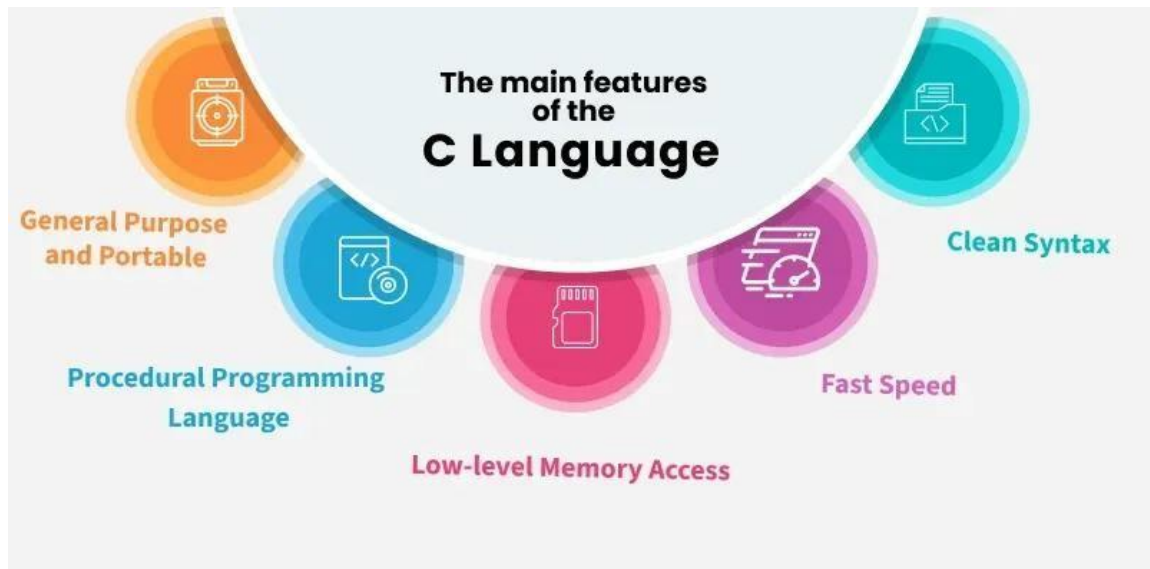


C Language Introduction

C is a general-purpose procedural programming language initially developed by **Dennis Ritchie** in **1972** at Bell Laboratories of AT&T Labs. It was mainly created as a system programming language to write the **UNIX operating system**.



Why Learn C?

- C is considered **mother of all programming languages** as many later languages like Java, PHP and JavaScript have borrowed syntax/features directly or indirectly from the C.
- If a person learns C programming first, it helps to learn any modern programming language as it provides a deeper **understanding of the fundamentals of programming** and underlying architecture of the operating system like pointers, working with memory locations etc.
- C is widely used in **operating systems**, embedded systems, compilers, databases, networking, game engines, and real-time systems for its efficiency to work in low resource environment and hardware-level support.

This simple program demonstrates the basic structure of a C program.

It will also help us understand the basic syntax of a C program. Code:

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

```

    // This prints "Hello World" printf("Hello World");
    return 0;
}

```

Output Hello World

Structure of the C program

After the above discussion, we can formally assess the basic structure of a C program. By structure, it is meant that any program can be written in this structure only. Writing a C program in any other structure will lead to a Compilation Error. The structure of a C program is as follows:

	1	<i>#include <stdio.h></i>	Header
	2	<i>int main(void)</i>	Main
BODY	3	<i>{</i>	
	4	<i>// This prints "Hello World"</i>	Comment
	5	<i>printf("Hello World");</i>	Statement
	6	<i>return 0;</i>	Return
	7	<i>}</i>	

Header Files Inclusion - Line 1 [*#include <stdio.h>*]

The first component is the Header files in a C program. A header file is a file with extension .h which contains C function declarations and macro definitions to be shared between several source files. All lines that start with # are processed by a preprocessor which is a program invoked by the compiler. In the above example, the preprocessor copies the preprocessed code of stdio.h to our file.

The .h files are called header files in C.

Some of the C Header files:

- `stddef.h` - Defines several useful types and macros.
- `stdint.h` - Defines exact width integer types.
- `stdio.h` - Defines core input and output functions
- `stdlib.h` - Defines numeric conversion functions, pseudo-random number generator, and memory allocation

- `string.h` - Defines string handling functions `math.h` - Defines common mathematical functions.

Main Method Declaration - Line 2 [`int main()`]

The next part of a C program is the [main\(\) function](#). It is the entry point of a C program and the execution typically begins with the first line of the `main()`. The empty brackets indicate that the main doesn't take any parameter (See [this](#) for more details). The `int` that was written before the `main` indicates the return type of `main()`. The value returned by the main indicates the status of program termination.

Body of Main Method - Line 3 to Line 6 [enclosed in `{}`]

The body of the main method in the C program refers to statements that are a part of the main function. It can be anything like manipulations, searching, sorting, printing, etc. A pair of curly brackets define the body of a function. All functions must start and end with curly brackets.

Comment - Line 7[`// This prints "Hello World"`]

The [comments](#) are used for the documentation of the code or to add notes in your program that are ignored by the compiler and are not the part of executable program .

Statement - Line 4 [`printf("Hello World");`]

Statements are the instructions given to the compiler. In C, a statement is always terminated by a **semicolon (;)**. In this particular case, we use [printf\(\)](#) function to instruct the compiler to display "Hello World" text on the screen.

Return Statement - Line 5 [`return 0;`]

The last part of any C function is the [return](#) statement. The return statement refers to the return values from a function. This return statement and return value depend upon the return type of the function. The return statement in our program returns the value from `main()`. The returned value may be used by an operating system to know the termination status of your program. The value 0 typically means successful termination.

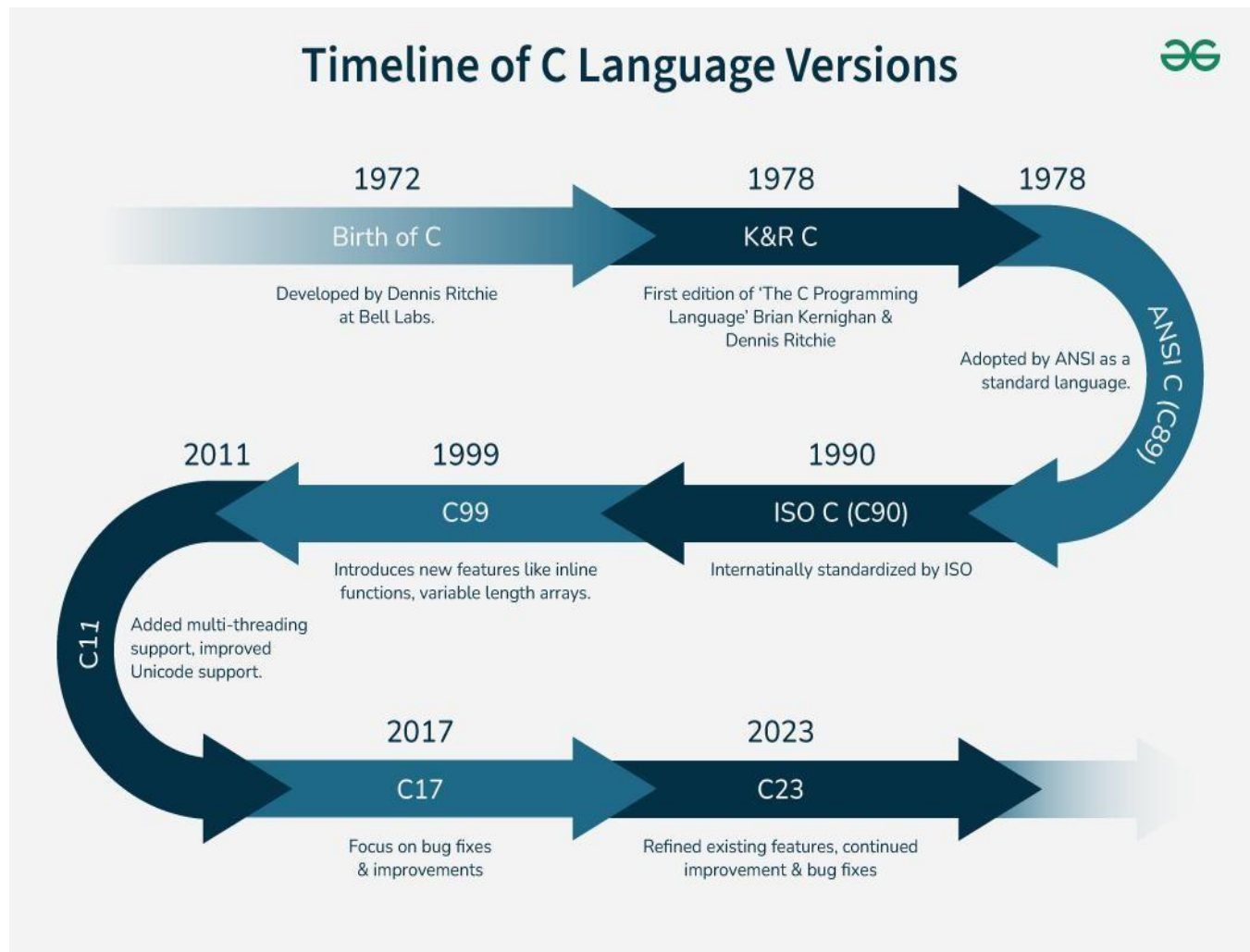
Application of C Language

C language is being used since the early days of computer programming and still is used in wide variety of applications such as:

- C is used to develop core components of **operating systems** such as Windows, Linux, and macOS.
- C is applied to program **embedded systems** in small devices such as washing machines, microwave ovens, and printers.
- C is utilized to create efficient and quick **game engines**. For example, the Doom video game engine was implemented using C.
- C is employed to construct **compilers, assemblers, and interpreters**. For example, the CPython interpreter is written partially using C.
- C is applied to develop efficient **database engines**. The MySQL database software is implemented using C and C++.
- C is employed to create **programs** for devices and sensors of **Internet of Things (IoT)**. A common example is a house automation system comprising temperature sensors and controllers that is often prepared with C.
- C is employed for creating **lightweight and speedy desktop applications**. The widely used text editor Notepad++ employs C for performance-sensitive sections.

History Of C

C is a **general-purpose** procedural programming language developed by Dennis Ritchie in 1972 at Bell Labs to build the **UNIX** operating system. It provides **low-level memory** access, high performance, and portability, making it ideal for system programming. Over the years, C has evolved through standards like **ANSI C**, C99, C11, and C23, adding modern features while retaining its simplicity. [Read more about history here.](#)



Features of C Languages

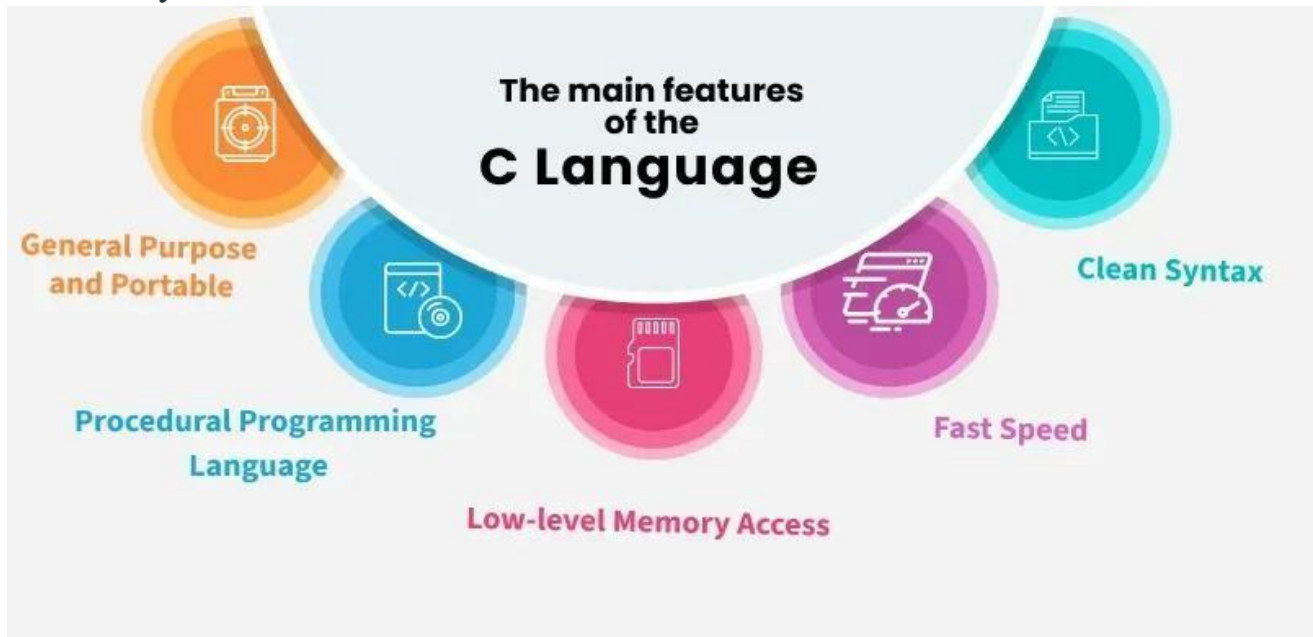
The main features of C language include low-level access to memory, a simple set of keywords, and a clean style, these features make C language suitable for system programming like an operating system or compiler development.

What are the Most Important Features of C Language?

Here are some of the most important features of the C language:

1. Procedural Language
2. Fast and Efficient
3. Modularity
4. Statically Type

5. General-Purpose Language
6. Rich set of built-in Operators
7. Libraries with Rich Functions
8. Middle-Level Language
9. Portability
10. Easy to Extend



Let discuss these features one by one:

1. Procedural Language

In a procedural language like C step by step, predefined instructions are carried out. C program may contain more than one function to perform a particular task. New people to programming will think that this is the only way a particular programming language works. There are other programming paradigms as well in the programming world. Most of the commonly used paradigm is an object-oriented programming language.

2. Fast and Efficient

Newer languages like Java, python offer more features than C programming language but due to additional processing in these languages, their performance rate gets down effectively. C programming language as the middle-level language provides programmers access to direct manipulation with the computer hardware but higher-level languages do not allow this. That's one of the reasons C language is considered the first

choice to start learning programming languages. It's fast because statically typed languages are faster than dynamically typed languages.

3. Modularity

The concept of storing C programming language code in the form of libraries for further future uses is known as modularity. This programming language can do very little on its own most of its power is held by its libraries. C language has its own [library](#) to solve common problems.

4. Statically Type

C programming language is a [statically typed language](#). Meaning the type of variable is checked at the time of compilation but not at run time. This means each time a programmer types a program they have to mention the type of variables used.

5. General-Purpose Language

From system programming to photo editing software, the C programming language is used in various applications. Some of the common applications where it's used are as follows:

- [Operating systems](#): Windows, [Linux](#), iOS, [Android](#), macOS
- [Databases](#): PostgreSQL, Oracle, [MySQL](#), MS SQL Server, etc.

6. Rich set of built-in Operators

It is a diversified language with a rich set of builtin [operators](#) which are used in writing complex or simplified C programs.

7. Libraries with Rich Functions

Robust libraries and [functions in C](#) help even a beginner coder to code with ease.

8. Middle-Level Language

As it is a middle-level language so it has the combined form of both capabilities of assembly language and features of the [high-level language](#).

9. Portability

C language is lavishly portable as programs that are written in C language can run and compile on any system with either no or small changes.

10. Easy to Extend

Programs written in C language can be extended means when a program is already written in it then some more features and operations can be added to it.

Tokens in C

In C programming, tokens are the smallest units in a program that have meaningful representations. Tokens are the building blocks of a C program, and they are recognized by the C compiler to form valid expressions and statements. Tokens can be classified into various categories, each with specific roles in the program.

Types of Tokens in C



The tokens of C language can be classified into six types based on the functions they are used to perform. The types of C tokens are as follows:

Table of Content

Punctuators

Keywords

Strings

Operators

Identifiers

Constants

1. Punctuators

The following special symbols are used in C having some special meaning and thus, cannot be used for some other purpose.

Some of these are listed below:

- **Brackets[]**: Opening and closing brackets are used as array element references. These indicate single and multidimensional subscripts.
- **Parentheses()**: These special symbols are used to indicate function calls and function parameters.
- **Braces{}:** These opening and ending curly braces mark the start and end of a block of code containing more than one executable statement.
- **Comma (,)**: It is used to separate more than one statement like for separating parameters in function calls.
- **Colon(:)**: It is an operator that essentially invokes something called an initialization list.
- **Semicolon(;)**: It is known as a statement terminator. It indicates the end of one logical entity. That's why each individual statement must be ended with a semicolon.
- **Asterisk (*)**: It is used to create a pointer variable and for the multiplication of variables.
- **Assignment operator(=)**: It is used to assign values and for logical operation validation.
- **Pre-processor (#)**: The preprocessor is a macro processor that is used automatically by the compiler to transform your program before actual compilation.
- **Dot (.)**: Used to access members of a structure or union. **Tilde(~)**: Bitwise One's Complement Operator.

Example:

```
#include <stdio.h>

int main() {

    // '\n' is a special symbol that //
    // represents a newline printf("Hello,
    // \n World!"); return 0;
}
```

Output

```
Hello,
World!
```

2. Keywords

Keywords are reserved words that have predefined meanings in C. These cannot be used as identifiers (variable names, function names, etc.). Keywords define the structure and behavior of the program C language supports **32** keywords such as int, for, if, ... etc.

Example:

```
#include <stdio.h>

int main() {

    // 'int' is a keyword used to define
    // variable type int x
    // = 5; printf("%d", x);

    // 'return' is a keyword used to exit
    // main function
    return 0;
}
```

Output

```
5
```

Note: The number of keywords may change depending on the version of C you are using. For example, keywords present in

ANSI C are 32 while in C11, it was increased to 44. Moreover, in the latest C23, it is increased to around 54.

3. Strings

Strings are nothing but an array of characters ended with a null character ('\0'). This null character indicates the end of the string. Strings are always enclosed in double quotes. Whereas a character is enclosed in single quotes in C and C++.

Examples:

```
#include <stdio.h>

int main() {

    // "Hello, World!" is a string literal char
    str[] = "Hello, World!";
    printf("%s", str);
    return 0; }
```

Output

```
Hello, World!
```

4. Operators

Operators are symbols that trigger an action when applied to C variables and other objects. The data items on which operators act are called operands. **Example:**

```
#include <stdio.h>

int main() {
    int a = 10, b = 5;

    // '+' is an arithmetic operator used
    // for addition int
    sum = a + b;
    printf("%d", sum);
    return 0;
}
```

Output

5. Identifiers

15

Identifiers are names given to variables, functions, arrays, and other user-defined items. They must begin with a letter (a-z, A-Z) or an underscore (_) and can be followed by letters, digits (0-9), and underscores. **Example:**

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

```
int main() {  
  
    // 'num' is an identifier used to name  
    // a variable int num  
    = 10; printf("%d",  
    num); return 0;  
}
```

Output

6. Constants

10

Constants are fixed values used in a C program. These values do not change during the execution of the program. Constants can be integers, floating-point numbers, characters, or strings. **Examples:**

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

```
int main() {  
  
    // 'MAX_VALUE' is a constant that holds  
    // a fixed value const int  
    MAX_VALUE = 100;  
    printf("%d", MAX_VALUE);  
    return 0;  
}
```

Output

C Identifiers

In C programming, identifiers are the names used to identify variables, functions, arrays, structures, or any other userdefined items. It is a name that uniquely identifies a program element and can be used to refer to it later in the program.

Example:

```
//Creating a variable int val = 10;
//Creating a function void func()
{}
```

In the above code snippet, "**val**" and "**func**" are identifiers.

Rules for Naming Identifiers in C

A programmer must follow a set of rules to create an identifier in C:

- Identifier can contain following characters:
 - Uppercase (A-Z) and lowercase (a-z) alphabets. ◦ Numeric digits (0-9).
 - Underscore (_).
- The first character of an identifier must be a letter or an underscore.
- Identifiers are case-sensitive.
- Identifiers cannot be keywords in C (such as int, return, if, while etc.).

The below image and table show some **valid** and **invalid** identifiers in C language.

Valid names	Invalid names
_srujan, srujan_poojari, srujan812, srujan_812	<p>srujan poojari <i>It contains a whitespace in between srujan and poojari.</i></p> <p>13srujan <i>It starts with a number so we cannot declare it as a variable.</i></p> <p>goto, for, switch <i>We can't declare them as variables because they are keywords of C language.</i></p>

Example

The following code examples demonstrate the creation and usage of identifiers in C:

Creating an Identifier for a Variable

```
#include <stdio.h>

int main() {

    // Creating an integer variable and // assign it
    the identifier 'var' int var;

    // Assigning value to the variable // using
    assigned name var = 10;
    // Referring to same variable using
    // assigned name printf("%d", var);
    return 0;

}
```

Output

10

If you are not familiar with [variables](#) and [functions](#), don't worry! We will discuss them in the later sections.

Creating an Identifier for a Function

```
#include <stdio.h>

// Function declaration which contains user
// defined identifier as its name int
sum(int a, int b) { return a + b;
}

int main() {

    // Calling the function using its name
    printf("%d", sum(10, 20)); return 0;
}
```

Output

30

Naming Conventions

In C programming, naming conventions are not strict rules but are commonly followed suggestions by the programming community for identifiers to improve readability and understanding of code. Below are some conventions that are commonly used:

For Variables:

- Use camelCase for variable names (e.g., frequencyCount, personName).
- Constants can use UPPER_SNAKE_CASE (e.g., MAX_SIZE, PI).
- Start variable names with a lowercase letter.
- Use descriptive and meaningful names.

For Functions:

- Use camelCase for function names (e.g., getName(), countFrequency()).
- Function names should generally be verbs or verb phrases that describe the action.

For Structures:

- Use PascalCase for structure names (e.g., Car, Person).
- Structure names should be nouns or noun phrases.

Keywords in C

Keywords are predefined or reserved words that have special meanings to the compiler. These are part of the syntax and cannot be used as identifiers in the program. A list of keywords in C or reserved words in the C programming language are mentioned below:

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
auto	break	case	char	const	continue	default	do
		enum	extern				continue

struc t	switc h	typed ef	unio n	unsign ed	void	volati le	whil e
---------	------------	-------------	--------	--------------	------	--------------	-----------

We cannot use these keywords as identifiers (such as variable names, function names, or struct names). The compiler will throw an **error** if we try to do so.

Example:

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

```
int main() { int return =
    10;
    printf("%d\n", return);
    return 0;
}
```

Output

```
./Solution.c: In function 'main':
```

```
./Solution.c:4:9: error: expected identifier or '(' before 'return'
```

```
    int return = 10;
```

```
        ^
```

```
./Solution.c:5:20: error: expected expression before
```

```
'return' printf("%d\n", return);
```

```
        ^
```

Let's categorize all keywords based on context for a more clear understanding.

