C-LANGUAGE

VS-Code

bonsoninstitutetechnology@.com

PROGRAMMING LANGUAGE

Comprehensive C Language Topics

1. Introduction to C Programming

- History of C Features of, Structure of a C program, Compilation process

2. Basic Syntax and Data Types

- Keywords Identifier , Data types (`int`, `char`, `float`, `double`)
- Constants and Literals , Variable declaration and initialization

3. Operators and Expressions

- Arithmetic operators, Relational operators, Logical operators, Bitwise operators
- Assignment operators, (Shifting operator)

4. Control Flow Statements

- Conditional statements
  - `if`, `if-else`, `nested if`, `switch`
- Looping statements
  - `for`, `while`, `do-while`
- Jump statements
  - `break`, `continue`, `goto`

5. Functions & C Recursion

- Function declaration and definition
- Function call
- Return values and `void`
- Parameter passing
  - Call by value
  - Call by reference
- Inline functions
- Recursion

6. Arrays and Strings and Pointer and Storage Class & Macros

- Array declaration, initialization, and access
- Structure in C
- Multidimensional arrays
- Strings (String manipulation functions from `<string.h>`)
- Memory Allocation in C & Macros
- Pointer

C- LANGUAGE

C, computer programming language developed in the early 1970 by American computer scientist Dennis M. Ritchie at Bell Laboratories (formerly AT&T Bell Laboratories).

The *current latest version* of the C language is C17, which was released in June 2018. No new features were added to this version only a few basic corrections were made and some bugs in the C11 version were fixed.

A new standard revision of C programming language is expected in 2023 which will be called C23 or C2x.

Programming Language

As we know, to communicate with a person, we need a specific language, similarly to communicate with computers, programmers also need a language is called Programming language.

Before learning the programming language, let's understand what is language?

What is Language?

Language is a mode of communication that is used to **share ideas, opinions with each other**. For example, if we want to teach someone, we need a language that is understandable by both communicators

What is a Programming Language?

A programming language is a **computer language** that is used by **programmers (developers) to communicate with computers**. It is a set of instructions written in any specific language

(C, C++, Java, Python) to perform a specific task.

A programming language is mainly used to **develop desktop applications, websites, and mobile applications**.

Types of programming language

1. Low-level programming language

Low-level language is **machine-dependent (0s and 1s)** programming language. The processor runs low- level programs directly without the need of a compiler or interpreter, so the programs written in low-level language can be run very fast.

- ➔ **Low-level language is further divided into two parts-**

I. Machine Language

Machine language is a type of low-level programming language. It is also called as **machine code or object code**. Machine language is easier to read because it is normally displayed in binary or hexadecimal form (base 16) form. It does not require a translator to convert the programs because computers directly understand the machine language programs. The advantage of machine language is that it helps the programmer to execute the programs faster than the high-level programming language.

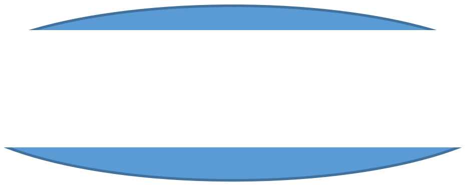
ii. Assembly Language

Assembly language (ASM) is also a type of low-level programming language that is designed for specific processors. It represents the set of instructions in a **symbolic and human-understandable form**. It uses an assembler to convert the assembly language to machine language.

2. High-level programming language

High-level programming language (HLL) is designed for **developing user-friendly software programs and websites**. This programming language requires a compiler or interpreter to translate the program into machine language (execute the program).

High-level programming language includes **Python, Java, JavaScript, PHP, C#, C++, Objective C, Cobol , Perl, Pascal, LISP, FORTRAN, and Swift programming language**.



COMPUTER LANGUAGES

Low Level Language (Machine Language): -> use1's & 0's to create instruction. (BINARY L)

Middle Level Language (Assembly Language): -> use mnemonics to create instructions .

High Level Language (HLL): -> Similar to Human Language .

➢ First C Program

```
#include <stdio.h>
 int main() {
 printf("Hello C Language");
```

```
        return 0; }
```

#include <stdio.h> includes the **standard input output** library functions. The printf() function is defined in stdio.h .   # define the preprocessor.

int main() The **main() function is the entry point of every program** in c language.

printf() The printf() function is **used to print data** on the console.

return 0 The return 0 statement, returns execution status to the OS. The 0 value is used for successful execution and 1 for unsuccessful execution.

Basically, every C program has **two** major sections.

- Header section
- Main section

```
#include<stdio.h>          //header section

int main()                //main section
{
    return 0;
}
```

Your First C Program

In the previous tutorial you learnt how to install C on your computer. Now, let's learn how to write a simple C program.

We will write a simple program that displays `Hello, World!` on the screen.

```
#include <stdio.h>

int main() {

    printf("Hello, World!");

    return 0;
}
```
Run Code

Output

```
Hello World!
```

Basic Structure of a C Program

As we have seen from the last example, a C program requires a lot of lines even for a simple program.

For now, just remember that every C program we will write will follow this structure:

```
#include <stdio.h>
int main() {
  // your code
  return 0;
}
```

And, we will write out code inside curly braces `{}`.

C Comments

In the previous tutorial you learned to write your first C program. Now, let's learn about C comments.

> **Tip**: We are introducing comments early in this tutorial series because, from now on, we will be using them to explain our code.

Comments are hints that we add to our code, making it easier to understand.

Comments are completely ignored by C compilers.

For example,

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

```
int main() {

  // print Hello World to the screen
  printf("Hello World");
  return 0;
}
```
Run Code

Output

```
Hello World
```

Here, `// print Hello World to the screen` is a comment in C programming. The C compiler ignores everything after the `//` symbol.

Note: You can ignore the programming concepts and simply focus on the **comments**. We will revisit these concepts in later tutorials.

Single-line Comments in C

In C, a single line comment starts with `//` symbol. It starts and ends in the same line. For example,

```
#include <stdio.h>

int main() {

  // create integer variable
  int age = 25;

  // print the age variable
  printf("Age: %d", age);

  return 0;
}
```
Run Code

Output

```
Age: 25
```

In the above example, we have used two single-line comments:

- `// create integer variable`
- `// print the age variable`

We can also use the single line comment along with the code.

```
int age = 25;  // create integer variable
```

Here, code before `//` are executed and code after `//` are ignored by the compiler.

Multi-line Comments in C

In C programming, there is another type of comment that allows us to comment on multiple lines at once, they are multi-line comments.

To write multi-line comments, we use the `/*....*/` symbol. For example,

```
/* This program takes age input from the user
It stores it in the age variable
And, print the value using printf() */

#include <stdio.h>

int main() {

  // create integer variable
  int age = 25;

  // print the age variable
  printf("Age: %d", age);

  return 0;
}
```

Run Code

Output

```
Age: 25
```

In this type of comment, the C compiler ignores everything from `/*` to `*/`.

Note: Remember the keyboard shortcut to use comments:

- Single Line comment: ctrl + / (windows) and cmd + / (mac)
- Multi line comment: ctrl + shift + / (windows) and cmd + shift + / (mac)

Prevent Executing Code Using Comments

While debugging there might be situations where we don't want some part of the code. For example,

In the program below, suppose we don't need data related to height. So, instead of removing the code related to height, we can simply convert them into comments.

```
#include <stdio.h>

int main() {
    int number1 = 10;
    int number2 = 15;
    int sum = number1 + number2;

    printf("The sum is: %d\n", sum);
    printf("The product is: %d\n", product);
    return 0;
}
```
Run Code

Here, the code throws an error because we have not defined a `product` variable.

We can comment out the code that's causing the error.

For example,

```
#include <stdio.h>

int main() {
    int number1 = 10;
    int number2 = 15;
    int sum = number1 + number2;

    printf("The sum is: %d\n", sum);
    // printf("The product is: %d\n", product);

    return 0;
}
```
Run Code

Now, the code runs without any errors.

Here, we have resolved the error by commenting out the code related to the product.

If we need to calculate the product in the near future, we can uncomment it.

C Keywords and Identifiers

Character set

A character set is a set of alphabets, letters and some special characters that are valid in C language.

Alphabets

```
Uppercase: A B C ................................. X Y Z

Lowercase: a b c .................................... x y z
```

C accepts both lowercase and uppercase alphabets as variables and functions.

Digits

```
0 1 2 3 4 5 6 7 8 9
```

Special Characters

,	<	>	.	_
(	)	;	$	:

%	[	]	#	?
'	&	{	}	"
^	!	*	/	|
-	\	~	+	

White space Characters
Blank space, newline, horizontal tab, carriage return and form feed.

C Keywords

Keywords are predefined, reserved words used in programming that have special meanings to the compiler. Keywords are part of the syntax and they cannot be used as an identifier. For example:

```
int money;
```

Here, `int` is a keyword that indicates `money` is a variable of type `int` (integer).
As C is a case sensitive language, all keywords must be written in lowercase. Here is a list of all keywords allowed in ANSI C.

C Keywords

`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`continue`	`for`	`signed`	`void`
`do`	`if`	`static`	`while`
`default`	`goto`	`sizeof`	`volatile`
`const`	`float`	`short`	`unsigned`

All these keywords, their syntax, and application will be discussed in their respective topics. However, if you want a brief overview of these keywords without going further, visit List of all keywords in C programming.

C Identifiers

Identifier refers to name given to entities such as variables, functions, structures etc.

Identifiers must be unique. They are created to give a unique name to an entity to identify it during the execution of the program. For example:

```c
int money;
double accountBalance;
```

Here, `money` and `accountBalance` are identifiers.
Also remember, identifier names must be different from keywords. You cannot use `int` as an identifier because `int` is a keyword.

Rules for naming identifiers

1. A valid identifier can have letters (both uppercase and lowercase letters), digits and underscores.
2. The first letter of an identifier should be either a letter or an underscore.
3. You cannot use keywords like `int`, `while` etc. as identifiers.
4. There is no rule on how long an identifier can be. However, you may run into problems in some compilers if the identifier is longer than 31 characters.

You can choose any name as an identifier if you follow the above rule, however, give meaningful names to identifiers that make sense.

C Variables, Constants and Literals

Variables

In programming, a variable is a container (storage area) to hold data.

To indicate the storage area, each variable should be given a unique name (identifier). Variable names are just the symbolic representation of a memory location. For example:

```c
int playerScore = 95;
```

Here, `playerScore` is a variable of `int` type. Here, the variable is assigned an integer value `95`.
The value of a variable can be changed, hence the name variable.

```c
char ch = 'a';
// some code
ch = 'l';
```

Rules for naming a variable

1. A variable name can only have letters (both uppercase and lowercase letters), digits and underscore.
2. The first letter of a variable should be either a letter or an underscore.
3. There is no rule on how long a variable name (identifier) can be. However, you may run into problems in some compilers if the variable name is longer than 31 characters.

Note: You should always try to give meaningful names to variables. For example: `firstName` is a better variable name than `fn`.

C is a strongly typed language. This means that the variable type cannot be changed once it is declared. For example:

```
int number = 5;      // integer variable
number = 5.5;        // error
double number;       // error
```

Here, the type of `number` variable is `int`. You cannot assign a floating-point (decimal) value `5.5` to this variable. Also, you cannot redefine the data type of the variable to `double`. By the way, to store the decimal values in C, you need to declare its type to either `double` or `float`.
Visit this page to learn more about different types of data a variable can store.

Literals

Literals are data used for representing fixed values. They can be used directly in the code. For example: `1`, `2.5`, `'c'` etc.
Here, `1`, `2.5` and `'c'` are literals. Why? You cannot assign different values to these terms.

1. Integers

An integer is a numeric literal(associated with numbers) without any fractional or exponential part. There are three types of integer literals in C programming:

- decimal (base 10)
- octal (base 8)
- hexadecimal (base 16)

For example:

```
Decimal: 0, -9, 22 etc

Octal: 021, 077, 033 etc

Hexadecimal: 0x7f, 0x2a, 0x521 etc
```

In C programming, octal starts with a `0`, and hexadecimal starts with a `0x`.

2. Floating-point Literals

A floating-point literal is a numeric literal that has either a fractional form or an exponent form. For example:

```
-2.0

0.0000234

-0.22E-5
```

Note: `E-5 = 10-5`

3. Characters

A character literal is created by enclosing a single character inside single quotation marks. For example: `'a'`, `'m'`, `'F'`, `'2'`, `'}'` etc.

4. Escape Sequences

Sometimes, it is necessary to use characters that cannot be typed or has special meaning in C programming. For example: newline(enter), tab, question mark etc.

