The C Programming Language

by Erica D. Perkins

June 7th, 2025

Version 1.0

Contents

1 Introduction to C Programming Language

C is a general-purpose, imperative, and programmer-oriented computer programming language that supports structured programming. It was invented in 1972 by Dennis Ritchie at Bell Labs. C is one of the most widely used programming languages of all time, and it has influenced many other popular programming languages, including C++, C#, and Java.

1.1 Fundamental tasks when authoring a C Program

i. **Editing:** This is where one writes and edits the source code of a C program. Afterwards, the source code is saved in a file with a .c file extension. To view the contents of a source file from the terminal, one can use the **cat** command.

   Example:

```
user@mac %: cat sourceFile.c
```

ii. **Compiling:** This is a two-stage process: preprocessing and compilation. The C compiler translates the source code into machine code, creating an executable file that can be run on a computer. It is here the source code is translated into Assembly Language (instructions) → Object Code .obj or .o file extension. Errors are detected in this period as well.

   Example:

```
user@mac %: gcc -c sourceFile.c
```

iii. **Linking:** After the program has been translated into object code, it is ready to be linked (McGoff, J., 2024). Most IDEs have made it so that this occurs automatically duing compilation; however, other times linking is a manual process that demands its own seperate command. The objective is to "combine the object modules generated by the compiler with additonal libraries needed by the program to creat the whole executable (McGoff, J., 2024)." **Example:**

```
user@mac %: gcc -c sourceFile.c
user@mac %: gcc sourceFile.o -o program
```

iv. **Execution:** There is usually a button in the GUI that will allow for one to execute the newly compiled program. If it is the case that execution need be done manuallyu, this can be initiated via double-tapping the file or writing the following command in linux terminal.

Example:

```
user@mac %: chmod +x sourceFile.c
user@mac %: sudo ./sourceFile.c
user@mac %: #In Windows type 'wine program.exe'
```

The **#include** statement is a preprocessor directive.
Example: #include <stdio.h>
This line tells the compiler to include the Standard Input Output library before compiling the program. Libraries are collections of precompiled routines that a program can use. The stdio.h library contains routines for input and output, such as printf and scanf.

It is not strictly part of the executable programl; however, the program will not work without it.
The symbol # indicates this is a preprocessing directive:

- This is similar to the import statement in Java.
- These usually are placed at the beginning of the program source file, but they can be anywhere.
- It is an instruction to the C compiler to do something before compiling the source code

1.2 Single and Multi-line Comments

Comments are utilized to help one read through the program, but it can also help point the way to the source of the logic mistake. Comments are an essential part of writing code, as they allow programmers to document their code and make it more understandable for others (and themselves) in the future.

In C, comments can be written in two ways:

- **Single-line comments:** These are used to provid shorter length comments. These comments begin with two forward slashes // and continue until the end of the line. They are used for brief explanations or notes.

Example of a single-line comment:

```
1  //This is a single-line comment.
```

- **Multi-line comments:** These comments begin with /* and end with */. They can span multiple lines and are useful for more detailed explanations.

Example of a multi-line comment:

```
1  /*
2   * This is a multi-line comment. I have much to say
3   * about this particular section of code, and so
4   * I have initiated a multi-line comment. Now I may
5   * provide useful commentary that spans multiple lines.
6   */
```

Comments are ignored by the compiler, meaning they do not affect the execution of the program. However, they are crucial for maintaining code quality and readability.
Best Practices:

- Use comments to explain why the code exists, not just what it does.
- Avoid obvious comments that repeat the code verbatim.
- Maintain up-to-date comments when modifying code.

2 Pointers

A pointer is low-level concept. They are a powerful feature in C that allow you to directly manipulate memory. A pointer is a variable that stores the memory address of another variable. It is similar to a variable in that way (stores a value and has a location in memory). Despite the similarities, pointers and variables have distinctly unique differences. The main differences are that pointers are prefixed with an asterisk symbol (*) and they store memory addresses instead of data values of another (different variable). This can be useful for a variety of tasks, such as dynamic memory allocation, arrays, and function arguments.

2.1 Declaring and Initializing Pointers

To declare a pointer in C, one must specify the type of data it will point to, followed by an asterisk (*) and the pointer's name. Below is the syntax for declaring a pointer.