Category	Keywords
	char , int , float , double , void , short, long , signed , unsigned

Data Type Keywords	
Operator & Utility Keywords	<u>sizeof</u> , <u>return</u> , <u>goto</u> , <u>typedef</u>
Control Flow Keywords	<u>if</u> , <u>else</u> , <u>switch</u> , <u>case</u> , <u>default</u> , <u>for</u> , <u>while</u> , <u>do</u> , <u>break</u> , <u>continue</u>
Storage Class	<u>auto</u> , <u>register</u> , <u>static</u> , <u>extern</u>
Category	<i>Keywords</i>
Keywords	
Type Qualifiers	<u>const</u> , <u>volatile</u>
User Defined Types	<u>struct</u> , <u>union</u> , <u>enum</u>

Difference Between Keywords and Identifiers

Keywords	Identifiers
-----------------	--------------------

Reserved Words in C that have a specific meaning and use in the syntax	Names given to variables, functions, structs, etc.
Cannot be used as variable names.	Can be used as variable names (if not a keyword).
Examples: int, return, if, while	Examples: x, total, count
Part of the C language grammar.	User-defined, meaningful names in the code.
Keywords	<i>Identifiers</i>
<i>Cannot be redefined or repurposed.</i>	Can be defined redefined and reused as needed.

Difference Between Variables and Keywords

Variables	<i>Keywords</i>
------------------	-----------------

Used to store data and manipulate data	Reserved words used to define structure and control flow
Created by the user using valid identifiers	Defined by C standard and recognized by the compiler
Examples: age, total, marks	Examples: for, while, int, return
Can hold values and be used in operations	Used to define the structure of C code.
Naming is flexible within identifier rules	Names are fixed and cannot be altered or used differently

Differences between the keywords and identifiers:

Parameters	<i>Keywords</i>	<i>Identifiers</i>
-------------------	------------------------	---------------------------

Definition	Keywords are predefined word that gets reserved for working program that have special meaning and cannot get used anywhere else.	Identifiers are the values used to define different programming items such as variables, integers, structures, unions and others and mostly have an alphabetic character.
Use	Specify the type/kind of entity.	Identify the name of a particular entity.
	It always starts with a lowercase letter.	First character can be a uppercase, lowercase letter or underscore.
Rules of	A keyword should be	An identifier can be in
Parameters	Keywords	Identifiers

Definition	in lower case and can only contains alphabetical characters.	upper case or lower case and can consist of alphabetical characters, digits and underscores.
Purpose	They help to identify a specific property that exists within a computer language.	They help to locate the name of the entity that gets defined along with a keyword.
Examples	int, char, if, while, do, class etc.	Test, count1, high_speed, etc.

Code Illustration

The below example illustrate the different purpose of the keywords and identifiers in C:

```
#include <stdio.h>

// 'main' is also an identifier //
although it is predefined int
main() {

    // Example of a keyword: int
    // Here, age is identifier
    int age = 25;

    printf("Age: %d\n", age);
```

```
    // return is a keyword that exits function return 0;  
}
```

Output

```
Age: 25
```

Explanation: In the above code, **age** is an identifier used to name a variable. The type of variable is integer that is specified using **int** keyword as it already is defined in C to indicate that the variable will store integer data.

C Variables

A **variable in C** is a named piece of memory which is used to store data and access it whenever required. It allows us to use the memory without having to memorize the exact memory address.

To create a variable in C, we have to specify a **name** and the **type of data** it is going to store in the syntax. `data_type name;`

C provides different [data types](#) that can store almost all kinds of data. For example, int, char, float, double, etc.

```
int num;  
char letter; float  
decimal;
```

In C, every variable must be declared before it is used. We can also declare multiple variables of same data type in a single statement by separating them using comma as shown:

```
data_type name1, name2, name3, ...;
```

Rules for Naming Variables in C

We can assign any name to a C variable as long as it follows the following rules:

- A variable name must only contain **letters**, **digits**, and **underscores**.
- It must **start with an alphabet** or an **underscore** only. It cannot start with a digit.

- **No white space** is allowed within the variable name.
- A variable name must **not** be any reserved word or **keyword**.
- The name must be unique in the program.

C Variable Initialization

Once the variable is declared, we can store useful values in it.

The first value we store is called initial value and the process is called **Initialization**. It is done using [assignment operator \(=\)](#).

```
int num;  
num = 3;
```

It is important to initialize a variable because a C variable only contains garbage value when it is declared. We can also initialize a variable along with declaration.

```
int num = 3;
```

Note: It is compulsory that the values assigned to the variables should be of the same data type as specified in the declaration.

Accessing Variables

The data stored inside a C variable can be easily accessed by using the variable's name.

Example:

```
#include <stdio.h>

int main() {

    // Create integer variable int num =
    3;
    // Access the value stored in
    // variable printf("%d",
    num);
    return 0;
}
```

Output

3

Changing Stored Values

We can also update the value of a variable with a new value whenever needed by using the assignment operator =.

Example:

```
#include <stdio.h>

int main() {

    // Create integer variable
    int n = 3;
    // Change the stored data n = 22;

    // Access the value stored in
    // variable printf("%d", n);
    return 0;
}
```

Output

22

How to use variables in C?

Variables act as name for memory locations that stores some value. It is valid to use the variable wherever it is valid to use its value. It means that a variable name can be used anywhere as a substitute in place of the value it stores. **Example:** An integer variable can be used in a mathematical expression in place of numeric values.

```

#include <stdio.h> int
main() {
    // Expression that uses values int sum1 =
    20 + 40;

    // Defining variables
    int a = 20, b = 40;

    // Expression that uses variables int sum2 =
    a + b;

    printf("%d\n%d", sum1, sum2); return 0;
}

```

Output

```

60
60

```

Memory Allocation of C Variables

When a variable is **declared**, the compiler is told that the variable with the given name and type exists in the program. But no memory is allocated to it yet. Memory is allocated when the variable is **defined**.

Most programming languages like C generally declare and define a variable in the single step. For example, in the above part where we create a variable, variable is declared and defined in a single statement.

The size of memory assigned for variables depends on the type of variable.

We can check the size of the variables using [sizeof operator](#).

Example:

```

#include <stdio.h>

int main() { int
num = 22;

```



```
    // Finding size of num
    printf("%d bytes", sizeof(num));
    return 0;
}
```

Output

4 bytes

Variables are also stored in different parts of the memory based on their [storage classes](#). **Scope of Variables in C**

We have told that a variable can be accessed anywhere once it is declared, but it is partially true. A variable can be accessed using its name anywhere in a specific region of the program called its [scope](#). It is the region of the program where the name assigned to the variable is valid.

A scope is generally the area inside the **{ curly braces**.

Example:

A **variable in C** is a named piece of memory which is used to store data and access it whenever required. It allows us to use the memory without having to memorize the exact memory address.

To create a variable in C, we have to specify a **name** and the **type of data** it is going to store in the syntax.

```
data_type name;
```

C provides different [data types](#) that can store almost all kinds of data. For example, int, char, float, double, etc.

```
int num; char letter;
float decimal;
```

In C, every variable must be declared before it is used. We can also declare multiple variables of same data type in a single statement by separating them using comma as shown:

```
data_type name1, name2, name3, ...;
```

Rules for Naming Variables in C

We can assign any name to a C variable as long as it follows the following rules:

- A variable name must only contain **letters**, **digits**, and **underscores**.

- It must **start with an alphabet** or an **underscore** only. It cannot start with a digit.
- **No white space** is allowed within the variable name.
- A variable name must **not** be any reserved word or **keyword**.
- The name must be unique in the program.

C Variable Initialization

Once the variable is declared, we can store useful values in it. The first value we store is called initial value and the process is called **Initialization**. It is done using assignment operator (=).

```
int num; num
= 3;
```

It is important to initialize a variable because a C variable only contains garbage value when it is declared. We can also initialize a variable along with declaration.

```
int num = 3;
```

Note: It is compulsory that the values assigned to the variables should be of the same data type as specified in the declaration.

Accessing Variables

The data stored inside a C variable can be easily accessed by using the variable's name. **Example:**

```
#include <stdio.h>

int main() {
    // Create integer variable int
    num = 3;

    // Access the value stored in
    // variable
    printf("%d", num);
    return 0;
}
```

Output

```
3
```

Changing Stored Values

We can also update the value of a variable with a new value whenever needed by using the assignment operator =. **Example:**

```
#include <stdio.h>

int main() {

    // Create integer variable int n
    = 3;

    // Change the stored data n =
    22;

    // Access the value stored in
    // variable
    printf("%d", n);
    return 0;
}
```

Output

How to use variables in C?

22

Variables act as name for memory locations that stores some value. It is valid to use the variable wherever it is valid to use its value. It means that a variable name can be used anywhere as a substitute in place of the value it stores. **Example:** An integer variable can be used in a mathematical expression in place of numeric values.

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

    // Expression that uses values int
    sum1 = 20 + 40;

    // Defining variables int
    a = 20, b = 40;

    // Expression that uses variables int
    sum2 = a + b;

    printf("%d\n%d", sum1, sum2);
    return 0;
}
```

```
}
```

Output

```
60  
60
```

Memory Allocation of C Variables

When a variable is **declared**, the compiler is told that the variable with the given name and type exists in the program. But no memory is allocated to it yet. Memory is allocated when the variable is **defined**.

Most programming languages like C generally declare and define a variable in the single step. For example, in the above part where we create a variable, variable is declared and defined in a single statement.

The size of memory assigned for variables depends on the type of variable. We can check the size of the variables using sizeof operator. **Example:**

```
#include <stdio.h>  
  
int main() {  
    int num = 22;  
  
    // Finding size of num printf("%d  
    bytes", sizeof(num)); return 0;  
}
```

Output

```
4 bytes
```

Variables are also stored in different parts of the memory based on their storage classes.

Scope of Variables in C

We have told that a variable can be accessed anywhere once it is declared, but it is partially true. A variable can be accessed using its name anywhere in a specific region of the program called its scope. It is the region of the program where the name assigned to the variable is valid.

A scope is generally the area inside the {} curly braces. **Example:**

```
// num cannot be accessed here int main() {
```

```
// num cannot be accessed here {  
    // Variable declaration int num;  
}  
  
// Cannot be accessed here either return 0;  
}
```

Constants in C

In C programming, **const** is a keyword used to declare a variable as constant, meaning its value cannot be changed after it is initialized. It is mainly used to protect variables from being accidentally modified, making the program safer and easier to understand. These constants can be of various types, such as integer, floating-point, string, or character constants. Let's take a look at an example:

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

```
int main() {  
  
    // Defining constant variable  
    const int a = 10;  
    printf("%d", a);  
    return 0;  
}
```

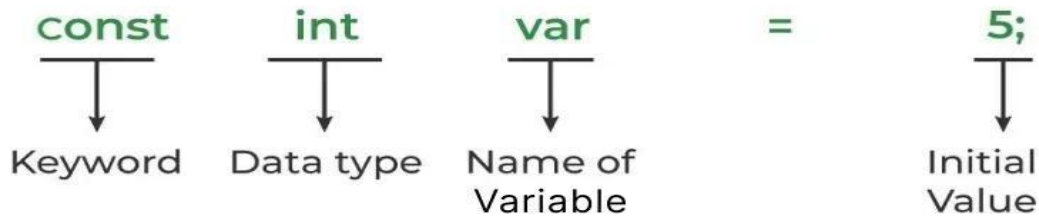
Output

```
10
```

Syntax

We define a constant in C using the **const** keyword. Also known as a const type qualifier, the const keyword is placed at the start of the variable declaration to declare that variable as a constant. **const**data_typevar_name=value;

Constants



Properties of Constant

The important properties of constant variables in C defined using the `const` keyword are as follows:

1. Initialization with Declaration

We can only initialize the constant variable in C at the time of its declaration. If we do not initialize it at the time of declaration, it will store the garbage value that was previously stored in the same memory.

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

```
int main() {  
  
    // Not initializing a constant variable const  
    int a;  
  
    // printing value printf("%d",  
    a);  
    return 0;  
}
```

Output

```
0
```

The garbage value here is 0 but it can be anything depending on your environment.

2. Immutability

The constant variables in C are immutable after its definition, i.e., they can be initialized only once in the whole program. After that, we cannot modify the value stored inside that variable.

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

```
// Declaring a constant variable const
int a;

// Initializing constant variable var after declaration a =
20;
printf("%d", a);
return 0;
}
```

Output

```
In function 'main':
10:9: error: assignment of read-only variable 'a'
10 |     a = 20;
    |     ^
```

Constants Using #define

In C, the **#define directive** can also be used to define symbolic constants that do not require a data type. They are called macros and are replaced by their values at compile time.

Syntax:

#define CONSTANT_NAME value **Example:**

```
#include <stdio.h>
#define PI 3.14
```

```
int main() { printf("%.2f",
    PI);
    return 0;
}
```

Output

```
3.14
```

Real world examples of Const

ATM Daily Withdrawal Limit

-> In banking software, a customer may have a fixed limit of ₹20,000 per day.

```
const int dailyLimit = 20000;
```

Mobile App – Max Login Attempts

-> Apps like WhatsApp or banking apps usually allow only 3 login attempts.

```
const int maxLoginAttempts = 3;
```

Mathematics – Value of Pi (π)

-> The value of π is always the same in mathematical formulas.

```
const float pi = 3.14159;
```

HR System – Maximum Allowed Leave Days

-> A company sets a fixed number of leave days (e.g., 15 days) for all employees.

```
const int maxLeaveDays = 15;
```

Types of Constants in C:

Constants in C are broadly categorized into two main groups:

- **Primary Constants:**

These are fundamental data types that represent single values.

- **Integer Constants:** Whole numbers without fractional parts. They can be:
- **Decimal:** (e.g., 10, -50, 123)
- **Octal:** Prefixed with 0 (e.g., 012 which is 10 in decimal)
- **Hexadecimal:** Prefixed with 0x or 0X (e.g., 0xA which is 10 in decimal, 0xFF)

Example `int decimal_const = 123;`

`int octal_const = 017; // Represents decimal 15 int`

`hex_const = 0xA5; // Represents decimal 165`

- **Real (Floating-Point) Constants:** Numbers with fractional parts, represented using a decimal point or exponential notation. (e.g., 3.14, -0.5, 1.2e-3)

Example `float pi = 3.14159f; double`

`speed_of_light = 2.998e8;`

- **Character Constants:** A single character enclosed in single quotes. (e.g., 'A', 'c', '7') This also includes Backslash Character Constants (escape sequences) like '\n' (newline), '\t' (tab), '\0' (null character).

Example

`char initial = 'J'; char`

`newline = '\n';`

- **String Constants:** A sequence of characters enclosed in double quotes. (e.g., "Hello World", "C Programming")

Example char greeting[] = "Hello, World!";

- **Secondary Constants:**

These are derived data types that can be declared constant.

- **Arrays:** An array declared with const cannot have its elements modified after initialization.
- **Pointers:** A pointer can be declared as constant in two ways:
 - A const pointer, where the pointer itself cannot be changed to point to a different memory location.
 - A pointer to a const value, where the value pointed to cannot be modified through that pointer.
- **Structures:** A struct declared with const cannot have its member values modified.
- **Unions:** Similar to structures, a union declared with const cannot have its member values modified.
- **Enumerations (enums):** Named integer constants defined using the enum keyword. (e.g., enum Color { RED, GREEN, BLUE };)

Operators in C

Operators are the basic components of C programming. They are symbols that represent some kind of operation, such as mathematical, relational, bitwise, conditional, or logical computations, which are to be performed on values or variables.

The values and variables used with operators are called **operands**. **Example:**

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

    // Expression for getting sum int
    sum = 10 + 20;

    printf("%d", sum); return
    0;
}
```

Output

30

In the above expression, '+' is the **addition operator** that tells the compiler to add both of the operands 10 and 20. To dive deeper into how operators are used with data structures, the [C Programming Course Online with Data Structures](#) covers this topic thoroughly.

Unary, Binary and Ternary Operators

On the basis of the number of operands they work on, operators can be classified into three types :

1. **Unary Operators:** Operators that work on single operand. Example:
Increment(++) , Decrement(--)
2. **Binary Operators:** Operators that work on two operands.
Example: Addition (+), Subtraction(-) , Multiplication (*)
3. **Ternary Operators:** Operators that work on three operands.
Example: Conditional Operator(? :)

Types of Operators in C

C language provides a wide range of built in operators that can be classified into 6 types based on their functionality:

Table of Content

[Arithmetic Operators](#)

[Relational Operators](#)

[Logical Operator](#)

[Bitwise Operators](#)

[Assignment Operators](#)

[Other Operators](#)

Arithmetic Operators

The [arithmetic operators](#) are used to perform arithmetic/mathematical operations on operands. There are **9 arithmetic** operators in C language:

Symbol	Operator	Description	Syntax
--------	----------	-------------	--------

+	Plus	Adds two numeric values.	a + b
-	Minus	Subtracts right operand from left operand.	a - b
*	Multiply	Multiply two numeric values.	a * b
/	Divide	Divide two numeric values.	a / b
%	Modulus	Returns the remainder after diving	a % b
Symbol	Operator	Description	<i>Syntax</i>
		the left operand with the right operand.	

+	Unary Plus	Used to specify the positive values.	+a
-	Unary Minus	Flips the sign of the value.	-a
++	Increment	Increases the value of the operand by 1.	a++
--	Decrement	Decreases the value of the operand by 1.	a--

Example of C Arithmetic Operators

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

```
int main() {
```

```
    int a = 25, b = 5;
```

```
    // using operators and printing results
```

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

```
    b = %d\n", a - b); printf("a * b = %d\n",
```

```
    a * b); printf("a / b = %d\n", a / b);
```

```
    printf("a % b = %d\n", a % b); printf("+a
```

```
    = %d\n", +a); printf("-a = %d\n", -a);
```

```
    printf("a++ = %d\n", a++);
```

```
    printf("a-- = %d\n", a--);
```

```
    return 0; }
```

Output

```
a + b = 30
a - b = 20
a * b = 125
a / b = 5
a % b = 0
+a = 25
-a = -25
a++ = 25
a-- = 26
```

Relational Operators

The relational operators in C are used for the comparison of the two operands. All these operators are binary operators that return true or false values as the result of comparison.

These are a total of 6 relational operators in C:

Symbol	Operator	Description	Syntax
--------	----------	-------------	--------

Symbol	Operator	Description	Syntax
<	Less than	Returns true if the left operand is less than the right operand. Else false	a < b

>	Greater than	Returns true if the left operand is greater than the right operand. Else false	a > b
<=	Less than or equal to	Returns true if the left operand is less than or equal to the right operand. Else false	a <= b
>=	Greater than	Returns true	a >= b
Symbol	Operator	Description	<i>Syntax</i>
	or equal to	if the left operand is greater than or equal to right operand. Else false	

==	Equal to	Returns true if both the operands are equal.	a == b
!=	Not equal to	Returns true if both the operands are NOT equal.	a != b

Example of C Relational Operators

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

```
int main() {
    int a = 25, b = 5;

    // using operators and printing results
    printf("a < b : %d\n", a < b); printf("a >
    b : %d\n", a > b); printf("a <= b: %d\n",
    a <= b); printf("a >= b: %d\n", a >= b);
    printf("a == b: %d\n", a == b);
    printf("a != b : %d\n", a != b);
    return 0;
}
```

Output

```
a < b   : 0
a > b   : 1
a <= b : 0
a >= b : 1
a == b : 0
a != b : 1
```

Here, 0 means false and 1 means true.

Logical Operator

Logical Operators are used to combine two or more conditions/constraints or to complement the evaluation of the original condition in consideration. The result of the operation of a logical operator is a Boolean value either **true** or **false**. There are **3** logical operators in C:

Symbol	Operator	Description	<i>Syntax</i>
&&	Logical AND	Returns true if both the operands are true.	a && b
 	Logical OR	Returns true if both or any of the	a b
Symbol	Operator	Description	<i>Syntax</i>
		operand is true.	
!	Logical NOT	Returns true if the operand is false.	!a

Example of Logical Operators in C

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

```
int main() {  
    int a = 25, b = 5;
```



```

// using operators and printing results
printf("a && b : %d\n", a && b); printf("a
|| b : %d\n", a || b);
printf("!a: %d\n", !a);

return 0;
}

```

Output

```

a && b : 1 a || b : 1
!a: 0

```

Bitwise Operators

The Bitwise operators are used to perform bit-level operations on the operands. The operators are first converted to bit-level and then the calculation is performed on the operands.

Note: Mathematical operations such as addition, subtraction, multiplication, etc. can be performed at the bit level for faster processing.

There are 6 bitwise operators in C:

Symbol	Operator	Description	Syntax
&	Bitwise AND	Performs bitby-bit AND operation and returns the result.	a & b
	Bitwise OR	Performs bitby-bit OR operation and returns the result.	a b

\wedge	Bitwise XOR	Performs bitby-bit XOR operation and returns the result.	$a \wedge b$
\sim	Bitwise First Complement	Flips all the set and unset bits on the number.	$\sim a$
Symbol	Operator	Description	Syntax
\ll	Bitwise Leftshift	Shifts bits to the left by a given number of positions; multiplies the number by 2 for each shift.	$a \ll b$
\gg	Bitwise Rightshilft	Shifts bits to the right by a given number of positions; divides the number by 2 for each shift.	$a \gg b$

Example of Bitwise Operators

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

```
int main() {  
    int a = 25, b = 5;  
  
    // using operators and printing results  
    printf("a & b: %d\n", a & b); printf("a |  
    b: %d\n", a | b); printf("a ^ b: %d\n", a ^  
    b); printf("~a: %d\n", ~a); printf("a >>  
    b: %d\n", a >> b); printf("a << b: %d\n",  
    a << b);  
  
    return 0;  
}
```

Output

```
a & b: 1  
a | b: 29  
a ^ b: 28  
~a: -26  
a >> b: 0  
a << b: 800
```

Assignment Operators

Assignment operators are used to assign value to a variable. The left side operand of the assignment operator is a variable and the right side operand of the assignment operator is a value. The value on the right side must be of the same data type as the variable on the left side otherwise the compiler will raise an error. The assignment operators can be combined with some other operators in C to provide multiple operations using single operator. These operators are called compound operators.

In C, there are 11 assignment operators:

Symbol	Operator	Description	Syntax
--------	----------	-------------	--------

=	Simple Assignment	Assign the value of the right operand to the left operand.	a = b
----------	--------------------------	--	--------------

Symbol	Operator	Description	<i>Syntax</i>
+=	Plus and assign	Add the right operand and left operand and assign this value to the left operand.	a += b
-=	Minus and assign	Subtract the right operand and left operand and assign this value to the left operand.	a -= b

*=	Multiply and assign	Multiply the right operand and left operand and assign this value to the left operand.	a *= b
/=	Divide and assign	Divide the left operand	a /= b

Symbol	Operator	Description	<i>Syntax</i>
		with the right operand and assign this value to the left operand.	

%=	Modulus and assign	Assign the remainder in the division of left operand with the right operand to the left operand.	a %= b
&=	AND and assign	Performs bitwise AND and assigns this value to the left operand.	a &= b
 =	OR and assign	Performs bitwise OR and assigns	a = b
Symbol	Operator	Description	<i>Syntax</i>
		this value to the left operand.	