In order to use these characters, escape sequences are used.

Escape Sequences

Escape Sequences	Character
`\b`	Backspace
`\f`	Form feed
`\n`	Newline
`\r`	Return
`\t`	Horizontal tab
`\v`	Vertical tab
`\\`	Backslash
`\'`	Single quotation mark
`\"`	Double quotation mark
`\?`	Question mark
`\0`	Null character

For example: `\n` is used for a newline. The backslash `\` causes escape from the normal way the characters are handled by the compiler.

5. String Literals

A string literal is a sequence of characters enclosed in double-quote marks. For example:

```
"good"                  //string constant
```

```
""                      //null string constant

"      "                //string constant of six white space

"x"                     //string constant having a single character.

"Earth is round\n"          //prints string with a newline
```

Constants

If you want to define a variable whose value cannot be changed, you can use the `const` keyword. This will create a constant. For example,

```
const double PI = 3.14;
```

Notice, we have added keyword `const`.
Here, `PI` is a symbolic constant; its value cannot be changed.

```
const double PI = 3.14;
PI = 2.9; //Error
```

C Data Types

In C programming, data types are declarations for variables. This determines the type and size of data associated with variables. For example,

```
int myVar;
```

Here, `myVar` is a variable of `int` (integer) type. The size of `int` is 4 bytes.

Basic types

Here's a table containing commonly used types in C programming for quick access.

Type	Size (bytes)	Format Specifier
`int`	at least 2, usually 4	`%d`, `%i`
`char`	1	`%c`
`float`	4	`%f`
`double`	8	`%lf`

Type	Size (bytes)	Format Specifier
`short int`	2 usually	`%hd`
`unsigned int`	at least 2, usually 4	`%u`
`long int`	at least 4, usually 8	`%ld`, `%li`
`long long int`	at least 8	`%lld`, `%lli`
`unsigned long int`	at least 4	`%lu`
`unsigned long long int`	at least 8	`%llu`
`signed char`	1	`%c`
`unsigned char`	1	`%c`
`long double`	at least 10, usually 12 or 16	`%Lf`

int

Integers are whole numbers that can have both zero, positive and negative values but no decimal values. For example, `0`, `-5`, `10`
We can use `int` for declaring an integer variable.

```
int id;
```

Here, `id` is a variable of type integer.
You can declare multiple variables at once in C programming. For example,

```
int id, age;
```

The size of `int` is usually 4 bytes (32 bits). And, it can take 2^{32} distinct states from `-2147483648` to `2147483647`.

float and double

`float` and `double` are used to hold real numbers.

```
float salary;
```

```
double price;
```

In C, floating-point numbers can also be represented in exponential. For example,

```
float normalizationFactor = 22.442e2;
```

What's the difference between `float` and `double`?
The size of `float` (single precision float data type) is 4 bytes. And the size of `double` (double precision float data type) is 8 bytes.

char

Keyword `char` is used for declaring character type variables. For example,

```
char test = 'h';
```

The size of the character variable is 1 byte.

void

`void` is an incomplete type. It means "nothing" or "no type". You can think of void as **absent**.
For example, if a function is not returning anything, its return type should be `void`.
Note that, you cannot create variables of `void` type.

short and long

If you need to use a large number, you can use a type specifier `long`. Here's how:

```
long a;
long long b;
long double c;
```

Here variables `a` and `b` can store integer values. And, `c` can store a floating-point number.
If you are sure, only a small integer (`[−32,767, +32,767]` range) will be used, you can use `short`.

```
short d;
```

You can always check the size of a variable using the `sizeof()` operator.

```
#include <stdio.h>
int main() {
  short a;
  long b;
  long long c;
  long double d;

  printf("size of short = %d bytes\n", sizeof(a));
  printf("size of long = %d bytes\n", sizeof(b));
  printf("size of long long = %d bytes\n", sizeof(c));
  printf("size of long double= %d bytes\n", sizeof(d));
  return 0;
}
```

Run Code

signed and unsigned

In C, `signed` and `unsigned` are type modifiers. You can alter the data storage of a data type by using them:

- `signed` - allows for storage of both positive and negative numbers

- `unsigned` - allows for storage of only positive numbers
  For example,

```
// valid codes
unsigned int x = 35;
int y = -35;  // signed int
int z = 36;  // signed int

// invalid code: unsigned int cannot hold negative integers
unsigned int num = -35;
```

Here, the variables `x` and `num` can hold only zero and positive values because we have used the `unsigned` modifier.
Considering the size of `int` is 4 bytes, variable `y` can hold values from -2^{31} to $2^{31}-1$, whereas variable `x` can hold values from `0` to $2^{32}-1$.

Derived Data Types

Data types that are derived from fundamental data types are derived types. For example: arrays, pointers, function types, structures, etc.

We will learn about these derived data types in later tutorials.

- bool type
- Enumerated type
- Complex types

C Input Output (I/O)

C Output

In C programming, `printf()` is one of the main output function. The function sends formatted output to the screen. For example,

Example 1: C Output

```
#include <stdio.h>
int main()
{
    // Displays the string inside quotations
    printf("C Programming");
    return 0;
}
```
Run Code

Output

```
C Programming
```

How does this program work?

- All valid C programs must contain the `main()` function. The code execution begins from the start of the `main()` function.

- The `printf()` is a library function to send formatted output to the screen. The function prints the string inside quotations.
- To use `printf()` in our program, we need to include `stdio.h` header file using the `#include <stdio.h>` statement.
- The `return 0;` statement inside the `main()` function is the "Exit status" of the program. It's optional.

Example 2: Integer Output

```
#include <stdio.h>
int main()
{
    int testInteger = 5;
    printf("Number = %d", testInteger);
    return 0;
}
```
Run Code

Output

```
Number = 5
```

We use `%d` format specifier to print `int` types. Here, the `%d` inside the quotations will be replaced by the value of `testInteger`.

Example 3: float and double Output

```
#include <stdio.h>
int main()
{
    float number1 = 13.5;
    double number2 = 12.4;

    printf("number1 = %f\n", number1);
    printf("number2 = %lf", number2);
    return 0;
}
```
Run Code

Output

```
number1 = 13.500000
number2 = 12.400000
```

To print `float`, we use `%f` format specifier. Similarly, we use `%lf` to print `double` values.

Example 4: Print Characters

```
#include <stdio.h>
int main()
{
    char chr = 'a';
    printf("character = %c", chr);
    return 0;
}
```
Run Code

Output

```
character = a
```

To print `char`, we use `%c` format specifier.

C Input

In C programming, `scanf()` is one of the commonly used function to take input from the user. The `scanf()` function reads formatted input from the standard input such as keyboards.

Example 5: Integer Input/Output

```
#include <stdio.h>
int main()
{
    int testInteger;
    printf("Enter an integer: ");
    scanf("%d", &testInteger);
    printf("Number = %d",testInteger);
    return 0;
}
```
Run Code

Output

```
Enter an integer: 4
Number = 4
```

Here, we have used `%d` format specifier inside the `scanf()` function to take `int` input from the user. When the user enters an integer, it is stored in the `testInteger` variable.

Notice, that we have used `&testInteger` inside `scanf()`. It is because `&testInteger` gets the address of `testInteger`, and the value entered by the user is stored in that address.

Example 6: Float and Double Input/Output

```
#include <stdio.h>
int main()
{
    float num1;
    double num2;

    printf("Enter a number: ");
    scanf("%f", &num1);
    printf("Enter another number: ");
    scanf("%lf", &num2);

    printf("num1 = %f\n", num1);
    printf("num2 = %lf", num2);

    return 0;
}
```
Run Code

Output

```
Enter a number: 12.523
Enter another number: 10.2
num1 = 12.523000
```

```
num2 = 10.200000
```

We use `%f` and `%lf` format specifier for `float` and `double` respectively.

Example 7: C Character I/O

```c
#include <stdio.h>
int main()
{
    char chr;
    printf("Enter a character: ");
    scanf("%c",&chr);
    printf("You entered %c.", chr);
    return 0;
}
```
Run Code

Output

```
Enter a character: g
You entered g
```

When a character is entered by the user in the above program, the character itself is not stored. Instead, an integer value (ASCII value) is stored.

And when we display that value using `%c` text format, the entered character is displayed. If we use `%d` to display the character, it's ASCII value is printed.

Example 8: ASCII Value

```c
#include <stdio.h>
int main()
{
    char chr;
    printf("Enter a character: ");
    scanf("%c", &chr);

    // When %c is used, a character is displayed
    printf("You entered %c.\n",chr);

    // When %d is used, ASCII value is displayed
    printf("ASCII value is %d.", chr);
    return 0;
}
```
Run Code

Output

```
Enter a character: g
You entered g.
ASCII value is 103.
```

I/O Multiple Values

Here's how you can take multiple inputs from the user and display them.

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

```
int main()
{
    int a;
    float b;

    printf("Enter integer and then a float: ");

    // Taking multiple inputs
    scanf("%d%f", &a, &b);

    printf("You entered %d and %f", a, b);
    return 0;
}
```

Run Code

Output

```
Enter integer and then a float: -3
3.4
You entered -3 and 3.400000
```

Format Specifiers for I/O

As you can see from the above examples, we use

- `%d` for `int`
- `%f` for `float`
- `%lf` for `double`
- `%c` for `char`

Here's a list of commonly used C data types and their format specifiers.

Data Type	Format Specifier
`int`	`%d`
`char`	`%c`
`float`	`%f`
`double`	`%lf`
`short int`	`%hd`
`unsigned int`	`%u`

Data Type	Format Specifier
`long int`	`%li`
`long long int`	`%lli`
`unsigned long int`	`%lu`
`unsigned long long int`	`%llu`
`signed char`	`%c`
`unsigned char`	`%c`
`long double`	`%Lf`

While debugging there might be situations where we don't want some part of the code. For example,

In the program below, suppose we don't need data related to height. So, instead of removing the code related to height, we can simply convert them into comments.

```c
// Program to take age and height input

#include <stdio.h>

int main() {

  int age;
  // double height;


  printf("Enter the age: ");
  scanf("%d", &age);

  // printf("Enter the height: ");
  // scanf("%lf", &height);


  printf("Age = %d", age);
  // printf("\nHeight = %.1lf", height);
  return 0;
}
```
Run Code

Now later on, if we need height again, all you need to do is remove the forward slashes. And, they will now become statements not comments.

C Programming Operators

An operator is a symbol that operates on a value or a variable. For example: `+` is an operator to perform addition.
C has a wide range of operators to perform various operations.

C Arithmetic Operators

An arithmetic operator performs mathematical operations such as addition, subtraction, multiplication, division etc on numerical values (constants and variables).

Operator	Meaning of Operator
+	addition or unary plus
-	subtraction or unary minus
*	multiplication
/	division
%	remainder after division (modulo division)

Example 1: Arithmetic Operators

```
// Working of arithmetic operators
#include <stdio.h>
int main()
{
    int a = 9,b = 4, c;

    c = a+b;
    printf("a+b = %d \n",c);
    c = a-b;
    printf("a-b = %d \n",c);
    c = a*b;
    printf("a*b = %d \n",c);
    c = a/b;
    printf("a/b = %d \n",c);
    c = a%b;
    printf("Remainder when a divided by b = %d \n",c);
    return 0;
}
```

Run Code

Output

```
a+b = 13
a-b = 5
a*b = 36
a/b = 2
Remainder when a divided by b=1
```

The operators `+`, `-` and `*` computes addition, subtraction, and multiplication respectively as you might have expected.
In normal calculation, `9/4 = 2.25`. However, the output is `2` in the program.
It is because both the variables `a` and b are integers. Hence, the output is also an integer. The compiler neglects the term after the decimal point and shows answer `2` instead of `2.25`.
The modulo operator `%` computes the remainder. When `a=9` is divided by `b=4`, the remainder is `1`.
The `%` operator can only be used with integers.
Suppose `a = 5.0`, `b = 2.0`, `c = 5` and `d = 2`. Then in C programming,

```
// Either one of the operands is a floating-point number

a/b = 2.5

a/d = 2.5

c/b = 2.5

// Both operands are integers

c/d = 2
```

C Increment and Decrement Operators

C programming has two operators increment `++` and decrement `--` to change the value of an operand (constant or variable) by 1.
Increment `++` increases the value by 1 whereas decrement `--` decreases the value by 1. These two operators are unary operators, meaning they only operate on a single operand.