Example:

```
1 int *ptr;
```

This declares a pointer to an integer. The pointer itself is not an integer; it simply holds the address of an integer variable. You can initialize a pointer by assigning it the address of a variable using the & operator:

Example:

```
1 #include <stdio.h>
2 int main() {
3       //An int variable
4       int baseTen = 774;
5       //A pointer that stores the address of `baseTen`
6       int* thePoint = &baseTen;
7       //Display the memory location of `baseTen`
8       printf("%p", thePoint);
9      return 0;
10 }
```

2.2 Dereferencing Pointers

Dereferencing a pointer in C means using the asterisk (*) operator to access the value at the memory address stored.

Example:

```
1 int value = *thePoint;
```

This assigns the value of thePoint (which is 774) to value.

Reasons to dereference a pointer in C:

1. Access the value at a memory address to read or modify the value that a pointer points to.

```
1 int num = 10;
2 int *ptr = &num;
3 *ptr = 20;            // Modifies the value of num
4 printf("num = %d\n", num);    // Output: num = 20
```

2. Implement dynamic memory operations to interact with memory allocated at runtime.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5       int *arr = (int*)malloc(3 * sizeof(int));
6       if (arr == NULL) {
7             printf("Memory allocation failed\n");
8             return 1;
9       }
10      arr[0] = 10; arr[1] = 20; arr[2] = 30;
11      printf("Values: %d, %d, %d\n", arr[0], arr[1],
arr[2]);
12      free(arr);
13      return 0;
14}
```

3. Work with function parameters efficiently to modify arguments passed to functions.

```c
void increment(int *num) {
    (*num)++;               // Dereference pointer to modify original value
}

int main() {
    int value = 5;
    increment(&value);
    printf("Value after increment: %d\n", value);
    return 0;
}
```

4. Traverse arrays or strings to iterate over elements using pointer arithmetic.

```c
int arr[] = {1, 2, 3, 4, 5};
int *ptr = arr;
for (int i = 0; i < 5; i++) {
    printf("Element %d: %d\n", i, *(ptr + i));
}
```

```c
char str[] = "Hello";
char *p = str;
while (*p != '\0') {
    printf("%c ", *p);
    p++;
}
```

5. Interact with data structures to access members of structures via pointers.

```c
#include <stdio.h>
struct Point {
    int x;
    int y;
};

int main() {
    struct Point p = {10, 20};
    struct Point *ptr = &p;
    printf("x = %d, y = %d\n", ptr->x, ptr->y);
    (*ptr).x = 30;            // Alternative way to access member
    printf("After update: x = %d\n", p.x);
    return 0;
}
```

6. Return multiple values from a function to allow a function to modify multiple variables.

```c
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 5, y = 10;
    swap(&x, &y);
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

7. Implement linked data structures to move through nodes in a linked list or tree.
   Example: Traversing a singly linked list

```c
#include <stdio.h>
struct Node {
      int data;
      struct Node *next;
};
int main() {
      struct Node n1 = {1, NULL};
      struct Node n2 = {2, NULL};
      struct Node n3 = {3, NULL};
      n1.next = &n2; n2.next = &n3;
      struct Node *current = &n1;
      while (current) {
            printf("Node data: %d\n", current->data);
            current = current->next;
      return 0;
}
```

8. Support polymorphism in C when using function pointers or pointers to structs.
   Example: Achieving polymorphism using function pointers in C

```c
typedef void (*SpeakFunc)();
struct Animal { SpeakFunc speak; };
void dogSpeak() { printf("Woof!\n"); }
void catSpeak() { printf("Meow!\n"); }
int main() {
      struct Animal dog = {dogSpeak};
      struct Animal cat = {catSpeak};
      dog.speak();          // Output: Woof!
      cat.speak();          // Output: Meow!
      return 0;
}
```

   Explanation: This example demonstrates a form of polymorphism in C using function pointers within a struct.

Each Animal struct can have a different speak function, allowing different behaviors at runtime.

9. Improve performance by avoiding expensive copying of large data structures.
   Example: Passing a large structure by pointer to avoid expensive copying

```c
struct LargeData {
    int data[10000];
};

void processLargeData(struct LargeData *ptr) {
    ptr->data[0] = 42;          // Modify data directly
}

int main() {
    struct LargeData big;
    processLargeData(&big);       // Pass by pointer, not by value
    printf("First element: %d\n", big.data[0]);
    return 0;
}
```

   Passing a pointer to a large structure avoids copying its contents, making the program faster and more memory efficient.

