

C PROGRAMMING

Programming Languages

The computer system is simply a machine and hence it cannot perform any work; therefore, in order to make it functional different languages are developed, which are known as programming languages or simply computer languages.

Over the last two decades, dozens of computer languages have been developed. Each of these languages comes with its own set of vocabulary and rules, better known as syntax. Furthermore, while writing the computer language, syntax has to be followed literally, as even a small mistake will result in an error and not generate the required output.

Following are the major categories of Programming Languages –

- Machine Language
- Assembly Language
- High Level Language
 - System Language
 - Scripting Language

Machine Language or Code

This is the language that is written for the computer hardware. Such language is affected directly by the central processing unit (CPU) of a computer system.

Assembly Language

It is a language of an encoding of machine code that makes simpler and readable.

High Level Language

The high-level language is simple and easy to understand and it is similar to English language. For example, COBOL, FORTRAN, BASIC, C, C++, Python, etc.

High-level languages are very important, as they help in developing complex software and they have the following advantages –

- Unlike assembly language or machine language, users do not need to learn the high-level language in order to work with it.
- High-level languages are similar to natural languages, therefore, easy to learn and understand.
- High-level language is designed in such a way that it detects the errors immediately.
- High-level language is easy to maintain and it can be easily modified.
- High-level language makes development faster.
- High-level language is comparatively cheaper to develop.
- High-level language is easier to document.

Although a high-level language has many benefits, yet it also has a drawback. It has poor control on machine/hardware.

The following table lists down the frequently used languages –

SQL
Java
Javascript
C#
Python
C++
PHP
IOS
Ruby/Rails
.Net

Hardware & Software:

The following table highlights the points that differentiate a hardware from a software.

Hardware	Software
It is the physical component of a computer system.	It is the programming language that makes hardware functional.
It has the permanent shape and structure, which cannot be modified.	It can be modified and reused, as it has no permanent shape and structure.
The external agents such as dust, mouse, insects, humidity, heat, etc. can affect the hardware (as it is tangible).	The external agents such as dust, mouse, insects, humidity, heat, etc. cannot affect (as it is not tangible).
It works with binary code (i.e., 1's to 0's) .	It functions with the help of high level language like COBOL, BASIC, JAVA, etc.
It takes in only machine language, i.e., lower level language.	It takes in higher level language, easily readable by a human being.
It is not affected by the computer bug or virus.	It is affected by the computer bug or virus.
It cannot be transferred from one place to other electronically.	It can transfer from one place to other electronically.
Duplicate copy of hardware cannot be created.	A user can create copies of a software as many as he wishes.

Software Programming

In order to make a computer functional, a set of instructions need to be programmed, as these programmed languages are carriers to the performance of a task.

Likewise, a computer accepts users' instructions in the form of computer programming and then carries out the given task.

Features of Software Programming

A computer program, which actually is a set of instructions and helps computer to perform a specific task, has the following basic features –

- It ensures the given instructions are performed successfully.

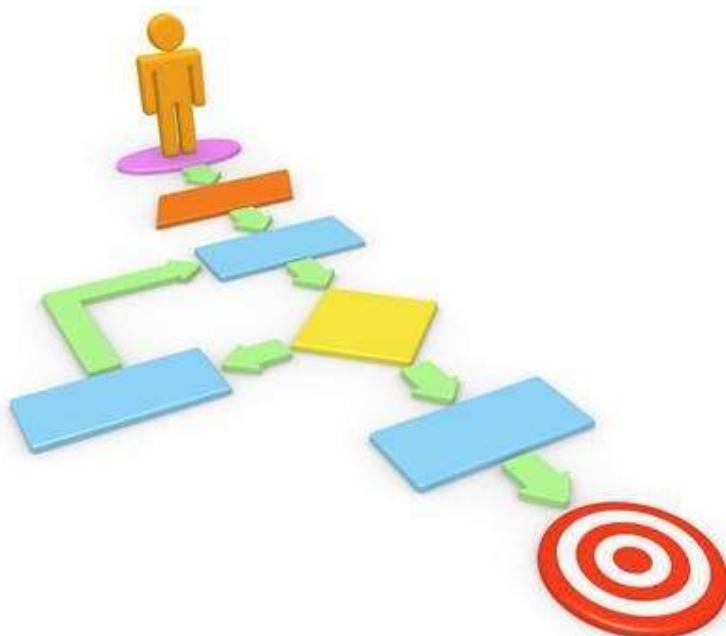
- It ensures the given instructions are performed in sequential order.
- Explains the input (data) given is correct or insufficient and accordingly gives the result.
- It is written with high level language.

Steps to Development of Program

Development of programming language is entirely dependent on the kind of problem and requirement. However, development of a programming language normally (not essentially, but) includes the following steps –

Defining the Problem

This the first step, wherein the problem has to be defined.



Analysis of Task and Methods

Once the problem is defined, the developer analyzes and develops various solutions in order to solve the problem and finally, the best feasible solution is developed.

Development of Algorithm

Algorithm is a proper technique that illustrates the right solution in logical and feasible steps. Algorithm is normally done in the form of flowcharts and pseudo codes.

Verification of Algorithm

Once the algorithm is developed, it cannot be applied directly rather primarily it needs to be tested specially for the accuracy. If there is any error, it is rectified and solved in the beginning itself. The verification process saves time, money, and energy.

Coding

Once the basic processes and steps are completed successfully, then the actual coding of a program starts in the given programming language.

Testing of Program

Testing of the development of program code is another essential feature, as it is bound with errors; hence, testing makes it error free. The developer keeps testing and correcting the coding until he/she develops it finally.

Documentation

Once the coding and programming is done successfully, it is the job of the developer to document all these features and steps. The documented program instructs users on how to run and operate the respective program.

Implementation

Once the above steps are executed successfully, the developed codes (programming language) are installed in the computer system for the end users. The users are also manuals - explaining how to run the respective programs.

TRANSLATORS: Compilers & Interpreters

What is Compiler?

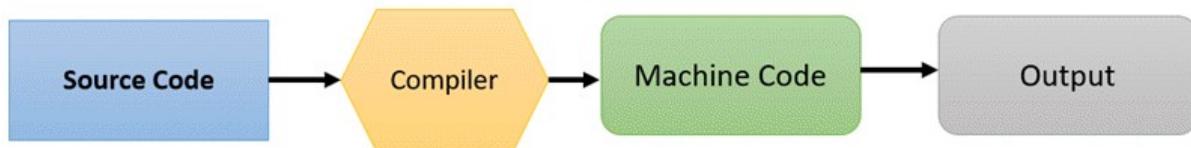
A compiler is a computer program that transforms code written in a high-level programming language into the machine code. It is a program which translates the human-readable code to a language a computer processor understands (binary 1 and 0 bits). The computer processes the machine code to perform the corresponding tasks.

A compiler should comply with the syntax rule of that programming language in which it is written. However, the compiler is only a program and can not fix errors found in that program. So, if you make a mistake, you need to make changes in the syntax of your program. Otherwise, it will not compile.

What is Interpreter?

An interpreter is a computer program, which converts each high-level program statement into the machine code. This includes source code, pre-compiled code, and scripts. Both compiler and interpreters do the same job which is converting higher level programming language to machine code. However, a compiler will convert the code into machine code (create an exe) before program run. Interpreters convert code into machine code when the program is run.

How Compiler Works



How Interpreter Works



Compiler	Interpreter
<ul style="list-style-type: none">• A compiler takes the entire program in one go.	<ul style="list-style-type: none">• An interpreter takes a single line of code at a time.
<ul style="list-style-type: none">• The compiler generates an intermediate machine code.	<ul style="list-style-type: none">• The interpreter never produces any intermediate machine code.
<ul style="list-style-type: none">• The compiler is best suited for the production environment.	<ul style="list-style-type: none">• An interpreter is best suited for a software development environment.
<ul style="list-style-type: none">• The compiler is used by programming languages such as C, C++, C#, Scala, Java, etc.	<ul style="list-style-type: none">• An interpreter is used by programming languages such as Python, PHP, Perl, Ruby, etc.

KEY DIFFERENCE

1. Compiler transforms code written in a high-level programming language into the machine code, at once, before program runs, whereas an Interpreter converts each high-level program statement, one by one, into the machine code, during program run.
2. Compiled code runs faster while interpreted code runs slower.
3. Compiler displays all errors after compilation, on the other hand, the Interpreter displays errors of each line one by one.
4. Compiler is based on translation linking-loading model, whereas Interpreter is based on Interpretation Method.
5. Compiler takes an entire program whereas the Interpreter takes a single line of code.

Problem Solving Using Computer

Computer based problem solving is a systematic process of designing, implementing and using programming tools during the problem solving stage. This method enables the computer system to be more intuitive with human logic than machine logic. Final outcome of this process is software tools which is dedicated to solve the problem under consideration. Software is just a collection of computer programs and programs are a set of instructions which guides computer's hardware. These instructions need to be well specified for solving the problem. After its creation, the software should be error free and well documented. Software development is the process of creating such software, which satisfies end user's requirements and needs.

The following six steps must be followed to solve a problem using computer.

1. Problem Analysis
2. Program Design - Algorithm, Flowchart and Pseudocode
3. Coding
4. Compilation and Execution
5. Debugging and Testing
6. Program Documentation

ALGORITHM:

An algorithm is an effective step-by-step procedure for solving a problem in a finite number of steps. In other words, it is a finite set of well-defined instructions or step-by-step description of the procedure written in human readable language for solving a given problem. An algorithm itself is division of a problem into small steps which are ordered in sequence and easily understandable. Algorithms are very important to the way computers process information, because a computer program is basically an algorithm that tells computer what specific tasks to perform in what specific order to accomplish a specific task. The same problem can be solved with different methods. So, for solving the same problem, different algorithms can be designed. In these algorithms, number of steps, time and efforts may vary more or less.

Characteristics of an Algorithm

An algorithm must possess following characteristics:

1. **Finiteness:** An algorithm should have finite number of steps and it should end after a finite time.
2. **Input:** An algorithm may have many inputs or no inputs at all.
3. **Output:** It should result at least one output.
4. **Definiteness:** Each step must be clear, well-defined and precise. There should be no any ambiguity.
5. **Effectiveness:** Each step must be simple and should take a finite amount of time.

Guidelines for Developing an Algorithm

Following guidelines must be followed while developing an algorithm:

1. An algorithm will be enclosed by START (or BEGIN) and STOP (or END).
2. To accept data from user, generally used statements are INPUT, READ, GET or OBTAIN.
3. To display result or any message, generally used statements are PRINT, DISPLAY, or WRITE.
4. Generally, COMPUTE or CALCULATE is used while describing mathematical expressions and based on situation relevant operators can be used.

Example of an Algorithm

Algorithm: Calculation of Simple Interest

```
Step 1: Start  
Step 2: Read principle (P), time (T) and rate (R)  
Step 3: Calculate I = P*T*R/100  
Step 4: Print I as Interest  
Step 5: Stop
```

Advantages of an Algorithm

Designing an algorithm has following advantages:

1. **Effective Communication:** Since algorithm is written in English like language, it is simple to understand step-by-step solution of the problems.
2. **Easy Debugging:** Well-designed algorithm makes debugging easy so that we can identify logical error in the program.
3. **Easy and Efficient Coding:** An algorithm acts as a blueprint of a program and helps during program development.
4. **Independent of Programming Language:** An algorithm is independent of programming languages and can be easily coded using any high level language.

Disadvantages of an Algorithm

An algorithm has following disadvantages :

1. Developing algorithm for complex problems would be time consuming and difficult to understand.
2. Understanding complex logic through algorithms can be very difficult.