Example 2: Increment and Decrement Operators

```
// Working of increment and decrement operators
#include <stdio.h>
int main()
{
    int a = 10, b = 100;
    float c = 10.5, d = 100.5;

    printf("++a = %d \n", ++a);
    printf("--b = %d \n", --b);
    printf("++c = %f \n", ++c);
    printf("--d = %f \n", --d);

    return 0;
}
```

Run Code

Output

```
++a = 11
```

```
--b = 99
++c = 11.500000
--d = 99.500000
```

Here, the operators `++` and `--` are used as prefixes. These two operators can also be used as postfixes like `a++` and `a--`. Visit this page to learn more about how increment and decrement operators work when used as postfix.

C Assignment Operators

An assignment operator is used for assigning a value to a variable. The most common assignment operator is `=`

Operator	Example	Same as
=	a = b	a = b
+=	a += b	a = a+b
-=	a -= b	a = a-b
*=	a *= b	a = a*b
/=	a /= b	a = a/b
%=	a %= b	a = a%b

Example 3: Assignment Operators

```
// Working of assignment operators
#include <stdio.h>
int main()
{
    int a = 5, c;

    c = a;      // c is 5
    printf("c = %d\n", c);
    c += a;     // c is 10
    printf("c = %d\n", c);
    c -= a;     // c is 5
    printf("c = %d\n", c);
    c *= a;     // c is 25
    printf("c = %d\n", c);
    c /= a;     // c is 5
    printf("c = %d\n", c);
    c %= a;     // c = 0
    printf("c = %d\n", c);

    return 0;
}
```
Run Code

Output

```
c = 5
c = 10
c = 5
c = 25
c = 5
c = 0
```

C Relational Operators

A relational operator checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0.

Relational operators are used in decision making and loops.

Operator	Meaning of Operator	Example
==	Equal to	`5 == 3` is evaluated to 0
>	Greater than	`5 > 3` is evaluated to 1
<	Less than	`5 < 3` is evaluated to 0
!=	Not equal to	`5 != 3` is evaluated to 1
>=	Greater than or equal to	`5 >= 3` is evaluated to 1
<=	Less than or equal to	`5 <= 3` is evaluated to 0

Example 4: Relational Operators

```
// Working of relational operators
#include <stdio.h>
int main()
{
    int a = 5, b = 5, c = 10;

    printf("%d == %d is %d \n", a, b, a == b);
    printf("%d == %d is %d \n", a, c, a == c);
    printf("%d > %d is %d \n", a, b, a > b);
    printf("%d > %d is %d \n", a, c, a > c);
    printf("%d < %d is %d \n", a, b, a < b);
    printf("%d < %d is %d \n", a, c, a < c);
    printf("%d != %d is %d \n", a, b, a != b);
    printf("%d != %d is %d \n", a, c, a != c);
    printf("%d >= %d is %d \n", a, b, a >= b);
    printf("%d >= %d is %d \n", a, c, a >= c);
    printf("%d <= %d is %d \n", a, b, a <= b);
    printf("%d <= %d is %d \n", a, c, a <= c);

    return 0;
}
```
Run Code

Output

```
5 == 5 is 1
5 == 10 is 0
5 > 5 is 0
5 > 10 is 0
5 < 5 is 0
5 < 10 is 1
5 != 5 is 0
5 != 10 is 1
5 >= 5 is 1
5 >= 10 is 0
5 <= 5 is 1
5 <= 10 is 1
```

C Logical Operators

An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false. Logical operators are commonly used in decision making in C programming.

Operator	Meaning	Example
&&	Logical AND. True only if all operands are true	If c = 5 and d = 2 then, expression `((c==5) && (d>5))` equals to 0.
||	Logical OR. True only if either one operand is true	If c = 5 and d = 2 then, expression `((c==5) || (d>5))` equals to 1.
!	Logical NOT. True only if the operand is 0	If c = 5 then, expression `!(c==5)` equals to 0.

Example 5: Logical Operators

```c
// Working of logical operators

#include <stdio.h>
int main()
{
    int a = 5, b = 5, c = 10, result;

    result = (a == b) && (c > b);
    printf("(a == b) && (c > b) is %d \n", result);

    result = (a == b) && (c < b);
    printf("(a == b) && (c < b) is %d \n", result);

    result = (a == b) || (c < b);
    printf("(a == b) || (c < b) is %d \n", result);

    result = (a != b) || (c < b);
    printf("(a != b) || (c < b) is %d \n", result);

    result = !(a != b);
    printf("!(a != b) is %d \n", result);

    result = !(a == b);
    printf("!(a == b) is %d \n", result);

    return 0;
}
```
Run Code

Output

```
(a == b) && (c > b) is 1
(a == b) && (c < b) is 0
(a == b) || (c < b) is 1
(a != b) || (c < b) is 0
!(a != b) is 1
!(a == b) is 0
```

Explanation of logical operator program

- `(a == b) && (c > 5)` evaluates to 1 because both operands `(a == b)` and `(c > b)` is 1 (true).
- `(a == b) && (c < b)` evaluates to 0 because operand `(c < b)` is 0 (false).
- `(a == b) || (c < b)` evaluates to 1 because `(a = b)` is 1 (true).
- `(a != b) || (c < b)` evaluates to 0 because both operand `(a != b)` and `(c < b)` are 0 (false).
- `!(a != b)` evaluates to 1 because operand `(a != b)` is 0 (false). Hence, !(a != b) is 1 (true).
- `!(a == b)` evaluates to 0 because `(a == b)` is 1 (true). Hence, `!(a == b)` is 0 (false).

C Bitwise Operators

During computation, mathematical operations like: addition, subtraction, multiplication, division, etc are converted to bit-level which makes processing faster and saves power.

Bitwise operators are used in C programming to perform bit-level operations.

Operators	Meaning of operators
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise complement
<<	Shift left
>>	Shift right

Visit bitwise operator in C to learn more.

Other Operators

Comma Operator

Comma operators are used to link related expressions together. For example:

```
int a, c = 5, d;
```

The sizeof operator

The `sizeof` is a unary operator that returns the size of data (constants, variables, array, structure, etc).

Bitwise Operators in C Programming

In the arithmetic-logic unit (which is within the CPU), mathematical operations like: addition, subtraction, multiplication and division are done in bit-level. To perform bit-level operations in C programming, bitwise operators are used.

Operators	Meaning of operators
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise complement
<<	Shift left
>>	Shift right

Bitwise AND Operator &

The output of bitwise AND is **1** if the corresponding bits of two operands is **1**. If either bit of an operand is **0**, the result of corresponding bit is evaluated to **0**.

In C Programming, the bitwise AND operator is denoted by `&`.

Let us suppose the bitwise AND operation of two integers **12** and **25**.

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

Bit Operation of 12 and 25
  00001100
& 00011001

  ________
  00001000  = 8 (In decimal)
```

Example 1: Bitwise AND

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

    int a = 12, b = 25;
    printf("Output = %d", a & b);

    return 0;
}
```

Run Code

Output

```
Output = 8
```

Bitwise OR Operator |

The output of bitwise OR is **1** if at least one corresponding bit of two operands is **1**. In C Programming, bitwise OR operator is denoted by `|`.

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

Bitwise OR Operation of 12 and 25
  00001100
| 00011001

  ________
  00011101  = 29 (In decimal)
```

Example 2: Bitwise OR

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

    int a = 12, b = 25;
    printf("Output = %d", a | b);


    return 0;
}
```

Run Code

Output

```
Output = 29
```

Bitwise XOR (exclusive OR) Operator ^

The result of bitwise XOR operator is **1** if the corresponding bits of two operands are opposite. It is denoted by `^`.

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

Bitwise XOR Operation of 12 and 25
  00001100
^ 00011001
  ________
  00010101  = 21 (In decimal)
```

Example 3: Bitwise XOR

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

    int a = 12, b = 25;
    printf("Output = %d", a ^ b);

    return 0;
}
```

Run Code

Output

```
Output = 21
```

Bitwise Complement Operator ~

Bitwise complement operator is a unary operator (works on only one operand). It changes **1** to **0** and **0** to **1**. It is denoted by `~`.

```
35 = 00100011 (In Binary)

Bitwise complement Operation of 35
~ 00100011

  ________
  11011100  = 220 (In decimal)
```

Twist in Bitwise Complement Operator in C Programming

The bitwise complement of **35** (`~35`) is **-36** instead of **220**, but why?
For any integer `n`, bitwise complement of `n` will be `-(n + 1)`. To understand this, you should have the knowledge of 2's complement.

2's Complement

Two's complement is an operation on binary numbers. The 2's complement of a number is equal to the complement of that number plus 1. For example:

```
 Decimal         Binary           2's complement
   0            00000000           -(11111111+1) = -00000000 = -0(decimal)
   1            00000001           -(11111110+1) = -11111111 = -256(decimal)
   12           00001100           -(11110011+1) = -11110100 = -244(decimal)
   220          11011100           -(00100011+1) = -00100100 = -36(decimal)

Note: Overflow is ignored while computing 2's complement.
```

The bitwise complement of **35** is **220** (in decimal). The 2's complement of **220** is **-36**. Hence, the output is `-36` instead of **220**.

Bitwise Complement of Any Number N is -(N+1). Here's how:

```
bitwise complement of N = ~N (represented in 2's complement form)
2'complement of ~N= -(~(~N)+1) = -(N+1)
```

Example 4: Bitwise complement

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

    printf("Output = %d\n", ~35);
    printf("Output = %d\n", ~-12);

    return 0;
}
```
Run Code

Output

```
Output = -36
Output = 11
```

Shift Operators in C programming

There are two shift operators in C programming:

- Right shift operator
- Left shift operator.

Right Shift Operator

Right shift operator shifts all bits towards right by certain number of specified bits. It is denoted by `>>`.

```
212 = 11010100 (In binary)
212 >> 2 = 00110101 (In binary) [Right shift by two bits]
212 >> 7 = 00000001 (In binary)
212 >> 8 = 00000000
212 >> 0 = 11010100 (No Shift)
```

Left Shift Operator

Left shift operator shifts all bits towards left by a certain number of specified bits. The bit positions that have been vacated by the left shift operator are filled with **0**. The symbol of the left shift operator is `<<`.

```
212 = 11010100 (In binary)
212<<1 = 110101000 (In binary) [Left shift by one bit]
212<<0 = 11010100 (Shift by 0)
212<<4 = 110101000000 (In binary) =3392(In decimal)
```

Example #5: Shift Operators

```
#include <stdio.h>

int main() {

    int num=212, i;

    for (i = 0; i <= 2; ++i) {
        printf("Right shift by %d: %d\n", i, num >> i);
    }
    printf("\n");
    for (i = 0; i <= 2; ++i) {
        printf("Left shift by %d: %d\n", i, num << i);
    }

    return 0;
}
```
Run Code

```
Right Shift by 0: 212
Right Shift by 1: 106
Right Shift by 2: 53

Left Shift by 0: 212
Left Shift by 1: 424
Left Shift by 2: 848
```

Example 6: size of Operator

```
#include <stdio.h>
int main()
{
    int a;
    float b;
    double c;
    char d;
    printf("Size of int=%lu bytes\n",sizeof(a));
    printf("Size of float=%lu bytes\n",sizeof(b));
    printf("Size of double=%lu bytes\n",sizeof(c));
    printf("Size of char=%lu byte\n",sizeof(d));

    return 0;
}
```
Run Code

Output

```
Size of int = 4 bytes
Size of float = 4 bytes
Size of double = 8 bytes
Size of char = 1 byte
```

C if...else Statement

C if Statement

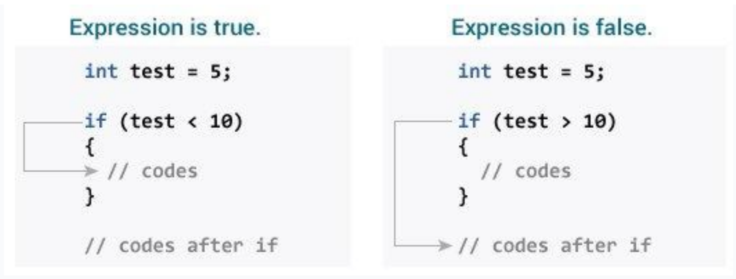
The syntax of the `if` statement in C programming is:

```
if (test expression)
{
   // code
}
```

How if statement works?

The `if` statement evaluates the test expression inside the parenthesis `()`.

- If the test expression is evaluated to true, statements inside the body of `if` are executed.
- If the test expression is evaluated to false, statements inside the body of `if` are not executed.