10. Hardware-level programming or systems programming to directly access specific memory addresses.
    Example: Accessing a specific hardware memory address (system programming)

```c
volatile unsigned int *reg = (unsigned int *)0x40021000;
*reg = 0x1;                // Write value to register
unsigned int value = *reg; // Read value from register
```

    Explanation: In embedded or systems programming, pointers

are often used to access specific memory-mapped hardware registers.

2.3 Pointer Arithmetic

Pointers can be incremented or decremented to point to different memory locations. Pointer arithmetic allows one to perform operations on pointers, such as incrementing or decrementing them. When a pointer is incremented, it moves to the next memory location of the type it points to.

Example:

```
ptr++;
```

This moves the pointer to the next integer location in memory.

For example, if you have a pointer to an integer and increment it, the pointer advances to the next integer in memory (typically 4 bytes ahead on most systems).

Example:

```
#include <stdio.h>
void main() {

    int nums[] = {10, 20, 30};
    int *ptr = nums;

    printf("First value: %d\n", *ptr);        // 10
    printf("Second value: %d\n", *(ptr + 1)); // 20
    printf("Third value: %d\n", *(ptr + 2));  // 30

return 0;

}
```

A null pointer is a pointer that does not point to any valid memory location. You can assign a pointer to NULL to indicate that it is not currently pointing to anything: **Example:**

```
1 *ptr = NULL;
```

2.4 Pointers and Arrays

In C, arrays and pointers are closely related. The name of an array acts as a pointer to its first element. For example:
Example:

```
1 int arr[5] = {1, 2, 3, 4, 5};
2 int *ptr = arr;
```

Arrays and pointers are closely related in C. The name of an array is essentially a constant pointer to its first element. For example:

```
1 int arr[5] = {1, 2, 3, 4, 5};
2 int *ptr = arr;
```

Here, ptr points to the first element of arr. You can use pointer arithmetic to access array elements:

```
1 int second = *(ptr + 1);          // second = 2
```

Passing Arrays to Functions:

When you pass an array to a function, what is actually passed is a pointer to its first element:

```
1 void printArray(int *arr, int size) {
2       for (int i = 0; i < size; i++) {
3             printf("%d ", arr[i]);
4       }
5 }
```

Dynamic Arrays:

For arrays whose size is not known at compile time, you can use dynamic memory allocation with pointers:

```c
int *dynamicArr = (int*)malloc(10 * sizeof(int));
// Use dynamicArr[0] ... dynamicArr[9]
free(dynamicArr);
```

Summary:

Arrays provide a way to store and access multiple values of the same type efficiently. Pointers allow flexible manipulation of arrays, dynamic memory allocation, and efficient function parameter passing. Mastery of arrays and pointers is essential for effective C programming.

2.5 Arrays in C

An **array** in C is a collection of elements of the same data type stored in contiguous memory locations. Arrays provide a way to group related data together and access elements using an index. The index of the first element is always 0.

Declaration and Initialization:

To declare an array, specify the data type, array name, and the number of elements in square brackets:

```c
int numbers[5];
```

This creates an array of 5 integers. You can also initialize an array at the time of declaration:

```c
int numbers[5] = {1, 2, 3, 4, 5};
```

If you omit the size, the compiler will determine it from the initializer:

```c
float grades[] = {98.5, 87.0, 92.3};
```

Accessing Array Elements:

Elements are accessed using the index in square brackets:

```
1 numbers[0] = 10;            // Assign 10 to the first
element
2 int x = numbers[2];         // Get the third element
```

Iterating Over Arrays:

A for loop is commonly used to process all elements:

```
1 for (int i = 0; i < 5; i++) {
2      printf("Element %d: %d\n", i, numbers[i]);
3 }
```

Multidimensional Arrays:

C supports multidimensional arrays (e.g., matrices):

```
1 int matrix[2][3] = {
2      {1, 2, 3},
3      {4, 5, 6}
4 };
```

Arrays and Memory:

Arrays are stored in contiguous memory, which allows efficient access but also means their size must be known at compile time (for static arrays). The name of the array acts as a pointer to its first element.

Limitations:

- The size of a static array cannot be changed after declaration.
- No bounds checking is performed, so accessing out-of-bounds elements leads to undefined behavior.

2.6 Pointers to Pointers

You can have pointers that point to other pointers. This is called a pointer to a pointer.