Examples:

1. Write an Algorithm to compute the sum and average of two numbers

Algorithm: To compute Sum and Average of Two Numbers:

```
Step 1: [Start the algorithm]  
Start
```

Step 2: [Read any Two Numbers]

 Read a, b

Step 3: [Add the Numbers to Compute Sum]

 Sum \leftarrow (a + b)

Step 4: [Compute the Average of Two numbers]

 Average \leftarrow Sum/2

Step 5: [Print the Computed results: Sum & Average]

 Print Sum, Average

Step 6: [Finished - end of the Algorithm]

 Stop

2) Write an algorithm to compute the sum and average of three numbers.

Algorithm: To compute Sum and Average of Three Numbers:

Step 1: [Read any three numbers]

 Read a, b, c

Step 2: [Add the numbers to compute Sum]

 Sum \leftarrow (a + b + c)

Step 3: [Compute the Average of three numbers]

 Average \leftarrow Sum/3

Step 4: [Print the computed results: Sum & Average]

 Write Sum, Average

Step 5: [Finished - end of the Algorithm]

 Exit

3) Write an algorithm to find the area and perimeter of a circle.

Formula: - Area = $3.142 * \text{radius} * \text{radius}$

 Perimeter = $2 * 3.142 * \text{radius}$

Algorithm: To find the Area and Perimeter of a Circle:

Step 1: [Read the radius]

 Read radius

Step 2: [Compute the area of circle]

 Area \leftarrow ($3.142 * \text{radius} * \text{radius}$)

Step 3: [Compute the perimeter of a circle]

Perimeter $\leftarrow (2 * 3.142 * \text{radius})$

Step 4: [Print the result: area and perimeter]

Write area, perimeter

Step 5: [Finished- End of the Algorithm]

Exit

4) Write an algorithm to add five numbers?

Algorithm: To compute five Numbers

Step 1: [Read any Five Numbers]

Read a, b, c, d, e

Step 2: [Compute the Sum]

sum $\leftarrow (a+b+c+d+e)$

Step 3: [Print the Result]

Write sum

Step 4: [Finished- End of Algorithm]

Exit

5) Write an algorithm to compute the sum and average of five numbers?

Algorithm: To compute the Sum and Average of Five Numbers:

Step 1: [Read any Five Numbers]

Read a, b, c, d, e

Step 2: [Compute the Sum]

sum $\leftarrow (a+b+c+d+e)$

Step 3: [Compute Average]

average $\leftarrow (\text{sum} / 5)$

Step 4: [Print the Result]

Write sum, average

Step 5: [Finished- End of Algorithm]

Exit

FLOW CHART

A flowchart is a blueprint that pictorially represents the algorithm and its steps. The steps of a flowchart do not have a specific size and shape rather it is designed in different shapes and sizes (see the image given below).



As shown in the above image, the boxes in different shapes and interconnected with arrows, are logically making a flow chart. A flow-chart represents the general steps in a process.

Benefits of Flowchart

Let us now discuss the benefits of a flowchart.

1. Simplify the Logic

As it provides the pictorial representation of the steps; therefore, it simplifies the logic and subsequent steps.

2. Makes Communication Better

Because of having easily understandable pictorial logic and steps, it is a better and simple way of representation.

3. Effective Analysis

Once the flow-chart is prepared, it becomes very simple to analyse the problem in an effective way.

4. Useful in Coding

The flow-chart also helps in coding process efficiently, as it gives directions on what to do, when to do, and where to do. It makes the work easier.

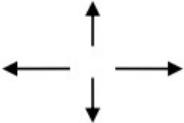
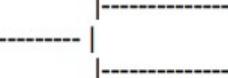
5. Proper Testing

Further, flowchart also helps in finding the error (if any) in program

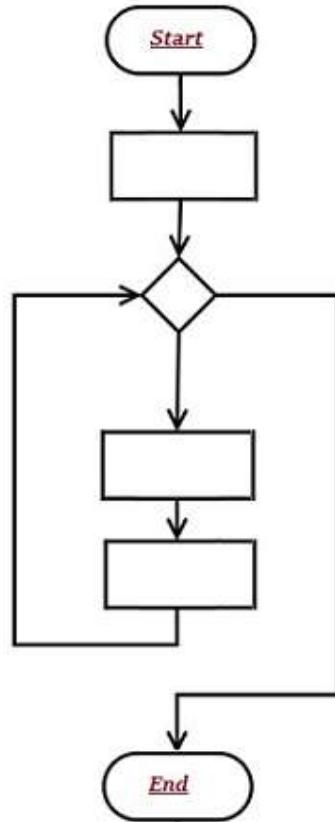
6. Applicable Documentation

Last but not the least, a flowchart also helps in preparing the proper document (once the codes are written).

Flow-Chart Symbols

Symbol Name	Symbol	function
Oval		Used to represent start and end of flowchart
Parallelogram		Used for input and output operation
Rectangle		Processing: Used for arithmetic operations and data-manipulations
Diamond		Decision making. Used to represent the operation in which there are two/three alternatives, true and false etc
Arrows		Flow line Used to indicate the flow of logic by connecting symbols
Circle		Page Connector
		Off Page Connector
		Predefined Process /Function Used to represent a group of statements performing one processing task.
		Preprocessor
		Comments

Sample of Flow Chart



Example of Flowchart

Flowchart example for calculating simple interest is shown below:

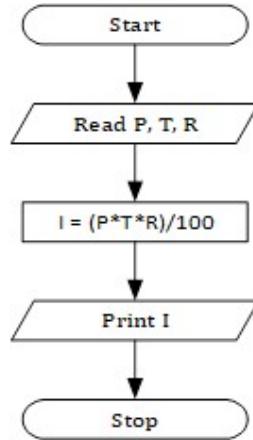


Figure: Flowchart for calculating simple interest

Advantages of Flowchart

Drawing flowchart while solving any problem has following **advantages**:

1. **Effective Communication:** Flowcharts are better way of communicating the logic of the system.
 2. **Effective Analysis:** Using flowchart problem can be analysed more efficiently
1. **Easy Debugging and Efficient Testing:** The Flowchart helps in debugging and testing process.
 2. **Efficient Coding:** The flowcharts are very useful during program development phase.
 3. **Proper Documentation:** Flowcharts serves as a good program documentation, which is needed for various purpose.
 4. **Efficient Program Maintenance:** Maintenance of operating programs becomes easy with the help of flowchart.

Disadvantages of Flowchart

Flowchart has following disadvantages:

1. **Complex Logic:** For complicated logic, flowchart becomes complex and clumsy.
2. **Difficulty in Modifications:** If change is required in the logic, then flowchart needs to be redrawn and requires a lot of time.

C Programming

The C Language is developed by Dennis Ritchie for creating system applications that directly interact with the hardware devices such as drivers, kernels, etc.

C programming is considered as the base for other programming languages, that is why it is known as mother language.

It can be defined by the following ways:

1. Mother language
2. System programming language
3. Procedure-oriented programming language
4. Structured programming language
5. Mid-level programming language

1) C as a mother language

C language is considered as the mother language of all the modern programming languages because **most of the compilers, JVMs, Kernels, etc. are written in C language**, and most of the programming languages follow C syntax, for example, C++, Java, C#, etc.

It provides the core concepts like the array, strings, functions, file handling, etc. that are being used in many languages like C++, Java, C#, etc.

2) C as a system programming language

A system programming language is used to create system software. C language is a system programming language because it **can be used to do low-level programming (for example driver and kernel)**. It is generally used to create hardware devices, OS, drivers, kernels, etc. For example, Linux kernel is written in C.

It can't be used for internet programming like Java, .Net, PHP, etc.

3) C as a procedural language

A procedure is known as a function, method, routine, subroutine, etc. A procedural language **specifies a series of steps for the program to solve the problem.**

A procedural language breaks the program into functions, data structures, etc.

C is a procedural language. In C, variables and function prototypes must be declared before being used.

4) C as a structured programming language

A structured programming language is a subset of the procedural language. **Structure means to break a program into parts or blocks** so that it may be easy to understand.

In the C language, we break the program into parts using functions. It makes the program easier to understand and modify.

5) C as a mid-level programming language

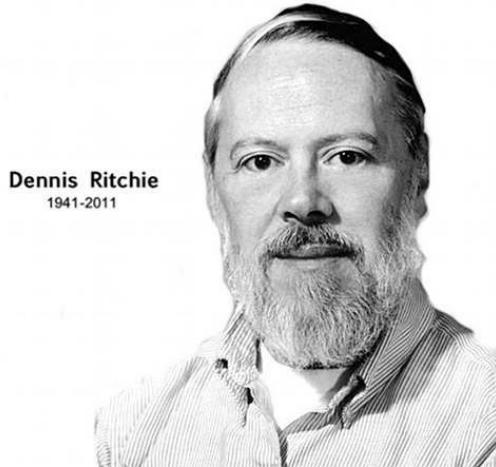
C is considered as a middle-level language because it **supports the feature of both low-level and high-level languages**. C language program is converted into assembly code, it supports pointer arithmetic (low-level), but it is machine independent (a feature of high-level).

A **Low-level language** is specific to one machine, i.e., machine dependent. It is machine dependent, fast to run. But it is not easy to understand.

A **High-Level language** is not specific to one machine, i.e., machine independent. It is easy to understand.

History of C Language:

C programming language was developed in 1972 by Dennis Ritchie at bell laboratories of AT&T (American Telephone & Telegraph), located in the U.S.A.



Dennis Ritchie is known as the **founder of the c language**.

It was developed to overcome the problems of previous languages such as B, BCPL, etc.

Here is the list of all the languages that were already developed before the invention of the C language along with details of the invention year and the name of the developer:

Language	Year	Developed By
Algol	1960	International Group
BCPL	1967	Martin Richard
B	1970	Ken Thompson
Traditional C	1972	Dennis Ritchie
K & R C	1978	Dennis Ritchie & Kernighan
ANSI C	1989	ANSI Committee
ANSI/ISO C	1990	ISO Committee
C99	1999	Standardization Committee

Features of the C Language

It is a well-known fact that C is the most widely used programming language, here are some the features of C language that separates it from the rest.

1. Simple
2. Mid-level programming language
3. Machine Independent or Portable
4. Rich Library
5. Structured programming language
6. Memory Management
7. Recursion

1. Simple

The C language is known as a simple language in the context that it delivers a structured approach (in order to break the problem into parts), the wide set of library functions, data types, etc.

2. Mid-level programming language

C is known to do the low-level programming but it is also used to develop the system applications like kernel, driver, etc. C is also known to support the features of a high-level language. Including all these qualities, C language is as a mid-level language.

3. Machine Independent or Portable

Just opposite of the assembly language, C programs are not limited to only machines, It can be executed on different machines with some machine specific changes. Hence, C language is also called as a machine independent language.

4. A Wide library

There are a lot of inbuilt functions in the C language that are known to make the development amazingly fast.

5. Structured Programming Language

C language is a structured programming language in the context that the user can break the program into parts using the functions. Hence, C language is easy

in understanding and modification. Functions in the C language also provide the user the reusability of code.

6. Memory Management

The feature of dynamic memory allocation is supported by the C language. You can free the allocated memory any time just by calling the `free()` function in the C language.

7. Recursion

In C language, a function can be called within the function and it also provides the code reusability for every function. Recursion generally enables the user to use the approach of backtracking.

8. Lightning-Fast Speed

The compilation and execution time of C language is amazingly fast as compared to the other languages due to the lesser inbuilt functions that leads to the lesser overhead.

9. Extensible

C language is known to be extensible because of its easily adopting of the new features.

10. Pointers

The C language has the feature of pointers that can directly interact with the memory by the use of the pointers. These functions can be used like pointers for memory, structures, functions, array, etc.