```
Expression is true.
int test = 5;

if (test < 10)
{
   // codes
}

// codes after if
```

```
Expression is false.
int test = 5;

if (test > 10)
{
   // codes
}

// codes after if
```

Working of if Statement

To learn more about when test expression is evaluated to true (non-zero value) and false (0), check relational and logical operators.

Example 1: if statement

```
// Program to display a number if it is negative

#include <stdio.h>
int main() {
    int number;

    printf("Enter an integer: ");
    scanf("%d", &number);

    // true if number is less than 0
    if (number < 0) {
        printf("You entered %d.\n", number);
    }

    printf("The if statement is easy.");

    return 0;
}
```

Run Code

Output 1

```
Enter an integer: -2
You entered -2.
The if statement is easy.
```

When the user enters -2, the test expression `number<0` is evaluated to true. Hence, `You entered -2` is displayed on the screen.

Output 2

```
Enter an integer: 5
The if statement is easy.
```

When the user enters 5, the test expression `number<0` is evaluated to false and the statement inside the body of `if` is not executed

C if...else Statement

The `if` statement may have an optional `else` block. The syntax of the `if..else` statement is:

```
if (test expression) {
    // run code if test expression is true
}
else {
    // run code if test expression is false
}
```

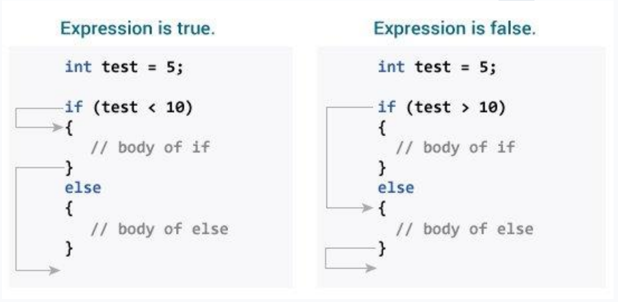
How if...else statement works?

If the test expression is evaluated to true,

- statements inside the body of `if` are executed.
- statements inside the body of `else` are skipped from execution.

If the test expression is evaluated to false,

- statements inside the body of `else` are executed
- statements inside the body of `if` are skipped from execution.



```
int test = 5;

if (test < 10)
{
   // body of if
}
else
{
   // body of else
}
```

```
int test = 5;

if (test > 10)
{
   // body of if
}
else
{
   // body of else
}
```

Working of if...else Statement

Example 2: if...else statement

```
// Check whether an integer is odd or even

#include <stdio.h>
int main() {
    int number;
    printf("Enter an integer: ");
    scanf("%d", &number);

    // True if the remainder is 0
    if  (number%2 == 0) {
        printf("%d is an even integer.",number);
    }
    else {
        printf("%d is an odd integer.",number);
    }

    return 0;
}
```
Run Code

Output

```
Enter an integer: 7
7 is an odd integer.
```

When the user enters 7, the test expression `number%2==0` is evaluated to false. Hence, the statement inside the body of `else` is executed.

C if...else Ladder

The `if...else` statement executes two different codes depending upon whether the test expression is true or false. Sometimes, a choice has to be made from more than 2 possibilities.
The if...else ladder allows you to check between multiple test expressions and execute different statements.

Syntax of if...else Ladder

```c
if (test expression1) {
   // statement(s)
}
else if(test expression2) {
   // statement(s)
}
else if (test expression3) {
   // statement(s)
}
.
.
else {
   // statement(s)
}
```

Example 3: C if...else Ladder

```c
// Program to relate two integers using =, > or < symbol

#include <stdio.h>
int main() {
    int number1, number2;
    printf("Enter two integers: ");
    scanf("%d %d", &number1, &number2);

    //checks if the two integers are equal.
    if(number1 == number2) {
        printf("Result: %d = %d",number1,number2);
    }

    //checks if number1 is greater than number2.
    else if (number1 > number2) {
        printf("Result: %d > %d", number1, number2);
    }

    //checks if both test expressions are false
    else {
        printf("Result: %d < %d",number1, number2);
    }

    return 0;
}
```

Run Code

Output

```
Enter two integers: 12
23
Result: 12 < 23
```

Nested if...else

It is possible to include an `if...else` statement inside the body of another `if...else` statement.

Example 4: Nested if...else

This program given below relates two integers using either `<`, `>` and `=` similar to the `if...else` ladder's example. However, we will use a nested `if...else` statement to solve this problem.

```
#include <stdio.h>
int main() {
    int number1, number2;
    printf("Enter two integers: ");
    scanf("%d %d", &number1, &number2);

    if (number1 >= number2) {
      if (number1 == number2) {
        printf("Result: %d = %d",number1,number2);
      }
      else {
        printf("Result: %d > %d", number1, number2);
      }
    }
    else {
        printf("Result: %d < %d",number1, number2);
    }

    return 0;
}
```

Run Code

If the body of an `if...else` statement has only one statement, you do not need to use brackets `{}`.

For example, this code

```
if (a > b) {
    printf("Hello");
}
printf("Hi");
```

is equivalent to

```
if (a > b)
    printf("Hello");
printf("Hi");
```

C for Loop

In programming, a loop is used to repeat a block of code until the specified condition is met.

C programming has three types of loops:

1. for loop
2. while loop
3. do...while loop

We will learn about `for` loop in this tutorial. In the next tutorial, we will learn about `while` and `do...while` loop.

for Loop

The syntax of the `for` loop is:

```
for (initializationStatement; testExpression; updateStatement)
{
    // statements inside the body of loop
}
```

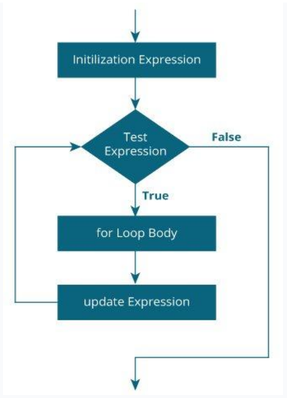
How for loop works?

- The initialization statement is executed only once.
- Then, the test expression is evaluated. If the test expression is evaluated to false, the `for` loop is terminated.
- However, if the test expression is evaluated to true, statements inside the body of the `for` loop are executed, and the update expression is updated.
- Again the test expression is evaluated.

This process goes on until the test expression is false. When the test expression is false, the loop terminates.

To learn more about test expression (when the test expression is evaluated to true and false), check out relational and logical operators.

for loop Flowchart



Example 1: for loop

```
// Print numbers from 1 to 10
#include <stdio.h>

int main() {
  int i;

  for (i = 1; i < 11; ++i)
  {
    printf("%d ", i);
  }
  return 0;
}
```
Run Code

Output

```
1 2 3 4 5 6 7 8 9 10
```

1. `i` is initialized to 1.
2. The test expression `i < 11` is evaluated. Since 1 less than 11 is true, the body of `for` loop is executed. This will print the **1** (value of `i`) on the screen.
3. The update statement `++i` is executed. Now, the value of `i` will be 2. Again, the test expression is evaluated to true, and the body of `for` loop is executed. This will print **2** (value of `i`) on the screen.
4. Again, the update statement `++i` is executed and the test expression `i < 11` is evaluated. This process goes on until `i` becomes 11.

5. When `i` becomes 11, `i < 11` will be false, and the `for` loop terminates.

Example 2: for loop

```
// Program to calculate the sum of first n natural numbers
// Positive integers 1,2,3...n are known as natural numbers

#include <stdio.h>
int main()
{
    int num, count, sum = 0;

    printf("Enter a positive integer: ");
    scanf("%d", &num);

    // for loop terminates when num is less than count
    for(count = 1; count <= num; ++count)
    {
        sum += count;
    }

    printf("Sum = %d", sum);

    return 0;
}
```
Run Code

Output

```
Enter a positive integer: 10
Sum = 55
```

The value entered by the user is stored in the variable `num`. Suppose, the user entered 10.

The `count` is initialized to 1 and the test expression is evaluated. Since the test expression `count<=num` (1 less than or equal to 10) is true, the body of `for` loop is executed and the value of `sum` will equal to 1.

Then, the update statement `++count` is executed and `count` will equal to 2. Again, the test expression is evaluated. Since 2 is also less than 10, the

test expression is evaluated to true and the body of the `for` loop is executed. Now, `sum` will equal 3.
This process goes on and the sum is calculated until the `count` reaches 11.
When the `count` is 11, the test expression is evaluated to 0 (false), and the loop terminates.
Then, the value of `sum` is printed on the screen.

C while and do...while Loop

In programming, loops are used to repeat a block of code until a specified condition is met.

C programming has three types of loops.

1. for loop
2. while loop
3. do...while loop

In the previous tutorial, we learned about `for` loop. In this tutorial, we will learn about `while` and `do..while` loop.

While loop

The syntax of the `while` loop is:

```
while (testExpression) {
  // the body of the loop
}
```

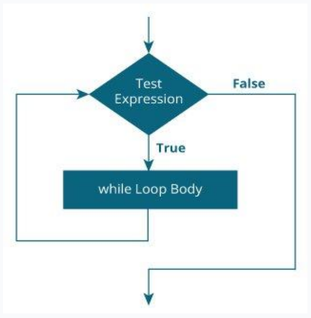
How while loop works?

- The `while` loop evaluates the `testExpression` inside the parentheses `()`.

- If `testExpression` is **true**, statements inside the body of `while` loop are executed. Then, `testExpression` is evaluated again.
- The process goes on until `testExpression` is evaluated to **false**.
- If `testExpression` is **false**, the loop terminates (ends).

To learn more about test expressions (when `testExpression` is evaluated to **true** and **false**), check out relational and logical operators.

Flowchart of while loop



Working of while loop

Example 1: while loop

```c
// Print numbers from 1 to 5

#include <stdio.h>
int main() {
  int i = 1;

  while (i <= 5) {
    printf("%d\n", i);
    ++i;
  }

  return 0;
}
```
Run Code

Output

```
1
2
```

```
3
4
5
```

Here, we have initialized `i` to 1.

1. When `i = 1`, the test expression `i <= 5` is **true**. Hence, the body of the `while` loop is executed. This prints `1` on the screen and the value of `i` is increased to `2`.
2. Now, `i = 2`, the test expression `i <= 5` is again **true**. The body of the `while` loop is executed again. This prints `2` on the screen and the value of `i` is increased to `3`.
3. This process goes on until `i` becomes 6. Then, the test expression `i <= 5` will be **false** and the loop terminates.

do...while loop

The `do..while` loop is similar to the `while` loop with one important difference. The body of `do...while` loop is executed at least once. Only then, the test expression is evaluated.

The syntax of the `do...while` loop is:

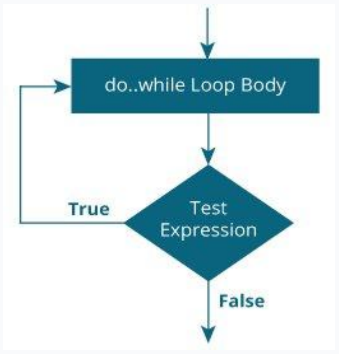
```
do {
  // the body of the loop
}
while (testExpression);
```

How do...while loop works?

- The body of `do...while` loop is executed once. Only then, the `testExpression` is evaluated.
- If `testExpression` is **true**, the body of the loop is executed again and `testExpression` is evaluated once more.
- This process goes on until `testExpression` becomes **false**.

- If `testExpression` is **false**, the loop ends.

Flowchart of do...while Loop



Working of do...while loop

Example 2: do...while loop

```c
// Program to add numbers until the user enters zero

#include <stdio.h>
int main() {
  double number, sum = 0;

  // the body of the loop is executed at least once
  do {
    printf("Enter a number: ");
    scanf("%lf", &number);
    sum += number;
  }
  while(number != 0.0);

  printf("Sum = %.2lf",sum);

  return 0;
}
```
Run Code

Output

```
Enter a number: 1.5
Enter a number: 2.4
Enter a number: -3.4
Enter a number: 4.2
Enter a number: 0
Sum = 4.70
```

Here, we have used a `do...while` loop to prompt the user to enter a number. The loop works as long as the input number is not `0`.

The `do...while` loop executes at least once i.e. the first iteration runs without checking the condition. The condition is checked only after the first iteration has been executed.