A **pointer to a pointer** is a variable that stores the address of another pointer variable. This allows you to create multiple

levels of indirection, which is useful in scenarios such as dynamic memory allocation for multidimensional arrays, or when you want a function to modify the value of a pointer passed to it.

Declaration:

```
int **pptr;
```

Here, pptr is a pointer to a pointer to an integer.

Example:

```
int value = 10;
int *ptr = &value;            // ptr points to value
int **pptr = &ptr;            // pptr points to ptr
printf("%d\n", **pptr);         // Outputs 10
```

Explanation:

- value stores the integer 10.
- ptr stores the address of value.
- pptr stores the address of ptr.
- **pptr dereferences twice to access the value stored in value.

Pointers to pointers are especially useful for:

- Passing a pointer to a function and allowing the function to modify the original pointer.
- Working with dynamically allocated multidimensional arrays.

3 File I/O

In C, input and output operations are typically performed using the Standard Input Output library, which provides functions for reading from and writing to the console. The most commonly used functions for input and output are **printf** and **scanf**.

3.1 Output

If you are familiar with generating output in Java (I/O), then you will be pleasantly surprised by how output is generated in C. To display output in the C programming language, one would use the printf("") function. To read input in the C programming language, one would use the scanf() function.

Example:

```
1 // The following will print "Hello World!"
2 printf("Hello World!");
3
4 Hello World!
```

The printf() function is the primary way to display output in C. It allows you to print formatted text, numbers, and variables to the console. The function uses format specifiers to control how values are displayed. Some common format specifiers include:

- %d -- Print an integer (decimal)
- %f -- Print a floating-point number
- %c -- Print a single character
- %s -- Print a string
- %x -- Print an integer in hexadecimal
- %p -- Print a pointer (memory address)

Example:

```c
#include <stdio.h>
int main() {
    int i = 42;
    float f = 3.1415;
    char c = 'A';
    char str[] = "Hello";

    printf("Integer: %d\n", i);
    printf("Float: %f\n", f);
     printf("Char: %c\n", c);
     printf("String: %s\n", str);
     return 0;
}
```

Controlling Output Format:

You can control the width, precision, and alignment of output using modifiers in the format string. For example, %.2f prints a floating-point number with two decimal places, and %10d prints an integer in a field at least 10 characters wide.

Example:

```c
printf("Pi to 2 decimals: %.2f\n", 3.14159);
printf("Right aligned: %10d\n", 123);
printf("Left aligned: %-10d\n", 123);
```

Escape Sequences:

Special characters can be printed using escape sequences:

- \n -- Newline
- \t -- Tab
- \\ -- Backslash
- \" -- Double quote

Example:

```
1 printf("Line1\nLine2\n");
```

Summary: printf() is a versatile function for displaying all types of data in C. Mastery of format specifiers and escape sequences is essential for producing clear and well-formatted output.

3.2 Input

The C library contains several input functions, and `scanf()` is the most general of them. It is a favorable option amongst many due to its flexibility and the various formats it can handle.

Input is essential in the operation of many programs. The scanf function reads from the standard input stream (stdin) and scans that input according to the format provided. The format can be a simple constant string, or you can specify as follows:

1. %s - String
2. %d - Base-10
3. %c - Character
4. %f - Float
5. %li - Long

If the stdin is input from the keyboard then text is read in. This is because the keys generate text characters: letters, digits, and punctuation. For illustration, when one enters the integer 2014, they type the characters 2 0 1 and 4.

If you want to store that as a numerical value rather than as a string, your program has to convert the string char-by-char to a numerical value. This is the job of the **scanf** function. Similar to **printf()**, **scanf()** uses a control string followed by a list of arguments. A control string indicates the destination data types for the input stream of characters.

Example:

```c
#include <stdio.h>
int main() {
    char str[100];
    int i;

    printf("Enter a value :");
    scanf("%s %d", str, &i);

    printf("You entered: %s %d", str, i);

     return 0;
}
```

Explanation:

This program demonstrates how to use scanf() to read both a string and an integer from the user. The %s format specifier tells scanf() to expect a string, while %d expects an integer. The variables str and i are passed to scanf(), with &i providing the address where the integer input should be stored. After reading the input, the program prints the values using printf().

- **Note:** When using %s, scanf() reads input until the first whitespace character (space, tab, or newline). To read a full line of text (including spaces), consider using fgets() instead.
- Always ensure that the input buffer is large enough to store the expected input to avoid buffer overflows.