<code>^=</code>	XOR and assign	Performs bitwise XOR and assigns this value to the left operand.	<code>a ^= b</code>
<code>>>=</code>	Rightshift and assign	Performs bitwise Rightshift and assign this value to the left operand.	<code>a >>= b</code>
<code><<=</code>	Leftshift and assign	Performs bitwise Leftshift and assign this value to the left operand.	<code>a <<= b</code>

Example of C Assignment Operators

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

```
int main() {
    int a = 25, b = 5;

    // using operators and printing results
    printf("a = b: %d\n", a = b); printf("a +=
b: %d\n", a += b); printf("a -= b: %d\n",
a -= b); printf("a *= b: %d\n", a *= b);
```

```

printf("a /= b: %d\n", a /= b);
printf("a %%= b: %d\n", a %= b);
printf("a &= b: %d\n", a &= b); printf("a
|= b: %d\n", a |= b); printf("a ^=
b: %d\n", a ^= b); printf("a >>= b: %d\n",
a >>= b); printf("a <<= b: %d\n", a <<=
b);

return 0;
}

```

Output

```

a = b: 5
a += b: 10
a -= b: 5
a *= b: 25
a /= b: 5
a %= b: 0
a &= b: 0
a |= b: 5
a ^= b: 0
a >>= b: 0
a <<= b: 0

```

Other Operators

Apart from the above operators, there are some other operators available in C used to perform some specific tasks.

Some of them are discussed here:

sizeof Operator

- sizeof is much used in the C programming language.
- It is a compile-time unary operator which can be used to compute the size of its operand.
- The result of sizeof is of the unsigned integral type which is usually denoted by `size_t`.
- Basically, the sizeof the operator is used to compute the size of the variable or datatype. **Syntax**

`sizeof (operand)`

Comma Operator (,)

The comma operator (represented by the token) is a binary operator that evaluates its first operand and discards the result, it then evaluates the second operand and returns this value (and type).

The comma operator has the lowest precedence of any C operator. It can act as both operator and separator. **Syntax**

```
operand1 , operand2
```

Conditional Operator (? :)

The conditional operator is the only ternary operator in C++. It is a conditional operator that we can use in place of if..else statements. **Syntax**
expression1 ? Expression2 : Expression3;

Here, **Expression1** is the condition to be evaluated. If the condition(**Expression1**) is **True** then we will execute and return the result of **Expression2** otherwise if the condition(**Expression1**) is **false** then we will execute and return the result of **Expression3**.

dot (.) and arrow (->) Operators

Member operators are used to reference individual members of classes, structures, and unions.

- The dot operator is applied to the actual object.
- The arrow operator is used with a pointer to an object.

Syntax

```
structure variable . member;  
structure pointer -> member;
```

Cast Operators

Casting operators convert one data type to another. For example, `int(2.2000)` would return 2.

A cast is a special operator that forces one data type to be converted into another.

Syntax

(new_type) operand;

addressof (&) and Dereference (*) Operators

Addressof operator & returns the address of a variable and the dereference operator * is a pointer to a variable. For example *var; will pointer to a variable var. **Example of Other C Operators**

// C Program to demonstrate the use of Misc operators

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

```
int main()
{
    // integer variable int num
    = 10; int* add_of_num =
    &num;

    printf("sizeof(num) = %d bytes\n", sizeof(num));
    printf("&num = %p\n", &num); printf("*add_of_num
    = %d\n", *add_of_num); printf("(10 < 5) ? 10 : 20
    = %d\n", (10 < 5) ? 10 : 20);
    printf("(float)num = %f\n", (float)num);

    return 0;
}
```

Output

```
sizeof(num) = 4 bytes
&num = 0x7ffdb58c037c
*add_of_num = 10
(10 < 5) ? 10 : 20 = 20
(float)num = 10.000000
```

Operator Precedence and Associativity

Operator Precedence and Associativity is the concept that decides which operator will be evaluated first in the case when there are multiple operators present in an expression to avoid ambiguity. As, it is very common for a C expression or statement to have multiple operators and in this expression. The below table describes the precedence order and associativity of operators in C. The precedence of the operator decreases from top to bottom.

Precedenc e	Operato r	Description	Associativit y
1	()	Parentheses (function call)	left-to-right
	[]	Brackets (array subscript)	left-to-right
	.	Member selection via object name	left-to-right
	->	Member selection via a pointer	left-to-right
	a++ , a--	Postfix increment/decrement (a is a variable)	left-to-right
2	++a , --a	Prefix increment/decrement (a is a variable)	right-to-left
	+ , -	Unary plus/minus	right-to-left
	! , ~	Logical negation/bitwise complement	right-to-left

Precedenc e	Operato r	Description	Associativit y
-------------	-----------	-------------	----------------

	(type)	Cast (convert value to temporary value of type)	right-to-left
	*	Dereference	right-to-left
	&	Address (of operand)	right-to-left
	sizeof	Determine size in bytes on this implementation	right-to-left
3	*, / , %	Multiplication/division/modulus	left-to-right
4	+, -	Addition/subtraction	left-to-right
5	<<, >>	Bitwise shift left, Bitwise shift right	left-to-right
6	<, <=	Relational less than/less than or equal to	left-to-right
	>, >=	Relational greater than/greater than or equal to	left-to-right
7	==, !=	Relational is equal to/is not	left-to-right

Precedence	Operator	Description	Associativity
		equal to	
8	&	Bitwise AND	left-to-right
9	^	Bitwise XOR	left-to-right
10		Bitwise OR	left-to-right
11	&&	Logical AND	left-to-right
12		Logical OR	left-to-right
13	?:	Ternary conditional	right-to-left
14	=	Assignment	right-to-left
	+=, -=	Addition/subtraction assignment	right-to-left
	*=, /=	Multiplication/division assignment	right-to-left

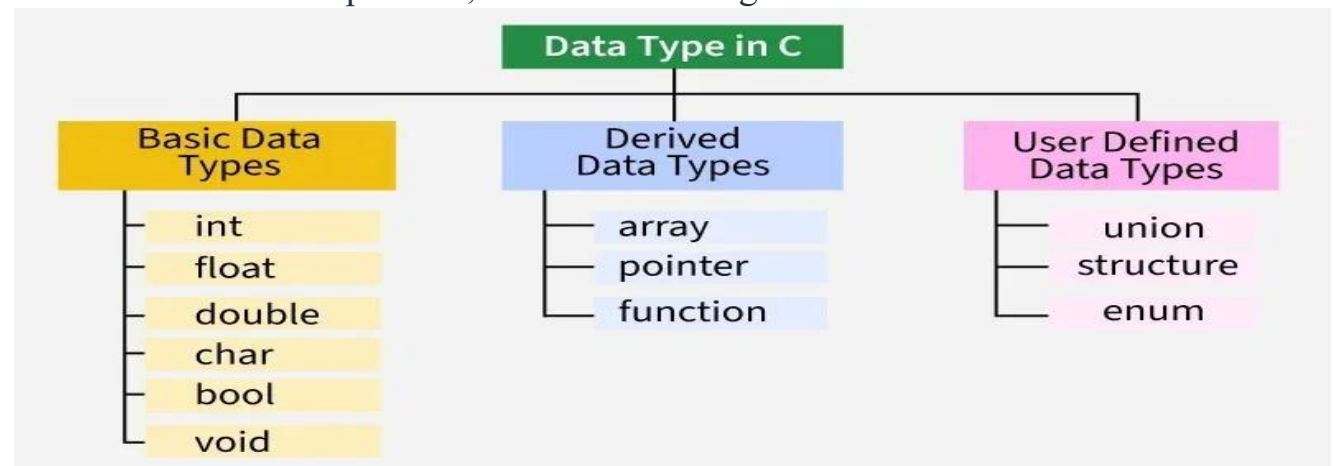
	<code>%=, &=</code>	Modulus/bitwise AND assignment	right-to-left
Precedence	Operator	Description	Associativity
	<code>^=, =</code>	Bitwise exclusive/inclusive OR assignment	right-to-left
	<code><<=, >>=</code>	Bitwise shift left/right assignment	right-to-left
15	<code>,</code>	expression separator	left-to-right

Data Types

Each variable in C has an associated data type. It specifies the type of data that the variable can store like integer, character, floating, double, etc. **Example:**
`int number;`

The above statement declares a variable with name **number** that can store **integer** values.

C is a statically type language where each variable's type must be specified at the declaration and once specified, it cannot be changed.



Please note that the Ranges and Sizes of different data types mentioned below are the most commonly used values. The actual values may vary from compiler to compiler. **Integer Data Type**

The integer datatype in C is used to store the integer numbers (any number including positive, negative and zero without decimal part). Octal values, hexadecimal values, and decimal values can also be stored in int data type in C.

- **Range:** -2,147,483,648 to 2,147,483,647
- **Size:** 4 bytes

Format Specifier: %d

Format specifiers are the symbols that are used for printing and scanning values of given data types.

Example:

We use **int keyword** to declare the integer variable:

```
int val;
```

We can store the integer values (literals) in this variable.

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

```
int main() { int  
    var = 22;  
    printf("var  
    = %d", var);  
    return 0;  
}
```

Output var

```
= 22
```

A variable of given data type can only contain the values of the same type. So, **var** can only store numbers, not text or anything else.

The integer data type can also be used as:

1. **unsigned int:** It can store the data values from zero to positive numbers, but it can't store negative values
2. **short int:** It is lesser in size than the int by 2 bytes so can only store values from -32,768 to 32,767.
3. **long int:** Larger version of the int datatype so can store values greater than int.

4. **unsigned short int:** Similar in relationship with short int as unsigned int with int.

Note: The size of an integer data type is compiler dependent.

We can use [sizeof operator](#) to check the actual size of any data type. In this article, we are discussing the sizes according to 64-bit compilers.

Character Data Type

Character data type allows its variable to store only a single character. The size of the character is **1 byte**. It is the most basic data type in C. It stores a single character and requires a single byte of memory in almost all compilers.

Range: (-128 to 127) or (0 to 255)

Size: 1 byte

Format Specifier: %c **Example:**

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

```
int main() {  
    char ch = 'A';  
  
    printf("ch = %c", ch);  
    return 0;  
}
```

Output

ch

Float Data Type

= A

In C programming, [float data type](#) is used to store single precision floating-point values. These values are decimal and exponential numbers.

- **Range:** 1.2E-38 to 3.4E+38
- **Size:** 4 bytes
- **Format Specifier:** %f **Example:**

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

```
int main() {  
    float val = 12.45;
```



```
printf("val = %f", val);
return 0;
}
```

Output

```
val = 12.450000
```

Double Data Type

The double data type in C is used to store decimal numbers (numbers with floating point values) with double precision. It can easily accommodate about 16 to 17 digits after or before a decimal point.

- **Range:** 1.7E-308 to 1.7E+308
- **Size:** 8 bytes
- **Format Specifier:** %lf **Example:**

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

```
int main() {
    double val = 1.4521;

    printf("val = %lf", val);
    return 0;
}
```

Output

```
val = 1.452100
```

Void Data Type

The void data type in C is used to indicate the absence of a value. Variables of void data type are not allowed. It can only be used for pointers and function return type and parameters. **Example:**

```
void fun(int a, int b){
    //function body
}
```

where function **fun** is a void type of function means it doesn't return any value.

Size of Data Types in C

The size of the data types in C is dependent on the size of the architecture, so we cannot define the universal size of the data types. For that, the C language provides the **sizeof()** operator to check the size of the data types. **Example**

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

```
int main(){  
  
    // Use sizeof() to know size //  
    the data types printf("The size  
of int: %d\n", sizeof(int));  
    printf("The size of char: %d\n", sizeof(char));  
    printf("The size of float: %d\n", sizeof(float));  
    printf("The size of double: %d",  
        sizeof(double));  
  
    return 0;  
}
```

Output

The size of int: 4

The size of char: 1

The size of float: 4

The size of double: 8

Different data types also have different ranges up to which can vary from compiler to compiler. Below is a list of ranges along with the memory requirement and format specifiers on the **64bit GCC compiler**.

Data Type	Size (bytes)	Range	Format Specifier
short int	2	-32,768 to 32,767	%hd

unsigned short int	2	0 to 65,535	%hu
unsigned int	4	0 to 4,294,967,295	%u
int	4	-2,147,483,648 to 2,147,483,647	%d
long int	4	-2,147,483,648 to 2,147,483,647	%ld
unsigned long int	4	0 to 4,294,967,295	%lu
long long int	8	$-(2^{63})$ to $(2^{63})-1$	%lld
			Format Specifier
Data Type	<i>Size (bytes)</i>	<i>Range</i>	

unsigned long long int	8	0 to 18,446,744,073,709,551,615	%llu
signed char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
float	4	1.2E-38 to 3.4E+38	%f
double	8	1.7E-308 to 1.7E+308	%lf
long double	16	3.4E-4932 to 1.1E+4932	%Lf

Note: The **long**, **short**, **signed** and **unsigned** are **datatype modifier** that can be used with some primitive datatype to change the size or length of the datatype.

Unit II

Structure of the C Program

The basic structure of a C program is divided into 6 parts which makes it easy to read, modify, document, and understand in a particular format. C program must follow the below-mentioned outline in order to successfully compile and execute. Debugging is easier in a well-structured C program.

Sections of the C Program

There are 6 basic sections responsible for the proper execution of a program. Sections are mentioned below:

1. Documentation
2. Preprocessor Section
3. Definition
4. Global Declaration
5. Main() Function
6. Sub Programs

1. Documentation

This section consists of the description of the program, the name of the program, and the creation date and time of the program. It is specified at the start of the program in the form of comments. Documentation can be represented as:

```
// description, name of the program, programmer name, date, time etc.
```

or

```
/*  
description, name of the program, programmer name, date, time etc.  
*/
```

Anything written as comments will be treated as documentation of the program and this will not interfere with the given code. Basically, it gives an overview to the reader of the program.

2. Preprocessor Section

All the header files of the program will be declared in the [preprocessor](#) section of the program. Header files help us to access other's improved code into our code. A copy of these multiple files is inserted into our program before the process of compilation.

Example:

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

3. Definition

Preprocessors are the programs that process our source code before the process of compilation. There are multiple steps which are involved in the writing and execution of the program. Preprocessor directives start with the '#' symbol. The #define preprocessor is used to create a constant throughout the program. Whenever this name is encountered by the compiler, it is replaced by the actual piece of defined code.

Example:

```
#define long long ll
```

4. Global Declaration

The global declaration section contains global variables, function declaration, and static variables. Variables and functions which are declared in this scope can be used anywhere in the program.

Example:

```
int num = 18;
```

5. Main() Function

Every C program must have a main function. The main() function of the program is written in this section. Operations like declaration and execution are performed inside the curly braces of the main program. The return type of the main() function can be int as well as void too. void() main tells the compiler that the program will not return any value. The int main() tells the compiler that the program will return an integer value.

Example:

```
void main()
```

or

```
int main()
```

6. Sub Programs

User-defined functions are called in this section of the program. The control of the program is shifted to the called function whenever they are called from the main or outside the main() function. These are specified as per the requirements of the programmer.

Example:

```
int sum(int x, int y)
{
    return x+y;
}
```

Structure of C Program with example

Example: Below C program to find the sum of 2 numbers:

```
// Documentation
```

```
/**
```

```
 * file: sum.c
```

```
 * author: you
```

```
 * description: program to find sum.
```

```
 */
```

```
// Link
```

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

```
// Definition
```

```
#define X 20
```

```
// Global Declaration
```

```
int sum(int y);
```

```
// Main() Function
```

```
int main(void)
```

```
{
```

```
    int y = 55;
```

```
    printf("Sum: %d", sum(y));
```

```
    return 0;
```

```
}
```

```
// Subprogram
```

```
int sum(int y)
```

```
{
```

```
    return y + X;
```

```
}
```

Output

```
Sum: 75
```

Explanation of the above Program

Below is the explanation of the above program. With a description explaining the program's meaning and use.

Sections	Description
<pre>/** * file: sum.c * author: you * description: program to * find sum. */</pre>	It is the comment section and is part of the description section of the code.
<pre>#include<stdio.h></pre>	Header file which is used for standard input-output. This is the preprocessor section.
<pre>#define X 20</pre>	This is the definition section. It allows the use of constant X in the code.
<pre>int sum(int y)</pre>	This is the Global declaration section includes the function declaration that can be used anywhere in the program.
<pre>int main()</pre>	main() is the first function that is executed in the C program.
<pre>{...}</pre>	These curly braces mark the beginning and end of the main function.
<pre>printf("Sum: %d", sum(y));</pre>	printf() function is used to print the sum on the screen.
<pre>return 0;</pre>	We have used int as the return type so we have to return 0 which states that the given program is free from the error and it can be exited successfully.
<pre>int sum(int y) { return y + X; }</pre>	This is the subprogram section. It includes the user-defined functions that are called in the main() function.

Basic Input and Output in C

In C, there are many input and output for different situations, but the most commonly used functions for Input/Output are scanf() and printf() respectively. These functions are part of the standard input/output library <stdio.h>. scanf() takes user inputs (typed using keyboard) and printf() displays output on the console or screen.

Basic Output in C

The **printf()** function is used to print formatted output to the standard output stdout (which is generally the console screen). It is one of the most commonly used functions in C.

Syntax

```
printf("formatted_string", variables/values);
```

Where,

- **Formatted String:** string defining the structure of the output and include **format specifiers**
- **variables/values:** arguments passed to printf() that will replace the format specifiers in the formatted string.

Examples

The following examples demonstrate the use of printf for output in different cases:

Printing Some Text

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

```
int main() {  
  
    // Prints some text  
    printf("First Print");  
  
    return 0;  
}
```

Output

```
First Print
```

Explanation: The text inside "" is called a string in C. It is used to represent textual information. We can directly pass strings to the printf() function to print them in console screen.

Printing Variables

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

```
int main() {  
    int age = 22;  
  
    // Prints Age  
    printf("%d\n", age);  
  
    return 0;  
}
```

Output

```
22
```

Here, the value of variable age is printed. You may have noticed **%d** in the formatted string. It is actually called format specifier which are used as placeholders for the value in the formatted string.

You may have also noticed '\n' character. This character is an escape sequence and is used to enter a newline.

Printing Variables Along with String

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

```
int main() {  
    int age = 22;
```



```
// Prints Age
printf("The value of the variable age is %d\n", age);

return 0;
}
```

Output

The value of the variable age is 22

The printf function in C allows us to format the output string to console. This type of string is called formatted string as it allows us to format (change the layout the article).

fputs()

The fputs() function is used to output strings to the files but we can also use it to print strings to the console screen.

Syntax:

```
fputs("your text here", stdout);
```

Where, the **stdout** represents that the text should be printed to console.

Example

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

```
int main() {
    fputs("This is my string", stdout);
    return 0;
}
```

Output

This is my string

Basic Input in C

scanf() is used to read user input from the console. It takes the format string and the addresses of the variables where the input will be stored.

Syntax

```
scanf("formatted_string", address_of_variables/values);
```

Remember that this function takes the address of the arguments where the read value is to be stored.

Examples of Reading User Input

The following examples demonstrate how to use the scanf for different user input in C:

Reading an Integer

```
#include <stdio.h>

int main() {
    int age;
    printf("Enter your age: ");

    // Reads an integer
    scanf("%d", &age);

    // Prints the age
    printf("Age is: %d\n", age);
    return 0;
}
```

Output

Enter your age:

25 *(Entered by the user)*

Age is: 25

Explanation: %d is used to read an integer; and &age provides the address of the variable where the input will be stored.

Reading a Character

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

```
int main() {  
    int ch;  
    printf("Enter a character: \n");  
  
    // Reads an integer  
    scanf("%c", &ch);  
  
    // Prints the age  
    printf("Entered character is: %d\n", ch);  
    return 0;  
}
```

Output

Enter a character:

a *(Entered by the user)*

Entered character is: a

Reading a string

The scanf() function can also be used to read string input from users. But it can only read single words.

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

```
int main() {  
    char str[100]; // Declare an array to hold the input string  
  
    printf("Enter a string: ");  
    scanf("%s", str); // Reads input until the first space or newline  
  
    printf("You entered: %s\n", str);  
  
    return 0;  
}
```

Output:

Enter a String:

Geeks *(Entered by the user)*

Entered string is: Geeks

The scanf() function can not handle spaces and stops at the first blank space. To handle this situation we can use [fgets\(\)](#) which is a better alternative as it can handle spaces and prevent buffer overflow.

fgets()

fgets() reads the given number of characters of a line from the input and stores it into the specified string. It can read multiple words at a time.

Syntax

```
fgets(str, n, stdin);
```

where **buff** is the string where the input will be stored and **n** is the maximum number of characters to read. **stdin** represents input reading from the keyboard.

Example:

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

int main() {

    // String variable
    char name[20];

    printf("Enter your name: \n");
    fgets(name, sizeof(name), stdin);

    printf("Hello, %s", name);
    return 0;
}
```

Output

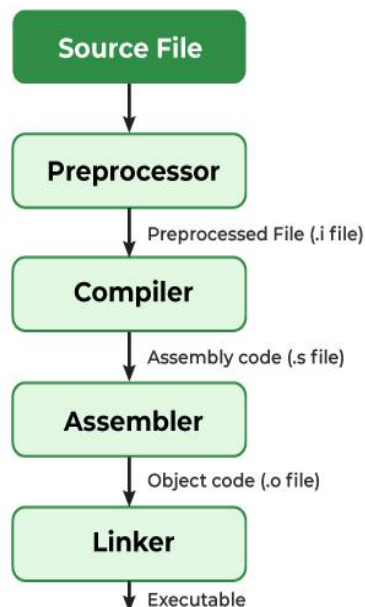
```
Enter your name:
John (Entered by User)
Hello, John
```

Compilation & Execution of C program

Compilation of C program

The compilation is the process of converting the source code of the C language into machine code. As C is a mid-level language, it needs a compiler to convert it into an executable code so that the program can be run on our machine.

The C program goes through the following phases during compilation:



Compilation Process in C

Understanding the compilation process in C helps developers optimize their programs.

How do we compile and run a C program?

We first need a compiler and a code editor to compile and run a C Program. The below example is of an Ubuntu machine with GCC compiler.

Step 1: Creating a C Source File

We first create a C program using an editor and save the file as filename.c In linux, we can use vi to create a file from the terminal using the command:

```
vi filename.c
```

In windows, we can use the Notepad to do the same. Then write a simple hello world program and save it.

Step 2: Compiling using GCC compiler

We use the following command in the terminal for compiling our **filename.c** source file.