```
do {
  printf("Enter a number: ");
  scanf("%lf", &number);
  sum += number;
}
while(number != 0.0);
```

So, if the first input is a non-zero number, that number is added to the `sum` variable and the loop continues to the next iteration. This process is repeated until the user enters `0`.

But if the first input is 0, there will be no second iteration of the loop and `sum` becomes `0.0`.

Outside the loop, we print the value of `sum`.

C break and continue

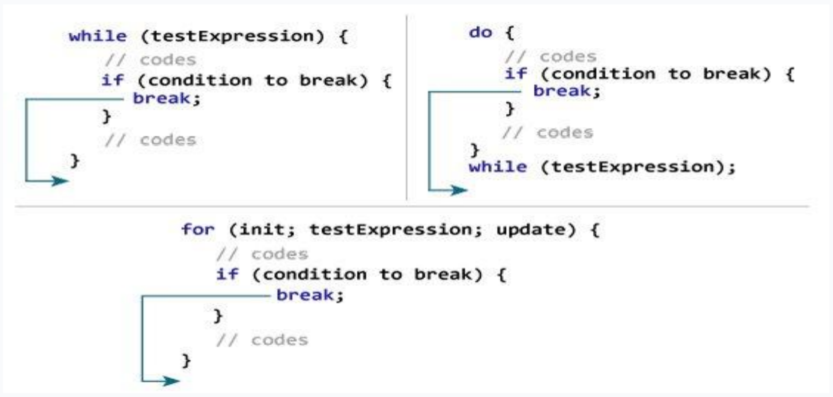
C break

The break statement ends the loop immediately when it is encountered. Its syntax is:

```
break;
```

The break statement is almost always used with `if...else` statement inside the loop.

How break statement works?



```
while (testExpression) {
   // codes
   if (condition to break) {
      break;
   }
   // codes
}
```

```
do {
   // codes
   if (condition to break) {
      break;
   }
   // codes
}
while (testExpression);
```

```
for (init; testExpression; update) {
   // codes
   if (condition to break) {
      break;
   }
   // codes
}
```

Working of break in C

Example 1: break statement

```
// Program to calculate the sum of numbers (10 numbers max)
// If the user enters a negative number, the loop terminates

#include <stdio.h>

int main() {
   int i;
   double number, sum = 0.0;

   for (i = 1; i <= 10; ++i) {
      printf("Enter n%d: ", i);
      scanf("%lf", &number);

      // if the user enters a negative number, break the loop
      if (number < 0.0) {
         break;
      }

      sum += number; // sum = sum + number;
   }

   printf("Sum = %.2lf", sum);
```

```
    return 0;
}
```
Run Code

Output

```
Enter n1: 2.4
Enter n2: 4.5
Enter n3: 3.4
Enter n4: -3
Sum = 10.30
```

This program calculates the sum of a maximum of 10 numbers. Why a maximum of 10 numbers? It's because if the user enters a negative number, the `break` statement is executed. This will end the `for` loop, and the `sum` is displayed.

In C, `break` is also used with the `switch` statement. This will be discussed in the next tutorial.

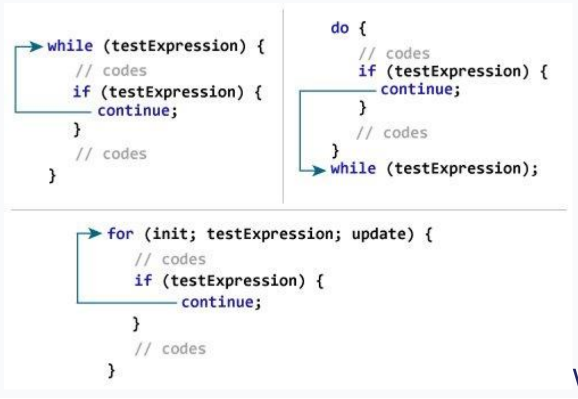
C continue

The `continue` statement skips the current iteration of the loop and continues with the next iteration. Its syntax is:

```
continue;
```

The `continue` statement is almost always used with the `if...else` statement.

How continue statement works?



```
while (testExpression) {
    // codes
    if (testExpression) {
        continue;
    }
    // codes
}
```

```
do {
    // codes
    if (testExpression) {
        continue;
    }
    // codes
}
while (testExpression);
```

```
for (init; testExpression; update) {
    // codes
    if (testExpression) {
        continue;
    }
    // codes
}
```

Working of Continue in C

Example 2: continue statement

```
// Program to calculate the sum of numbers (10 numbers max)
// If the user enters a negative number, it's not added to the result

#include <stdio.h>
int main() {
   int i;
   double number, sum = 0.0;

   for (i = 1; i <= 10; ++i) {
      printf("Enter a n%d: ", i);
      scanf("%lf", &number);

      if (number < 0.0) {
         continue;
      }
      sum += number; // sum = sum + number;
   }

   printf("Sum = %.2lf", sum);

   return 0;
}
```
Run Code

Output

```
Enter n1: 1.1
Enter n2: 2.2
Enter n3: 5.5
Enter n4: 4.4
Enter n5: -3.4
Enter n6: -45.5
Enter n7: 34.5
Enter n8: -4.2
Enter n9: -1000
Enter n10: 12
Sum = 59.70
```

In this program, when the user enters a positive number, the sum is calculated using `sum += number;` statement.

When the user enters a negative number, the `continue` statement is executed and it skips the negative number from the calculation.

C switch Statement

The switch statement allows us to execute one code block among many alternatives.

You can do the same thing with the `if...else..if` ladder. However, the syntax of the `switch` statement is much easier to read and write.

Syntax of switch...case

```
switch (expression)
{
    case constant1:
      // statements
      break;

    case constant2:
      // statements
      break;
    .
    .
    .
    default:
      // default statements
}
```

How does the switch statement work?

The `expression` is evaluated once and compared with the values of each `case` label.

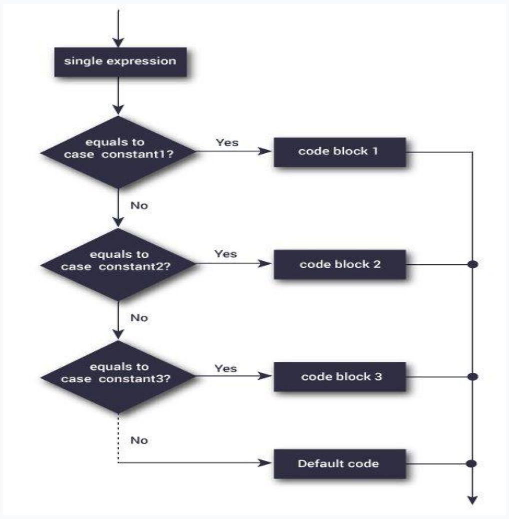
- If there is a match, the corresponding statements after the matching label are executed. For example, if the value of the expression is equal to `constant2`, statements after `case constant2:` are executed until `break` is encountered.

- If there is no match, the default statements are executed.

Notes:

- If we do not use the `break` statement, all statements after the matching label are also executed.
- The `default` clause inside the `switch` statement is optional.

switch Statement Flowchart



Switch Statement Flowchart

Example: Simple Calculator

```
// Program to create a simple calculator
#include <stdio.h>

int main() {
    char operation;
    int n1, n2;

    printf("Enter an operator (+, -, *, /): ");
    scanf("%c", &operation);
    printf("Enter two operands: ");
    scanf("%d %d",&n1, &n2);
    switch(operation)
    {
        case '+':
            printf("%d + %d = %d",n1, n2, n1+n2);
            break;
        case '-':
            printf("%d - %d = %d",n1, n2, n1-n2);
            break;
        case '*':
            printf("%d * %d = %d",n1, n2, n1*n2);
            break;

        case '/':
            printf("%d / %d = %d",n1, n2, n1/n2);
            break;

        // operator doesn't match any case constant +, -, *, /
        default:
            printf("Error! operator is not correct");
    }
    return 0;
}
```

Run Code

Output

```
Enter an operator (+, -, *, /): +
Enter two operands: 32
12
32 - 12 = 44
```

The `-` operator entered by the user is stored in the `operation` variable. And, two operands `32` and `12` are stored in variables `n1` and `n2` respectively. Since the `operation` is `-`, the control of the program jumps to

```
printf("%d - %d = %d", n1, n2, n1-n2);
```

Finally, the break statement terminates the `switch` statement.

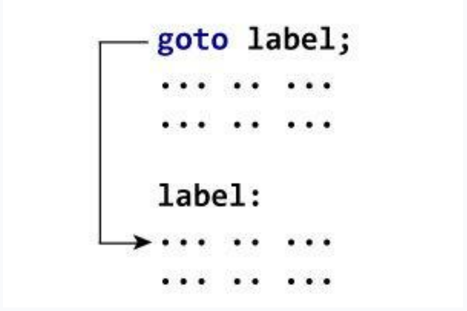
C goto Statement

The `goto` statement allows us to transfer control of the program to the specified `label`.

Syntax of goto Statement

```
goto label;
label:
statement;
```

The `label` is an identifier. When the `goto` statement is encountered, the control of the program jumps to `label:` and starts executing the code.



Working of goto Statement

Example: goto Statement

```
// Program to calculate the sum and average of positive numbers
// If the user enters a negative number, the sum and average are displayed.

#include <stdio.h>

int main() {

   const int maxInput = 100;
   int i;
   double number, average, sum = 0.0;

   for (i = 1; i <= maxInput; ++i) {
      printf("%d. Enter a number: ", i);
      scanf("%lf", &number);

      // go to jump if the user enters a negative number
      if (number < 0.0) {
            goto jump;
        }
        sum += number;
    }

jump:
    average = sum / (i - 1);
    printf("Sum = %.2f\n", sum);
    printf("Average = %.2f", average);

    return 0;
}
```
Run Code

Output

```
1. Enter a number: 3
2. Enter a number: 4.3
3. Enter a number: 9.3
4. Enter a number: -2.9
Sum = 16.60
Average = 5.53
```

Reasons to avoid goto

The use of `goto` statement may lead to code that is buggy and hard to follow. For example,

```
one:
for (i = 0; i < number; ++i)
{
    test += i;
    goto two;
}
two:
if (test > 5) {
  goto three;
}
... .. ...
```

Also, the `goto` statement allows you to do bad stuff such as jump out of the scope.
That being said, `goto` can be useful sometimes. For example: to break from nested loops.

Should you use goto?

If you think the use of `goto` statement simplifies your program, you can use it. That being said, `goto` is rarely useful and you can create any C program without using `goto` altogether.
Here's a quote from Bjarne Stroustrup, creator of C++, **"The fact that 'goto' can do anything is exactly why we don't use it."**

C Functions

A function is a block of code that performs a specific task.
Suppose, you need to create a program to create a circle and color it. You can create two functions to solve this problem:

- create a circle function
- create a color function

Dividing a complex problem into smaller chunks makes our program easy to understand and reuse.

Types of function

There are two types of function in C programming:

- Standard library functions
- User-defined functions

Standard library functions

The standard library functions are built-in functions in C programming.

These functions are defined in header files. For example,

- The `printf()` is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in the `stdio.h` header file.
  Hence, to use the `printf()`function, we need to include the `stdio.h` header file using `#include <stdio.h>`.
- The `sqrt()` function calculates the square root of a number. The function is defined in the `math.h` header file.

User-defined function

You can also create functions as per your need. Such functions created by the user are known as user-defined functions.

How user-defined function works?

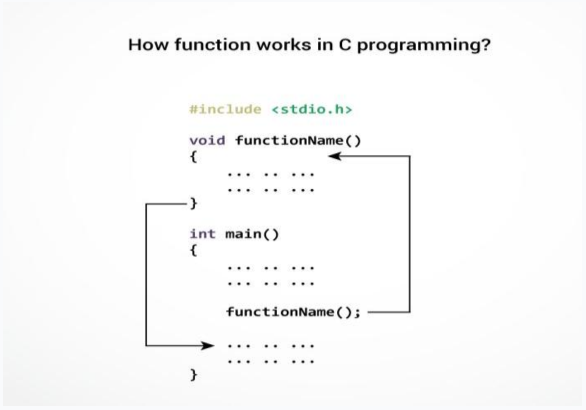
```
#include <stdio.h>

void functionName()

{

}

int main()

{

    functionName();

}
```

The execution of a C program begins from the `main()` function.

When the compiler encounters `functionName();`, control of the program jumps to

```
void functionName()
```

And, the compiler starts executing the codes inside `functionName()`.
The control of the program jumps back to the `main()` function once code inside the function definition is executed.