Example Character:

```c
#include <stdio.h>
int main() {
    char c;
    float f;
    long l;

    printf("Enter a char, a float, and a long int:");
    scanf("%c %f %li", &c, &f, &l);

    printf("You entered: %c, %f, %li", c, f, l);

    return 0;
}
```

4 Data Types and Variables

C Supports man types of variables and each type of ariable is used for storing a specific kind of data. There is no string data type, nor is there a string object. A string is a character array. That is to say, in C a **string** is literally an array of characters. The difference between the types is in the amount of memory they occupy, and the range of values they can hold. The amount of storage that is allocated to store a particular type of data depends on the computer you are running (machine-dependent). An integer might take up 32 bits on your computer, or it might be stored in 64 bits.

For example, the size of an int or a long may vary between different systems or compilers. This means that a program compiled on one machine may use a different amount of memory for the same data type than when compiled on another machine. As a result, it is important to be aware of these differences when writing portable C code, and to use standard types (such as those defined in **<stdint.h>**) when exact sizes are required.
The following are some examples of basic data types in C:

1. Int - integer
2. Float - floating point
3. Double - long floating point
4. _Bool - true/false value
5. Enum - Enumeration
6. Char - character

4.1 Integer

A variable of type **int** can be used to contain integral values. If a minus sign precedes the integer, it negates the value. That is to say, it makes the number that follows minus sign a negative number. The **int** type sets aside space for whole numbers. It must be an integer which can be **positive**, **negative**, or **zero**. Variables of type int are either 2 bytes or 4 bytes.

If it is the case that an integer is preceded by a szero and the letter x, the value is taken as being expressed in hexadecimal (Base-16) notation.
Example:

```
1 int rgbColor = 0xFFEF0D;
```

This declares an integer variable **rgbColor** and assigns it the hexadecimal value **0xFFEF0D**. Again, in the C Programming Language hexadecimal numbers are prefixed with **0x**.

The values 1908, -100, and 0 are *all valid* examples of integer constants. An integer constant is a whole number value written directly in the code, without any fractional or decimal part. Integer constants can be positive, negative, or zero, and are typically written in decimal (e.g., 42), hexadecimal (e.g., 0x2A), or octal (e.g., 052) notation.

NOTE: Values cannot be expressed using commas. For illustration, the number 10,000 would have to be written as 10000.

C offers many other integer types, providing for more flexibility when programming. This is especially useful when precision is necessary. Moreover, C offers **three adjective** keywords to modify the basic integer types:

- `short`
- `long`
- `unsigned`

Short types use lesser storage than type long, ultimately saving space. This can be particularly useful when larger numbers are not needed. Long types use more storage, but serve their own purpose in providing the allocation of larger numbers when needed.

4.2 Floating-Point

A variable of type **float** can be used for storing floating-point numbers. That is to say, numerical values that contain decimal

places (fractions) are of the type float. "Float in C is used to represent real numbers with decimal points (prepbytes.com)." **Float** is utilized to store single-precision floating point numbers. The values 4.0, 199.0, and -.111 are all sound examples of floating-point constants and would qualify for variable assignment.

Example:

```
1// The variable `floatVariable` stores the value 3.7
2 float floatVariable =  3.70;
```

Furthermore, it is permissible to express floating-point constants in scientific notation.

Example:

```
1 float myFloat =  2.5e4;
2 // This represents the value 2.5 * 10^4
```

NOTE: Floating-point numbers can store decimal values with precision up to 6-7 decimal places. For numbers that require greater precision a double might suffice. This is owing to the fact that it has twice the precision of a floating point number.

4.3 Double

The **Double** type is the same as type **Float**, but with roughtly twice the precision. Doubles are used whenever the range provided by a float variable is insufficient. It can store twice as many significant digits. This is important as the allowance of twice as many significant digits increases precision, and overall accuracy.

Example:

```
1 double applePi =  3.141592653589;
2 // `applePi` stores the value 3.141592653589
```

Moreover, most computers represent double values using 64 bits.

NOTE: All floating-point constants are taken as double values by the C compiler. To explicitly express a floating constant, append either an **f** or **F** to the end of the number.

Example:

```c
float myFloat = 12.5f;
// This is a float constant
```

The same type specifiers used for integers can also be applied to doubles. **Aside:** A long double constant is written as a floating constant with the letter `l` or `L` immediately following.