```
gcc filename.c -o filename
```

We can pass many instructions to the GCC compiler to different tasks such as:

- The option -Wall enables all compiler's warning messages. This option is recommended to generate better code.
- The option -o is used to specify the output file name. If we do not use this option, then an output file with the name a.out is generated.

If there are no errors in our C program, the executable file of the C program will be generated.

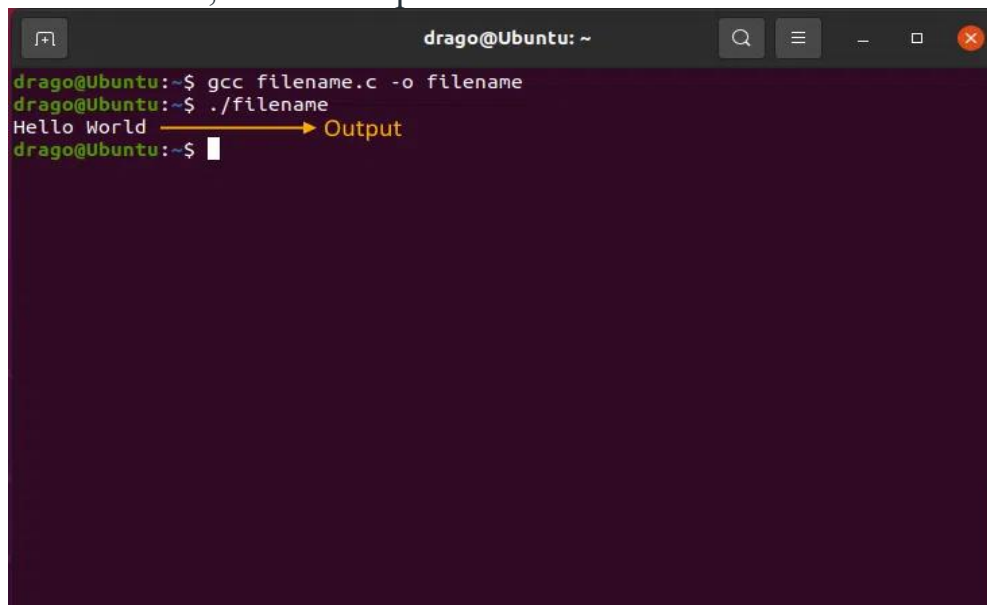
Step 3: Executing the program

After compilation executable is generated and we run the generated executable using the below command.

```
./filename // for linux
```

```
filename // for windows
```

The program will be executed, and the output will be shown in the terminal.

A screenshot of a terminal window titled 'drago@Ubuntu: ~'. The terminal shows the following commands and output:

```
drago@Ubuntu:~$ gcc filename.c -o filename
drago@Ubuntu:~$ ./filename
Hello World
```

An orange arrow points from 'Hello World' to the word 'Output'.

What goes inside the compilation process?

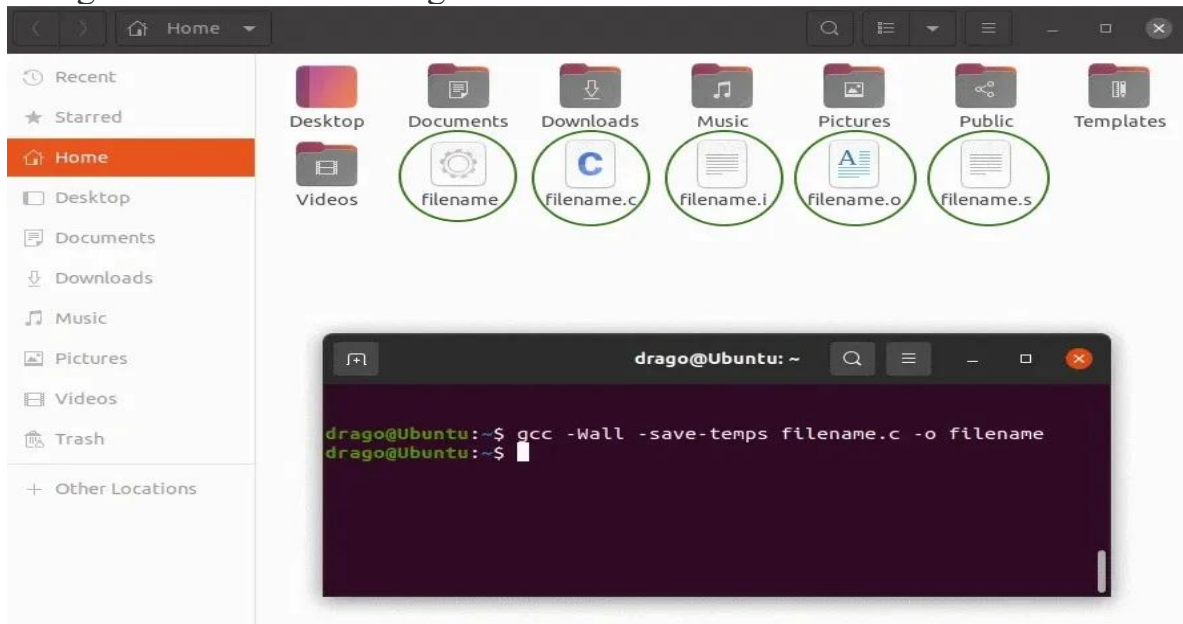
A compiler converts a C program into an executable. There are four phases for a C program to become an executable:

1. **Pre-processing**
2. **Compilation**
3. **Assembly**
4. **Linking**

By executing the below command while compiling the code, we get all intermediate files in the current directory along with the executable.

```
gcc -Wall -save-temps filename.c -o filename
```

The following screenshot shows all generated intermediate files.



Let us one by one see what these intermediate files contain.

1. Pre-processing

This is the first phase through which source code is passed. This phase includes:

- Removal of Comments
- Expansion of Macros
- Expansion of the included files.
- Conditional compilation

The pre-processed output is stored in the **filename.i**. Let's see what's inside filename.i: using **\$vi filename.i**

In the above output, the source file is filled with lots and lots of info, but in the end, our code is preserved.

- printf contains now `a + b` rather than `add(a, b)` that's because macros have expanded.
- Comments are stripped off.
- **#include<stdio.h>** is missing instead we see lots of code. So header files have been expanded and included in our source file.

2. Compiling

The next step is to compile filename.i and produce an; intermediate compiled output file **filename.s**. This file is in assembly-level instructions. Let's see through this file using **\$nano filename.s** terminal command.

Assembly Code File

The snapshot shows that it is in assembly language, which the assembler can understand.

3. Assembling

In this phase the filename.s is taken as input and turned into **filename.o** by the assembler. This file contains machine-level instructions. At this phase, only existing code is converted into machine language, and the function calls like `printf()` are not resolved. Let's view this file using **\$vi filename.o**

Binary Code

4. Linking

This is the final phase in which all the linking of function calls with their definitions is done. Linker knows where all these functions are implemented. Linker does some extra work also: it

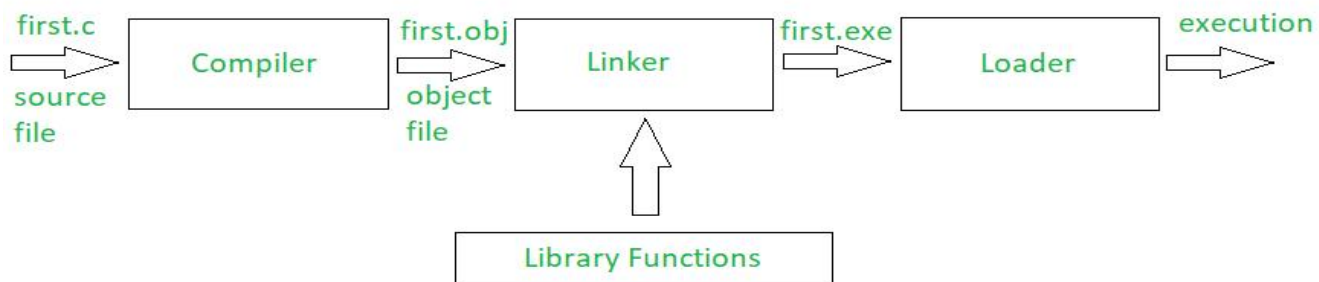
adds some extra code to our program which is required when the program starts and ends. For example, there is a code that is required for setting up the environment like passing command line arguments. This task can be easily verified by using **size filename.o** and **size filename**. Through these commands, we know how the output file increases from an object file to an executable file. This is because of the extra code that Linker adds to our program.

Linking can be of two types:

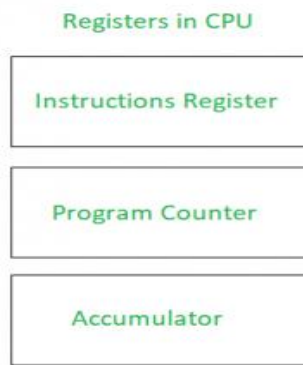
- **Static Linking:** All the code is copied to the single file and then executable file is created.
- **Dynamic Linking:** Only the names of the shared libraries is added to the code and then it is referred during the execution.

How does a C program executes?

Whenever a C program file is compiled and executed, the compiler generates some files with the same name as that of the C program file but with different extensions. So, what are these files and how are they created? Below image shows the compilation process with the files created at each step of the compilation process:



Every file that contains a C program must be saved with '.c' extension. This is necessary for the compiler to understand that this is a C program file. Suppose a program file is named, first.c. The file first.c is called the source file which keeps the code of the program. Now, when we compile the file, the C compiler looks for errors. If the C compiler reports no error, then it stores the file as a .obj file of the same name, called the object file. So, here it will create the first.obj. This .obj file is not executable. The process is continued by the Linker which finally gives a .exe file which is executable. **Linker:** First of all, let us know that library functions are not a part of any C program but of the C software. Thus, the compiler doesn't know the operation of any function, whether it be printf or scanf. The definitions of these functions are stored in their respective library which the compiler should be able to link. This is what the Linker does. So, when we write #include, it includes stdio.h library which gives access to Standard Input and Output. The linker links the object files to the library functions and the program becomes a .exe file. Here, first.exe will be created which is in an executable format. **Loader:** Whenever we give the command to execute a particular program, the loader comes into work. The loader will load the .exe file in RAM and inform the CPU with the starting point of the address where this program is loaded



CPU Registers

Instruction Register: It holds the current instructions to be executed by the CPU. **Program Counter:** It contains the address of the next instructions to be executed by the CPU. **Accumulator:** It stores the information related to calculations. The loader informs Program Counter about the first instruction and initiates the execution. Then onwards, Program Counter handles the task. **Difference between Linker and Loader**

Linker	Loader
Linker generates the executable module of a source program.	Loader loads the executable module to the main memory for execution.
Linker takes the object code generated by an assembler, as input.	Loader takes executable module generated by a linker as input.
Linker combines all the object modules of a source code to generate an executable module.	Loader allocates the addresses to an executable module in main memory for execution.
The types of Linker are Linkage Editor, Dynamic linker.	The types of Loader are Absolute loading, Relocatable loading and Dynamic Run-time loading.

Escape Sequence in C

The escape sequence in C is the characters or the sequence of characters that can be used inside the string literal. The purpose of the escape sequence is to represent the characters that cannot be used normally using the keyboard. Some escape sequence characters are the part of ASCII charset but some are not.

Different escape sequences represent different characters but the output is dependent on the compiler you are using.

Escape Sequence List

The table below lists some common escape sequences in C language.

Escape Sequence	Name	Description
\a	Alarm or Beep	It is used to generate a bell sound in the C program.

Escape Sequence	Name	Description
\b	Backspace	It is used to move the cursor one place backward.
\f	Form Feed	It is used to move the cursor to the start of the next logical page.
\n	New Line	It moves the cursor to the start of the next line.
\r	Carriage Return	It moves the cursor to the start of the current line.
\t	Horizontal Tab	It inserts some whitespace to the left of the cursor and moves the cursor accordingly.
\v	Vertical Tab	It is used to insert vertical space.
\\	Backlash	Use to insert backslash character.
\'	Single Quote	It is used to display a single quotation mark.
\"	Double Quote	It is used to display double quotation marks.
\?	Question Mark	It is used to display a question mark.
\ooo	Octal Number	It is used to represent an octal number.
\xhh	Hexadecimal Number	It represents the hexadecimal number.
\0	NULL	It represents the NULL character.

Out of all these escape sequences, \n and \0 are used the most. In fact, escape sequences like \f, \a, are not even used by programmers nowadays.

Example of Escape sequence characters

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

```
int main() {
    // 1. Alarm / Beep
    printf("1. Alarm / Beep (\\a): Beep sound\\a\\n");

    // 2. Backspace
    printf("2. Backspace (\\b): ABC\\bD (C is erased)\\n");

    // 3. Form Feed
```



```

printf("3. Form Feed (\f): First Page\fSecond Page (form feed)\n");

// 4. New Line
printf("4. New Line (\n): Line1\nLine2\n");

// 5. Carriage Return
printf("5. Carriage Return (\r): Hello\rWorld (overwrites)\n");

// 6. Horizontal Tab
printf("6. Horizontal Tab (\t): Hello\tWorld\n");

// 7. Vertical Tab
printf("7. Vertical Tab (\v): Line1\vLine2\n");

// 8. Backslash
printf("8. Backslash (\\): This is a backslash -> \\n");

// 9. Single Quote
printf("9. Single Quote (\'): This is a single quote -> \'A\'\n");

// 10. Double Quote
printf("10. Double Quote (\\"): This is a double quote -> \"Hello\"\n");

// 11. Question Mark
printf("11. Question Mark (\?): To avoid trigraph issues -> \?\n");

// 12. Octal number (\ooo)
printf("12. Octal (\ooo): \\101 = %c (letter A)\n", '\101');

// 13. Hexadecimal number (\xhh)
printf("13. Hexadecimal (\xhh): \\x41 = %c (letter A)\n", '\x41');

// 14. Null Character
printf("14. Null (\0): This string ends here\0This part is hidden\n");

return 0;
}

```

Output

1. Alarm / Beep (\a): Beep sound [Beep sound plays if supported]
2. Backspace (\b): ABD (C is erased)
3. Form Feed (\f): First Page
Second Page (form feed)
4. New Line (\n): Line1
Line2
5. Carriage Return (\r): World (overwrites Hello)
6. Horizontal Tab (\t): Hello World
7. Vertical Tab (\v): Line1
Line2
8. Backslash (\\): This is a backslash -> \
9. Single Quote (\'): This is a single quote -> 'A'
10. Double Quote (\"): This is a double quote -> "Hello"
11. Question Mark (\?): To avoid trigraph issues -> ?

- 12. Octal (\ooo): \101 = A (letter A)
- 13. Hexadecimal (\xhh): \x41 = A (letter A)
- 14. Null (\0): This string ends here
(The null \0 ends the string, so text after it doesn't display.)

Decision Making in C

In C, programs can choose which part of the code to execute based on some condition. This ability is called **decision making** and the statements used for it are called **conditional statements**. These statements evaluate one or more conditions and make the decision whether to execute a block of code or not.

For example, consider that there is a show that starts only when certain number of people are present in the audience. So, you can write a program like as shown:

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

```
int main() {  
  
    // Number of people in the audience  
    int num = 100;  
  
    // Conditional code inside decision making statement  
    if (num > 50) {  
        printf("Start the show");  
    }  
  
    return 0;  
}
```

Output

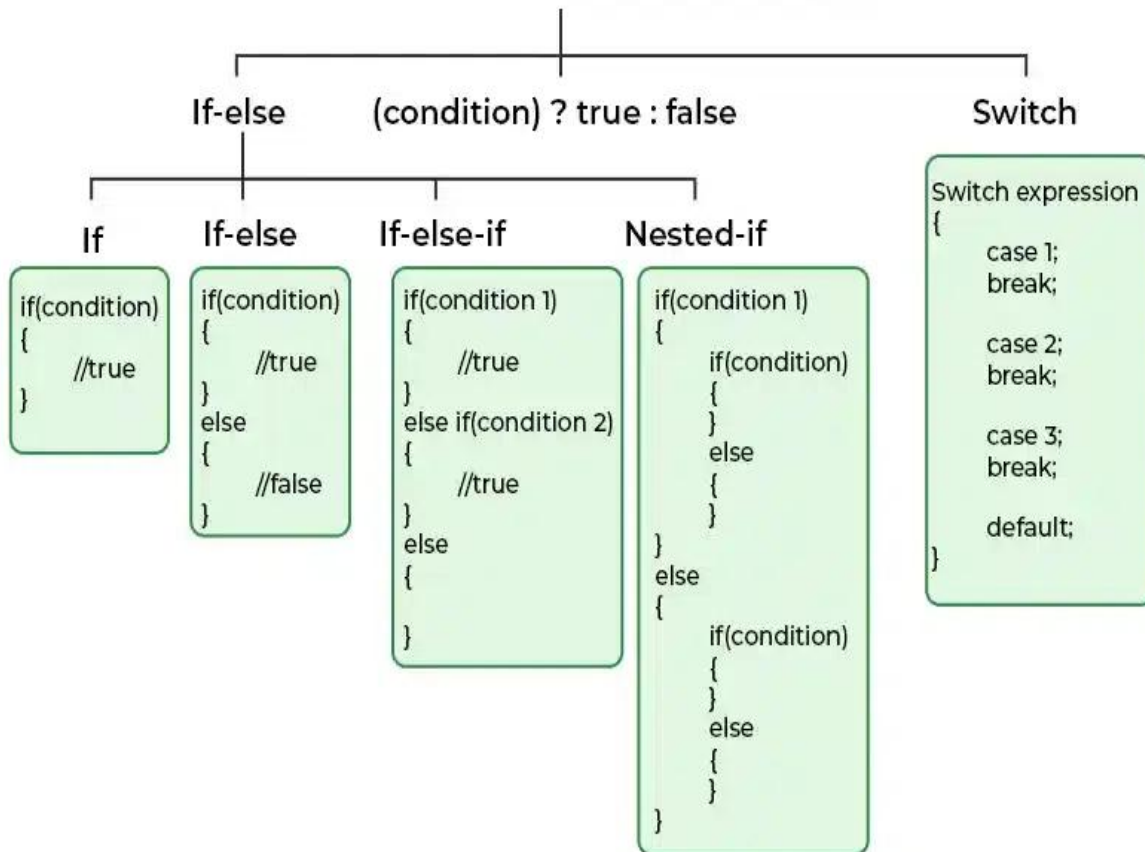
```
Start the show
```

In the above program, the show only starts when the number of people is greater than **50**. It is specified in the **if statement** (a type of conditional statement) as a condition (**num > 50**). You can decrease the value of **num** to less than **50** and try rerunning the code.

Types of Conditional Statements in C

In the above program, we have used if statement, but there are many different types of conditional statements available in C language:

Conditional Statements in C



1. if in C

The if statement is the simplest decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statements is executed otherwise not.

A **condition** is any expression that evaluates to either a true or false (or values convertible to true or false).

Example

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

```
int main() {

    int i = 10;

    // If statement
    if (i < 18) {
        printf("Eligible for vote");
    }
}
```

Output

```
Eligible for vote
```

The expression inside **() parenthesis** is the **condition** and set of statements inside **{ } braces** is its **body**. If the condition is true, only then the body will be executed.

*If there is only a single statement in the body, **{ } braces** can be omitted.*

2. if-else in C

The **if** statement alone tells us that if a condition is true, it will execute a block of statements and if the condition is false, it won't. But what if we want to do something else when the condition is false? Here comes the C **else** statement. We can use the **else** statement with the **if** statement to execute a block of code when the condition is false. The [if-else statement](#) consists of two blocks, one for false expression and one for true expression.

Example

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

```
int main() {
    int i = 10;

    if (i > 18) {
        printf("Eligible for vote");
    }
    else {
        printf("Not Eligible for vote");
    }
    return 0;
}
```

Output

```
Not Eligible for vote
```

The block of code following the *else* statement is executed as the condition present in the *if* statement is false.

3. Nested if-else in C

A nested if in C is an if statement that is the target of another if statement. Nested if statements mean an if statement inside another if statement. Yes, C allow us to nested if statements within if statements, i.e, we can place an if statement inside another if statement.

Example

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

```
int main(){
    int i = 10;

    if (i == 10) {
        if (i < 18)
            printf("Still not eligible for vote");
        else
            printf("Eligible for vote\n");
    }
    else {
        if (i == 20) {
            if (i < 22)
                printf("i is smaller than 22 too\n");
            else
                printf("i is greater than 25");
        }
    }

    return 0;
}
```

Output

Still not eligible for vote

4. if-else-if Ladder in C

The if else if statements are used when the user has to decide among multiple options. The C if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the C else-if ladder is bypassed. If none of the conditions is true, then the final else statement will be executed. if-else-if ladder is similar to the switch statement.

Example

```
#include <stdio.h>

int main() {
    int i = 20;

    // If else ladder with three conditions
    if (i == 10)
        printf("Not Eligible");
    else if (i == 15)
        printf("wait for three years");
    else if (i == 20)
        printf("You can vote");
    else
        printf("Not a valid age");

    return 0;
}
```

Output

You can vote

5. switch Statement in C

The switch case statement is an alternative to the if else if ladder that can be used to execute the conditional code based on the value of the variable specified in the switch statement. The switch block consists of cases to be executed based on the value of the switch variable.

Example

```
#include <stdio.h>

int main() {

    // variable to be used in switch statement
    int var = 18;

    // declaring switch cases
    switch (var) {
    case 15:
        printf("You are a kid");
        break;
    case 18:
        printf("Eligible for vote");
    }
```

```

        break;
default:
    printf("Default Case is executed");
    break;
}

return 0;
}

```

Output

Eligible for vote

Note: The switch expression should evaluate to either integer or character. It cannot evaluate any other data type.

6. Conditional Operator in C

The conditional operator is used to add conditional code in our program. It is similar to the if-else statement. It is also known as the ternary operator as it works on three operands.

Example:

```

#include <stdio.h>

int main() {
    int var;
    int flag = 0;

    // using conditional operator to assign the value to var
    // according to the value of flag
    var = flag == 0 ? 25 : -25;
    printf("Value of var when flag is 0: %d\n", var);

    return 0;
}

```

Output

Value of var when flag is 0: 25

7. Jump Statements in C

These statements are used in C for the unconditional flow of control throughout the functions in a program. They support four types of jump statements:

A) break

This loop control statement is used to terminate the loop. As soon as the break statement is encountered from within a loop, the loop iterations stop there, and control returns from the loop immediately to the first statement after the loop.

Example

```

#include <stdio.h>

int main() {
    int arr[] = { 1, 2, 3, 4, 5, 6 };
    int key = 3;
    int size = 6;

    for (int i = 0; i < size; i++) {
        if (arr[i] == key) {

```

```

        printf("Element found at position: %d",
            (i + 1));
        break;
    }
}

return 0;
}

```

Output

```
Element found at position: 3
```

B) continue

This loop control statement is just like the break statement. The continue statement is opposite to that of the break *statement*, instead of terminating the loop, it forces to execute the next iteration of the loop.

As the name suggests the continue statement forces the loop to continue or execute the next iteration. When the continue statement is executed in the loop, the code inside the loop following the continue statement will be skipped and the next iteration of the loop will begin.

Example:

```

#include <stdio.h>

int main() {
    for (int i = 1; i <= 10; i++) {

        // If i is equals to 6,
        // continue to next iteration
        // without printing
        if (i == 6)
            continue;
        else
            printf("%d ", i);
    }

    return 0;
}

```

Output

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

C) goto

The goto statement in C also referred to as the unconditional jump statement can be used to jump from one point to another within a function.

Examples:

```

#include <stdio.h>

int main() {
    int n = 1;
label:
    printf("%d ", n);
    n++;
    if (n <= 10)
        goto label;
}

```

```
    return 0;
}
```

Output

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

D) return

The return in C returns the flow of the execution to the function from where it is called. This statement does not mandatorily need any conditional statements. As soon as the statement is executed, the flow of the program stops immediately and returns the control from where it was called. The return statement may or may not return anything for a void function, but for a non-void function, a return value must be returned.

Example:

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

```
int sum(int a, int b) {
    int s1 = a + b;
    return s1;
}

int main()
{
    int num1 = 10;
    int num2 = 10;
    int sumOf = sum(num1, num2);
    printf("%d", sumOf);
    return 0;
}
```

Output