```
#include <stdio.h>

void functionName()
{
    ... .. ...
    ... .. ...
}

int main()
{
    ... .. ...
    ... .. ...

    functionName();

    ... .. ...
    ... .. ...
}
```

Working of C Function

Note, function names are identifiers and should be unique.

This is just an overview of user-defined functions. Visit these pages to learn more on:

Advantages of user-defined function

1. The program will be easier to understand, maintain and debug.
2. Reusable codes that can be used in other programs. A large program can be divided into smaller modules. Hence, a large project can be divided among many programmers. Types of User-defined Functions in C Programming

Advantage of functions in C

There are the following advantages of C functions.

- By using functions, we can avoid rewriting same logic/code again and again in a program.
- We can call C functions any number of times in a program and from any place in a program.
- We can track a large C program easily when it is divided into multiple functions.
- Reusability is the main achievement of C functions.
- However, Function calling is always a overhead in a C program.

Function Aspects

There are three aspects of a C function.

- **Function declaration** A function must be declared globally in a c program to tell the compiler about the function name, function parameters, and return type.
- **Function call** Function can be called from anywhere in the program. The parameter list must not differ in function calling and function declaration. We must pass the same number of functions as it is declared in the function declaration.
- **Function definition** It contains the actual statements which are to be executed. It is the most important aspect to which the control comes when the function is called. Here, we must notice that only one value can be returned from the function.

SN	C function aspects	Syntax
1	Function declaration	return_type function_name (argument list);
2	Function call	function_name (argument_list)
3	Function definition	return_type function_name (argument list) {function body;}

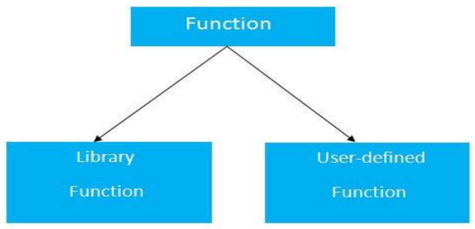
The syntax of creating function in c language is given below:

```
return_type function_name(data_type parameter...){
//code to be executed
}
```

Types of Functions

There are two types of functions in C programming:

1. **Library Functions:** are the functions which are declared in the C header files such as scanf(), printf(), gets(), puts(), ceil(), floor() etc.
2. **User-defined functions:** are the functions which are created by the C programmer, so that he/she can use it many times. It reduces the complexity of a big program and optimizes the code.



Return Value

A C function may or may not return a value from the function. If you don't have to return any value from the function, use void for the return type.

Let's see a simple example of C function that doesn't return any value from the function.

Example without return value:

```
void hello(){
printf("hello c");
}
```

If you want to return any value from the function, you need to use any data type such as int, long, char, etc. The return type depends on the value to be returned from the function.

Let's see a simple example of C function that returns int value from the function.

Example with return value:

```
int get(){
return 10;
}
```

In the above example, we have to return 10 as a value, so the return type is int. If you want to return floating-point value (e.g., 10.2, 3.1, 54.5, etc), you need to use float as the return type of the method.

```c
float get(){
return 10.2;
}
```

Now, you need to call the function, to get the value of the function.

Different aspects of function calling

A function may or may not accept any argument. It may or may not return any value. Based on these facts, There are four different aspects of function calls.

- function without arguments and without return value
- function without arguments and with return value
- function with arguments and without return value
- function with arguments and with return value

Example for Function without argument and return value

Example 1

```c
#include<stdio.h>
void printName();
void main ()
{
    printf("Hello ");
    printName();
}
void printName()
{
    printf("Hello123");
}
```

Output

```
Hello Hello123
```

Example 2

```c
#include<stdio.h>
void sum();
void main()
{
    printf("\nGoing to calculate the sum of two numbers:");
    sum();
}
void sum()
{
    int a,b;
    printf("\nEnter two numbers");
    scanf("%d %d",&a,&b);
    printf("The sum is %d",a+b);
}
```

Output

```
Going to calculate the sum of two numbers:

Enter two numbers 10
24

The sum is 34
```

Example for Function without argument and with return value

Example 1

```c
#include<stdio.h>
int sum();
void main()
{
    int result;
    printf("\nGoing to calculate the sum of two numbers:");
    result = sum();
    printf("%d",result);
}
int sum()
{
    int a,b;
    printf("\nEnter two numbers");
    scanf("%d %d",&a,&b);
    return a+b;
}
```

Output

```
Going to calculate the sum of two numbers:

Enter two numbers 10
24

The sum is 34
```

Example 2: program to calculate the area of the square

```
#include<stdio.h>
int sum();
void main()
{
    printf("Going to calculate the area of the square\n");
    float area = square();
    printf("The area of the square: %f\n",area);
}
int square()
{
    float side;
    printf("Enter the length of the side in meters: ");
    scanf("%f",&side);
    return side * side;
}
```

Output

```
Going to calculate the area of the square
Enter the length of the side in meters: 10
The area of the square: 100.000000
```

Example for Function with argument and without return value

Example 1

```c
#include<stdio.h>
void sum(int, int);
void main()
{
    int a,b,result;
    printf("\nGoing to calculate the sum of two numbers:");
    printf("\nEnter two numbers:");
    scanf("%d %d",&a,&b);
    sum(a,b);
}
void sum(int a, int b)
{
    printf("\nThe sum is %d",a+b);
}
```

Output

```
Going to calculate the sum of two numbers:

Enter two numbers 10
24

The sum is 34
```

Example 2: program to calculate the average of five numbers.

```c
#include<stdio.h>
void average(int, int, int, int, int);
void main()
{
    int a,b,c,d,e;
    printf("\nGoing to calculate the average of five numbers:");
    printf("\nEnter five numbers:");
    scanf("%d %d %d %d %d",&a,&b,&c,&d,&e);
    average(a,b,c,d,e);
}
void average(int a, int b, int c, int d, int e)
{
    float avg;
    avg = (a+b+c+d+e)/5;
    printf("The average of given five numbers : %f",avg);
}
```

Output

```
Going to calculate the average of five numbers:
Enter five numbers:10
20
30
40
50
The average of given five numbers : 30.000000
```

Example for Function with argument and with return value

Example 1

```c
#include<stdio.h>
int sum(int, int);
void main()
{
    int a,b,result;
    printf("\nGoing to calculate the sum of two numbers:");
    printf("\nEnter two numbers:");
    scanf("%d %d",&a,&b);
    result = sum(a,b);
    printf("\nThe sum is : %d",result);
}
```

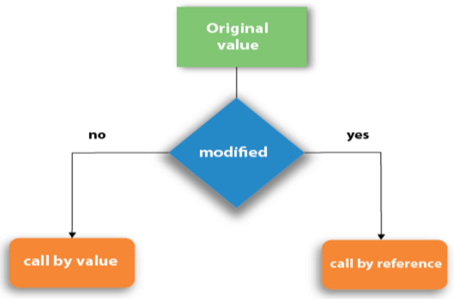
```c
int sum(int a, int b)
{
    return a+b;
}
```

Output

```
Going to calculate the sum of two numbers:
Enter two numbers:10
20
The sum is : 30
```

Call by value and Call by reference in C

There are two methods to pass the data into the function in C language, i.e., *call by value* and *call by reference*.



Let's understand call by value and call by reference in c language one by one.

Call by value in C

- In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.
- In call by value method, we can not modify the value of the actual parameter by the formal parameter.

- In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
- The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

Let's try to understand the concept of call by value in c language by the example given below:

```c
#include<stdio.h>
void change(int num) {
    printf("Before adding value inside function num=%d \n",num);
    num=num+100;
    printf("After adding value inside function num=%d \n", num);
}
int main() {
    int x=100;
    printf("Before function call x=%d \n", x);
    change(x);//passing value in function
    printf("After function call x=%d \n", x);
return 0;
}
```

Output

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=100
```

Call by Value Example: Swapping the values of the two variables

```c
#include <stdio.h>
void swap(int , int); //prototype of the function
int main()
{
    int a = 10;
    int b = 20;
    printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a a
nd b in main
```

```c
    swap(a,b);
    printf("After swapping values in main a = %d, b = %d\n",a,b); // The value of actual parameters do not change by changing the formal parameters in call by value, a = 10, b = 20
}
void swap (int a, int b)
{
    int temp;
    temp = a;
    a=b;
    b=temp;
    printf("After swapping values in function a = %d, b = %d\n",a,b); // Formal parameters, a = 20, b = 10
}
```

Output

```
Before swapping the values in main a = 10, b = 20
After swapping values in function a = 20, b = 10
After swapping values in main a = 10, b = 20
```

Call by reference in C

- In call by reference, the address of the variable is passed into the function call as the actual parameter.
- The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
- In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

Consider the following example for the call by reference.

```c
#include<stdio.h>
void change(int *num) {
    printf("Before adding value inside function num=%d \n",*num);
    (*num) += 100;
    printf("After adding value inside function num=%d \n", *num);
}
```

```c
int main() {
    int x=100;
    printf("Before function call x=%d \n", x);
    change(&x);//passing reference in function
    printf("After function call x=%d \n", x);
return 0;
}
```

Output

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=200
```

Call by reference Example: Swapping the values of the two variables

```c
#include <stdio.h>
void swap(int *, int *); //prototype of the function
int main()
{
    int a = 10;
    int b = 20;
    printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and b in main
    swap(&a,&b);
    printf("After swapping values in main a = %d, b = %d\n",a,b); // The values of actual parameters do change in call by reference, a = 10, b = 20
}
void swap (int *a, int *b)
{
    int temp;
    temp = *a;
    *a=*b;
    *b=temp;
    printf("After swapping values in function a = %d, b = %d\n",*a,*b); // Formal parameters, a = 20, b = 10
}
```

Output

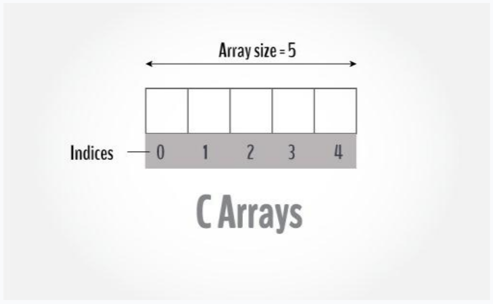
```
Before swapping the values in main a = 10, b = 20
```

```
After swapping values in function a = 20, b = 10
After swapping values in main a = 20, b = 10
```

Difference between call by value and call by reference in c

No.	Call by value	Call by reference
1	A copy of the value is passed into the function	An address of value is passed into the function
2	Changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters.	Changes made inside the function validate outside of the function also. The values of the actual parameters do change by changing the formal parameters.
3	Actual and formal arguments are created at the different memory location	Actual and formal arguments are created at the same memory location

C Arrays



Arrays in C

An array is a variable that can store multiple values. For example, if you want to store 100 integers, you can create an array for it.

```
int data[100];
```

How to declare an array?

```
dataType arrayName[arraySize];
```

For example,

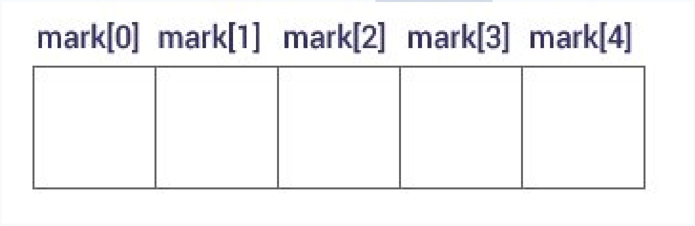
```
float mark[5];
```

Here, we declared an array, `mark`, of floating-point type. And its size is 5. Meaning, it can hold 5 floating-point values.
It's important to note that the size and type of an array cannot be changed once it is declared.

Access Array Elements

You can access elements of an array by indices.

Suppose you declared an array `mark` as above. The first element is `mark[0]`, the second element is `mark[1]` and so on.



Declare an Array

Few keynotes:

- Arrays have 0 as the first index, not 1. In this example, `mark[0]` is the first element.

- If the size of an array is `n`, to access the last element, the `n-1` index is used. In this example, `mark[4]`
- Suppose the starting address of `mark[0]` is **2120d**. Then, the address of the `mark[1]` will be **2124d**. Similarly, the address of `mark[2]` will be **2128d** and so on.
  This is because the size of a `float` is 4 bytes.

How to initialize an array?

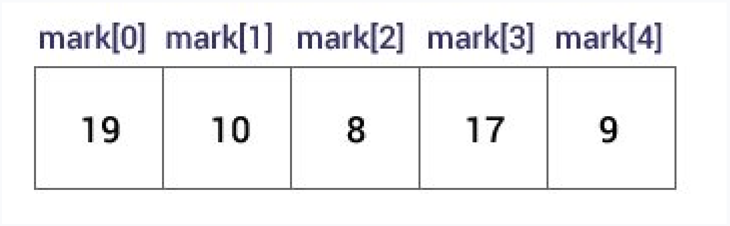
It is possible to initialize an array during declaration. For example,

```
int mark[5] = {19, 10, 8, 17, 9};
```

You can also initialize an array like this.