C Program Structure:

Before we study the basic building blocks of the C programming language, let us look at a bare minimum C program structure so that we can take it as a reference in the upcoming chapters.

Hello World Example

A C program basically consists of the following parts –

- Pre-processor Commands

- Functions
- Variables
- Statements & Expressions
- Comments

Let us look at a simple code that would print the words "Hello World" –

```
#include <stdio.h>

int main() {
    /* my first program in C */
    printf("Hello, World! \n");
    return 0;
}
```

Let us take a look at the various parts of the above program –

- The first line of the program `#include <stdio.h>` is a preprocessor command, which tells a C compiler to include stdio.h file before going to actual compilation.
- The next line `int main()` is the main function where the program execution begins.
- The next line `/*...*/` will be ignored by the compiler and it has been put to add additional comments in the program. So such lines are called comments in the program.
- The next line `printf(...)` is another function available in C which causes the message "Hello, World!" to be displayed on the screen.
- The next line **return 0;** terminates the main() function and returns the value 0.

Compile and Execute C Program

Let us see how to save the source code in a file, and how to compile and run it. Following are the simple steps –

- Open a text editor and add the above-mentioned code.
- Save the file as `hello.c`
- Open a command prompt and go to the directory where you have saved the file.
- Type `gcc hello.c` and press enter to compile your code.

- If there are no errors in your code, the command prompt will take you to the next line and would generate *a.out* executable file.
- Now, type *a.out* to execute your program.
- You will see the output "Hello World" printed on the screen.

```
$ gcc hello.c
```

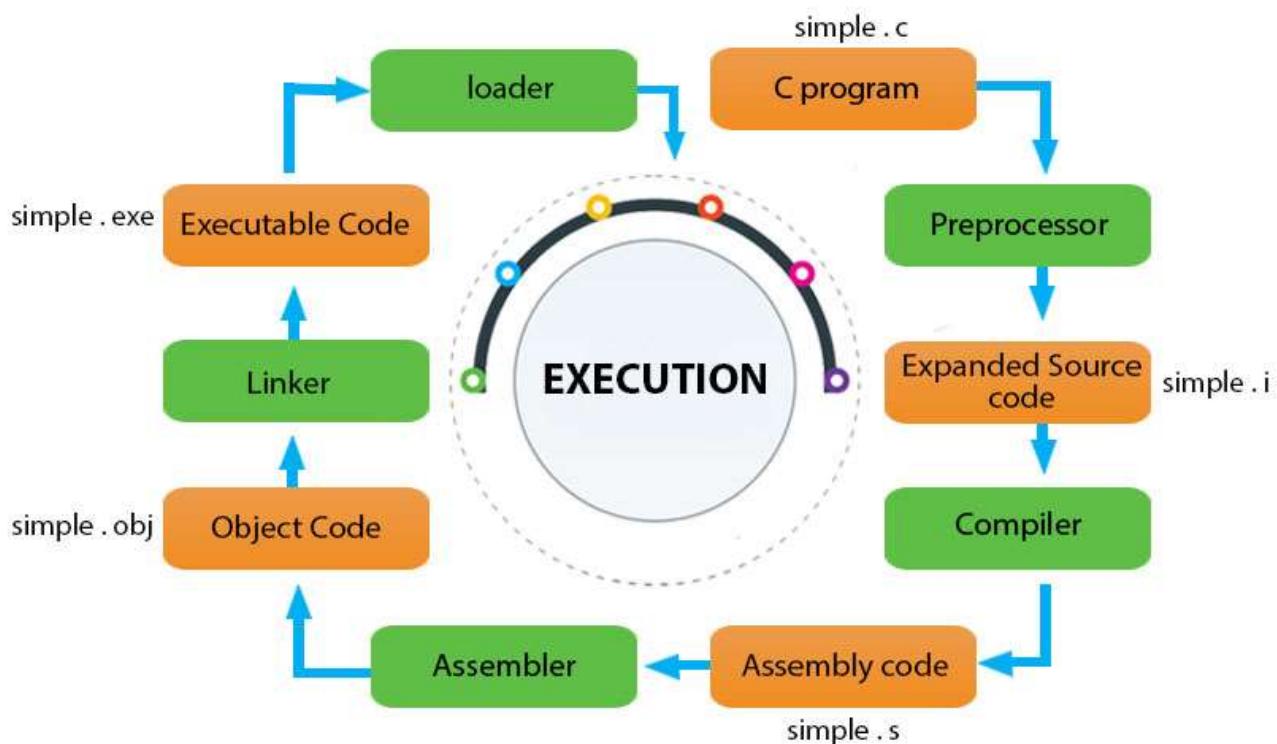
```
$ ./a.out
```

Hello, World!

Make sure the gcc compiler is in your path and that you are running it in the directory containing the source file *hello.c*.

The Execution Flow

Here is the flow chart depicting the flow of the above-mentioned program:



1. The C program (source code) firstly, is sent to the preprocessor. To convert the preprocessor directives into their respective values, the preprocessor is responsible. An expanded source code is generated by the preprocessor.
2. Then the expanded source code is sent to the compiler that compiles the code and converts into the assembly code.

3. The assembly code is then sent to the assembler that assembles the code and converts into the object code. Then, a simple.obj file is generated.
4. The object code is then sent to the linker that links it to the library like header files. In the next step, it is converted into an executable code. Then, a simple.exe file is generated.
5. The executable code is then sent to the loader that loads the code into the memory followed by the execution of the code. The output is then sent to the console, after execution

Functions as building block in C Programming

C Programming is known as **procedure-oriented programming language**.

It contains different procedures or functions which accomplish some specified tasks.

Any C program must contain a function named **main ()**, which actually acts as an entry point or it is the first program to be executed.

Functions can share and exchange some data.

Functions should be made to achieve one specific task.

One function should not be assigned to many functionalities. This is considered as bad programming practice.

The concept of giving only one task to a function is known as cohesion.

Functions can share data by passing arguments and returning values.

Passing data between functions is known as coupling i.e., concept of joining two function.

Function is of two types:

1. Library functions
2. User defined functions

Library function are the ready-made functions provide by C compiler.

To use them we need to include the specific library file in which that function has been defined.

User defined functions are created by programmer for some specific requirement.

A user defined function needs to be declared, used and defined.

Program development cycle in C language

When we want to develop a program by using any programming language, we have to follow a sequence of steps. These steps are called phases in program development.

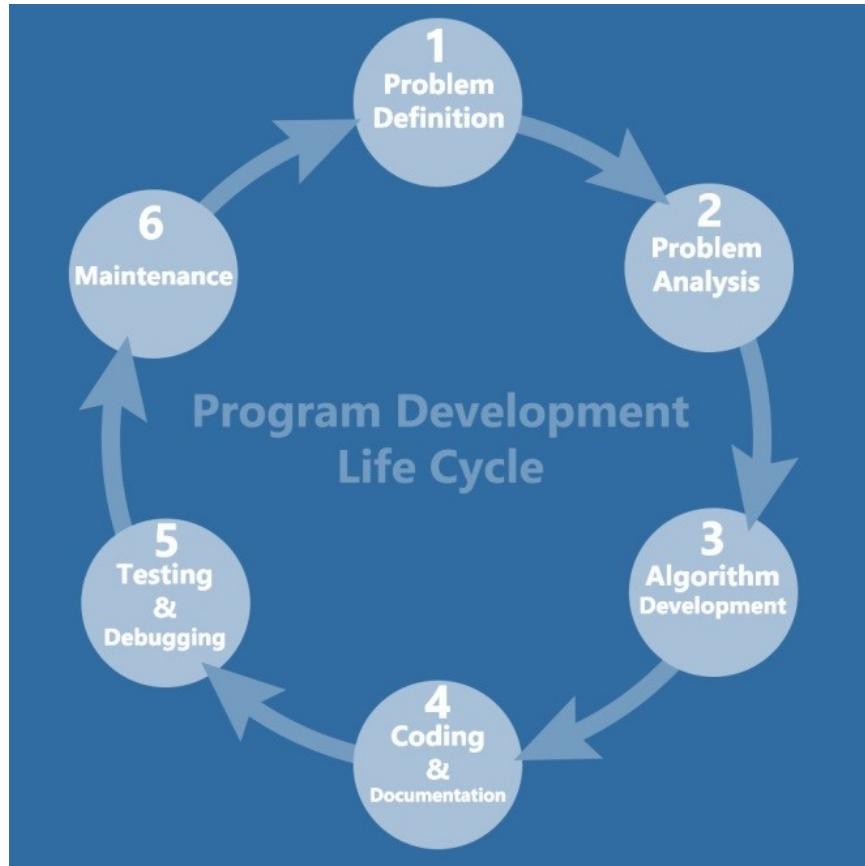
The program development life cycle is a set of steps or phases which are used to develop a program in any programming language.

Phases of program development

Program development life cycle contains 6 phases, which are as follows –

1. Problem Definition.
2. Problem Analysis.
3. Algorithm Development.
4. Coding & Documentation.
5. Testing & Debugging.
6. Maintenance.

These six phases are depicted in the diagram given below –



1. Problem Definition

Here, we define the problem statement and decide the boundaries of the problem.

In this phase, we need to understand what is the problem statement, what is our requirement and what is the output of the problem solution. All these are included in the first phase of program development life cycle.

2. Problem Analysis

Here, we determine the requirements like variables, functions, etc. to solve the problem. It means that we gather the required resources to solve the problem, which are defined in the problem definition phase. Here, we also determine the bounds of the solution.

3. Algorithm Development

Here, we develop a step-by-step procedure that is used to solve the problem by using the specification given in the previous phase. It is very important phase for the program development. We write the solution in step-by-step statements.

4. Coding & Documentation

Here, we use a programming language to write or implement the actual programming instructions for the steps defined in the previous phase. We construct the actual program in this phase. We write the program to solve the given problem by using the programming languages like C, C++, Java, etc.

5. Testing & Debugging

In this phase, we check whether the written code in the previous step is solving the specified problem or not. This means, we try to test the program whether it is solving the problem for various input data values or not. We also test if it is providing the desired output or not.

6. Maintenance

In this phase, we make the enhancements. Therefore, the solution is used by the end-user. If the user gets any problem or wants any enhancement, then we need to repeat all these phases from the starting, so that the encountered problem is solved or enhancement is added.

Tokens in C

A C program consists of various tokens and a token is either a keyword, an identifier, a constant, a string literal, or a symbol. For example, the following C statement consists of five tokens –

```
printf("Hello, World! \n");
```

The individual tokens are –

```
printf  
(  
    "Hello, World! \n"  
)  
;
```

Semicolons

In a C program, the semicolon is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity.

Given below are two different statements –

```
printf("Hello, World! \n");  
return 0;
```

Comments

Comments are like helping text in your C program and they are ignored by the compiler. They start with /* and terminate with the characters */ as shown below –

```
/* my first program in C */
```

You cannot have comments within comments and they do not occur within a string or character literals.

Identifiers

A C identifier is a name used to identify a variable, function, or any other user-defined item. An identifier starts with a letter A to Z, a to z, or an underscore '_' followed by zero or more letters, underscores, and digits (0 to 9).

C does not allow punctuation characters such as @, \$, and % within identifiers. C is a **case-sensitive** programming language. Thus, *Manpower* and *manpower* are two different identifiers in C. Here are some examples of acceptable identifiers –

```
mohd    zara   abc  move_name a_123  
myname50 _temp  j   a23b9    retVal
```

Keywords

The following list shows the reserved words in C. These reserved words may not be used as constants or variables or any other identifier names.

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

Whitespace in C

A line containing only whitespace, possibly with a comment, is known as a blank line, and a C compiler totally ignores it.