```
20
```

Loops Statements

In C programming, there is often a need for repeating the same part of the code multiple times. For example, to print a text three times, we have to use printf() three times as shown in the code:

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

```
int main() {
    printf( "Hello GfG\n");
    printf( "Hello GfG\n");
    printf( "Hello GfG");
    return 0;
}
```

Output

```
Hello GfG
```

```
Hello GfG
```

```
Hello GfG
```


But if we say to write this 20 times, it will take some time to write statement. Now imagine writing it 100 or 1000 times. Then it becomes a really hectic task to write same statements again and again. To solve such kind of problems, we have loops in programming languages.

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

```
int main() {  
  
    // Loop to print "Hello GfG" 3 times  
    for (int i = 0; i < 3; i++) {  
        printf("Hello GfG\n");  
    }  
    return 0;  
}
```

Output

Hello GfG

Hello GfG

Hello GfG

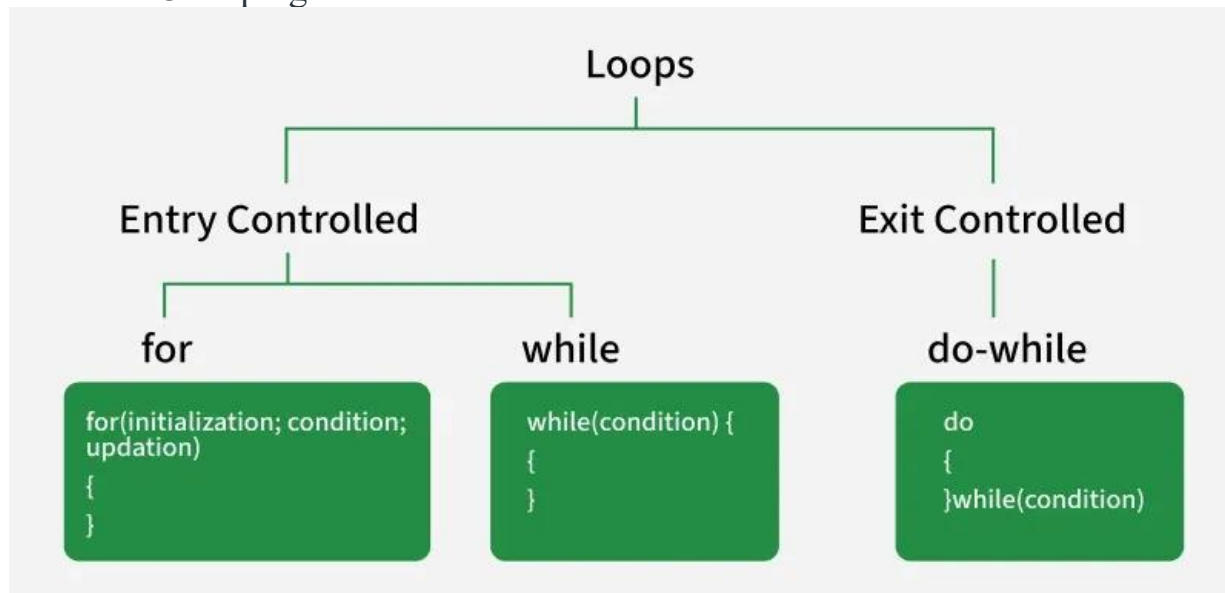
In the above code, we have used a loop to print text 3 times. We could have done it for 100 or even 1000 times in the same number of code lines.

What are loops in C?

Loops in C programming are used to repeat a block of code until the specified condition is met. It allows programmers to execute a statement or group of statements multiple times without writing the code again and again.

Types of Loops in C

There are 3 looping statements in C:



Let's discuss all 3 types of loops in C one by one.

for Loop

for loop is an **entry-controlled** loop, which means that the condition is checked before the loop's body executes.

Syntax

```
for (initialization; condition; updation) {  
    // body of for loop
```

```
}
```

The various parts of the for loop are:

- **Initialization:** Initialize the variable to some initial value.
- **Test Condition:** This specifies the test condition. If the condition evaluates to true, then body of the loop is executed. If evaluated false, loop is terminated.
- **Update Expression:** After the execution loop's body, this expression increments/decrements the loop variable by some value.
- **Body of Loop:** Statements to repeat. Generally enclosed inside {} braces.

Example:

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

```
int main() {
```

```
    // Loop to print numbers from 1 to 5
```

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

```
        printf( "%d ", i + 1);
```

```
    }
```

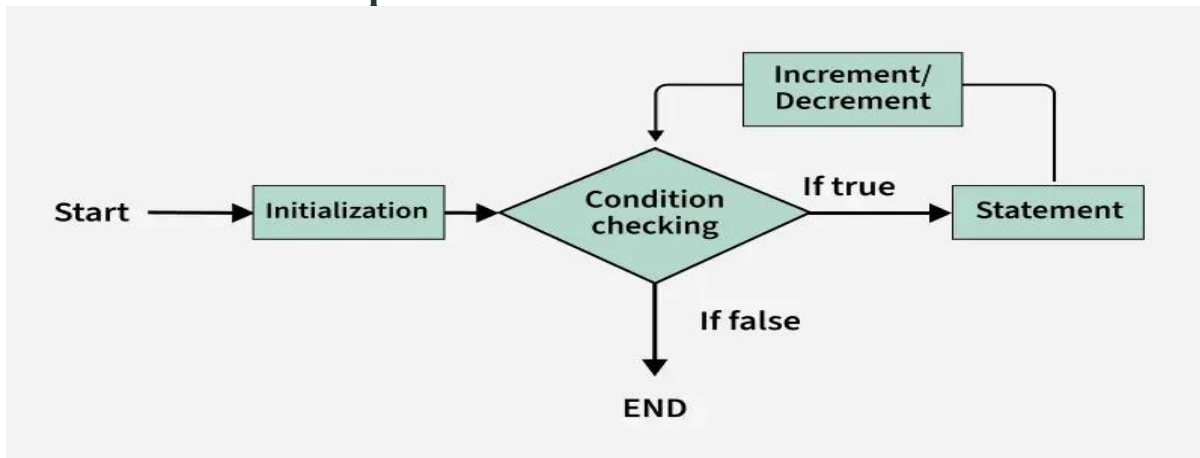
```
    return 0;
```

```
}
```

Output

```
1 2 3 4 5
```

Flowchart of for Loop



while Loop

A **while loop** is also an **entry-controlled loop** in which the condition is checked before entering the body.

Syntax

```
while (condition) {  
    // Body of the loop
```

```
}
```

Only the **condition** is the part of **while loop** syntax, we have to initialize and update loop variable manually.

Example:

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

```
int main() {
```

```
    // Initialization expression
```

```
    int i = 0;
```

```
// Test expression
while(i <= 5) {
    printf("%d ", i + 1);

    // update expression
    i++;
}

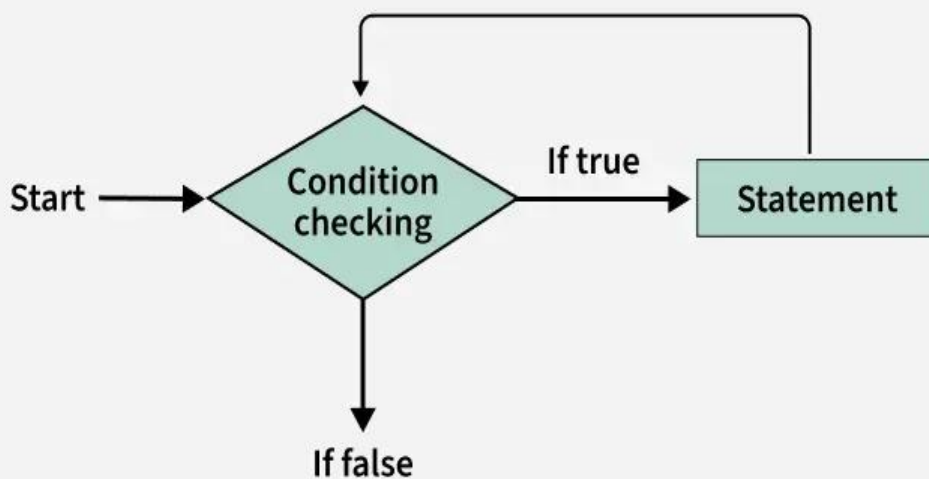
return 0;
}
```

Output

```
1 2 3 4 5 6
```

Flowchart of while Loop

The below flowchart demonstrates execution flow of the **while loop**.



do-while Loop

The do-while loop is an **exit-controlled loop**, which means that the condition is checked after executing the loop body. Due to this, the loop body will **execute at least once** irrespective of the test condition.

Syntax

```
do {
    // Body of the loop
} while (condition);
```

Like while loop, only the condition is the part of **do while loop** syntax, we have to do the initialization and updating of loop variable manually.

Example:

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

```
int main() {
```

```
    // Initialization expression
    int i = 0;
```

```
    do
```

```

{
    // loop body
    printf( "%d ", i);

    // Update expression
    i++;

    // Condition to check
} while (i <= 10);

return 0;
}

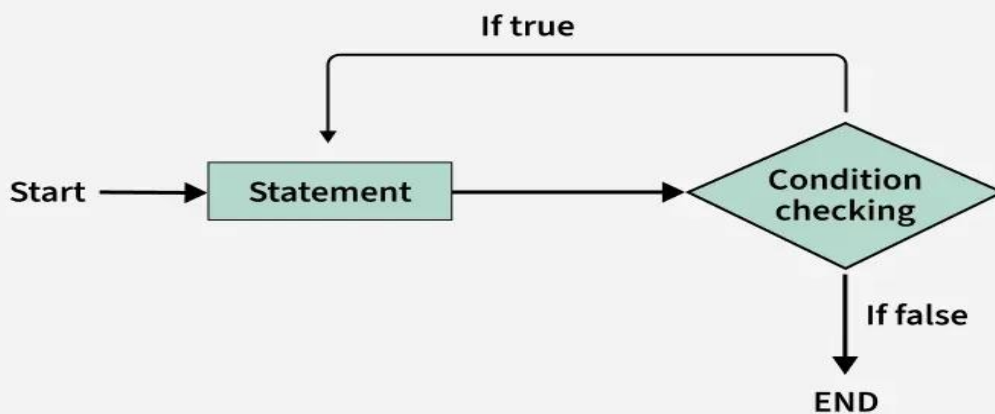
```

Output

```
0 1 2 3 4 5 6 7 8 9 10
```

Flowchart of do-while Loop

The below flowchart demonstrates execution flow of the do while loop.



Infinite Loop

An **infinite loop** is executed when the test expression never becomes false, and the body of the loop is executed repeatedly. A program is stuck in an Infinite loop when the condition is always true. Mostly this is an error that can be resolved by using Loop Control statements.

Using for loop:

```

#include <stdio.h>

int main () {

    // This is an infinite for loop
    // as the condition expression
    // is blank
    for ( ;; ) {
        printf("This loop will run forever.");
    }
    return 0;
}

```

Output

```
This loop will run forever.
This loop will run forever.
```

This loop will run forever.

...

Using While loop:

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

```
int main() {  
    while (1)  
        printf("This loop will run forever.\n");  
    return 0;  
}
```

Output

This loop will run forever.

This loop will run forever.

This loop will run forever.

...

Using the do-while loop:

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

```
int main() {  
    do {  
        printf("This loop will run forever.");  
    } while (1);  
    return 0;  
}
```

Output

This loop will run forever.

This loop will run forever.

This loop will run forever.

...

Nested Loops

Nesting loops means placing one loop inside another. The inner loop runs fully for each iteration of the outer loop. This technique is helpful when you need to perform multiple iterations within each cycle of a larger loop, like when working with a two-dimensional array or performing tasks that require multiple levels of iteration.

Example:

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

```
int main() {  
    // Outer loop runs 3 times  
    for (int i = 0; i < 3; i++) {  
        // Inner loop runs 2 times for each  
        // outer loop iteration  
        for (int j = 0; j < 2; j++) {  
            printf("i = %d, j = %d\n", i, j);  
        }  
    }  
    return 0;  
}
```

Output

```
i = 0, j = 0
i = 0, j = 1
i = 1, j = 0
i = 1, j = 1
i = 2, j = 0
i = 2, j = 1
```

Loop Control Statements

Loop control statements in C programming are used to change execution from its normal sequence.

Name	Description
<u>break</u>	The break statement is used to terminate the loop statement.
<u>continue</u>	When encountered, the continue statement skips the remaining body and jumps to the next iteration of the loop.
<u>goto</u>	goto statement transfers the control to the labeled statement.

Example:

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

```
int main() {
    for (int i = 0; i < 5; i++) {
        if (i == 3) {

            // Exit the loop when i equals 3
            break;
        }
        printf("%d ", i);
    }
    printf("\n");

    for (int i = 0; i < 5; i++) {
        if (i == 3) {

            // Skip the current iteration
            // when i equals 3
            continue;
        }
        printf("%d ", i);
    }
    printf("\n");
    for (int i = 0; i < 5; i++) {
        if (i == 3) {

            // Jump to the skip label when
            // i equals 3
            goto skip;
        }
    }
}
```

```

    printf("%d ", i);
}

skip:
printf("\nJumped to the 'skip' label %s",
"when i equals 3.");

return 0;
}

```

Output

```

0 1 2
0 1 2 4
0 1 2
Jumped to the 'skip' label when i equals 3.

```

Operators in C

Operators are the basic components of C programming. They are symbols that represent some kind of operation, such as mathematical, relational, bitwise, conditional, or logical computations, which are to be performed on values or variables. The values and variables used with operators are called **operands**.

Example:

```

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

    // Expression for getting sum
    int sum = 10 + 20;

    printf("%d", sum);
    return 0;
}

```

Output

```

30

```

In the above expression, '+' is the **addition operator** that tells the compiler to add both of the operands 10 and 20. To dive deeper into how operators are used with data structures, the [C Programming Course Online with Data Structures](#) covers this topic thoroughly.

Unary, Binary and Ternary Operators

On the basis of the number of operands they work on, operators can be classified into three types :

1. **Unary Operators:** Operators that work on single operand.
Example: Increment(++) , Decrement(--)
2. **Binary Operators:** Operators that work on two operands.
Example: Addition (+), Subtraction(-) , Multiplication (*)
3. **Ternary Operators:** Operators that work on three operands.
Example: Conditional Operator(? :)

Types of Operators in C

C language provides a wide range of built in operators that can be classified into 6 types based on their functionality:

Table of Content

- [Arithmetic Operators](#)
- [Relational Operators](#)
- [Logical Operator](#)
- [Bitwise Operators](#)
- [Assignment Operators](#)
- [Other Operators](#)

Arithmetic Operators

The **arithmetic operators** are used to perform arithmetic/mathematical operations on operands. There are **9 arithmetic** operators in C language:

Symbol	Operator	Description	Syntax
+	Plus	Adds two numeric values.	a + b
-	Minus	Subtracts right operand from left operand.	a - b
*	Multiply	Multiply two numeric values.	a * b
/	Divide	Divide two numeric values.	a / b
%	Modulus	Returns the remainder after dividing the left operand with the right operand.	a % b
+	Unary Plus	Used to specify the positive values.	+a
-	Unary Minus	Flips the sign of the value.	-a
++	Increment	Increases the value of the operand by 1.	a++
--	Decrement	Decreases the value of the	a--

Symbol	Operator	Description	Syntax
		operand by 1.	

Example of C Arithmetic Operators

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

```
int main() {  
  
    int a = 25, b = 5;  
  
    // using operators and printing results  
    printf("a + b = %d\n", a + b);  
    printf("a - b = %d\n", a - b);  
    printf("a * b = %d\n", a * b);  
    printf("a / b = %d\n", a / b);  
    printf("a % b = %d\n", a % b);  
    printf("+a = %d\n", +a);  
    printf("-a = %d\n", -a);  
    printf("a++ = %d\n", a++);  
    printf("a-- = %d\n", a--);  
  
    return 0;  
}
```

Output

```
a + b = 30  
a - b = 20  
a * b = 125  
a / b = 5  
a % b = 0  
+a = 25  
-a = -25  
a++ = 25  
a-- = 26
```

Relational Operators

The relational operators in C are used for the comparison of the two operands. All these operators are binary operators that return true or false values as the result of comparison. **These are a total of 6 relational operators in C:**

Symbol	Operator	Description	Syntax
<	Less than	Returns true if the left operand is less than the right operand. Else false	a < b

Symbol	Operator	Description	Syntax
>	Greater than	Returns true if the left operand is greater than the right operand. Else false	a > b
<=	Less than or equal to	Returns true if the left operand is less than or equal to the right operand. Else false	a <= b
>=	Greater than or equal to	Returns true if the left operand is greater than or equal to right operand. Else false	a >= b
==	Equal to	Returns true if both the operands are equal.	a == b
!=	Not equal to	Returns true if both the operands are NOT equal.	a != b

Example of C Relational Operators

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

```
int main() {
    int a = 25, b = 5;

    // using operators and printing results
    printf("a < b : %d\n", a < b);
    printf("a > b : %d\n", a > b);
    printf("a <= b: %d\n", a <= b);
    printf("a >= b: %d\n", a >= b);
    printf("a == b: %d\n", a == b);
    printf("a != b : %d\n", a != b);

    return 0;
}
```

Output

```
a < b : 0
a > b : 1
a <= b: 0
a >= b: 1
a == b: 0
a != b : 1
```

Here, 0 means false and 1 means true.

Logical Operator

Logical Operators are used to combine two or more conditions/constraints or to complement the evaluation of the original condition in consideration. The result of the operation of a logical operator is a Boolean value either **true** or **false**.

There are **3** logical operators in C:

Symbol	Operator	Description	Syntax
&&	Logical AND	Returns true if both the operands are true.	a && b
 	Logical OR	Returns true if both or any of the operand is true.	a b
!	Logical NOT	Returns true if the operand is false.	!a

Example of Logical Operators in C

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

```
int main() {
    int a = 25, b = 5;

    // using operators and printing results
    printf("a && b : %d\n", a && b);
    printf("a || b : %d\n", a || b);
    printf("!a: %d\n", !a);

    return 0;
}
```

Output

```
a && b : 1
a || b : 1
!a: 0
```

Bitwise Operators

The **Bitwise operators** are used to perform bit-level operations on the operands. The operators are first converted to bit-level and then the calculation is performed on the operands. **Note:** Mathematical operations such as addition, subtraction, multiplication, etc. can be performed at the bit level for faster processing.

There are 6 bitwise operators in C:

Symbol	Operator	Description	Syntax
&	Bitwise AND	Performs bit-by-bit AND operation and returns the result.	a & b
	Bitwise OR	Performs bit-by-bit OR operation and returns the result.	a b
^	Bitwise XOR	Performs bit-by-bit XOR operation and returns the result.	a ^ b
~	Bitwise First Complement	Flips all the set and unset bits on the number.	~a
<<	Bitwise Leftshift	Shifts bits to the left by a given number of positions; multiplies the number by 2 for each shift.	a << b
>>	Bitwise Rightshilft	Shifts bits to the right by a given number of positions; divides the number by 2 for each shift.	a >> b

Example of Bitwise Operators

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

```
int main() {  
    int a = 25, b = 5;
```

```
    // using operators and printing results  
    printf("a & b: %d\n", a & b);
```

```

printf("a | b: %d\n", a | b);
printf("a ^ b: %d\n", a ^ b);
printf("~a: %d\n", ~a);
printf("a >> b: %d\n", a >> b);
printf("a << b: %d\n", a << b);

return 0;
}

```

Output

```

a & b: 1
a | b: 29
a ^ b: 28
~a: -26
a >> b: 0
a << b: 800

```

Assignment Operators

Assignment operators are used to assign value to a variable. The left side operand of the assignment operator is a variable and the right side operand of the assignment operator is a value. The value on the right side must be of the same data type as the variable on the left side otherwise the compiler will raise an error.

The assignment operators can be combined with some other operators in C to provide multiple operations using single operator. These operators are called compound operators.

In C, there are 11 assignment operators:

Symbol	Operator	Description	Syntax
=	Simple Assignment	Assign the value of the right operand to the left operand.	a = b
+=	Plus and assign	Add the right operand and left operand and assign this value to the left operand.	a += b
-=	Minus and assign	Subtract the right operand and left operand and assign this value to the left operand.	a -= b
*=	Multiply and assign	Multiply the right operand and	a *= b

Symbol	Operator	Description	Syntax
		left operand and assign this value to the left operand.	
/=	Divide and assign	Divide the left operand with the right operand and assign this value to the left operand.	a /= b
%=	Modulus and assign	Assign the remainder in the division of left operand with the right operand to the left operand.	a %= b
&=	AND and assign	Performs bitwise AND and assigns this value to the left operand.	a &= b
=	OR and assign	Performs bitwise OR and assigns this value to the left operand.	a = b
^=	XOR and assign	Performs bitwise XOR and assigns this value to the left operand.	a ^= b
>>=	Rightshift and assign	Performs bitwise Rightshift and assign this value to the left operand.	a >>= b
<<=	Leftshift and assign	Performs bitwise Leftshift and assign this value to the left operand.	a <<= b

Example of C Assignment Operators

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

```

int main() {
    int a = 25, b = 5;

    // using operators and printing results
    printf("a = b: %d\n", a = b);
    printf("a += b: %d\n", a += b);
    printf("a -= b: %d\n", a -= b);
    printf("a *= b: %d\n", a *= b);
    printf("a /= b: %d\n", a /= b);
    printf("a %= b: %d\n", a %= b);
    printf("a &= b: %d\n", a &= b);
    printf("a |= b: %d\n", a |= b);
    printf("a ^= b: %d\n", a ^= b);
    printf("a >>= b: %d\n", a >>= b);
    printf("a <<= b: %d\n", a <<= b);

    return 0;
}

```

Output

```

a = b: 5
a += b: 10
a -= b: 5
a *= b: 25
a /= b: 5
a %= b: 0
a &= b: 0
a |= b: 5
a ^= b: 0
a >>= b: 0
a <<= b: 0

```

Other Operators

Apart from the above operators, there are some other operators available in C used to perform some specific tasks. Some of them are discussed here:

sizeof Operator

- **sizeof** is much used in the C programming language.
- It is a compile-time unary operator which can be used to compute the size of its operand.
- The result of sizeof is of the unsigned integral type which is usually denoted by `size_t`.
- Basically, the sizeof the operator is used to compute the size of the variable or datatype.

Syntax

sizeof (operand)

Comma Operator (,)

The **comma operator** (represented by the token) is a binary operator that evaluates its first operand and discards the result, it then evaluates the second operand and returns this value (and type).

The comma operator has the lowest precedence of any C operator. It can act as both operator and separator.

Syntax

```
operand1 , operand2
```

Conditional Operator (? :)

The **conditional operator** is the only ternary operator in C++. It is a conditional operator that we can use in place of if..else statements.

Syntax

```
expression1 ? Expression2 : Expression3;
```

Here, **Expression1** is the condition to be evaluated. If the condition(**Expression1**) is *True* then we will execute and return the result of **Expression2** otherwise if the condition(**Expression1**) is *false* then we will execute and return the result of **Expression3**.

dot (.) and arrow (->) Operators

Member operators are used to reference individual members of classes, structures, and unions.

- The **dot operator** is applied to the actual object.
- The **arrow operator** is used with a pointer to an object.

Syntax

