro constants). It is replaced by the preprocessor before compilation.

Example:

```
#define MAX_SIZE 100   // Define a constant for MAX_SIZE

#define PI 3.14159     // Define a constant for PI
```

- With #define, constants are replaced by their values throughout the program during preprocessing.
- No data type is associated with #define constants, as the preprocessor simply substitutes the text.

c) Using const vs #define

- const constants are variables, and they occupy memory, whereas #define constants are replaced by the preprocessor before compilation and do not occupy memory.
- const variables respect data types, but #define constants do not have data types.

In general, using const is preferred over #define for type safety and readability, but #define is sometimes used for defining symbolic constants or macros in larger programs.

3. Why Use Constants?

- **Readability**: Constants make your code easier to read because they represent meaningful names rather than raw numbers or strings. For example, MAX_HEIGHT is more descriptive than using 100 directly in the code.
- **Maintainability**: If you need to change a constant value, you can do so in one place, and it will reflect throughout the program, making maintenance easier.
- **Prevents Errors**: Since constants cannot be modified, using them reduces the risk of accidental modification, ensuring the integrity of the value.
- **Efficiency**: Constants (especially #define constants) are often used for values that will not change, providing faster access and saving memory.

4. Constant Example in C

Here’s an example program that demonstrates the use of constants in C:

```
#include <stdio.h>


// Define a constant using #define

#define PI 3.14159

#define MAX_SIZE 100
```

```
// Define a constant using const keyword
const int MAX_VALUE = 500;
const char NEWLINE = '\n';

int main() {
  // Using const variables
  printf("Maximum Value: %d\n", MAX_VALUE);  // Output: Maximum Value: 500

  // Using #define constants
  printf("Value of PI: %.5f\n", PI);  // Output: Value of PI: 3.14159

  // Using character constant
  printf("Character constant: %c\n", NEWLINE);  // Output: Character constant: (New Line)

  // Attempting to modify a const variable (this will result in a compile-time error)
  // MAX_VALUE = 1000;  // Error: assignment of read-only variable 'MAX_VALUE'

  return 0;
}
```

In this example:

- MAX_SIZE and PI are defined using #define.
- MAX_VALUE and NEWLINE are defined using the const keyword.
- Attempting to modify MAX_VALUE after initialization will cause a compile-time error because it is a constant.

5. Best Practices for Using Constants

- **Naming Convention**: For #define constants, it’s a common practice to use uppercase letters with underscores to separate words (e.g., MAX_SIZE, PI). For const variables, camelCase or PascalCase can be used (e.g., maxValue, piValue).
- **Use Constants for Fixed Values**: Always use constants for values that should not change, such as configuration settings, mathematical constants, and maximum limits.
- **Prefer const Over #define**: Prefer const variables over #define for better type safety and to take advantage of debugging tools, as const variables are real variables that the compiler can check for errors.