Whitespace is the term used in C to describe blanks, tabs, newline characters and comments. Whitespace separates one part of a statement from another and enables the compiler to identify where one element in a statement, such as int, ends and the next element begins. Therefore, in the following statement –

```
int age;
```

there must be at least one whitespace character (usually a space) between int and age for the compiler to be able to distinguish them. On the other hand, in the following statement –

```
fruit = apples + oranges; // get the total fruit
```

no whitespace characters are necessary between fruit and =, or between = and apples, although you are free to include some if you wish to increase readability.

C - Data Types

Data types in c refer to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

ANSI C provides three types of data types:

1. Primary (Built-in) Data Types:

void, int, char, double and float.

2. Derived Data Types:

Array, References, and Pointers.

3. User Defined Data Types:

Structure, Union, and Enumeration.

The array types and structure types are referred collectively as the aggregate types. The type of a function specifies the type of the function's return value. We will see the basic types in the following section, whereas other types will be covered in the upcoming chapters.

Integer Types

The following table provides the details of standard integer types with their storage sizes and value ranges –

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255

signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	8 bytes or (4bytes for 32 bit OS)	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615

To get the exact size of a type or a variable on a particular platform, you can use the **sizeof** operator. The expression `sizeof(type)` yields the storage size of the object or type in bytes. Given below is an example to get the size of various type on a machine using different constant defined in limits.h header file –

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

int main(int argc, char** argv) {

    printf("CHAR_BIT : %d\n", CHAR_BIT);
    printf("CHAR_MAX : %d\n", CHAR_MAX);
    printf("CHAR_MIN : %d\n", CHAR_MIN);
    printf("INT_MAX : %d\n", INT_MAX);
    printf("INT_MIN : %d\n", INT_MIN);
    printf("LONG_MAX : %ld\n", (long) LONG_MAX);
    printf("LONG_MIN : %ld\n", (long) LONG_MIN);
    printf("SCHAR_MAX : %d\n", SCHAR_MAX);
    printf("SCHAR_MIN : %d\n", SCHAR_MIN);
    printf("SHRT_MAX : %d\n", SHRT_MAX);
    printf("SHRT_MIN : %d\n", SHRT_MIN);
```

```

printf("UCHAR_MAX : %d\n", UCHAR_MAX);
printf("UINT_MAX : %u\n", (unsigned int) UINT_MAX);
printf("ULONG_MAX : %lu\n", (unsigned long) ULONG_MAX);
printf("USHRT_MAX : %d\n", (unsigned short) USHRT_MAX);

return 0;
}

```

When you compile and execute the above program, it produces the following result on Linux –

```

CHAR_BIT : 8
CHAR_MAX : 127
CHAR_MIN : -128
INT_MAX : 2147483647
INT_MIN : -2147483648
LONG_MAX : 9223372036854775807
LONG_MIN : -9223372036854775808
SCHAR_MAX : 127
SCHAR_MIN : -128
SHRT_MAX : 32767
SHRT_MIN : -32768
UCHAR_MAX : 255
UINT_MAX : 4294967295
ULONG_MAX : 18446744073709551615
USHRT_MAX : 65535

```

Floating-Point Types

The following table provide the details of standard floating-point types with storage sizes and value ranges and their precision –

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places

long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places
-------------	---------	------------------------	-------------------

The header file float.h defines macros that allow you to use these values and other details about the binary representation of real numbers in your programs. The following example prints the storage space taken by a float type and its range values –

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

int main(int argc, char** argv) {

    printf("Storage size for float : %d \n", sizeof(float));
    printf("FLT_MAX      :  %g\n", (float) FLT_MAX);
    printf("FLT_MIN      :  %g\n", (float) FLT_MIN);
    printf("-FLT_MAX     :  %g\n", (float) -FLT_MAX);
    printf("-FLT_MIN     :  %g\n", (float) -FLT_MIN);
    printf("DBL_MAX      :  %g\n", (double) DBL_MAX);
    printf("DBL_MIN      :  %g\n", (double) DBL_MIN);
    printf("-DBL_MAX     :  %g\n", (double) -DBL_MAX);
    printf("Precision value: %d\n", FLT_DIG );

    return 0;
}
```

When you compile and execute the above program, it produces the following result on Linux –

```
Storage size for float : 4
FLT_MAX      :  3.40282e+38
FLT_MIN      :  1.17549e-38
-FLT_MAX     :  -3.40282e+38
-FLT_MIN     :  -1.17549e-38
DBL_MAX      :  1.79769e+308
```

```
DBL_MIN      : 2.22507e-308
-DBL_MAX     : -1.79769e+308
```

Precision value: 6

The void Type

The void type specifies that no value is available. It is used in three kinds of situations –

Sr.No.	Types & Description
1	Function returns as void There are various functions in C which do not return any value or you can say they return void. A function with no return value has the return type as void. For example, void exit (int status);
2	Function arguments as void There are various functions in C which do not accept any parameter. A function with no parameter can accept a void. For example, int rand(void);
3	Pointers to void A pointer of type void * represents the address of an object, but not its type. For example, a memory allocation function void *malloc(size_t size); returns a pointer to void which can be casted to any data type.

```
#include <stdio.h>
int main()
{
    int a = 4000; // positive integer data type
    float b = 5.2324; // float data type
    char c = 'Z'; // char data type
    long d = 41657; // long positive integer data type
    long e = -21556; // long -ve integer data type
    int f = -185; // -ve integer data type
    short g = 130; // short +ve integer data type
    short h = -130; // short -ve integer data type
    double i = 4.1234567890; // double float data type
    float j = -3.55; // float data type
}
```

The storage representation and machine instructions differ from machine to machine. **sizeof** operator can be used to get the exact size of a type or a variable on a particular platform.

Example:

```
#include <stdio.h>
#include <limits.h>
int main()
{
    printf("Storage size for int is: %d \n", sizeof(int));
    printf("Storage size for char is: %d \n", sizeof(char));
    return 0;
}
```

Derived Data Types:

C supports three derived data types:

Data Types	Description
Arrays	Arrays are sequences of data items having homogeneous values. They have adjacent memory locations to store values.
References	Function pointers allow referencing functions with a particular signature.
Pointers	These are powerful C features which are used to access the memory and deal with their addresses.

User Defined Data Types:

C allows the feature called **type definition** which allows programmers to define their identifier that would represent an existing data type. There are three such types:

Data Types	Description
Structure	It is a package of variables of different types under a single name. This is done to handle data efficiently. "struct" keyword is used to define a structure.
Union	These allow storing various data types in the same memory location. Programmers can define a union with different members, but only a single member can contain a value at a given time.
Enum	Enumeration is a special data type that consists of integral constants, and each of them is assigned with a specific name. "enum" keyword is used to define the enumerated data type.

C - Variables

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C is case-sensitive. Based on the basic types explained in the previous chapter, there will be the following basic variable types –

Sr.No.	Type & Description
1	char Typically, a single octet (one byte). It is an integer type.
2	int The most natural size of integer for the machine.
3	float A single-precision floating point value.
4	double A double-precision floating point value.
5	void Represents the absence of type.

C programming language also allows to define various other types of variables, which we will cover in subsequent chapters like Enumeration, Pointer, Array, Structure, Union, etc. For this chapter, let us study only basic variable types.

Variable Definition in C

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type and contains a list of one or more variables of that type as follows –

```
type variable_list;
```

Here, **type** must be a valid C data type including char, w_char, int, float, double, bool, or any user-defined object; and **variable_list** may consist of one or more identifier names separated by commas. Some valid declarations are shown here –

```
int i, j, k;  
char c, ch;  
float f, salary;  
double d;
```

The line **int i, j, k;** declares and defines the variables i, j, and k; which instruct the compiler to create variables named i, j and k of type int.

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows –

```
type variable_name = value;
```

Some examples are –

```
extern int d = 3, f = 5; // declaration of d and f.  
int d = 3, f = 5; // definition and initializing d and f.  
byte z = 22; // definition and initializes z.  
char x = 'x'; // the variable x has the value 'x'.
```

For definition without an initializer: variables with static storage duration are implicitly initialized with NULL (all bytes have the value 0); the initial value of all other variables are undefined.

Variable Declaration in C

A variable declaration provides assurance to the compiler that there exists a variable with the given type and name so that the compiler can proceed for further compilation without requiring the complete detail about the variable. A variable definition has its meaning at the time of compilation only, the compiler needs actual variable definition at the time of linking the program.

A variable declaration is useful when you are using multiple files and you define your variable in one of the files which will be available at the time of linking of the program. You will use the keyword **extern** to declare a variable at any place. Though you can declare a variable multiple times in your C program, it can be defined only once in a file, a function, or a block of code.

Example

Try the following example, where variables have been declared at the top, but they have been defined and initialized inside the main function –

```
#include <stdio.h>

// Variable declaration:
extern int a, b;
extern int c;
extern float f;

int main () {

    /* variable definition: */
    int a, b;
    int c;
    float f;

    /* actual initialization */
    a = 10;
    b = 20;
```

```

c = a + b;
printf("value of c : %d \n", c);

f = 70.0/3.0;
printf("value of f : %f \n", f);

return 0;
}

```

When the above code is compiled and executed, it produces the following result –

```

value of c : 30
value of f : 23.333334

```

The same concept applies on function declaration where you provide a function name at the time of its declaration and its actual definition can be given anywhere else. For example –

```

// function declaration
int func();

int main() {

    // function call
    int i = func();
}

// function definition
int func() {
    return 0;
}

```

C - Constants and Literals

Constants refer to fixed values that the program may not alter during its execution. These fixed values are also called **literals**.

Constants can be of any of the basic data types like *an integer constant*, *a floating constant*, *a character constant*, or *a string literal*. There are enumeration constants as well.

Constants are treated just like regular variables except that their values cannot be modified after their definition.

Integer Literals

An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.

An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

Here are some examples of integer literals –

```
212      /* Legal */
215u     /* Legal */
0xFeeL   /* Legal */
078      /* Illegal: 8 is not an octal digit */
032UU    /* Illegal: cannot repeat a suffix */
```

Following are other examples of various types of integer literals –

```
85       /* decimal */
0213     /* octal */
0x4b     /* hexadecimal */
30       /* int */
30u      /* unsigned int */
30l      /* long */
30ul     /* unsigned long */
```

Floating-point Literals

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

While representing decimal form, you must include the decimal point, the exponent, or both; and while representing exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

Here are some examples of floating-point literals –

```
3.14159 /* Legal */  
314159E-5L /* Legal */  
510E /* Illegal: incomplete exponent */  
210f /* Illegal: no decimal or exponent */  
.e55 /* Illegal: missing integer or fraction */
```

Character Constants

Character literals are enclosed in single quotes, e.g., 'x' can be stored in a simple variable of **char** type.

A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t'), or a universal character (e.g., '\u02C0').

There are certain characters in C that represent special meaning when preceded by a backslash for example, newline (\n) or tab (\t).