```
structure_variable . member;  
structure_pointer -> member;
```

Cast Operators

Casting operators convert one data type to another. For example, `int(2.2000)` would return 2.

- A cast is a special operator that forces one data type to be converted into another.

Syntax

```
(new_type) operand;
```

addressof (&) and Dereference (*) Operators

Addressof operator & returns the address of a variable and the **dereference operator *** is a pointer to a variable. For example `*var`; will pointer to a variable `var`.

Example of Other C Operators

// C Program to demonstrate the use of Misc operators

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

```
int main()  
{  
    // integer variable  
    int num = 10;  
    int* add_of_num = &num;  
  
    printf("sizeof(num) = %d bytes\n", sizeof(num));  
    printf("&num = %p\n", &num);  
    printf("*add_of_num = %d\n", *add_of_num);  
    printf("(10 < 5) ? 10 : 20 = %d\n", (10 < 5) ? 10 : 20);  
    printf("(float)num = %f\n", (float)num);  
  
    return 0;  
}
```

Output

```
sizeof(num) = 4 bytes
```



```

&num = 0x7ffdb58c037c
*add_of_num = 10
(10 < 5) ? 10 : 20 = 20
(float)num = 10.000000

```

Operator Precedence and Associativity

Operator Precedence and Associativity is the concept that decides which operator will be evaluated first in the case when there are multiple operators present in an expression to avoid ambiguity. As, it is very common for a C expression or statement to have multiple operators and in this expression.

The below table describes the precedence order and associativity of operators in C. The precedence of the operator decreases from top to bottom.

Precedence	Operator	Description	Associativity
1	()	Parentheses (function call)	left-to-right
	[]	Brackets (array subscript)	left-to-right
	.	Member selection via object name	left-to-right
	->	Member selection via a pointer	left-to-right
	a++ , a--	Postfix increment/decrement (a is a variable)	left-to-right
2	++a , --a	Prefix increment/decrement (a is a variable)	right-to-left
	+ , -	Unary plus/minus	right-to-left
	! , ~	Logical negation/bitwise complement	right-to-left
	(type)	Cast (convert value to temporary value of type)	right-to-left
	*	Dereference	right-to-left
	&	Address (of operand)	right-to-left
	sizeof	Determine size in bytes on this implementation	right-to-left
3	* , / , %	Multiplication/division/modulus	left-to-right

Precedence	Operator	Description	Associativity
4	+, -	Addition/subtraction	left-to-right
5	<<, >>	Bitwise shift left, Bitwise shift right	left-to-right
6	<, <=	Relational less than/less than or equal to	left-to-right
	>, >=	Relational greater than/greater than or equal to	left-to-right
7	==, !=	Relational is equal to/is not equal to	left-to-right
8	&	Bitwise AND	left-to-right
9	^	Bitwise XOR	left-to-right
10		Bitwise OR	left-to-right
11	&&	Logical AND	left-to-right
12		Logical OR	left-to-right
13	?:	Ternary conditional	right-to-left
14	=	Assignment	right-to-left
	+=, -=	Addition/subtraction assignment	right-to-left
	*=, /=	Multiplication/division assignment	right-to-left
	%=, &=	Modulus/bitwise AND assignment	right-to-left
	^=, =	Bitwise exclusive/inclusive OR assignment	right-to-left
	<<=, >>=	Bitwise shift left/right assignment	right-to-left

Precedence	Operator	Description	Associativity
15	,	expression separator	left-to-right

Loop Interruption statements

Break Statement

The break statement in C is a loop control statement that breaks out of the loop when encountered. It can be used inside loops or switch statements to bring the control out of the block. The break statement can only break out of a single loop at a time.

Example:

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

```
int main() {  
    for (int i = 1; i <= 10; i++) {  
  
        // Exit the loop when i equals 5  
        if (i == 5) {  
            break;  
        }  
        printf("%d ", i);  
    }  
    return 0;  
}
```

Output

```
1 2 3 4
```

Explanation: In this program, the break statement exits the loop when `i == 5`. As a result, the loop stops prematurely, and only the numbers 1 to 4 are printed.

Syntax of break in C

```
// in a block {  
    break;  
}
```

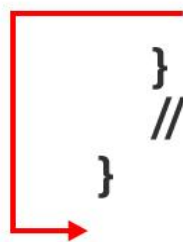
We just put the break wherever we want to terminate the execution of the loop.

How break in C Works?

```

for( init; condition; operation)
{
    // code
    if(condition to break)
    {
        break;
    }
    // code
}

```



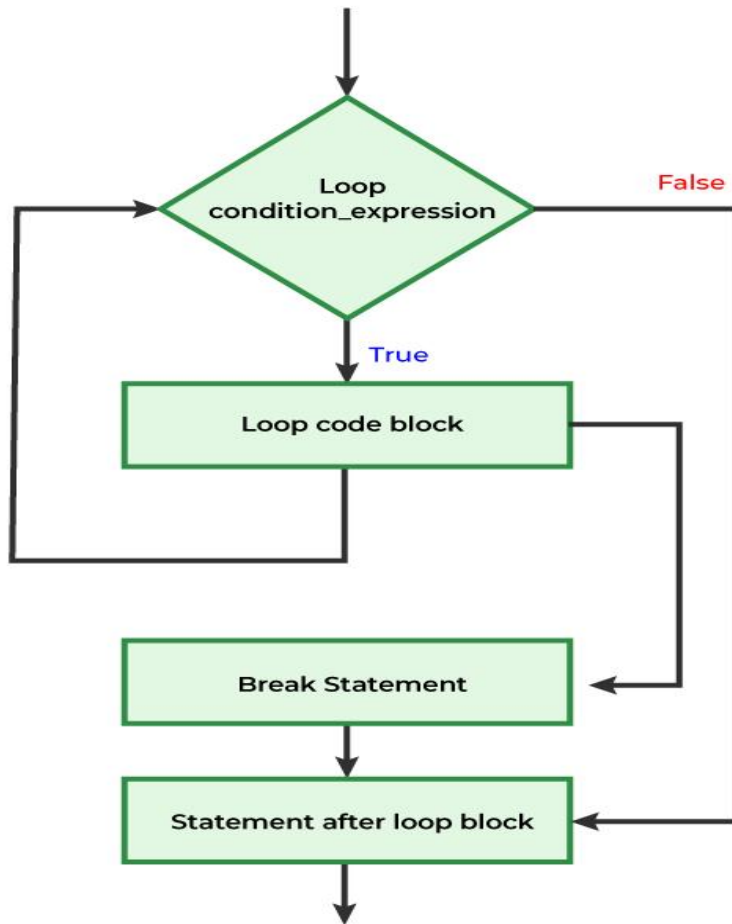
Working of break statement in C

The working of the break statement in C is described below:

1. **STEP 1:** The loop execution starts after the test condition is evaluated.
2. **STEP 2:** If the break condition is present the condition will be evaluated.
3. **STEP 3A:** If the condition is true, the program control reaches the break statement and skips the further execution of the loop by jumping to the statements directly below the loop.
4. **STEP 3B:** If the condition is false, the normal flow of the program control continues.

Flowchart of break Statement

Break Statement Flow Diagram



Flowchart for Break

Statement in C

Examples of break in C

The following examples illustrate the use of break in C programming:

break with Nested Loops

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

```
int main() {  
  
    // Nested for loops with break statement  
    // at inner loop  
    for (int i = 1; i <= 6; ++i) {  
        for (int j = 1; j <= i; ++j) {  
            if (i <= 4) {  
                printf("%d ", j);  
            }  
            else {  
  
                // If i > 4 then this innermost loop will  
                // break  
                break;  
            }  
        }  
        printf("\n");  
    }  
    return 0;  
}
```

```
}
```

Output

```
1
1 2
1 2 3
1 2 3 4
```

Explanation: In the above program the inner loop breaks when the value of i becomes equal to 4, which stops the printing of the value , although the outer loop continues to run but as the inner loop encounters a break statement for every iteration of i after the value of i becomes equal to 4, no values are printed after the fourth line.

break with Simple Loops

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

```
int main(){
```

```
    // Using break inside for loop to terminate
```

```
    // after 2 iterations
```

```
    printf("break in for loop\n");
```

```
    for (int i = 1; i < 5; i++) {
```

```
        if (i == 3) {
```

```
            break;
```

```
        }
```

```
        else {
```

```
            printf("%d ", i);
```

```
        }
```

```
    }
```

```
    // using break inside while loop to terminate
```

```
    // after 2 iterations
```

```
    printf("\nbreak in while loop\n");
```

```
    int i = 1;
```

```
    while (i < 20) {
```

```
        if (i == 3)
```

```
            break;
```

```
        else
```

```
            printf("%d ", i);
```

```
        i++;
```

```
    }
```

```
    return 0;
```

```
}
```

Output

```
break in for loop
```

```
1 2
```

```
break in while loop
```

```
1 2
```

Explanation: In this code, the break statement is used in both a for and while loop to terminate after two iterations. In both loops, numbers 1 and 2 are printed, and when i reaches

3, the break statement exits the loop, stopping further iterations. This demonstrates how break can be used to stop execution based on a specific condition.

Break in C switch case

In general, the Switch case statement evaluates an expression, and depending on the value of the expression, it executes the statement associated with the value. Not only that, all the cases after the matching case after the matching case will also be executed. To prevent that, we can use the break statement in the switch case as shown:

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

int main() {
    char c;
    float x, y;

    while (1) {
        printf("Enter an operator (+, -), if want to exit "
            "press x: ");
        scanf(" %c", &c);
        // to exit
        if (c == 'x')
            exit(0);

        printf("Enter Two Values:\n ");
        scanf("%f %f", &x, &y);

        switch (c) {

            // For Addition
            case '+':
                printf("%.1f + %.1f = %.1f\n", x, y, x + y);
                break;

            // For Subtraction
            case '-':
                printf("%.1f - %.1f = %.1f\n", x, y, x - y);
                break;
            default:
                printf(
                    "Error! please write a valid operator\n");
        }
    }
}
```

Output:

```
Enter an operator (+, -), if want to exit press x: +
Enter Two Values:
10
20
10.0 + 20.0 = 30.0
```

Explanation: This C program uses a while loop to repeatedly prompt the user for an operator and two numbers, performing addition or subtraction based on the operator entered. The break statement is used in the switch case to exit after executing the relevant operation, and the

program exits when the user enters 'x'. If an invalid operator is provided, an error message is displayed.

break vs continue

The difference between the break and continue in C is listed in the below table:

break	continue
The break statement terminates the loop and brings the program control out of the loop.	The continue statement terminates only the current iteration and continues with the next iterations.
The syntax is: break;	The syntax is: continue;
The break can also be used in switch case.	Continue can only be used in loops.

Unit III

Arrays

An array in C is a fixed-size collection of similar data items.

- Items are stored in contiguous memory locations.
- Can be used to store the collection of primitive data types such as int, char, float, etc., as well as derived and user-defined data types such as pointers, structures, etc.

// A simple C program to demonstrate

// working of arrays

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

```
int main() {
```

```
    int arr[] = {2, 4, 8, 12, 16, 18};
```

```
    int n = sizeof(arr)/sizeof(arr[0]);
```

```
    // Printing array elements
```

```
    for (int i = 0; i < n; i++) {
```

```
        printf("%d ", arr[i]);
```

```
    }
```

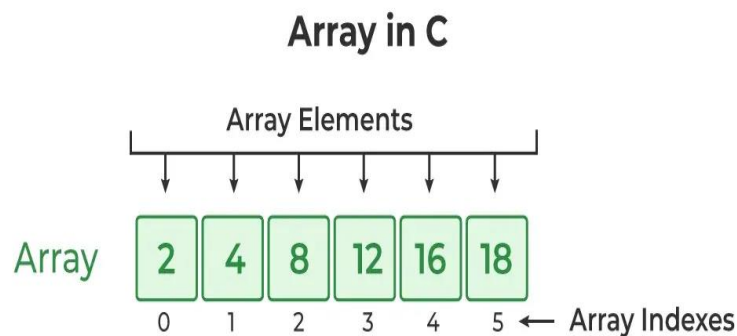
```
    return 0;
```

```
}
```

Output

```
2 4 8 12 16 18
```

The below image shows the array created in the above program.



Creating an Array in C

The whole process of creating an array in C language can be divided into two primary sub processes i.e.

1. Array Declaration

Array declaration is the process of specifying the type, name, and size of the array. In C, we have to declare the array like any other variable before using it.

```
data_type array_name[size];
```

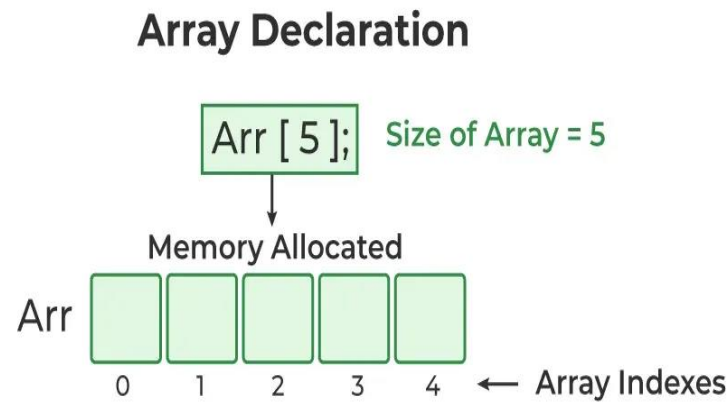
The above statements create an array with the name **array_name**, and it can store a specified number of elements of the same data type.

Example:

// Creates array arr to store 5 integer values.

```
int arr[5];
```

When we declare an array in C, the compiler allocates the memory block of the specified size to the array name.



2. Array Initialization

Initialization in C is the process to assign some initial value to the variable. When the array is declared or allocated memory, the elements of the array contain some garbage value. So, we need to initialize the array to some meaningful values.

Syntax:

```
int arr[5] = {2, 4, 8, 12, 16};
```

The above statement creates an array **arr** and assigns the values **{2, 4, 8, 12, 16}** at the time of declaration.

We can skip mentioning the size of the array if declaration and initialisation are done at the same time. This will create an array of size n where n is the number of elements defined during array initialisation. We can also **partially initialize** the array. In this case, the remaining elements will be assigned the value 0 (or equivalent according to the type).

//Partial Initialisation

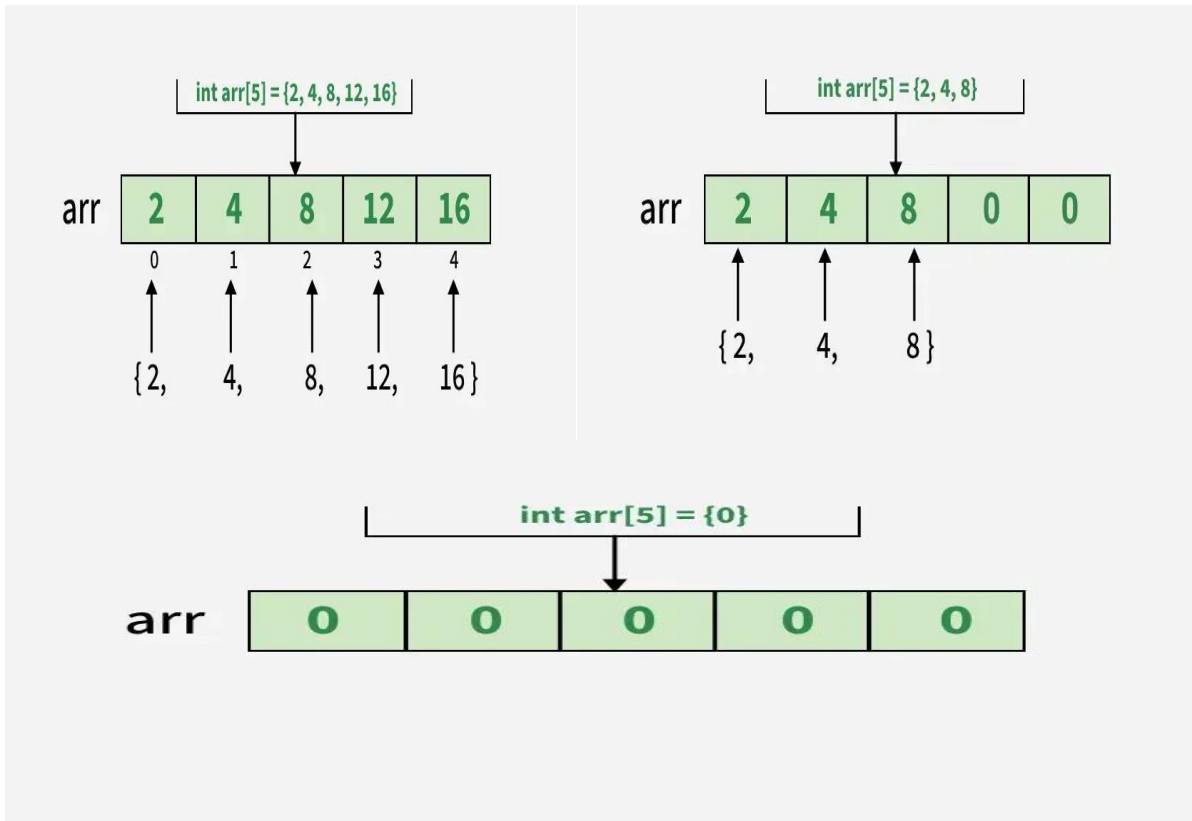
```
int arr[5] = {2, 4, 8};
```

//Skipping the size of the array.

```
int arr[] = {2, 4, 8, 12, 16};
```

//initialize an array with all elements set to 0.

```
int arr[5] = {0};
```



Accessing Array Elements

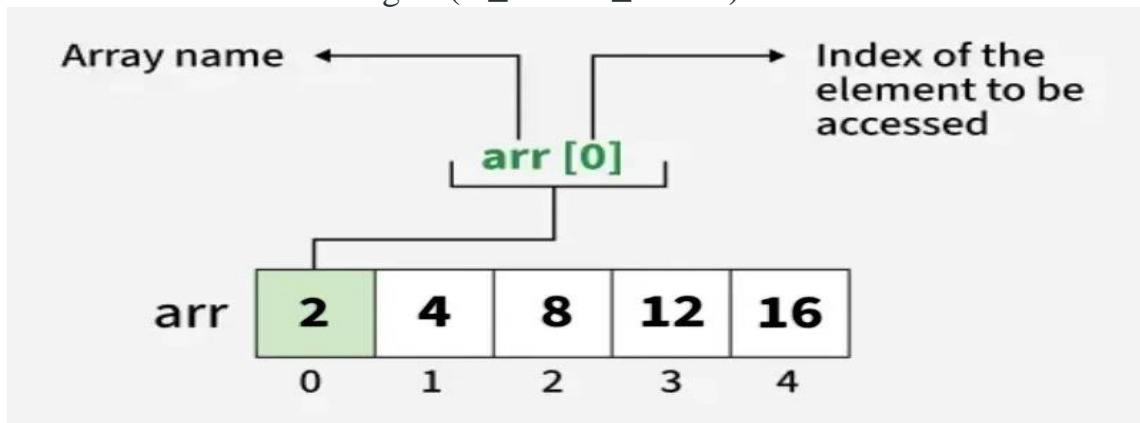
Array in C provides random access to its elements, which means that we can access any element of the array by providing the position of the element, called the index.

Syntax:

The index values start from **0** and goes up to **array_size-1**. We pass the index inside **square brackets []** with the name of the array.

`array_name [index];`

where, index value lies into this range - $(0 \leq \text{index} \leq \text{size}-1)$.



Example:

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

```
int main() {
```

```
    // array declaration and initialization
```

```
    int arr[5] = {2, 4, 8, 12, 16};
```

```
    // accessing element at index 2 i.e 3rd element
```

```
    printf("%d ", arr[2]);
```

```

// accessing element at index 4 i.e last element
printf("%d ", arr[4]);

// accessing element at index 0 i.e first element
printf("%d ", arr[0]);
return 0;
}

```

Output

```
8 16 2
```

Update Array Element

We can update the value of array elements at the given index *i* in a similar way to accessing an element by using the array **square brackets []** and **assignment operator (=)**.

```
array_name[i] = new_value;
```

Example:

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

```

int main() {
    int arr[5] = {2, 4, 8, 12, 16};

    // Update the first value
    // of the array
    arr[0] = 1;
    printf("%d", arr[0]);
    return 0;
}

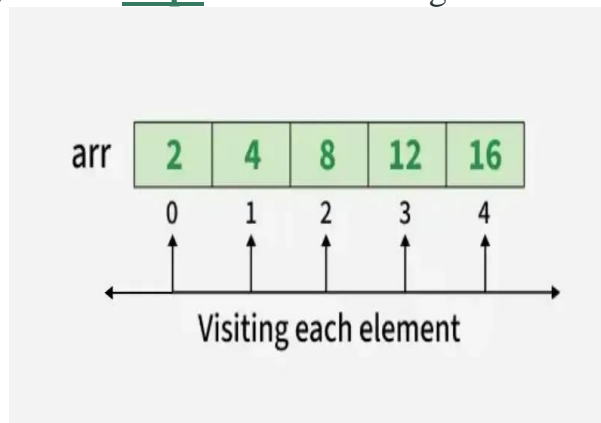
```

Output

```
1
```

C Array Traversal

Array Traversal is the process in which we visit every element of the array in a specific order. For C array traversal, we use loops to iterate through each element of the array.



Traversing An Array

Example:

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

```
int main() {  
    int arr[5] = {2, 4, 8, 12, 16};  
  
    // Print each element of  
    // array using loop  
    printf("Printing Array Elements\n");  
    for(int i = 0; i < 5; i++){  
        printf("%d ", arr[i]);  
    }  
    printf("\n");  
    // Printing array element in reverse  
    printf("Printing Array Elements in Reverse\n");  
    for(int i = 4; i >= 0; i--){  
        printf("%d ", arr[i]);  
    }  
    return 0;  
}
```

Output

```
Printing Array Elements
```

```
2 4 8 12 16
```

```
Printing Array Elements in Reverse
```

```
16 12 8 4 2
```

Size of Array

The size of the array refers to the number of elements that can be stored in the array. The array does not contain the information about its size but we can extract the size using sizeof() operator.

Example:

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

```
int main() {  
    int arr[5] = {2, 4, 8, 12, 16};  
  
    // Size of the array  
    int size = sizeof(arr)/sizeof(arr[0]);  
    printf("%d", size);  
    return 0;  
}
```

Output

```
5
```

The sizeof() operator returns the size in bytes. sizeof(arr) returns the total number of bytes of the array. In an array, each element is of type int, which is 4 bytes. Therefore, we can calculate the size of the array by dividing the total number of bytes by the byte size of one element.

Arrays and Pointers

Arrays and Pointers are closely related to each other such that we can use pointers to perform all the possible operations of the array. The array name is a constant pointer to the first element of the array and the array decays to the pointers when passed to the function.