- `short`
- `long`
- `unsigned`

Example:

```c
long double gradePoint = 3.88;
```

4.4 Boolean

The _Bool data type can be used to store just the values 0 or 1. Boolean values are used for indicating an **on/off**, **yes/no**, or **true/false** condition. This data type provides binary choices only.

The **Bool** data type in C is represented using the keyword _Bool (introduced in C99). To use the more familiar bool, true, and false keywords, include the <stdbool.h> header.

Example:

```c
#include <stdbool.h>
bool isActive =  true;
bool isFinished =  false;
```

true is equivalent to 1, and false is equivalent to 0. You can

use boolean variables in conditional statements and logical operations.

Example with _Bool:

```
1 _Bool flag = 1;
2 if (flag) { /* ... */ }
```

_Bool variables are used in programs that need to indicate a Boolean condition. For example, a variable of this type might be used to indicate whether all data was read from a file.
Example:

```
1 // 0 is used to indicate a false value
2 // 1 is used to indicate a true value
3 _Bool myBool = 0;
```

4.5 Enumeration

Enum is short for enumeration. One might enumerate to access an individual element in the `**enum**` itself (**enumeration**). Enum is a "data type that allows a programmer to define a variable and specify the valid values that could be stored into that variable (McGoff, J. 2024)." // To create an **enum** use the enum keyword,

followed by the name of the enum, and seperate the enum items with a comma:

```
1 #include <stdio.h>.
2 int main() {
3     enum GameLevel {
4         EASY,
5         MEDIUM,
6         HARD
7     };
8
9     return 0;
10 }
```

NOTE: Not dissimilar to JavaScript Object Notation, the last element *_does not require* a comma after declaration opf the value.

Enum declaration and definition process is as follows:

1. Define the data type (**enum** in this case)
2. Provide a name for the enumerated data type.
3. Provide an opening curly brace `{`
4. List all of the permissible values
5. Provide a closing curly brace `}` followed by a semicolon

The compiler actually treats enumeration identifiers as integer constants. If you want to have a specific integer value associated with an enumeration identifier, the integer can be assined to the identifier when the datat type is defined.

4.6 Character

Characters represent a single character such as the letter `A`, the digit character `6`, or a semicolon (;). Again, in C there are no `string` objects, instead one would use an array of characters. Character literals use single quotes such as `A` or `Z`. You can also declare variables of type char to be unsigned. This can be used to explicitly tell the compiler that a particular variable is a signed quantity.

Characters in C are stored using the `char` data type, which typically occupies 1 byte (8 bits) of memory. This allows it to represent 256 different values, which is sufficient for the standard ASCII character set. Each character is internally represented by its corresponding integer ASCII value.

Example:

```
1 char letter = 'A';
2 char digit = '7';
3 char symbol = '$';
```

You can use the printf() function with the %c format specifier to display characters, and with %d to display their ASCII values.

Example:

```
1 printf("Character: %c, ASCII: %d\n", letter, letter);
```

Character Arrays (Strings):

Although C does not have a built-in string type, you can create strings using arrays of characters terminated by the null character `\0`.

Example:

```
1 char greeting[] = "Hello";
```

This creates a character array with the contents `H`, `e`, `l`, `l`, `o`, `\0`.

Escape Sequences:

Special characters can be represented using escape sequences, such as:

- \n -- Newline
- \t -- Tab
- \' -- Single quote
- \" -- Double quote
- \0 -- Null character (string terminator)

Example:

```
1 char newline = '\n';
```

Following are both valid and invalid ways to declare a define a variable of type `char`.

Example

```c
#include <stdio.h>.
int main() {
    char broiled;               // Here we declare variable `broiled` to be of type char

    broiled = 'T';               // This is a valid definition
    broiled = T;                   // This definition is invalid (compiler assumes T is a variable)
    broiled = "T";               // This definition is invalid (compiler assumes "T" is a string)

    return 0;
}
```

Summary: Characters are fundamental in C for representing textual data, and understanding how to manipulate both individual characters and arrays of characters (strings) is essential for effective C programming.

Escape Characters: C contains special characters that represent actions such as:

- Backspacing
- Going to the next line
- Making the terrminal bell ring (or speaker beep)

We can represent these actions by using special sequences called **escape sequences**. Escape sequences must be enclosed in *single quotes* when assigned to a variable of type char.

- '\a' → Alert (ANSI C).
- '\b' → Backspace
- '\f' → Form feed
- '\n' → Newline
- '\r' → Carriage return
- '\t' → Horizontal tab

- '\\' → Backslash
- '\'' → Single quote
- '\?' → Question mark (?).