List of Escape Sequence Codes:

Escape sequence	Meaning
\\"	\ character
'	' character
\"	" character
\?	? character
\a	Alert or bell
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab

Following is the example to show a few escape sequence characters –

```
#include <stdio.h>

int main() {
    printf("Hello\tWorld\n\n");

    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

Hello World

String Literals

String literals or constants are enclosed in double quotes "". A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.

You can break a long line into multiple lines using string literals and separating them using white spaces.

Here are some examples of string literals. All the three forms are identical strings.

"hello, dear"

"hello, \
dear"

"hello, " "d" "ear"

Defining Constants

There are two simple ways in C to define constants –

- Using **#define** preprocessor.
- Using **const** keyword.

The **#define** Preprocessor

Given below is the form to use #define preprocessor to define a constant –

```
#define identifier value
```

The following example explains it in detail –

```
#include <stdio.h>

#define LENGTH 10
#define WIDTH 5
#define NEWLINE '\n'

int main() {
    int area;

    area = LENGTH * WIDTH;
    printf("value of area : %d", area);
    printf("%c", NEWLINE);

    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
value of area : 50
```

The const Keyword

You can use **const** prefix to declare constants with a specific type as follows –

```
const type variable = value;
```

The following example explains it in detail –

```
#include <stdio.h>

int main() {
    const int LENGTH = 10;
    const int WIDTH = 5;
    const char NEWLINE = '\n';
```

```
int area;

area = LENGTH * WIDTH;
printf("value of area : %d", area);
printf("%c", NEWLINE);

return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
value of area : 50
```

C - Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C language is rich in built-in operators and provides the following types of operators –

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Misc Operators

Arithmetic Operators

The following table shows all the arithmetic operators supported by the C language. Assume variable **A** holds 10 and variable **B** holds 20 then –

Show Examples

Operator	Description	Example
+	Adds two operands.	$A + B = 30$
-	Subtracts second operand from the first.	$A - B = -10$
*	Multiplies both operands.	$A * B = 200$
/	Divides numerator by de-numerator.	$B / A = 2$
%	Modulus Operator and remainder of after an integer division.	$B \% A = 0$
++	Increment operator increases the integer value by one.	$A++ = 11$
--	Decrement operator decreases the integer value by one.	$A-- = 9$

Example 1: Arithmetic Operators

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

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

    return 0;
}
```

```
}
```

Output

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

Example 2: Increment and Decrement Operators

```
// Working of increment and decrement operators  
#include <stdio.h>  
int main()  
{  
    int a = 10, b = 100;  
    float c = 10.5, d = 100.5;  
  
    printf("++a = %d \n", ++a);  
    printf("--b = %d \n", --b);  
    printf("++c = %f \n", ++c);  
    printf("--d = %f \n", --d);  
  
    return 0;  
}
```

Output

```
++a = 11  
--b = 99  
++c = 11.500000  
--d = 99.500000
```

Example:

```
#include <stdio.h>
//stdio.h is a header file used for input.output purpose.
void main()
{
    //set a and b both equal to 5.
    int a=5, b=5;

    //Print them and decrementing each time.
    //Use postfix mode for a and prefix mode for b.
    printf("\n%d %d",a--,--b);
    printf("\n%d %d",a--,--b);
    printf("\n%d %d",a--,--b);
    printf("\n%d %d",a--,--b);
    printf("\n%d %d",a--,--b);

}
```

Program Output:

```
5 4
4 3
3 2
2 1
1 0
```

Relational Operators

The following table shows all the relational operators supported by C. Assume variable **A** holds 10 and variable **B** holds 20 then –

Show Examples

Operator	Description	Example
<code>==</code>	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	$(A == B)$ is not true.
<code>!=</code>	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	$(A != B)$ is true.
<code>></code>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	$(A > B)$ is not true.
<code><</code>	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	$(A < B)$ is true.
<code>>=</code>	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	$(A >= B)$ is not true.
<code><=</code>	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	$(A <= B)$ is true.

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

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

```

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

return 0;
}

```

Output

```

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

```

Logical Operators

Following table shows all the logical operators supported by C language. Assume variable **A** holds 1 and variable **B** holds 0, then –

Show Examples

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.

	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A B) is true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

```
// Working of logical operators

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

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

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

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

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

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

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

    return 0;
}
```

```
}
```

Output

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

Explanation of logical operator program

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

Bitwise Operators

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for $\&$, $|$, and $^$ is as follows –

p	q	p & q	p q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume A = 60 and B = 13 in binary format, they will be as follows –

$$A = 0011\ 1100$$

$$B = 0000\ 1101$$

$$A \& B = 0000\ 1100$$

$$A | B = 0011\ 1101$$

$$A ^ B = 0011\ 0001$$

$$\sim A = 1100\ 0011$$

The following table lists the bitwise operators supported by C. Assume variable 'A' holds 60 and variable 'B' holds 13, then –

Show Examples

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
	Binary OR Operator copies a bit if it exists in either operand.	(A B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001

\sim	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.	$(\sim A) = \sim(60)$, i.e., -0111101
$<<$	Binary Left Shift Operator. The left operand's value is moved left by the number of bits specified by the right operand.	$A << 2 = 240$ i.e., 1111 0000
$>>$	Binary Right Shift Operator. The left operand's value is moved right by the number of bits specified by the right operand.	$A >> 2 = 15$ i.e., 0000 1111

Assignment Operators

The following table lists the assignment operators supported by the C language –

Show Examples

Operator	Description	Example
$=$	Simple assignment operator. Assigns values from right side operands to left side operand	$C = A + B$ will assign the value of $A + B$ to C
$+=$	Add AND assignment operator. It adds the right operand to the left operand and assigns the result to the left operand.	$C += A$ is equivalent to $C = C + A$
$-=$	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	$C -= A$ is equivalent to $C = C - A$
$*=$	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	$C *= A$ is equivalent to $C = C * A$

$/=$	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	$C /= A$ is equivalent to $C = C / A$
$\%=$	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	$C \%= A$ is equivalent to $C = C \% A$
$<<=$	Left shift AND assignment operator.	$C <<= 2$ is same as $C = C << 2$
$>>=$	Right shift AND assignment operator.	$C >>= 2$ is same as $C = C >> 2$
$\&=$	Bitwise AND assignment operator.	$C \&= 2$ is same as $C = C \& 2$
$\^=$	Bitwise exclusive OR and assignment operator.	$C \^= 2$ is same as $C = C \^ 2$
$ =$	Bitwise inclusive OR and assignment operator.	$C = 2$ is same as $C = C 2$

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

    c = a;      // c is 5
    printf("c = %d\n", c);
    c += a;     // c is 10
    printf("c = %d\n", c);
```

```

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

return 0;
}

```

Output

```

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

```

Misc Operators: sizeof & ternary

Besides the operators discussed above, there are a few other important operators including **sizeof** and **? :** supported by the C Language.

Show Examples

Operator	Description	Example
sizeof()	Returns the size of a variable.	sizeof(a), where a is integer, will return 4.
&	Returns the address of a variable.	&a; returns the actual address of the variable.

*	Pointer to a variable.	*a;
? :	Conditional Expression.	If Condition is true ? then value X : otherwise value Y

Example : sizeof Operator

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

    return 0;
}
```

Output

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

Operators Precedence in C

Operator precedence determines the grouping of terms in an expression and decides how an expression is evaluated. Certain operators have higher precedence than

others; for example, the multiplication operator has a higher precedence than the addition operator.

For example, $x = 7 + 3 * 2$; here, x is assigned 13, not 20 because operator $*$ has a higher precedence than $+$, so it first gets multiplied with $3*2$ and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Show Examples

Category	Operator	Associativity
Postfix	() [] -> . ++ --	Left to right
Unary	+ - ! ~ ++ -- (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Left to right
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= =	Right to left

Comma	,	Left to right
-------	---	---------------

Conditional Operator:

C offers a ternary operator which is the conditional operator (?: in combination) to construct conditional expressions.

Operator	Description
? :	Conditional Expression

Special Operator:

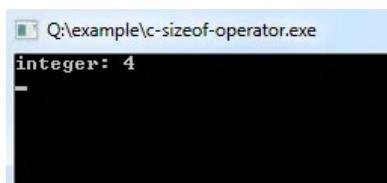
C supports some special operators

Operator	Description
sizeof()	Returns the size of a memory location.
&	Returns the address of a memory location.
*	Pointer to a variable.

Example:

```
#include <stdio.h>
void main()
{
    int i=10; // Variables Defining and Assign values
    printf("integer: %d\n", sizeof(i));
}
```

Program Output:



```
Q:\example\c-sizeof-operator.exe
integer: 4
```

C - Input and Output

When we say Input, it means to feed some data into a program. An input can be given in the form of a file or from the command line. C programming provides a set of built-in functions to read the given input and feed it to the program as per requirement.

When we say Output, it means to display some data on screen, printer, or in any file. C programming provides a set of built-in functions to output the data on the computer screen as well as to save it in text or binary files.

Input means to provide the program with some data to be used in the program and **Output** means to display data on the screen or write the data to a printer or a file.

The C programming language provides standard library functions to read any given input and to display data on the console.

The functions used for standard input and output are present in the **stdio.h** header file. Hence to use the functions we need to include the **stdio.h** header file in our program, as shown below.

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

Following are the functions used for standard input and output:

1. **printf()** function - Show Output
2. **scanf()** function - Take Input
3. **getchar()** and **putchar()** function
4. **gets()** and **puts()** function

Formatted I/O Functions

Formatted I/O functions are used to take various inputs from the user and display multiple outputs to the user. These types of I/O functions can help to display the output to the user in different formats using the format specifiers. These I/O supports all data types like int, float, char, and many more.

Why they are called formatted I/O?

These functions are called formatted I/O functions because we can use format specifiers in these functions and hence, we can format these functions according to our needs.

Format Specifiers for I/O

We use the following format specifiers:

1. %d for int
2. %f for float
3. %lf for double
4. %c for char

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

Data Type	FormatSpecifier
int	%d
char	%c
float	%f
double	%lf
short int	%hd
unsigned int	%u
long int	%li
long long int	%lli
unsigned long int	%lu
unsigned long long int	%llu

Data Type	Format Specifier
signed char	%c
unsigned char	%c
long double	%Lf

The following are the formatted I/O functions:

1. printf()
2. scanf()
3. sprintf()
4. sscanf()

printf() function - Show Output

The **printf()** function is the most used function in the C language. This function is defined in the **stdio.h** header file and is used to show output on the console (standard output).

This function is used to print a simple text sentence or value of any variable which can be of **int**, **char**, **float**, or any other datatype.

Syntax:

```
printf("Format Specifier", var1, var2, ...., varn);
```

printf() Example - Print a statement

Let's print a simple sentence using the printf() function.

```
#include <stdio.h>

int main() {
    // using printf()
```

```
printf("Welcome to C Programming Language");

return 0;

}
```

printf() Example - Print Integer

We can use the `printf()` function to print integer value coming from variable using the `%d` format specifier.

For example,

```
#include <stdio.h>
int main() {
    int x = 10;
    // using printf()
    printf("Value of x is: %d", x);

    return 0;
}
```

Output:

```
Value of x is: 10
```

printf() Example - Print Character

The `%c` format specifier is used to print character variable value using the `printf()` function.

```
#include <stdio.h>
int main() {
    // using printf()
    char gender = 'M';

    printf("John's Gender is: %c", gender);
    return 0;
}
```

```
}
```

Output:

```
John's Gender is: M
```

printf() Example - Print Float and Double

In the code example below, we have used the `printf()` function to print values of a `float` and `double` variable.

For `float` value we use the `%f` format specifier and for `double` value we use the `%lf` format specifier.

```
#include <stdio.h>
int main() {
    // using printf()
    float num1 = 15.50;
    double num2 = 15556522.0978678;

    printf("Value of num1 is: %f \n", num1);
    printf("Value of num2 is: %lf", num2);

    return 0;
}
```

Output:

```
Value of num1 is: 15.500000
```

```
Value of num2 is: 15556522.097868
```

printf() Example - Print multiple outputs

We can use a single `printf()` function to display values of multiple variables.

```
#include <stdio.h>
int main() {
    // using printf() for multiple outputs
```

```
int day = 09;
int month = 03;
int year = 2022;

printf("The date is: %d-%d-%d", day, month, year);

return 0;
}
```

Output:

```
The date is: 09-03-2022
```

scanf() function - Take Input

When we want to take input from the user, we use the **scanf()** function. When we take input from the user, we **store the input value** into a **variable**.