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

```
int main() {  
  
    int arr[5] = { 10, 20, 30, 40, 50 };  
    int* ptr = &arr[0];  
  
    // Address store inside  
    // name  
    printf("%p\n", arr);  
  
    // Print the address which  
    // is pointed by pointer ptr  
    printf("%p\n", ptr);  
    return 0;  
}
```

Output

```
0x7ffa0c00c50
```

```
0x7ffa0c00c50
```

Passing Array to Function

In C, arrays are passed to functions using pointers, as the array name decays to a pointer to the first element. So, we also need to pass the size of the array to the function.

Example:

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

```
// Functions that take array as argument  
void printArray(int arr[], int n) {  
    for (int i = 0; i < n; i++)  
        printf("%d ", arr[i]);  
}  
  
int main() {  
    int arr[] = {2, 4, 8, 12, 16};  
    int n = sizeof(arr) / sizeof(arr[0]);  
  
    // Passing array  
    printArray(arr, n);  
    return 0;  
}
```

Output

There are also other ways to pass array to functions. Refer to the article - [Pass Array to Functions in C](#)

Multidimensional Array in C

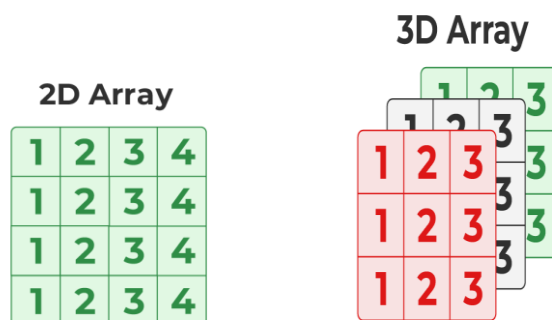
Multi-dimensional arrays in C are arrays that grow in multiple directions or dimensions. A one-dimensional array grows linearly, like parallel to the X-axis, while in a multi-dimensional array, such as a two-dimensional array, the elements can grow along both the X and Y axes.

Syntax

```
type array_name[size1][size2] .. [sizeN];
```

Some commonly used multidimensional arrays are:

- **Two-Dimensional Array:** It is an array that has exactly two dimensions. It can be visualized in the form of rows and columns organized in a two-dimensional plane.
- **Three-Dimensional Array:** A 3D array has exactly three dimensions. It can be visualized as a collection of 2D arrays stacked on top of each other to create the third dimension.



Functions

A function is a named block of code that performs a specific task. It allows you to write a piece of logic once and reuse it wherever needed in the program. This helps keep your code clean, organized, and easier to understand.

Functions play a vital role in building modular programs. They allow you to break down complex problems into smaller, manageable parts.

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

```
// Void function definition
```

```
void hello() {
    printf("GeeksforGeeks\n");
}
```

```
// Return-type function definition
```

```
int square(int x) {
    return x * x;
}
```

```

int main() {

    // Calling the void function
    hello();

    // Calling the return-type function
    int result = square(5);
    printf("Square of 5 is: %d", result);

    return 0;
}

```

Output

GeeksforGeeks

Square of 5 is: 25

In the above example, there are three functions:

- **main() function:** This is the starting point of every C program. When the program runs, execution begins from the main function.
- **hello() function:** This is a user-defined function that does not take any input and does not return a value. Its purpose is to print "GeeksforGeeks" to the screen. It is called inside the main function using hello();.
- **square() function:** This is another user-defined function, but unlike hello(), it has a return type. It takes one integer as input and returns the square of that number. In main(), we call square(5) and store the returned result in a variable to print it.

How Functions Work in C?

Function Syntax

Here is the basic structure:

```

return_type function_name(parameter_list) {
    // body of the function
}

```

Explanation of each part:

- **Return type:** Specifies the type of value the function will return. Use void if the function does not return anything.
- **Function name:** A unique name that identifies the function. It follows the same naming rules as variables.
- **Parameter list:** A set of input values passed to the function. If the function takes no inputs, this can be left empty or written as void.
- **Function body:** The block of code that runs when the function is called. It is enclosed in curly braces { }.

Function Declaration vs Definition

It's important to understand the difference between declaring a function and defining it. Both play different roles in how the compiler understands and uses your function.

Function Declaration

A declaration tells the compiler about the function's name, return type, and parameters before it is actually used. It does not contain the function's body. This is often placed at the top of the program or in a header file.

```
// function declaration  
int add(int a, int b);
```

Function Definition

A definition provides the actual implementation of the function. It includes the full code or logic that runs when the function is called.

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

Why is declaration needed?

If a function is defined after the main function or another function that uses it, then a declaration is needed before it is called. This helps the compiler recognize the function and check for correct usage.

In short, the declaration introduces the function to the compiler, and the definition explains what it actually does.

Calling a Function

Once a function is defined, you can use it by simply calling its name followed by parentheses. This tells the program to execute the code inside that function.

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

```
// Function definition
```

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

```
int main() {
```

```
    // Function call
```

```
    int result = add(5, 3);  
    printf("The sum is: %d", result);  
    return 0;  
}
```

Output

```
The sum is: 8
```

In this example, the function add is called with the values 5 and 3. The function runs its logic (adding the numbers) and returns the result, which is then stored in the variable result.

You can call a function as many times as needed from main or other functions. This helps avoid writing the same code multiple times and keeps your program clean and organized.

Types of Function in C

In C programming, functions can be grouped into two main categories: [library functions](#) and [user-defined functions](#). Based on how they handle input and output, user-defined functions can be further classified into different types.

1. Library Functions: These are built-in functions provided by C, such as [printf\(\)](#), [scanf\(\)](#), [sqrt\(\)](#), and many others. You can use them by including the appropriate [header file](#), like `#include <stdio.h>` or `#include <math.h>`.

2. User-Defined Functions: These are functions that you create yourself to perform specific tasks in your program. Depending on whether they take input or return a value, they can be of four types:

- **No arguments, no return value:** The function neither takes input nor returns any result.
- **Arguments, no return value:** The function takes input but does not return anything.
- **No arguments, return value:** The function does not take input but returns a result.
- **Arguments and return value:** The function takes input and returns a result.

Each type serves different purposes depending on what the program needs. Using the right type helps make your code more organized and efficient.

Memory Management of Functions

When a function is called, memory for its variables and other data is allocated in a separate block in a stack called a **stack frame**. The stack in which it is created is called **function call stack**. When the function completes its execution, its stack frame is deleted from the stack, freeing up the memory.

Refer to this article to know more [Function Call Stack in C](#)

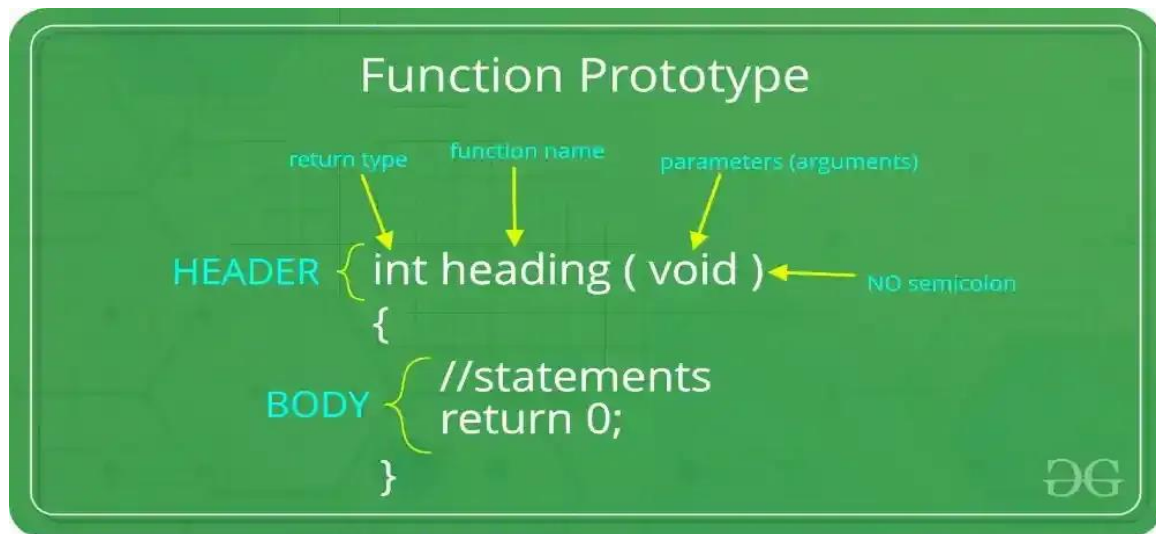
Disadvantages of Functions

While functions offer many benefits like modularity and code reuse, there are also a few limitations:

- **Overhead of function calls:** Every time a function is called, it adds a small overhead due to jumping between different parts of the program and managing memory (like pushing arguments to the stack).
- **Less efficient for very small tasks:** For simple one line operations, using a function might introduce unnecessary complexity and reduce performance slightly, especially in tight loops.
- **Code spread across multiple locations:** When functions are used heavily, especially in large projects, it can become harder to trace the complete flow of logic since different pieces are in different places.
- **Too many functions may reduce readability:** If not well organized or named properly, excessive use of small functions can make the code harder to follow instead of easier.
- **Limited access to local variables:** Variables defined inside a function are local to that function. Sharing data across multiple functions often requires global variables or parameters, which can complicate program design.

Function Prototype in C

A **function prototype** is a statement that tells the compiler about the function's name, its return type, numbers, and data types of its parameters. **Function prototype** provides a means for the compiler to cross-check function parameters and their data type with the function definition and the function call.



For example:

*// Not a prototype as it does not contain the number
// and type of parameters. It is only declaration*

```
int sum();
```

*// Valid prototype as it contains function name,
// return type, number and type of parameters*

```
int sum(int, int);
```

// Also a valid prototype

```
int sum(int a, int b);
```

// Function definition inherently contains function

// prototype

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

The above program contains a function prototype for a function named sum that adds two numbers which it takes as arguments and returns their sum.

Syntax For Function Prototype in C

```
return_type name(type1, type2 ....);
```

where,

- **return_type** : Type of value the function returns.
- **name**: Name of the function
- **type1, type2**: Type of the parameters. Specifying their names is optional.

Function prototype can be used in place of function declaration in cases where the function reference or call is present before the function definition but if the function definition is present before the function call in the program then the function prototype is not necessary.

Example:

The following program demonstrates the usage of function prototype.

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

```
int sum(int x, int y);
```

```

int main() {
    int x = 5, y=10;

    printf("Sum of x and y is %d\n",sum(x, y));

    return 0;
}

int sum (int x , int y)
{
    return x+y;
}

```

In the above program the function sum is called before its definition, since the program contains prototype for the sum function the program executes without any errors.

Benifits of Function Prototypes

Following are the major benefits of including function prototypes in your program:

- **Function Declaration Before Definition:** It allows a function to be called before [function definition](#). In large programs, it is common to place function prototypes at the beginning of the code or in header files, enabling function calls to occur before the function's actual implementation.
- **Type Checking:** A function prototype allows the compiler to check that the correct number and type of arguments are passed to the function. If the function is called with the wrong type or number of parameters, the compiler can catch the error.
- **Code Clarity:** By declaring prototypes, you inform the programmer about the function's purpose and expected parameters, which helps in understanding the code structure.

Function Declaration vs Prototype

The terms function declaration and function prototypes are often used interchangeably but they are different in the purpose and their meaning. Following are the major differences between the function declaration and function prototype in C:

Function Declaration	Function Prototype
Function Declaration is used to tell the existence of a function.	The function prototype tells the compiler about the existence and signature of the function.
A function declaration is valid even with only function name and return type.	A function prototype is a function delcaration that provides the function's name, return type, and parameter list without including the function body.
Typically used in header files to declare functions.	Used to declare functions before their actual definitions.

Function Declaration	Function Prototype
Syntax: return_type function_name();	Syntax: return_type function_name(parameter_list);

Call by Value

Call by value in C is where in the arguments we pass value and that value can be used in function for performing the operation. Values passed in the function are stored in temporary memory so the changes performed in the function don't affect the actual value of the variable passed.

Example:

```
// C Program to implement
// Call by value
#include <stdio.h>
```

```
// Call by value
int sum(int x, int y)
{
    int c;
    c = x + y;

    // Integer value returned
    return c;
}
```

```
// Driver Code
int main()
{
    // Integer Declared
    int a = 3, b = 2;

    // Function Called
    int c = sum(a, b);
    printf("Sum of %d and %d : %d", a, b, c);

    return 0;
}
```

Output

```
Sum of 3 and 2 : 5
```

Call by Reference

Call by reference is the method in C where we call the function with the passing address as arguments. We pass the address of the memory blocks which can be further stored in a pointer variable that can be used in the function. Now, changes performed in the values inside the function can be directly reflected in the main memory.

Example:

// C Program to implement

// Call by reference

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

// Call by reference

```
void swap(int* x, int* y)
```

```
{  
    int temp = *x;  
    *x = *y;  
    *y = temp;  
}
```

// Driver Code

```
int main()
```

```
{  
    // Declaring Integer  
    int x = 1, y = 5;  
    printf("Before Swapping: x:%d , y:%d\n", x, y);
```

// Calling the function

```
    swap(&x, &y);  
    printf("After Swapping: x:%d , y:%d\n", x, y);  
  
    return 0;  
}
```

Output

Before Swapping: x:1 , y:5

After Swapping: x:5 , y:1

Types of Function According to Arguments and Return Value

Functions can be differentiated into 4 types according to the arguments passed and value returns these are:

1. Function with arguments and return value
2. Function with arguments and no return value
3. Function with no arguments and with return value
4. Function with no arguments and no return value

1. Function with arguments and return value

Syntax:

Function declaration : int function (int);

Function call : function(x);

Function definition:

```
int function( int x )  
{  
    statements;  
    return x;  
}
```

Example:

// C code for function with arguments

// and return value

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

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

```
int function(int, int[]);
```

```
int main()
```

```
{
```

```
    int i, a = 20;
```

```
    int arr[5] = { 10, 20, 30, 40, 50 };
```

```
    a = function(a, &arr[0]);
```

```
    printf("value of a is %d\n", a);
```

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

```
        printf("value of arr[%d] is %d\n", i, arr[i]);
```

```
    }
```

```
    return 0;
```

```
}
```

```
int function(int a, int* arr)
```

```
{
```

```
    int i;
```

```
    a = a + 20;
```

```
    arr[0] = arr[0] + 50;
```

```
    arr[1] = arr[1] + 50;
```

```
    arr[2] = arr[2] + 50;
```

```
    arr[3] = arr[3] + 50;
```

```
    arr[4] = arr[4] + 50;
```

```
    return a;
```

```
}
```

Output

```
value of a is 40
```

```
value of arr[0] is 60
```

```
value of arr[1] is 70
```

```
value of arr[2] is 80
```

```
value of arr[3] is 90
```

```
value of arr[4] is 100
```

2. Function with arguments but no return value

When a function has arguments, it receives any data from the calling function but it returns no values. These are void functions with no return values.

Syntax:

Function declaration : void function (int);

Function call : function(x);

Function definition:

```
void function( int x )
{
    statements;
}
```

Example:

// C code for function

// with argument but no return value

#include <stdio.h>

```
void function(int, int[], char[]);
```

```
int main()
```

```
{
```

```
    int a = 20;
```

```
    int ar[5] = { 10, 20, 30, 40, 50 };
```

```
    char str[30] = "geeksforgeeks";
```

```
        // function call
```

```
    function(a, &ar[0], &str[0]);
```

```
    return 0;
```

```
}
```

```
void function(int a, int* ar, char* str)
```

```
{
```

```
    int i;
```

```
    printf("value of a is %d\n\n", a);
```

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

```
        printf("value of ar[%d] is %d\n", i, ar[i]);
```

```
    }
```

```
    printf("\nvalue of str is %s\n", str);
```

```
}
```

Output

```
value of a is 20
```

```
value of ar[0] is 10
```

```
value of ar[1] is 20
```

```
value of ar[2] is 30
```

```
value of ar[3] is 40
```

```
value of ar[4] is 50
```


value of str is geeksforgeeks

3. Function with no argument and no return value

When a function has no arguments, it does not receive any data from the calling function. Similarly, when it does not return a value, the calling function does not receive any data from the called function.

Syntax:

Function declaration : void function();

Function call : function();

Function definition :

```
void function()
{
    statements;
}
```

Example:

```
// C code for function with no
// arguments and no return value
#include <stdio.h>
```

```
void value(void);
```

```
void main() {
    value();
}
```

```
void value(void)
{
    float year = 1, period = 5, amount = 5000,
        inrate = 0.12;
    float sum;
    sum = amount;
    while (year <= period) {
        sum = sum * (1 + inrate);
        year = year + 1;
    }
    printf(" The total amount is :%f", sum);
}
```

Output

The total amount is :8811.708984

4. Function with no arguments but returns a value

There could be occasions where we may need to design functions that may not take any arguments but returns a value to the calling function. An example of this is getchar function which has no parameters but it returns an integer and integer-type data that represents a character.

Syntax:

Function declaration : int function();

Function call : function();

Function definition :

```
int function()
{
    statements;
    return x;
}
```

Example:

// C code for function with no arguments

// but have return value

```
#include <math.h>
```

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

```
int sum();
```

```
int main()
```

```
{
    int num;
    num = sum();
    printf("Sum of two given values = %d", num);
    return 0;
}
```

```
int sum()
```

```
{
    int a = 50, b = 80, sum;
    sum = sqrt(a) + sqrt(b);
    return sum;
}
```

Output

```
Sum of two given values = 16
```

Recursion

Recursion is a programming technique where a function calls itself repeatedly until a specific base condition is met. A function that performs such self-calling behavior is known as a **recursive function**, and each instance of the function calling itself is called a **recursive call**.

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

```
void rec(int n) {
```

```
    // Base Case
```

```
    if (n == 6) return;
```

```
    printf("Recursion Level %d\n", n);
```

```
    rec(n + 1);
```

```
}
```

```
int main() {
```

```
    rec(1);
```

```
    return 0;
```

```
}
```

Output

Recursion Level 1

Recursion Level 2

Recursion Level 3

Recursion Level 4

Recursion Level 5

This code demonstrates a simple recursive function that prints the current recursion level from 1 to 5. The function `rec(n)` calls itself with an incremented value of `n` until it reaches the base case `n == 6`, at which point the recursion stops.

Recursive Functions

A **recursive function** is a function that solves a problem by calling itself on a smaller subproblem. It typically consists of two main parts:

1. **Base Case** – the condition under which the recursion stops.
2. **Recursive Case** – where the function calls itself with modified arguments, gradually approaching the base case.

```
returntype function(parameters) {  
  
    // base case  
    if (base condition) {  
        return base value;  
    }  
  
    // recursive case  
    return recursive expression involving function(modified parameters);  
}
```

This structure allows problems to be broken down into simpler versions of themselves, making recursion a powerful tool for solving problems that can be defined in terms of smaller instances.

How Recursion works?

To understand how recursion works internally, it's important to see how the **call stack** behaves during recursive calls. Each time a function calls itself, the current state is saved on the stack, and the new call begins. Once the base case is reached, the function starts returning back, one call at a time.

The following example demonstrates both the **descending phase** (going deeper into recursion) and the **ascending phase** (returning back from recursion):

```
#include <stdio.h>  
  
void f(int n) {  
    printf("F(%d)'s Stack Frame Pushed\n", n);  
  
    if (n > 1) {  
        f(n - 1);  
        f(n - 1);  
    }  
}
```

```

    printf("F(%d)'s Stack Frame Removed\n", n);
}

int main() {
    f(3);
    return 0;
}

```

Output

```

F(3)'s Stack Frame Pushed
F(2)'s Stack Frame Pushed
F(1)'s Stack Frame Pushed
F(1)'s Stack Frame Removed
F(1)'s Stack Frame Pushed
F(1)'s Stack Frame Removed
F(2)'s Stack Frame Removed
F(2)'s Stack Fr...

```

The function `f` begins by printing a message indicating that the current call's stack frame has been pushed. It then recursively calls itself twice with $n - 1$, provided $n > 1$. This continues until the base case is implicitly reached at $n == 1$, where no further recursive calls are made.

Once the base case is reached, the recursion starts to unwind. Each function call resumes after both recursive calls complete, and a message is printed showing that the corresponding stack frame is being removed.

This example illustrates the two key phases in recursion:

Descending Phase: The function continues to call itself with smaller arguments ($n - 1$) twice, creating a binary recursion. Each call pushes a new stack frame onto the call stack, deepening the recursion.

Ascending Phase: After reaching $n == 1$, the recursion begins to return. Each call completes and prints a message as its stack frame is removed, showing how the stack unwinds in reverse order. Recursion comes in various forms based on how recursive calls are structured—like tail recursion, head recursion, linear recursion, and more.

Memory Management in Recursion

Like other functions, the data of a recursive function is stored in stack memory as a stack frame. This stack frame is removed once the function returns a value. In recursion,

- The function call occurs before returning a value, so the stack frames for each recursive call are placed on top of the existing stack frames in memory.
- Once the topmost function returns a value, its stack frame is removed, and control is passed back to the function just before it, resuming execution after the point where the recursive call was made.
- The compiler uses an instruction pointer to keep track of the location to return to after the function execution is complete.
- Unlike iteration, recursion relies on the call stack, making memory management a key differentiator between the two.

Stack Overflow

The program's call stack has limited memory assigned to it by the operating system and is generally enough for program execution. But if we have a recursive function that goes on for infinite times, sooner or later, the memory will be exhausted, and no more data can be stored. This is called stack overflow. In other words,
Stack overflow is the error that occurs when the call stack of the program cannot store more data resulting in program termination.

Applications of Recursion

Recursion is widely used to solve different kinds of problems from simple ones like printing linked lists to being extensively used in AI. Some of the common uses of recursion are:

- Tree-Graph Algorithms
- Mathematical Problems
- Divide and Conquer
- Dynamic Programming
- In Postfix to Infix Conversion
- Searching and Sorting Algorithms

Advantages of Recursion

The advantages of using recursive methods over other methods are:

- Recursion can effectively reduce the length of the code.
- Some problems are easily solved by using recursion like the tower of Hanoi and tree traversals.
- Data structures like linked lists, trees, etc. are recursive by nature so recursive methods are easier to implement for these data structures.

Disadvantages of Recursion

As with almost anything in the world, recursion also comes with certain limitations some of which are:

1. Recursive functions make our program a bit slower due to function call overhead.
2. Recursion functions always take extra space in the function call stack due to separate stack frames.
3. Recursion methods are difficult to understand and implement.

Pointers

A pointer is a variable that stores the memory address of another variable. Instead of holding a direct value, it holds the address where the value is stored in memory. It is the backbone of low-level memory manipulation in C. Accessing the pointer directly will just give us the address that is stored in the pointer. For example,

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

```
int main() {
```

```
    // Normal Variable
```

```
    int var = 10;
```

```
    // Pointer Variable ptr that
```

```
    // stores address of var
```

```

int* ptr = &var;

// Directly accessing ptr will
// give us an address
printf("%d", ptr);

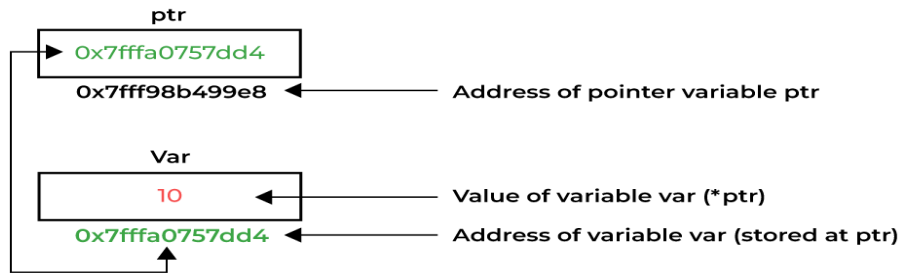
return 0;
}

```

Output

0x7ffa0757dd4

This hexadecimal integer (starting with 0x) is the memory address.