Following as an example of declaring a variable of type char and assigning an escape sequence as the value.

Example:

```c
#include <stdio.h>.
int main() {
    char x = '\n';

    return 0;
}
```

5 Operators

Operators in C programming are symbols that specify the type of operatoion to be performed on operands. They are categorized into several types. Operators are fundamental building blocks in C, allowing developers to manipulate data and variables to achieve desired outcomes. these operators enable a wide range of operations and control structures within C programs.

NOTE: understanding these operators is crucial for writing efficient and effective C code.

Following are several types for which operators are categorized:

5.1 Arithmetic operators:

Perform basic mathematical operations like addition, subtraction, muliplication, division, and modulus.

```c
int a = 10, b = 3;
int sum = a + b;          // Addition (13)
int diff = a - b;         // Subtraction (7)
int prod = a * b;         // Multiplication (30)
int quot = a / b;          // Division (3)
int mod = a % b;           // Modulus (1)
```

5.2 Rational Operators:

Compare two values and determine the relational relationship such as equal to, not equal to, greater than, and less than.

```c
int x = 5, y = 10;
if (x < y) {
      printf("x is less than y\n");
}
```

5.3 Bitwise Operators:

Perform operationss on binary representations of data such as **AND**, **OR**, **XOR**, **NOT**, **left shift**, and **right shift**.

```c
int a = 5;       // 0101 in binary
int b = 3;       // 0011 in binary
int andResult = a & b;    // Bitwise AND (1)
int orResult = a | b;     // Bitwise OR (7)
int xorResult = a ^ b;    // Bitwise XOR (6)
int notResult = ~a;       // Bitwise NOT (depends on system)
int leftShift = a << 1;   // Left shift (10)
int rightShift = a >> 1;  // Right shift (2)
```

5.4 Logical Operators

: Used to combine conditional statements, including **AND**, **OR**, and **NOT**.

```c
int a = 1, b = 0;
if (a && b) {
      printf("Both are true\n");
}
if (a || b) {
      printf("At least one is true\n");
}
if (!b) {
      printf("b is false\n");
}
```

5.5 Assignment Operators:

Assign values to variables, with variations that combine assignment with arithmetic or bitwise operations.

```c
int x = 5;                // Simple assignment
x += 3;                   // x = x + 3 (x becomes 8)
x -= 2;                   // x = x - 2 (x becomes 6)
x *= 4;                   // x = x * 4 (x becomes 24)
x /= 6;                   // x = x / 6 (x becomes 4)
x %= 3;                   // x = x % 3 (x becomes 1)
```

5.6 Increment and Decrement Operators:

Increase or decrease a variables value by one.

```
int count = 10;
count++;                    // Increment (count becomes 11)
count--;                    // Decrement (count becomes 10)
++count;                    // Pre-increment (count becomes 11)
--count;                    // Pre-decrement (count becomes 10)
```

5.7 Conditional Operators:

Allows multiple expressions to be evaluated in a single statement.

```
int a = 10, b = 20;
int max = (a > b) ? a : b;          // max is 20
printf("The maximum value is: %d\n", max);
```

5.8 Sizeof Operator:

Returns the size of a data type or a variable

```
int num;
printf("Size of int: %zu bytes\n", sizeof(num));
printf("Size of double: %zu bytes\n", sizeof(double));
```

5.9 Pointer Operators:

These are used to reference and dereference pointers (address-of and indirection operators).

```c
int value = 42;
int *ptr = &value;            // & is the address-of
operator
int result = *ptr;                // * is the
dereference operator
printf("Value: %d, Address: %p\n", result, ptr);
```

Explanation: The & operator obtains the address of a variable, and the * operator accesses the value at a given address (dereferencing the pointer).

5.10 Member Access Operators:

Access memebers of structuress and unions, including dot and arrow operators.

```c
#include <stdio.h>
struct Person {
      char name[20];
      int age;
};

int main() {
      struct Person p1;
      strcpy(p1.name, "Alice");
      p1.age = 30;
      struct Person *ptr = &p1;
      printf("Name: %s, Age: %d\n", ptr->name,
ptr->age);
      return 0;
}
```

Explanation: The . operator accesses members of a struct variable (e.g., p1.age), while the -> operator accesses members via a pointer to a struct (e.g., ptr->name).