The **scanf()** function can be used to take any datatype input from user, all we have to take care is that the variable in which we store the value has the same datatype. In **scanf()** function we use &(address-of operator) which is used to store the variable value on the memory location of that variable.

Syntax:

```
scanf("Format Specifier", &var1, &var2, ...., &varn);
```

scanf() Example - Take Integer value input

If we have to take an integer value input from the user, we have to define an integer variable and then use the **scanf()** function.

```
#include <stdio.h>
int main() {
    // using scanf()
    int user_input;

    printf("Please enter a number: ");
```

```
scanf("%d", &user_input);
printf("You entered: %d", user_input);

return 0;
}
```

scanf() Example - Take Float value input

Just like integer value, we can take input for any different datatype. Let's see an example for float type value.

```
#include <stdio.h>

int main() {
    // using scanf()
    float user_input;

    printf("Please enter a decimal number: ");
    scanf("%f", &user_input);
    printf("You entered: %f", user_input);

    return 0;
}
```

Output:

```
Please enter a decimal number: 7.007
```

```
You entered: 7.007
```

scanf() Example - Take Character value input

Let's see how we can take a simple character input from user.

```
#include <stdio.h>

int main() {
    // using scanf()
```

```

char gender;

printf("Please enter your gender (M, F or O): ");
scanf("%c", &gender);
printf("Your gender: %c", gender);

return 0;
}

```

scanf() Example - Take Multiple Inputs

```

#include <stdio.h>
int main() {
    // using scanf() for multiple inputs
    char gender;
    int age;

    printf("Enter your age and then gender(M, F or O): ");
    scanf("%d %c", &age, &gender);
    printf("You entered: %d and %c", age, gender);

    return 0;
}

```

Output:

Enter your age and then gender(M, F or O): 32 M

You entered: 32 and M

sprintf() function:

sprintf stands for “string print”. This function is similar to printf() function but this function prints the string into a character array instead of printing it on the console screen.

Syntax:

sprintf(array_name, "format specifier", variable_name);

Example:

```
#include <stdio.h>
int main()
{
    char str[50];
    int a = 2, b = 8;

    // The string "2 and 8 are even number"
    // is now stored into str
    sprintf(str, "%d and %d are even number", a, b);

    // Displays the string
    printf("%s", str);
    return 0;
}
```

Output

2 and 8 are even number

sscanf() function:

sscanf stands for “**string scanf**”. This function is similar to scanf() function but this function reads data from the string or character array instead of the console screen.

Syntax:

```
sscanf(array_name, "format specifier", &variable_name);
```

Example:

```
// C program to implement sscanf() function
#include <stdio.h>
int main()
{
    char str[50];
    int a = 2, b = 8, c, d;
```

```

// The string "a = 2 and b = 8" is now stored into str character array
sprintf(str, "a = %d and b = %d", a, b);

// The value of a and b is now in
// c and d
sscanf(str, "a = %d and b = %d", &c, &d);
// Displays the value of c and d
printf("c = %d and d = %d", c, d);
return 0;
}

```

Output

c = 2 and d = 8

Unformatted Input/Output functions

Unformatted I/O functions are used only for character data type or character array/string and cannot be used for any other datatype. These functions are used to read single input from the user at the console and it allows to display the value at the console.

Why they are called unformatted I/O?

These functions are called unformatted I/O functions because we cannot use format specifiers in these functions and hence, cannot format these functions according to our needs.

The following are unformatted I/O functions:

1. getch()
2. getche()
3. getchar()
4. putchar()
5. gets()
6. puts()
7. putch()

getch(): Unformatted Input Function

getch() function reads a single character from the keyboard by the user but doesn't display that character on the console screen and immediately returned without pressing enter key. This function is declared in conio.h(header file). getch() is also used for hold the screen.

Syntax:

```
getch();  
or  
variable-name = getch();
```

Example:

```
// C program to implement getch() function  
  
#include <conio.h>  
  
#include <stdio.h>  
  
int main()  
{  
    printf("Enter any character: ");  
    // Reads a character but  
    // not displays  
    getch();  
    return 0;  
}
```

Output:

Enter any character:

getche(): Unformatted Input Function

getche() function reads a single character from the keyboard by the user and displays it on the console screen and immediately returns without pressing the enter key. This function is declared in conio.h(header file).

Syntax:

getche();
or
variable_name = getche();

Example:

```
// C program to implement the getche() function
#include <conio.h>
#include <stdio.h>
int main()
{
    printf("Enter any character: ");

    // Reads a character and displays immediately
    getch();
    return 0;
}
```

Output:

Enter any character: g

getchar(): Unformatted Input Function

The getchar() function is used to read only a first single character from the keyboard whether multiple characters is typed by the user and this function reads one character at one time until and unless the enter key is pressed. This function is declared in stdio.h(header file)

Syntax:

Variable-name = getchar();

```
// C program to implement the getchar() function
#include <conio.h>
#include <stdio.h>

// Driver code
int main()
{
    // Declaring a char type variable
```

```

char ch;

printf("Enter the character: ");

// Taking a character from keyboard
ch = getchar();

// Displays the value of ch
printf("%c", ch);
return 0;
}

```

Output:

Enter the character: a

a

putchar(): Unformatted Output Function

The putchar() function is used to display a single character at a time by passing that character directly to it or by passing a variable that has already stored a character. This function is declared in stdio.h(header file)

Syntax:

putchar(variable_name);

Example:

```

// C program to implement the putchar() function
#include <conio.h>
#include <stdio.h>

// Driver code
int main()
{
    char ch;
    printf("Enter any character: ");

    // Reads a character
    ch = getchar();

    // Displays that character
    putchar(ch);
    return 0;
}

```

Output:

Enter any character: Z

Z

gets(): Unformatted Input Function

gets() function reads a group of characters or strings from the keyboard by the user and these characters get stored in a character array. This function allows us to write space-separated texts or strings. This function is declared in stdio.h(header file).

Syntax:

```
char str[length of string in number];
//Declare a char type variable of any length
gets(str);
```

```
// C program to implement the gets() function
#include <conio.h>
#include <stdio.h>

// Driver code
int main()
{
    // Declaring a char type array
    // of length 50 characters
    char name[50];

    printf("Please enter some texts: ");

    // Reading a line of character or
    // a string
    gets(name);

    // Displaying this line of character
    // or a string
    printf("You have entered: %s",
           name);
    return 0;
}
```

Output:

Please enter some texts: C Programming Language

You have entered: C Programming Language

Difference between `scanf()` and `gets()`

The main difference between these two functions is that `scanf()` **stops reading characters when it encounters a space**, but `gets()` **reads space as a character too**.

If you enter a name as **Study Tonight** using `scanf()` it will only read and store **Study** and will leave the part after space. But `gets()` function will read it completely.

`puts()`: Unformatted Output Function

In C programming `puts()` function is used to display a group of characters or strings which is already stored in a character array. This function is declared in stdio.h (header file).

Syntax:

```
puts(identifier_name );
```

Example:

```
// C program to implement the puts() function
#include <stdio.h>

// Driver code
int main()
{
    char name[50];
    printf("Enter your text: ");

    // Reads string from user
    gets(name);

    printf("Your text is: ");

    // Displays string
    puts(name);
```

```
    return 0;  
}
```

Output:

Enter your text: CProgramming

Your text is: CProgramming

putch(): Unformatted Output Function

putch() function is used to display a single character which is given by the user and that character prints at the current cursor location. This function is declared in conio.h(header file)

Syntax:

```
putch(variable_name);
```

Example:

```
// C program to implement the putch() functions  
#include <conio.h>  
#include <stdio.h>  
int main()  
{  
    char ch;  
    printf("Enter any character:\n ");  
  
    // Reads a character from the keyboard  
    ch = getch();  
  
    printf("\nEnterd character is: ");  
  
    // Displays that character on the console  
    putch(ch);  
    return 0;  
}
```

Output:

Enter any character:

Entered character is: d

Formatted I/O vs Unformatted I/O

S No.	Formatted I/O functions	Unformatted I/O functions
1	These functions allow us to take input or display output in the user's desired format.	These functions do not allow to take input or display output in user desired format.
2	These functions support format specifiers.	These functions do not support format specifiers.
3	These are used for storing data more user friendly	These functions are not more user-friendly.
4	Here, we can use all data types.	Here, we can use only character and string data types.
5	printf(), scanf, sprintf() and sscanf() are examples of these functions.	getch(), getche(), gets() and puts(), are some examples of these functions

Example:

```
#include <stdio.h>

void main()
{
    /* character array of length 100 */
```

```
char str[100];
printf("Enter a string: ");
gets(str);
puts(str);
getch();
return 0;
}
```

Enter a string: Studytonight

Studytonight

```
#include <stdio.h>
int main()
{
    // Declare the variables
    int num;
    char ch;
    float f;

    // --- Integer ---
    // Input the integer
    printf("Enter the integer: ");
    scanf("%d", &num);

    // Output the integer
    printf("\nEnterd integer is: %d", num);

    // --- Float ---
}
```

```

// Input the float
printf("\n\nEnter the float: ");
scanf("%f", &f);

// Output the float
printf("\nEnterd float is: %f", f);

// --- Character ---

// Input the Character
printf("\n\nEnter the Character: ");
scanf("%c", &ch);

// Output the Character
printf("\nEnterd integer is: %c", ch);

return 0;
}

```

Output:

```

Enter the integer: 10
Entered integer is: 10

```

```

Enter the float: 2.5
Entered float is: 2.500000

```

```

Enter the Character: A
Entered Character is: A

```

```

// C program to show input and output

#include <stdio.h>

int main()
{

```

```

// Declare string variable
// as character array
char str[50];

// --- String ---
// To read a word

// Input the Word
printf("Enter the Word: ");
scanf("%s\n", str);

// Output the Word
printf("\nEntered Word is: %s", str);

// --- String ---
// To read a Sentence

// Input the Sentence
printf("\n\nEnter the Sentence: ");
scanf("%[^\\n]\\ns", str);

// Output the String
printf("\nEnterd Sentence is: %s", str);

return 0;
}

```

Output:

Enter the Word: GeeksForGeeks

Entered Word is: GeeksForGeeks

Enter the Sentence: Geeks For Geeks

Entered Sentence is: Geeks For Geeks

Example:

```

#include<stdio.h>

void main ()
{
    char s[30];
    printf("Enter the string? ");
    gets(s);
    printf("You entered %s",s);
}

```

```
}
```

Output:

```
Enter the string? Sai Pali Institute of Technology & Management
```

```
You entered Sai Pali Institute of Technology & Management
```

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

Output:

```
Enter the string? Sai Pali Institute of Technology & Management
```

```
Sai Pali Institute
```

Let's see an example to read a string using gets() and print it on the console using puts().

```
#include<stdio.h>
#include <string.h>
int main(){
    char name[50];
    printf("Enter your name: ");
    gets(name); //reads string from user
    printf("Your name is: ");
    puts(name); //displays string
    return 0;
}
```

Output:

```
Enter your name: Bill Robert Gates
```

```
Your name is: Bill Robert Gates
```

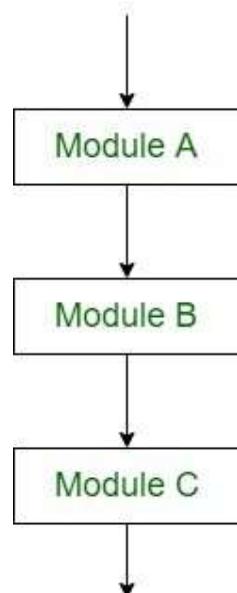
Control Structures

Control Structures are just a way to specify flow of control in programs. Any algorithm or program can be clearer and more understood if they use self-contained modules called as logic or control structures. It basically analyses and chooses in which direction a program flows based on certain parameters or conditions. There are three basic types of logic, or flow of control, known as:

1. Sequence logic, or sequential flow
2. Selection logic, or conditional flow
3. Iteration logic, or repetitive flow

1. Sequential Logic (Sequential Flow)

Sequential logic as the name suggests follows a serial or sequential flow in which the flow depends on the series of instructions given to the computer. Unless new instructions are given, the modules are executed in the obvious sequence. The sequences may be given, by means of numbered steps explicitly. Also, implicitly follows the order in which modules are written. Most of the processing, even some complex problems, will generally follow this elementary flow pattern.