```
int mark[] = {19, 10, 8, 17, 9};
```

Here, we haven't specified the size. However, the compiler knows its size is 5 as we are initializing it with 5 elements.



Initialize an Array

Here,

```
mark[0] is equal to 19

mark[1] is equal to 10

mark[2] is equal to 8

mark[3] is equal to 17
```

```
mark[4] is equal to 9
```

Change Value of Array elements

```
int mark[5] = {19, 10, 8, 17, 9}

// make the value of the third element to -1
mark[2] = -1;

// make the value of the fifth element to 0
mark[4] = 0;
```

Input and Output Array Elements

Here's how you can take input from the user and store it in an array element.

```
// take input and store it in the 3rd element
scanf("%d", &mark[2]);

// take input and store it in the ith element
scanf("%d", &mark[i-1]);
```

Here's how you can print an individual element of an array.

```
// print the first element of the array
printf("%d", mark[0]);

// print the third element of the array
printf("%d", mark[2]);

// print ith element of the array
printf("%d", mark[i-1]);
```

Example 1: Array Input/Output

```c
// Program to take 5 values from the user and store them in an array
// Print the elements stored in the array

#include <stdio.h>

int main() {

  int values[5];

  printf("Enter 5 integers: ");

  // taking input and storing it in an array
  for(int i = 0; i < 5; ++i) {
     scanf("%d", &values[i]);
  }

  printf("Displaying integers: ");

  // printing elements of an array
  for(int i = 0; i < 5; ++i) {
     printf("%d\n", values[i]);
  }
  return 0;
}
```
Run Code

Output

```
Enter 5 integers: 1
-3
34
0
3
Displaying integers: 1
-3
34
0
3
```

Here, we have used a `for` loop to take 5 inputs from the user and store them in an array. Then, using another `for` loop, these elements are displayed on the screen.

Example 2: Calculate Average

```c
// Program to find the average of n numbers using arrays

#include <stdio.h>

int main() {

  int marks[10], i, n, sum = 0;
  double average;

  printf("Enter number of elements: ");
  scanf("%d", &n);

  for(i=0; i < n; ++i) {
    printf("Enter number%d: ",i+1);
    scanf("%d", &marks[i]);

    // adding integers entered by the user to the sum variable
    sum += marks[i];
  }

  // explicitly convert sum to double
  // then calculate average
  average = (double) sum / n;

  printf("Average = %.2lf", average);

  return 0;
}
```
Run Code

Output

```
Enter number of elements: 5
Enter number1: 45
Enter number2: 35
Enter number3: 38
Enter number4: 31
Enter number5: 49
Average = 39.60
```

Here, we have computed the average of `n` numbers entered by the user.

Access elements out of its bound!

Suppose you declared an array of 10 elements. Let's say,

```
int testArray[10];
```

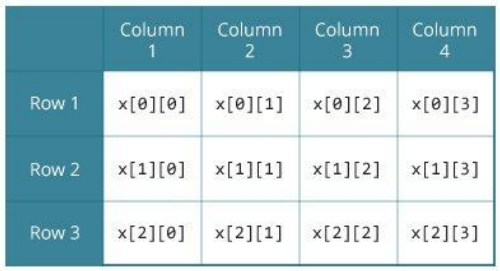
You can access the array elements from `testArray[0]` to `testArray[9]`.
Now let's say if you try to access `testArray[12]`. The element is not available. This may cause unexpected output (undefined behavior). Sometimes you might get an error and some other time your program may run correctly.
Hence, you should never access elements of an array outside of its bound.

C Multidimensional Arrays

In C programming, you can create an array of arrays. These arrays are known as multidimensional arrays. For example,

```
float x[3][4];
```

Here, `x` is a two-dimensional (2d) array. The array can hold 12 elements. You can think the array as a table with 3 rows and each row has 4 columns.



	Column 1	Column 2	Column 3	Column 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

Two dimensional Array

Similarly, you can declare a three-dimensional (3d) array. For example,

```
float y[2][4][3];
```

Here, the array `y` can hold 24 elements.

Initializing a multidimensional array

Here is how you can initialize two-dimensional and three-dimensional arrays:

Initialization of a 2d array

```
// Different ways to initialize two-dimensional array

int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};

int c[][3] = {{1, 3, 0}, {-1, 5, 9}};

int c[2][3] = {1, 3, 0, -1, 5, 9};
```

Initialization of a 3d array

You can initialize a three-dimensional array in a similar way to a two-dimensional array. Here's an example,

```
int test[2][3][4] = {
    {{3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2}},
    {{13, 4, 56, 3}, {5, 9, 3, 5}, {3, 1, 4, 9}}};
```

Example 1: Two-dimensional array to store and print values

```c
// C program to store temperature of two cities of a week and display it.
#include <stdio.h>
const int CITY = 2;
const int WEEK = 7;
int main()
{
  int temperature[CITY][WEEK];

  // Using nested loop to store values in a 2d array
  for (int i = 0; i < CITY; ++i)
  {
    for (int j = 0; j < WEEK; ++j)
    {
      printf("City %d, Day %d: ", i + 1, j + 1);
      scanf("%d", &temperature[i][j]);
    }
  }
  printf("\nDisplaying values: \n\n");

  // Using nested loop to display vlues of a 2d array
  for (int i = 0; i < CITY; ++i)
  {
    for (int j = 0; j < WEEK; ++j)
    {
      printf("City %d, Day %d = %d\n", i + 1, j + 1, temperature[i][j]);
    }
  }
  return 0;
}
```

Run Code

Output

```
City 1, Day 1: 33
City 1, Day 2: 34
City 1, Day 3: 35
City 1, Day 4: 33
City 1, Day 5: 32
City 1, Day 6: 31
City 1, Day 7: 30
City 2, Day 1: 23
City 2, Day 2: 22
City 2, Day 3: 21
City 2, Day 4: 24
City 2, Day 5: 22
City 2, Day 6: 25
City 2, Day 7: 26

Displaying values:

City 1, Day 1 = 33
City 1, Day 2 = 34
City 1, Day 3 = 35
City 1, Day 4 = 33
City 1, Day 5 = 32
City 1, Day 6 = 31
City 1, Day 7 = 30
City 2, Day 1 = 23
City 2, Day 2 = 22
City 2, Day 3 = 21
City 2, Day 4 = 24
City 2, Day 5 = 22
City 2, Day 6 = 25
```

Sum of two matrices

```
City 2, Day 7 = 26
```

```
// C program to find the sum of two matrices of order 2*2

#include <stdio.h>
int main()
{
  float a[2][2], b[2][2], result[2][2];

  // Taking input using nested for loop
  printf("Enter elements of 1st matrix\n");
  for (int i = 0; i < 2; ++i)
    for (int j = 0; j < 2; ++j)
    {
```

```c
      printf("Enter a%d%d: ", i + 1, j + 1);
      scanf("%f", &a[i][j]);
    }

  // Taking input using nested for loop
  printf("Enter elements of 2nd matrix\n");
  for (int i = 0; i < 2; ++i)
    for (int j = 0; j < 2; ++j)
    {
      printf("Enter b%d%d: ", i + 1, j + 1);
      scanf("%f", &b[i][j]);
    }

  // adding corresponding elements of two arrays
  for (int i = 0; i < 2; ++i)
    for (int j = 0; j < 2; ++j)
    {
      result[i][j] = a[i][j] + b[i][j];
    }

  // Displaying the sum
  printf("\nSum Of Matrix:");

  for (int i = 0; i < 2; ++i)
    for (int j = 0; j < 2; ++j)
    {
      printf("%.1f\t", result[i][j]);

      if (j == 1)
        printf("\n");
    }
  return 0;
}
```

```
o/P ->>

Enter elements of 1st matrix
Enter a11: 2;
Enter a12: 0.5;
Enter a21: -1.1;
Enter a22: 2;
Enter elements of 2nd matrix
Enter b11: 0.2;
Enter b12: 0;
Enter b21: 0.23;
Enter b22: 23;

Sum Of Matrix:
2.2     0.5
-0.9    25.0
```

❖ What is Recursion?

Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem. It is useful for problems that can be broken down into similar sub-problems. A recursive function typically includes:

1. **Base Case:** This condition terminates the recursion. Without it, the function would call itself indefinitely.
2. **Recursive Case:** This involves the function calling itself with modified arguments to work towards the base case.

Types of Recursion

3. **Direct Recursion:**
    1. **Definition:** The function calls itself directly.
    2. **Example:** A function `factorial` that calculates the factorial of a number.

4. **Code Example:**

```
#include <stdio.h>

// Function to calculate factorial
int factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;                      // Base case
    } else {
        return n * factorial(n - 1); // Recursive case
    }
}

int main() {
    int num = 5;
16.      printf("Factorial of %d is %d\n", num, factorial(num));
17.      return 0;
18.  }
```

19. **Output:**

```
20.  Copy code
21.  Factorial of 5 is 120
```

22. **Explanation:** `factorial(5)` results in `5 * factorial(4)`, which continues until `factorial(1)` returns 1.

Indirect Recursion:

- **Definition:** Function `A` calls function `B`, and function `B` calls function `A`.
- **Example:** Functions `functionA` and `functionB` that call each other.

Code Example:

```
#include <stdio.h>
void functionB(int n);

void functionA(int n) {
    if (n > 0) {
        printf("%d ", n);
        functionB(n - 1);                 // Calls functionB
    }
}
void functionB(int n) {
    if (n > 0) {
        printf("%d ", n);
        functionA(n - 1);                 // Calls functionA
    }
}

int main() {

    functionA(5);

    return 0;
}
```

Output:

```
Copy code
5 4 3 2 1 1 2 3 4 5
```

Explanation: `functionA` prints numbers from 5 to 1, then `functionB` prints numbers from 1 back to 5.

Important Key Points

1. **Base Case:** Essential to prevent infinite recursion. The base case ensures that the recursion stops and does not lead to a stack overflow.
2. **Recursive Case:** Must progress towards the base case. Each recursive call should work on a smaller or simpler problem.
3. **Stack Overflow:** Each recursive call consumes stack space. Deep recursion can lead to stack overflow errors if the recursion depth is too large.
4. **Performance:** Recursive functions can be less efficient than iterative solutions due to the overhead of multiple function calls and stack usage. Tail recursion can be optimized by some compilers into iterative code.
5. **Memoization:** For problems with overlapping sub-problems (like in dynamic programming), memoization can be used to store results of sub-problems to avoid redundant computations.
6. **Debugging:** Recursive functions can be challenging to debug. Print intermediate results or use a debugger to trace recursive calls.

C Strings

The string can be defined as the one-dimensional array of characters terminated by a null ('\0'). The character array or the string is used to manipulate text such as word or sentences. Each character in the array occupies one byte of memory, and the last character must always be 0. The termination character ('\0') is important in a string since it is the only way to identify where the string ends. When we define a string as char s[10], the character s[10] is implicitly initialized with the null in the memory.

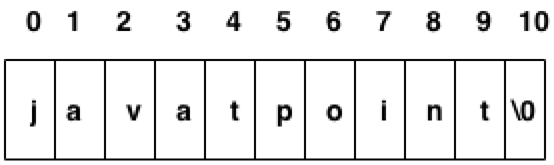
There are two ways to declare a string in c language.

1. By char array
2. By string literal

Let's see the example of declaring **string by char array** in C language.

```
char ch[10]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};
```

As we know, array index starts from 0, so it will be represented as in the figure given below.



While declaring string, size is not mandatory. So we can write the above code as given below:

```
char ch[]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};
```

We can also define the **string by the string literal** in C language. For example:

```
char ch[]="javatpoint";
```

In such case, '\0' will be appended at the end of the string by the compiler.

Difference between char array and string literal

There are two main differences between char array and literal.

- We need to add the null character '\0' at the end of the array by ourself whereas, it is appended internally by the compiler in the case of the character array.
- The string literal cannot be reassigned to another set of characters whereas, we can reassign the characters of the array.

String Example in C

Let's see a simple example where a string is declared and being printed. The '%s' is used as a format specifier for the string in c language.

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

```
int main(){
  char ch[11]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};
   char ch2[11]="javatpoint";

   printf("Char Array Value is: %s\n", ch);
   printf("String Literal Value is: %s\n", ch2);
 return 0;
}
```

Output

```
Char Array Value is: javatpoint
String Literal Value is: javatpoint
```

Traversing String

Traversing the string is one of the most important aspects in any of the programming languages. We may need to manipulate a very large text which can be done by traversing the text. Traversing string is somewhat different from the traversing an integer array. We need to know the length of the array to traverse an integer array, whereas we may use the null character in the case of string to identify the end the string and terminate the loop.

Hence, there are two ways to traverse a string.

- By using the length of string
- By using the null character.

Let's discuss each one of them.

Using the length of string

Let's see an example of counting the number of vowels in a string.