Let us understand different steps of the above program.

Declare a Pointer

A pointer is declared by specifying its name and type, just like simple variable declaration but with an **asterisk (*)** symbol added before the pointer's name.

```
data_type* name
```

Here, **data_type** defines the type of data that the pointer is pointing to. An integer type pointer can only point to an integer. Similarly, a pointer of float type can point to a floating-point data, and so on.

Example:

```
int *ptr;
```

In the above statement, pointer **ptr** can store the address of an integer. It is pronounced as pointer to integer.

Initialize the Pointer

Pointer initialization means assigning some address to the pointer variable. In C, the **(&)** **addressof operator** is used to get the memory address of any variable. This memory address is then stored in a pointer variable.

Example:

```
int var = 10;
```

```
// Initializing ptr
```

```
int *ptr = &var;
```

In the above statement, pointer **ptr** store the address of variable **var** which was determined using address-of operator (**&**).

Note: We can also declare and initialize the pointer in a single step. This is called **pointer definition**.

Dereference a Pointer

We have to first dereference the pointer to access the value present at the memory address. This is done with the help of **dereferencing operator(*)** (same operator used in declaration).

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

```
int main() {
    int var = 10;

    // Store address of var variable
    int* ptr = &var;

    // Dereferencing ptr to access the value
    printf("%d", *ptr);

    return 0;
}
```

Output

```
10
```

***Note:** Earlier, we used %d for printing pointers, but C provides a separate format specifier %p for printing pointers.*

Size of Pointers

The **size of a pointer in C** depends on the **architecture (bit system)** of the machine, **not the data type** it points to.

- On a **32-bit system**, all pointers typically occupy **4 bytes**.
- On a **64-bit system**, all pointers typically occupy **8 bytes**.

The size remains **constant regardless of the data type** (int*, char*, float*, etc.). We can verify this using the sizeof operator.

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

```
int main() {
    int *ptr1;
    char *ptr2;

    // Finding size using sizeof()
    printf("%zu\n", sizeof(ptr1));
    printf("%zu", sizeof(ptr2));

    return 0;
}
```

Output

```
8
```

```
8
```

The reason for the same size is that the pointers store the memory addresses, no matter what type they are. As the space required to store the addresses of the different memory locations is the same, the memory required by one pointer type will be equal to the memory required by other pointer types.

*Note: The actual size of the pointer may vary depending on the **compiler and system architecture**, but it is always **uniform across all data types** on the same system.*

Special Types of Pointers

There are 4 special types of pointers that used or referred to in different contexts:

NULL Pointer

The [NULL Pointers](#) are those pointers that do not point to any memory location. They can be created by assigning **NULL** value to the pointer. A pointer of any type can be assigned the **NULL** value.

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

```
int main() {  
    // Null pointer  
    int *ptr = NULL;  
  
    return 0;  
}
```

NULL pointers are generally used to represent the absence of any address. This allows us to check whether the pointer is pointing to any valid memory location by checking if it is equal to **NULL**.

Void Pointer

The [void pointers](#) in C are the pointers of type **void**. It means that they do not have any associated data type. They are also called **generic pointers** as they can point to any type and can be typecasted to any type.

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

```
int main() {  
    // Void pointer  
    void *ptr;  
  
    return 0;  
}
```

Wild Pointers

The [wild pointers](#) are pointers that have not been initialized with something yet. These types of C-pointers can cause problems in our programs and can eventually cause them to crash. If values are updated using wild pointers, they could cause data abort or data corruption.

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

```
int main() {  
  
    // Wild Pointer  
    int *ptr;  
  
    return 0;  
}
```

Dangling Pointer

A pointer pointing to a memory location that has been deleted (or freed) is called a dangling pointer. Such a situation can lead to unexpected behavior in the program and also serve as a source of bugs in C programs.

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

```
int main() {
    int* ptr = (int*)malloc(sizeof(int));

    // After below free call, ptr becomes a dangling pointer
    free(ptr);
    printf("Memory freed\n");

    // removing Dangling Pointer
    ptr = NULL;

    return 0;
}
```

Output

```
Memory freed
```

C Pointer Arithmetic

The pointer arithmetic refers to the arithmetic operations that can be performed on a pointer. It is slightly different from the ones that we generally use for mathematical calculations as only a limited set of operations can be performed on pointers. These operations include:

- **Increment/Decrement**
- **Addition/Subtraction of Integer**
- **Subtracting Two Pointers of Same Type**
- **Comparing/Assigning Two Pointers of Same Type**
- **Comparing/Assigning with NULL**

C Pointers and Arrays

In C programming language, pointers and arrays are closely related. An array name acts like a pointer constant. The value of this pointer constant is the address of the first element. For example, if we have an array named **val**, then **val** and **&val[0]** can be used interchangeably. If we assign this value to a non-constant pointer to array of the same type, then we can access the elements of the array using this pointer. Not only that, as the array elements are stored continuously, we can use pointer arithmetic operations such as increment, decrement, addition, and subtraction of integers on pointer to move between array elements. This concept is not limited to the one-dimensional array, we can refer to a multidimensional array element as well using pointers.

Constant Pointers

In **constant pointers**, the memory address stored inside the pointer is constant and cannot be modified once it is defined. It will always point to the same memory address.

Example:

```
#include <stdio.h>

int main() {
    int a = 90;
    int b = 50;

    // Creating a constant pointer
    int* const ptr = &a;

    // Trying to reassign it to b
    ptr = &b;

    return 0;
}
```

Output

```
solution.c: In function 'main':
solution.c:11:9: error: assignment of read-only variable 'ptr'
  11 |     ptr = &b;
      |     ^
```

Pointer to Function

A function pointer is a type of pointer that stores the address of a function, allowing functions to be passed as arguments and invoked dynamically. It is useful in techniques such as callback functions, event-driven programs.

Example:

```
#include <stdio.h>

int add(int a, int b) {
    return a + b;
}

int main() {

    // Declare a function pointer that matches
    // the signature of add() function
    int (*fptr)(int, int);

    // Assign address of add()
    fptr = &add;

    // Call the function via ptr
    printf("%d", fptr(10, 5));

    return 0;
}
```

Output

```
15
```

Multilevel Pointers

In C, we can create multi-level pointers with any number of levels such as – `***ptr3`, `****ptr4`, `*****ptr5` and so on. Most popular of them is double pointer (pointer to pointer). It stores the memory address of another pointer. Instead of pointing to a data value, they point to another pointer.

Example:

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

```
int main() {
    int var = 10;

    // Pointer to int
    int *ptr1 = &var;

    // Pointer to pointer (double pointer)
    int **ptr2 = &ptr1;

    // Accessing values using all three
    printf("var: %d\n", var);
    printf("*ptr1: %d\n", *ptr1);
    printf("**ptr2: %d", **ptr2);

    return 0;
}
```

Output

```
var: 10
*ptr1: 10
**ptr2: 10
```

Uses of Pointers in C

The C pointer is a very powerful tool that is widely used in C programming to perform various useful operations. It is used to achieve the following functionalities in C:

- Pass Arguments by Pointers
- Accessing Array Elements
- Return Multiple Values from Function
- Dynamic Memory Allocation
- Implementing Data Structures
- In System-Level Programming where memory addresses are useful.
- To use in Control Tables.

Advantages of Pointers

Following are the major advantages of pointers in C:

- Pointers are used for dynamic memory allocation and deallocation.
- An Array or a structure can be accessed efficiently with pointers
- Pointers are useful for accessing memory locations.
- Pointers are used to form complex data structures such as linked lists, graphs, trees, etc.
- Pointers reduce the length of the program and its execution time as well.

Issues with Pointers

Pointers are vulnerable to errors and have following disadvantages:

- Memory corruption can occur if an incorrect value is provided to pointers.
- Pointers are a little bit complex to understand.
- Pointers are majorly responsible for [memory leaks in C](#).
- Accessing using pointers are comparatively slower than variables in C.
- Uninitialized pointers might cause a [segmentation fault](#).

Address Operator & in C

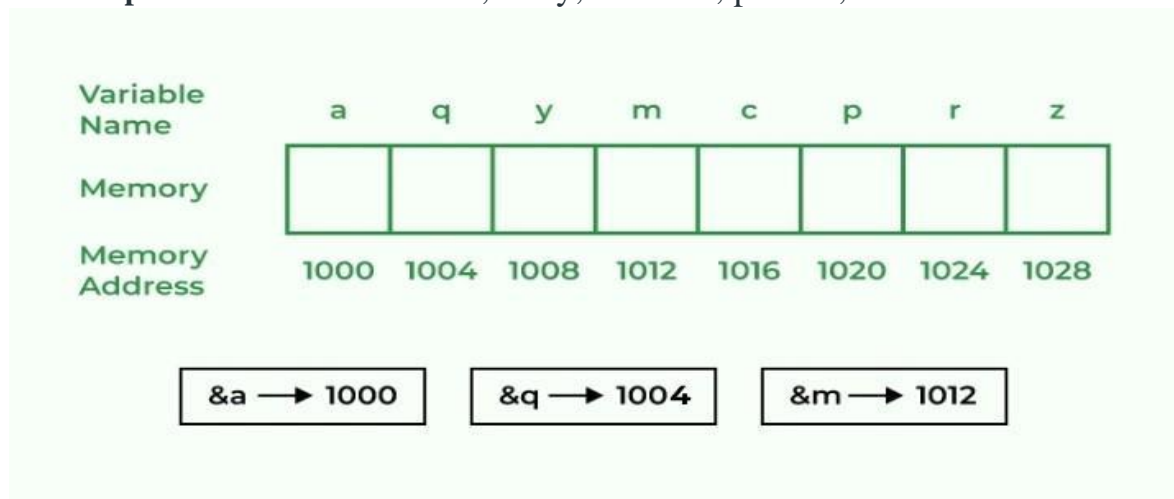
The **Address Operator in C** is a special unary operator that returns the address of a variable. It is denoted as the **Ampersand Symbol (&)**. This operator returns an integer value which is the **address of its operand** in the memory. We can use the address operator (&) with any kind of variables, array, strings, functions, and even pointers.

Syntax

The address operator is generally used as a prefix to its operand:

&operand

where **operand** can be a variable, array, function, pointer, etc.



Examples of Address Operators

Example 1:

Simple C example to demonstrate how to use the address operator in our program.

// C program to illustrate the use of address operator

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

```
int main()
{
    // declaring a variable
    int x = 100;

    // printing the address of the variable
    printf("The address of x is %p", &x);
    return 0;
}
```

Output

The address of x is 0x7ffe8f5591c

Explanation

A variable **x** was defined and initialized with the value 100 in the program above. We retrieved the address of this variable **x** by using the address operator (&) as the prefix and printed it using `printf()` function.

Note: The %p format specifier to print the address in hexadecimal form.

Generally, the value returned by the address operator is stored in the [pointer](#) variable and then the pointer is dereferenced to get the value stored in that address.

Example 2:

Using a pointer to store the address returned by the address operator and then dereferencing it.

```
// C program to illustrate the use of address operator with
// pointer
#include <stdio.h>
```

```
int main()
{
    // integer variable
    int x = 1;
    // integer pointer
    int* ptrX;
    // pointer initialization with the address of x
    ptrX = &x;

    // accessing value of x using pointer
    printf("Value of x: %d\n", *ptrX);

    return 0;
}
```

Output

```
Value of x: 1
```

Example 3:

Some standard functions like `scanf()` also require the address of the variable. In these cases, we use the address operator.

```
// C Program to illustrate the use of address operator with
// scanf()
#include <stdio.h>
```

```
int main()
{
    // defining variable
    int number;

    printf("Enter any number: ");
    // using address operator & in scanf() to get the value
    // entered by the user in the console
    scanf("%d", &number);

    // printing the entered number
    printf("The entered number is: %d", number);
    return 0;
}
```

Output

```
Enter any number: 10
The entered number is: 10
```

Address Operator Incompatible Entities in C

There are some entities in C for which we cannot use the address operator i.e. we cannot get the address of those entities in C. Some of them are:

1. Register Variables
2. Bit Fields
3. Literals
4. Expressions

Applications of Address Operator (&):

The address operator (&) is widely used in C programs to get the addresses of different entities. Some of the major and most common applications are:

1. [Passing Pointers as Function Arguments](#)
2. Pointer Arithmetic
3. Implementing Data Structures

Indirection (*) Operator

In C programming, the indirection operator, denoted by the asterisk *, is a unary operator used to dereference a pointer. This means it allows you to access the value stored at the memory location pointed to by a pointer variable.

How it works:

- **Pointer Declaration:** When you declare a pointer, the * symbol is used to indicate that the variable is a pointer type, for example:

C

```
int *ptr; // Declares 'ptr' as a pointer to an integer
```

- **Dereferencing with Indirection Operator:** Once a pointer variable holds a valid memory address, you can use the * operator before the pointer variable name to access or modify the value at that address.

C

```
int value = 10;
```

```
int *ptr = &value; // 'ptr' now holds the address of 'value'
```

```
printf("%d\n", *ptr); // This will print the value stored at the address 'ptr' points to, which is 10.
```

```
*ptr = 20; // This modifies the value at the address 'ptr' points to, so 'value' becomes 20.
```

Key Points:

- **Dereferencing:**

The primary purpose of the indirection operator is to "dereference" a pointer, meaning to retrieve the data that the pointer "points to."

- **L-value:**

When applied to a pointer to a storage location, the result of the indirection expression is an "l-value," meaning it represents a memory location that can be assigned a value.

- **Undefined Behavior:**

Applying the indirection operator to an invalid pointer (e.g., a null pointer, a pointer to deallocated memory, or an uninitialized pointer) results in undefined behavior, which can lead to crashes or unpredictable program behavior.

- **Distinction from Multiplication:**

It is important to note that the `*` symbol also serves as the multiplication operator in C. Its meaning is determined by the context: as a unary operator preceding a pointer variable, it signifies indirection; as a binary operator between two operands, it signifies multiplication.

Structures

A **structure** is a user-defined data type that can be used to group items of possibly different types into a single type. The **struct** keyword is used to define a structure. The items in the structure are called its **members** and they can be of any valid data type. Applications of structures involve creating data structures Linked List and Tree. Structures in C are also used to represent real world objects in a software like Students and Faculty in a college management software.

Example:

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

```
// Defining a structure
```

```
struct A {  
    int x;  
};
```

```
int main() {
```

```
    // Creating a structure variable
```

```
    struct A a;
```

```
    // Initializing member
```

```
    a.x = 11;
```

```
    printf("%d", a.x);
```

```
    return 0;
```

```
}
```

Output

```
11
```

Explanation: In this example, a structure **A** is defined to hold an integer member **x**. A variable **a** of type **struct A** is created and its member **x** is initialized to **11** by accessing it using dot operator. The value of **a.x** is then printed to the console.

Structures are used when you want to store a collection of different data types, such as integers, floats, or even other structures under a single name. To understand how structures are foundational to building complex data structures, the [C Programming Course Online with Data Structures](#) provides practical applications and detailed explanations.

Syntax of Structure

There are two steps of creating a structure in C:

1. Structure Definition
2. Creating Structure Variables

Structure Definition

A structure is defined using the **struct** keyword followed by the structure name and its members. It is also called a structure **template** or structure **prototype**, and no memory is allocated to the structure in the declaration.

```
struct structure_name {  
    data_type1 member1;  
    data_type2 member2;  
    ...  
};
```

- **structure_name**: Name of the structure.
- **member1, member2, ...**: Name of the members.
- **data_type1, data_type2, ...**: Type of the members.

Be careful not to forget the semicolon at the end.

Basic Operations of Structure

Following are the basic operations commonly used on structures:

1. Access Structure Members

To access or modify members of a structure, we use the (.) dot operator. This is applicable when we are using structure variables directly.

```
structure_name . member1;  
structure_name . member2;
```

In the case where we have a pointer to the structure, we can also use the **arrow operator** to access the members.

```
structure_ptr -> member1  
structure_ptr -> member2
```

2. Initialize Structure Members

Structure members **cannot be** initialized with the declaration. For example, the following C program fails in the compilation.

```
struct structure_name {  
    data_type1 member1 = value1; // COMPILER ERROR: cannot initialize members here  
    data_type2 member2 = value2; // COMPILER ERROR: cannot initialize members here  
    ...  
};
```

The reason for the above error is simple. When a datatype is declared, no memory is allocated for it. Memory is allocated only when variables are created. So there is no space to store the value assigned.

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

```
// Defining a structure to represent a student
```

```
struct Student {  
    char name[50];  
    int age;  
    float grade;  
};
```



```

int main() {

    // Declaring and initializing a structure
    // variable
    struct Student s1 = {"Rahul", 20, 18.5};

    // Designated Initializing another structure
    struct Student s2 = { .age = 18, .name =
        "Vikas", .grade = 22 };

    // Accessing structure members
    printf("%s\t%d\t%.2f\n", s1.name, s1.age,
        s1.grade);
    printf("%s\t%d\t%.2f\n", s2.name, s2.age,
        s2.grade);

    return 0;
}

```

Output

Rahul	20	18.50
Vikas	18	22.00

We can initialize structure members in 4 ways which are as follows:

Default Initialization

By default, structure members are not automatically initialized to 0 or NULL. Uninitialized structure members will contain garbage values. However, when a structure variable is declared with an initializer, all members not explicitly initialized are zero-initialized.

struct structure_name s = {0}; // Both x and y are initialized to 0

Initialization using Assignment Operator

```

struct structure_name str;
str.member1 = value1;
....

```

Note: We cannot initialize the arrays or strings using assignment operator after variable declaration.

Initialization using Initializer List

struct structure_name str = {value1, value2, value3};

In this type of initialization, the values are assigned in sequential order as they are declared in the structure template.

Initialization using Designated Initializer List

Designated Initialization allows structure members to be initialized in any order. This feature has been added in the [C99 standard](#).

struct structure_name str = { .member1 = value1, .member2 = value2, .member3 = value3 };

The Designated Initialization is only supported in C but not in C++.

3. Copy Structure

Copying structure is simple as copying any other variables. For example, s1 is copied into s2 using assignment operator.

```
s2 = s1;
```

But this method only creates a shallow copy of s1 i.e. if the structure **s1** have some dynamic resources allocated by malloc, and it contains pointer to that resource, then only the pointer will be copied to **s2**. If the dynamic resource is also needed, then it has to be copied manually (deep copy).

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

```
struct Student {
    int id;
    char grade;
};

int main() {
    struct Student s1 = {1, 'A'};

    // Create a copy of student s1
    struct Student s1c = s1;

    printf("Student 1 ID: %d\n", s1c.id);
    printf("Student 1 Grade: %c", s1c.grade);
    return 0;
}
```

Output

```
Student 1 ID: 1
Student 1 Grade: A
```

4. Passing Structure to Functions

Structure can be passed to a function in the same way as normal variables. Though, it is recommended to pass it as a pointer to avoid copying a large amount of data.

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

```
// Structure definition
```

```
struct A {
    int x;
};
```

```
// Function to increment values
```

```
void increment(struct A a, struct A* b) {
    a.x++;
    b->x++;
}
```

```
int main() {
    struct A a = { 10 };
    struct A b = { 10 };
}
```

```
// Passing a by value and b by pointer
increment(a, &b);

printf("a.x: %d \tb.x: %d", a.x, b.x);
return 0;
}
```

Output

```
a.x: 10   b.x: 11
```

5. typedef for Structures

The [typedef](#) keyword is used to define an alias for the already existing datatype. In structures, we have to use the struct keyword along with the structure name to define the variables. Sometimes, this increases the length and complexity of the code. We can use the typedef to define some new shorter name for the structure.

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

```
// Defining structure
```

```
typedef struct {
    int a;
} str1;
```

```
// Another way of using typedef with structures
```

```
typedef struct {
    int x;
} str2;
```

```
int main() {
```

```
    // Creating structure variables using new names
```

```
    str1 var1 = { 20 };
    str2 var2 = { 314 };
```

```
    printf("var1.a = %d\n", var1.a);
    printf("var2.x = %d\n", var2.x);
    return 0;
```

```
}
```

Output

```
var1.a = 20
```

```
var2.x = 314
```

Explanation: In this code, **str1** and **str2** are defined as aliases for the unnamed structures, allowing the creation of structure variables (**var1** and **var2**) using these new names. This simplifies the syntax when declaring variables of the structure.

Size of Structures

Technically, the size of the structure in C should be the sum of the sizes of its members. But it may not be true for most cases. The reason for this is Structure Padding.

Structure padding is the concept of adding multiple empty bytes in the structure to naturally align the data members in the memory. It is done to minimize the CPU read cycles to retrieve different data members in the structure.

There are some situations where we need to pack the structure tightly by removing the empty bytes. In such cases, we use **Structure Packing**. C language provides two ways for structure packing:

1. Using `#pragma pack(1)`
2. Using `__attribute__((packed))`

Nested Structures

In C, a nested structure refers to a structure that contains another structure as one of its members. This allows you to create more complex data types by grouping multiple structures together, which is useful when dealing with related data that needs to be grouped within a larger structure.

There are two ways in which we can nest one structure into another:

- **Embedded Structure Nesting:** The structure being nested is also declared inside the parent structure.
- **Separate Structure Nesting:** Two structures are declared separately and then the member structure is nested inside the parent structure.

Accessing Nested Members

We can access nested Members by using the same (.) dot operator two times as shown:
str_parent . str_child . member;

Example

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

```
// Child structure declaration
```

```
struct child {  
    int x;  
    char c;  
};
```

```
// Parent structure declaration
```

```
struct parent {  
    int a;  
    struct child b;  
};
```

```
int main() {
```

```
    struct parent p = { 25, 195, 'A' };
```

```
// Accessing and printing nested members
```

```
printf("p.a = %d\n", p.a);  
printf("p.b.x = %d\n", p.b.x);  
printf("p.b.c = %c", p.b.c);  
return 0;  
}
```

Output

```
p.a = 25
```

```
p.b.x = 195
```

```
p.b.c = A
```

Explanation: In this code, the structure **parent** contains another structure **child** as a member. The **parent** structure is then initialized with values, including the values for the child structure's members.

Structure Pointer

A pointer to a structure allows us to access structure members using the (->) arrow operator instead of the dot operator.

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

```
// Structure declaration
```

```
struct Point {  
    int x, y;  
};
```

```
int main() {  
    struct Point p = { 1, 2 };
```

```
// ptr is a pointer to structure p
```

```
struct Point* ptr = &p;
```

```
// Accessing structure members using structure pointer
```

```
printf("%d %d", ptr->x, ptr->y);
```

```
    return 0;
```

```
}
```

Output

```
1 2
```

Explanation: In this example, **ptr** is a pointer to the structure **Point**. It holds the address of the structure variable **p**. The structure members **x** and **y** are accessed using the **-> operator**, which is used to dereference the pointer and access the members of the structure.