2. Selection Logic (Conditional Flow)

Selection Logic simply involves a number of conditions or parameters which decides one out of several written modules. The structures which use this type of logic are known as **Conditional Structures**. These structures can be of three types:

- **Single Alternative:** This structure has the form:

If (condition) then:

[Module A]

[End of If structure]

- **Double Alternative:** This structure has the form:

If (Condition), then:

[Module A]

Else:

[Module B]

[End if structure]

- **Multiple Alternatives:** This structure has the form:

If (condition A), then:

[Module A]

Else if (condition B), then:

[Module B]

..

..

Else if (condition N), then:

[Module N]

[End If structure]

3. Iteration Logic (Repetitive Flow)

The Iteration logic employs a loop which involves a repeat statement followed by a

module known as the body of a loop.

The two types of these structures are:

- **Repeat-For Structure**

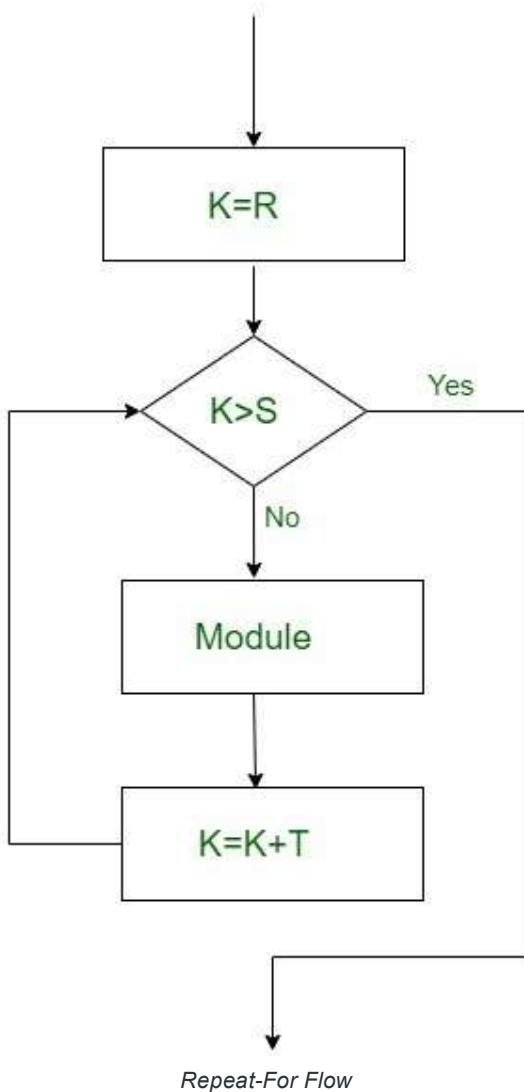
This structure has the form:

Repeat for i = A to N by I:

[Module]

[End of loop]

Here, A is the initial value, N is the end value and I is the increment. The loop ends when A>B. K increases or decreases according to the positive and negative value of I respectively.



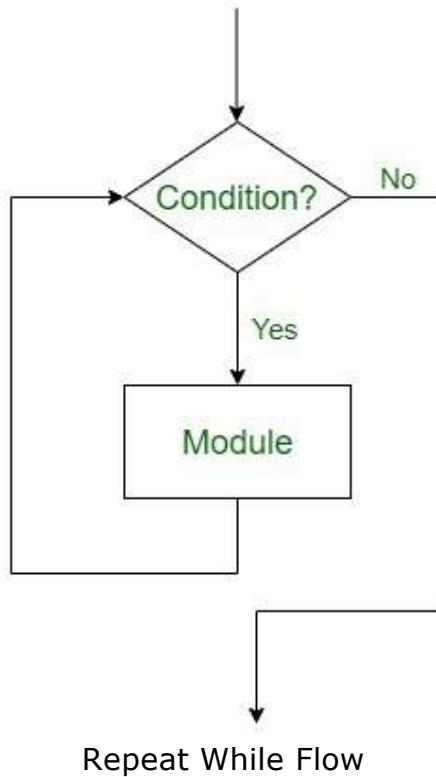
- **Repeat-While Structure**

It also uses a condition to control the loop. This structure has the form:

Repeat while condition:

[Module]

[End of Loop]



IF Statements:

If statements in C is used to control the program flow based on some condition, it's used to execute some statement code block if the expression is evaluated to true. Otherwise, it will get skipped. This is the simplest way to modify the control flow of the program.

There are four different types of if statement in C. These are:

1. Simple if Statement
2. if-else Statement
3. Nested if-else Statement

4. else-if Ladder

Simple if statement:

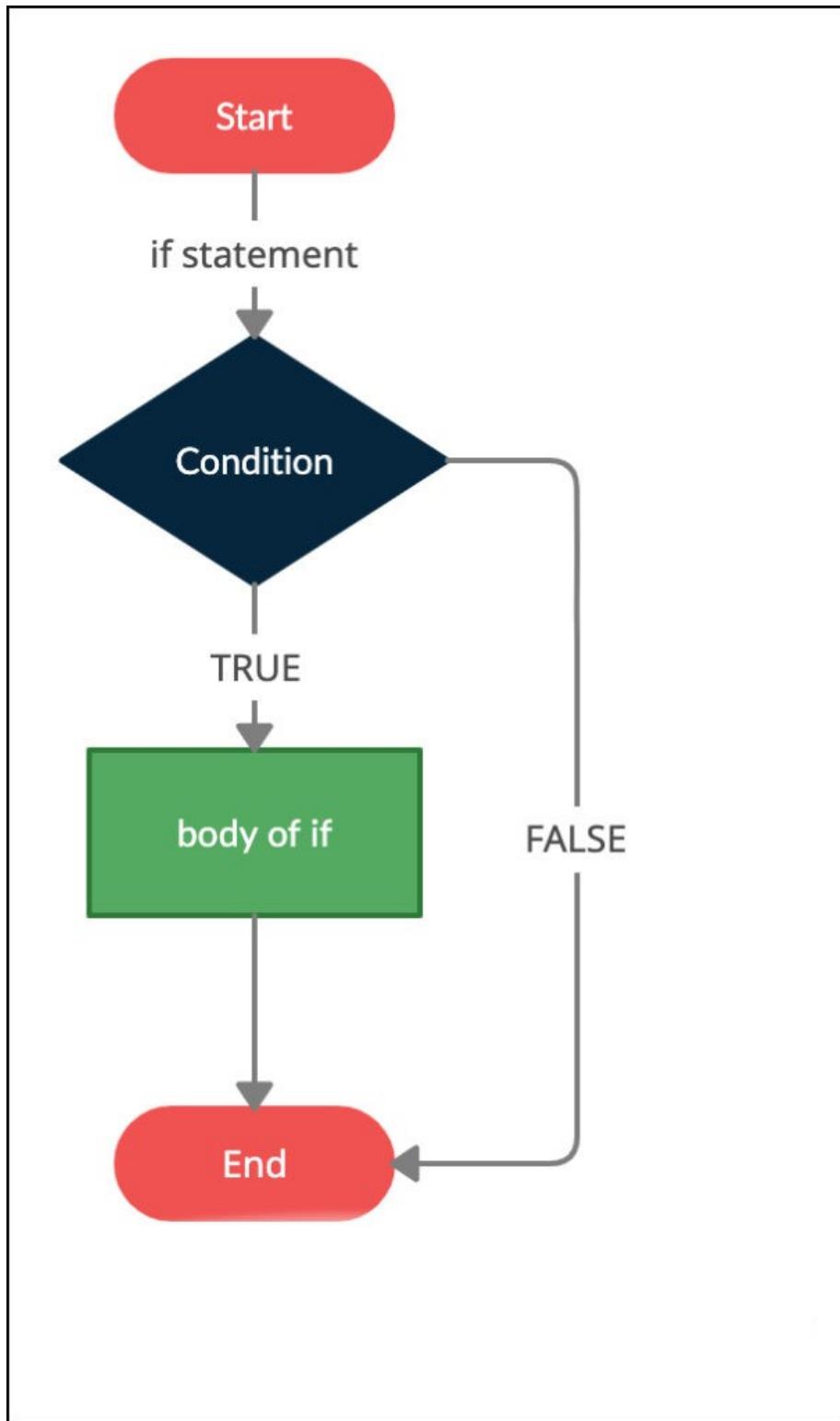
The statements inside the body of “if” only execute if the given condition returns true. If the condition returns false then the statements inside “if” are skipped.

The basic format of if statement is:

Syntax:

```
if(test_expression)
{
    statement 1;
    statement 2;
    ...
}
```

'Statement n' can be a statement or a set of statements, and if the test expression is evaluated to **true**, the statement block will get executed, or it will get skipped.



Examples:

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

```
void main()
```

```

{
int a = 15, b = 20;

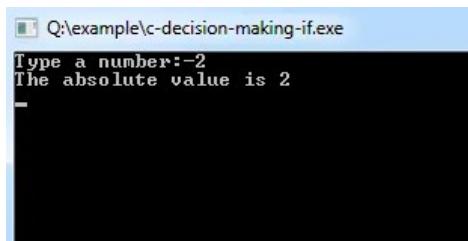
if (b > a) {
    printf("b is greater");
}
}

#include<stdio.h>
void main()
{
    int number;
    printf("Type a number:");
    scanf("%d", &number);

    if (number < 0) { // check whether the number is negative number.
        number = -number; // If it is a negative then convert it into positive.
        printf("The absolute value is %d\n", number);
    }
}

```

Program Output:



Find the Largest Number

Here is a program depicting to find the largest number of the three provided number:

```
#include <stdio.h>
int main()
{
    int a, b, c;
    printf("Enter three numbers?");
    scanf("%d %d %d",&a,&b,&c);
    if(a>b && a>c)
    {
        printf("%d is largest",a);
    }
}
```

```

}
if(b>a && b > c)
{
    printf("%d is largest",b);
}
if(c>a && c>b)
{
    printf("%d is largest",c);
}
if(a == b && a == c)
{
    printf("All are equal");
}
}

```

Output :

Enter three numbers?

12 23 34

34 is largest

Write a program to convert negative number to positive number.