```
#include<stdio.h>
void main ()
{
    char s[11] = "javatpoint";
    int i = 0;
```

```c
    int count = 0;
    while(i<11)
    {
        if(s[i]=='a' || s[i] == 'e' || s[i] == 'i' || s[i] == 'u' || s[i] == 'o')
        {
            count ++;
        }
        i++;
    }
    printf("The number of vowels %d",count);
}
```

Output

```
The number of vowels 4
```

Using the null character

Let's see the same example of counting the number of vowels by using the null character.

```c
#include<stdio.h>
void main ()
{
    char s[11] = "javatpoint";
    int i = 0;
    int count = 0;
    while(s[i] != NULL)
    {
        if(s[i]=='a' || s[i] == 'e' || s[i] == 'i' || s[i] == 'u' || s[i] == 'o')
        {
            count ++;
        }
        i++;
    }
    printf("The number of vowels %d",count);
}
```

Output

```
The number of vowels 4
```

Accepting string as the input

Till now, we have used scanf to accept the input from the user. However, it can also be used in the case of strings but with a different scenario. Consider the below code which stores the string while space is encountered.

```c
#include<stdio.h>
void main ()
{
    char s[20];
    printf("Enter the string?");
    scanf("%s",s);

    printf("You entered %s",s);
}
```

Output

```
Enter the string?javatpoint is the best
You entered javatpoint
```

```c
#include<stdio.h>
void main ()
{
    char s[20];
    printf("Enter the string?");
    scanf("%[^\n]s",s);
    printf("You entered %s",s);
}
```

Output

```
Enter the string? javatpoint is the best
You entered javatpoint is the best
```

String In C-Language

In C, strings are handled as arrays of characters ending with a null character (`'\0'`). There are no built-in string types in C; instead, strings are managed using arrays and standard library functions. Here's a brief overview and examples of working with strings in C.

Working with Strings in C

1. Declaring and Initializing Strings

You can declare and initialize strings in C using character arrays or string literals.

```
#include <stdio.h>

int main() {
    // Declaring and initializing a string
    char str1[] = "Hello, World!";

    // Declaring a string with specific size
    char str2[20] = "Hello";

    printf("str1: %s\n", str1);
    printf("str2: %s\n", str2);

    return 0;
}
```

Output:

```
str1: Hello, World!
str2: Hello
```

2. Basic String Operations

You can perform various operations on strings using standard library functions from **`<string.h>`**.

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

int main() {
    char str1[50] = "Hello";
    char str2[] = "World";
    char str3[50];

    // Concatenate str1 and str2
    strcat(str1, " ");
    strcat(str1, str2);
    printf("Concatenated String: %s\n", str1);
    // Copy str1 to str3
    strcpy(str3, str1);
    printf("Copied String: %s\n", str3);

    // Find length of str1
    int length = strlen(str1);
    printf("Length of str1: %d\n", length);

    // Compare two strings
    int cmp = strcmp(str1, "Hello World");
    printf("Comparison result: %d\n", cmp);

    return 0;
}
```

Output:

```
Concatenated String: Hello World
Copied String: Hello World
Length of str1: 11
Comparison result: 0
```

3. Reading and Writing Strings

Strings can be read from the user and printed using standard I/O functions.

```
#include <stdio.h>

int main() {
    char str[100];

    // Reading a string from user input
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);   // Using fgets to read a line of input
    printf("You entered: %s", str);
    return 0;
}
```

Output Example:

```
Enter a string: Hello, C Programming!
You entered: Hello, C Programming!
```

Key Points

- **String Declaration**: Strings in C are arrays of characters with a null terminator.
- **Standard Library Functions**: Functions like `strcat`, `strcpy`, `strlen`, and `strcmp` are used for string manipulation.
- **Input/Output**: Use functions like `fgets` for safe input and `printf` for outputting strings.

Structure union in C-Language

Structures (`struct`)

A structure is a user-defined data type that groups related variables of different types into a single unit. Each member of a structure has its own memory location.

Example Code

```c
#include <stdio.h>
struct Person {
    char name[50];
    int age;
    float height;
};
int main() {
    // Declare and initialize a structure variable
    struct Person person1 = {"Alice", 30, 5.7};

    // Access and print the members of the structure
    printf("Name: %s\n", person1.name);
    printf("Age: %d\n", person1.age);
    printf("Height: %.2f\n", person1.height);

    return 0;
}
```

Output:

```
Name: Alice
Age: 30
Height: 5.70
```

Unions (`union`)

A union is a user-defined data type that allows storing different data types in the same memory location. Only one member of a union can be accessed at a time. All members share the same memory space.

Example Code

```c
#include <stdio.h>
union Data {
    int intValue;
    float floatValue;
    char charValue;
};

int main() {
    // Declare and initialize a union variable
    union Data data;

    // Set an integer value
    data.intValue = 10;

    printf("Integer value: %d\n", data.intValue);

    // Set a float value (overwrites the integer value)
    data.floatValue = 5.75;

    printf("Float value: %.2f\n", data.floatValue);

    // Set a char value (overwrites the float value)
    data.charValue = 'A';

    printf("Char value: %c\n", data.charValue);

    return 0;
}
```

Output:

```
Integer value: 10
Float value: 5.75
Char value: A
```

Key Differences

- **Memory Allocation**:
  - **Structure**: Each member has its own memory location. The total size of the structure is the sum of the sizes of its members.
  - **Union**: All members share the same memory location. The total size of the union is the size of its largest member.
- **Access**:
  - **Structure**: All members can be accessed and used simultaneously.
  - **Union**: Only one member can be accessed at a time. Changing one member affects the others.

Usage Scenarios

- **Structures** are typically used when you need to group related data together and access all the data at once. For example, you might use a structure to represent a `Person`, `Book`, or `Car` with multiple attributes.
- **Unions** are useful when you need to store different data types in the same memory location but only use one at a time. For example, a union could be used in a variant record, where different types of data are stored but only one type is relevant at a given time.

Macros

In C, macros are a feature of the preprocessor that allow you to define symbolic names or functions that are replaced by specific values or code snippets before the actual compilation process. They are defined using the `#define` directive and are commonly used to simplify code, enhance readability, and manage constants or repetitive code.

Types of Macros in C

1. **Object-like Macros**: These are used to define constants or symbolic names.
2. **Function-like Macros**: These define code snippets that take arguments, resembling functions but with text substitution.

Object-like Macros

Object-like macros are used to create constants or simple symbolic names. They are replaced by their defined value or code snippet wherever they appear in the code.

Example Code
```
#include <stdio.h>
#define PI 3.14159
#define MAX_LENGTH 100

int main() {

    printf("Value of PI: %.5f\n", PI);

    printf("Maximum length: %d\n", MAX_LENGTH);
    return 0;
}
```

Output:

```
Value of PI: 3.14159
Maximum length: 100
```

Function-like Macros

Function-like macros are used to define code snippets that can take arguments. They are replaced by the specified code with the arguments substituted.

```
#include <stdio.h>
#define SQUARE(x) ((x) * (x))
#define MAX(a, b) ((a) > (b) ? (a) : (b))

int main() {
    int num = 5;
    int result = SQUARE(num); // Expands to (5 * 5)

    int a = 10, b = 20;
    int max = MAX(a, b); // Expands to ((10) > (20) ? (10) : (20))

    printf("Square of %d: %d\n", num, result);
    printf("Maximum of %d and %d: %d\n", a, b, max);
    return 0;
}
```

Output:

```
Square of 5: 25
Maximum of 10 and 20: 20
```

Key Points

1. **Syntax**:
   - Object-like Macro: `#define NAME value`
   - Function-like Macro: `#define MACRO_NAME(parameters) (expression)`
2. **Substitution**:
   - Macros are replaced by their definition before compilation. This is done via textual substitution.
3. **Parentheses**:
   - For function-like macros, it is good practice to use parentheses around the macro parameters and in the macro body to avoid unintended precedence issues.
4. **Scope**:
   - Macros are global and are valid throughout the file where they are defined. They do not respect scope rules as functions do.
5. **Debugging**:
   - Macros can sometimes make debugging difficult because they do not have a type and are replaced before compilation. Care should be taken to avoid complex macros or those that can lead to unexpected behavior.

Example of Complex Macro Issues

```
#include <stdio.h>

#define ADD(a, b) a + b
#define SAFE_ADD(a, b) ((a) + (b))

int main() {
    int result1 = ADD(5, 3 * 2);     // Expands to 5 + 3 * 2 -> 5 + 6 -> 11
    int result2 = SAFE_ADD(5, 3 * 2); // Expands to (5) + (3 * 2) -> 5 + 6 ->
11

    printf("Result1: %d\n", result1);
    printf("Result2: %d\n", result2);

    return 0;
}
```

Output:

```
Result1: 11
Result2: 11
```

Storage Class

In C, storage classes define the scope, visibility, and lifetime of variables and functions. They influence how and where variables are stored and how long they persist. C provides several storage classes:

1. **Automatic Storage Class (`auto`)**
2. **Static Storage Class (`static`)**
3. **External Storage Class (`extern`)**
4. **Register Storage Class (`register`)**

1. Automatic Storage Class (`auto`)

- **Default Behavior**: Variables declared inside a function are automatically of type `auto`, meaning they are local to that function and are created when the function is called and destroyed when it exits.
- **Visibility**: Local to the block or function in which it is declared.
- **Lifetime**: Exists only during the execution of the function or block.

Example Code

```
#include <stdio.h>

void function() {
    auto int localVar = 10; // 'auto' is optional here
    printf("Inside function: %d\n", localVar);
}

int main() {
    function();
    return 0;
}
```

Output:

```
Inside function: 10
```

2. Static Storage Class (`static`)

- **Behavior**: Variables declared with `static` retain their value between function calls. The variable is initialized only once and retains its value throughout the lifetime of the program.
- **Visibility**: Restricted to the function or file where it is declared.
- **Lifetime**: Exists for the lifetime of the program.

Example Code

```
#include <stdio.h>

void counter() {
    static int count = 0; // Retains its value between function calls
    count++;
    printf("Counter: %d\n", count);
}

int main() {
    counter();
    counter();
    counter();
    return 0;
}
```

Output:

```
Counter: 1
Counter: 2
Counter: 3
```

3. External Storage Class (`extern`)

- **Behavior**: The `extern` keyword is used to declare a variable or function that is defined in another file. It is used to access global variables and functions defined outside the current file.
- **Visibility**: Global; can be accessed by any function or file.
- **Lifetime**: Exists for the lifetime of the program.

Example Code

File1.c:

```
#include <stdio.h>

int globalVar = 100; // Global variable definition

void display() {
    printf("Global Variable: %d\n", globalVar);
}
```

File2.c:

```
#include <stdio.h>

extern int globalVar; // Declaration of the global variable

void modify() {
    globalVar = 200;
}

int main() {
    modify();
    display(); // Access the global variable
    return 0;
}
```

Output (after linking File1.c and File2.c):

```
Global Variable: 200
```

4. Register Storage Class (`register`)

- **Behavior**: `register` suggests that the variable be stored in a CPU register instead of RAM for faster access. The actual placement is implementation-dependent.
- **Visibility**: Local to the block or function in which it is declared.
- **Lifetime**: Exists only during the execution of the function or block.
- **Note**: `register` variables cannot be used with the `&` operator (address-of operator) because they may not have a memory address.

Example Code

```c
#include <stdio.h>

void function() {
    register int i;

    for (i = 0; i < 5; i++) {

        printf("i: %d\n", i);
    }
}

int main() {

    function();

    return 0;
}
```

Output:

```
i: 0
i: 1
i: 2
i: 3
i: 4
```

Summary

- **`auto`**: Default storage class for local variables; local scope and automatic lifetime.
- **`static`**: Retains the value between function calls; limited scope and extended lifetime.

- **`extern`**: Used to declare variables or functions defined in other files; global scope and extended lifetime.
- **`register`**: Suggests storing variables in a CPU register for faster access; limited scope and automatic lifetime.

1. Solve & Practice Question 100+