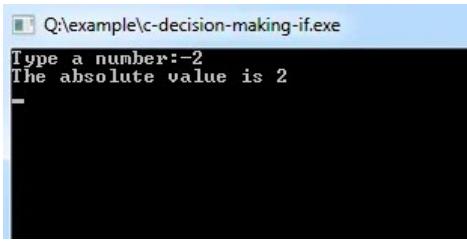
```

#include<stdio.h>
void main()
{
    int number;
    printf("Type a number:");
    scanf("%d", &number);

    if (number < 0) { // check whether the number is negative number.
        number = -number; // If it is a negative then convert it into positive.
        printf("The absolute value is %d\n", number);
    }
}

```

Output:



```
Q:\example\c-decision-making-if.exe
Type a number:-2
The absolute value is 2
```

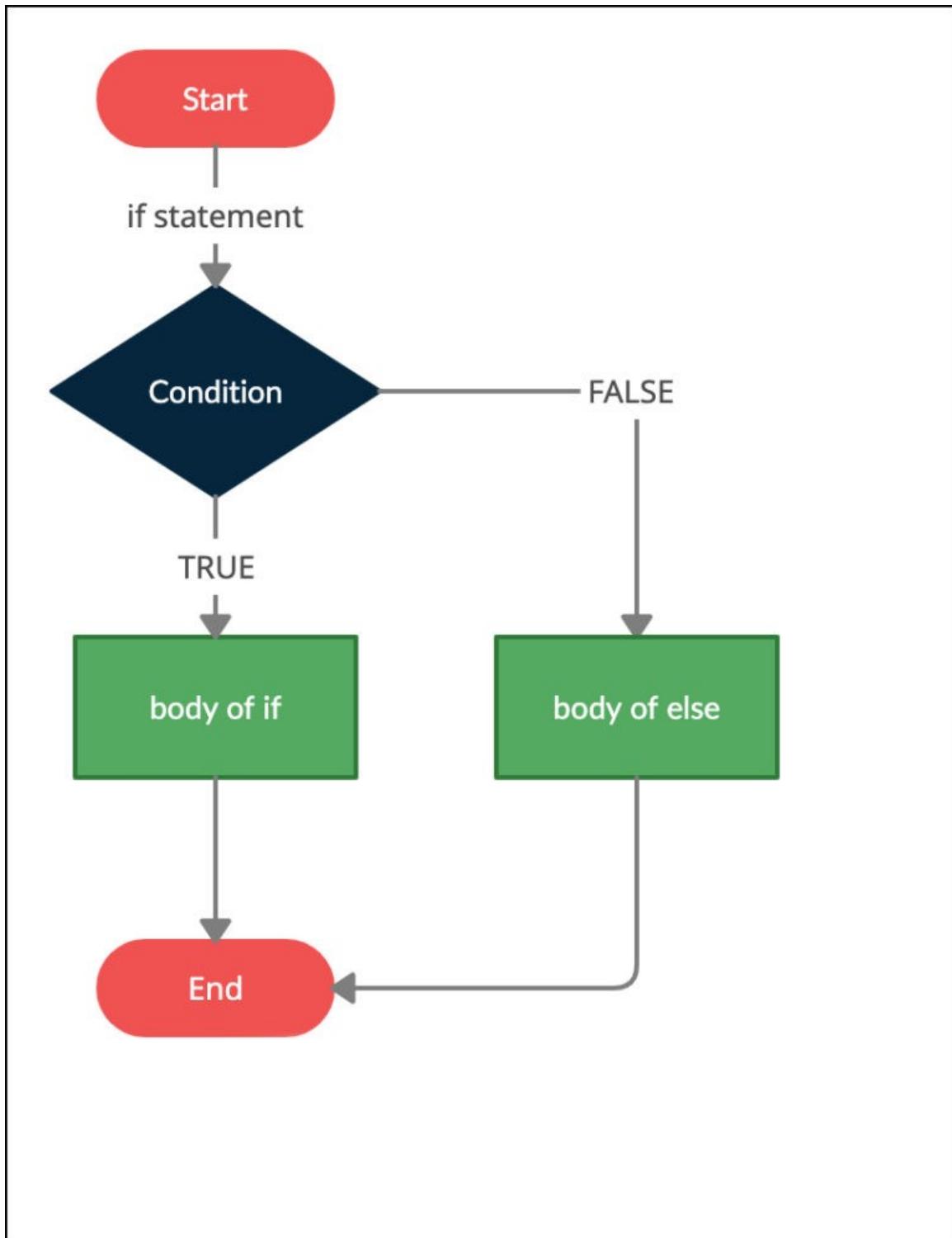
If...else statement:

If condition returns true then the statements inside the body of "if" are executed and the statements inside body of "else" are skipped.

If condition returns false then the statements inside the body of "if" are skipped and the statements in "else" are executed.

Syntax:

```
if(condition) {
    // Statements inside body of if
}
else {
    //Statements inside body of else
}
```



Example of if else statement

In this program user is asked to enter the age and based on the input, the if..else statement checks whether the entered age is greater than or equal to 18. If this condition meet then display message "You are eligible for voting", however if the condition doesn't meet then display a different message "You are not eligible for voting".

```
#include <stdio.h>
int main()
{
    int age;
    printf("Enter your age:");
    scanf("%d",&age);
    if(age >=18)
    {
        /* This statement will only execute if the
         * above condition (age>=18) returns true
         */
        printf("You are eligible for voting");
    }
    else
    {
        /* This statement will only execute if the
         * condition specified in the "if" returns false.
         */
        printf("You are not eligible for voting");
    }
    return 0;
}
```

Output:

```
Enter your age:14
You are not eligible for voting
```

Note: If there is **only one statement** is present in the “if” or “else” body then you do not need to use the braces (parenthesis). For example the above program can be rewritten like this:

```
#include <stdio.h>
int main()
{
    int age;
    printf("Enter your age:");
    scanf("%d",&age);
    if(age >=18)
        printf("You are eligible for voting");
    else
        printf("You are not eligible for voting");
    return 0;
}
```

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

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

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

    return 0;
}
```

Output

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

```
#include<stdio.h>

void main()
{
    int a, b;

    printf("Please enter the value for a:");
    scanf("%d", &a);

    printf("\nPlease enter the value for b:");
    scanf("%d", &b);

    if (a > b) {
        printf("\n a is greater");
    } else {
        printf("\n b is greater");
    }
}
```

```
#include<stdio.h>

void main() {
    int num;
    printf("Enter the number:");
    scanf("%d", &num);

    /* check whether the number is negative number */
    if (num < 0)
        printf("The number is negative.");
    else
        printf("The number is positive.");
}
```

Nested If..else statement

When an if else statement is present inside the body of another "if" or "else" then this is called nested if else.

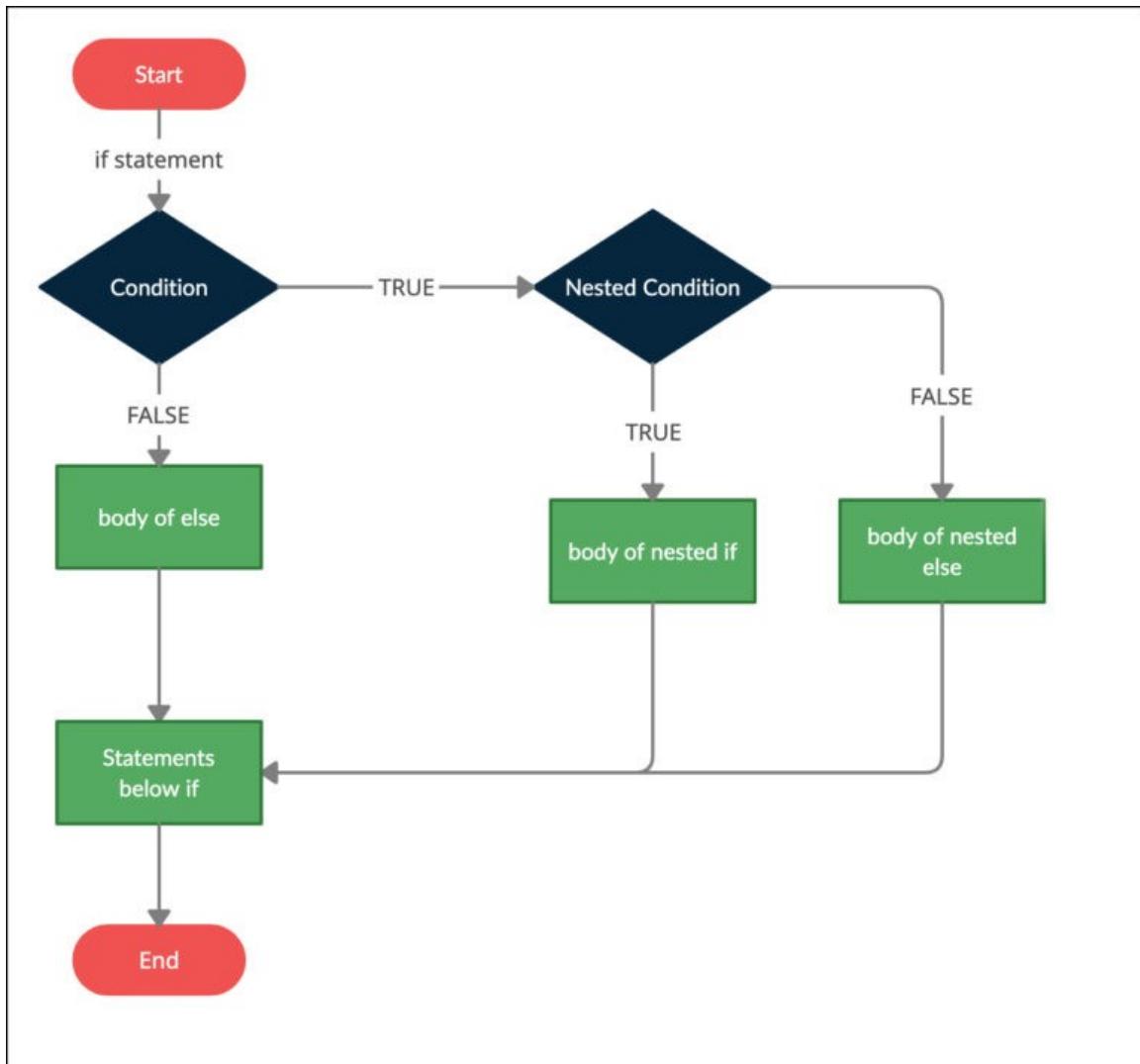
Syntax

```
if(condition) {
    //Nested if else inside the body of "if"
```

```

if(condition2) {
    //Statements inside the body of nested "if"
}
else {
    //Statements inside the body of nested "else"
}
else {
    //Statements inside the body of "else"
}

```



```

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

```

```

int number1, number2;
printf("Enter two integers: ");
scanf("%d %d", &number1, &number2);

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

return 0;
}

```

```

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

    printf("Please Enter Your Age Here:\n");
    scanf("%d", &age);

    if ( age < 18 )
    {
        printf("You are Minor.\n");
        printf("Not Eligible to Work");
    }
    else

```

```

{

if (age >= 18 && age <= 60 )
{
    printf("You are Eligible to Work \n");
    printf("Please fill in your details and apply\n");
}
else
{
    printf("You are too old to work as per the Government rules\n");
    printf("Please Collect your pension! \n");
}

}

return 0;
}

```

Within this example, If the age of a person is less than 18, he is not eligible to work. If the age is greater than or equal to 18, then the first condition fails, it will check the else statement.

In the Else statement, there is another if condition called Nested If in C to check further.

- In this example, the Nested IF Statement checks the person's age greater than or equal to 18 and less than or equal to 60. When the condition is TRUE, then he can apply for the job.
- If the condition is FALSE, then the statement – he is too old to work as per the government.

1st Output: age=10

Please Enter Your Age Here:

10

You are Minor.

Not eligible to work

2nd OUTPUT : age = 27. The first If condition is FALSE. Here, Nested IF condition is TRUE

```
Please Enter Your Age Here:
```

```
27
```

```
You are Eligible to Work
```

```
Please fill in your details and apply
```

OUTPUT 3: age as 61. It means both If condition and also Nested IF condition failed here

```
Please Enter Your Age Here:
```

```
61
```

```
You are too old to work as per the Government rules
```

```
Please Collect your pension!
```

else..if ladder statement:

The else..if statement is useful when you need to check multiple conditions within the program, nesting of if-else blocks can be avoided using else..if statement.

Syntax of else..if statement:

```
if (condition1)
{
    //These statements would execute if the condition1 is true
}
else if(condition2)
{
    //These statements would execute if the condition2 is true
}
else if (condition3)
{
    //These statements would execute if the condition3 is true
```

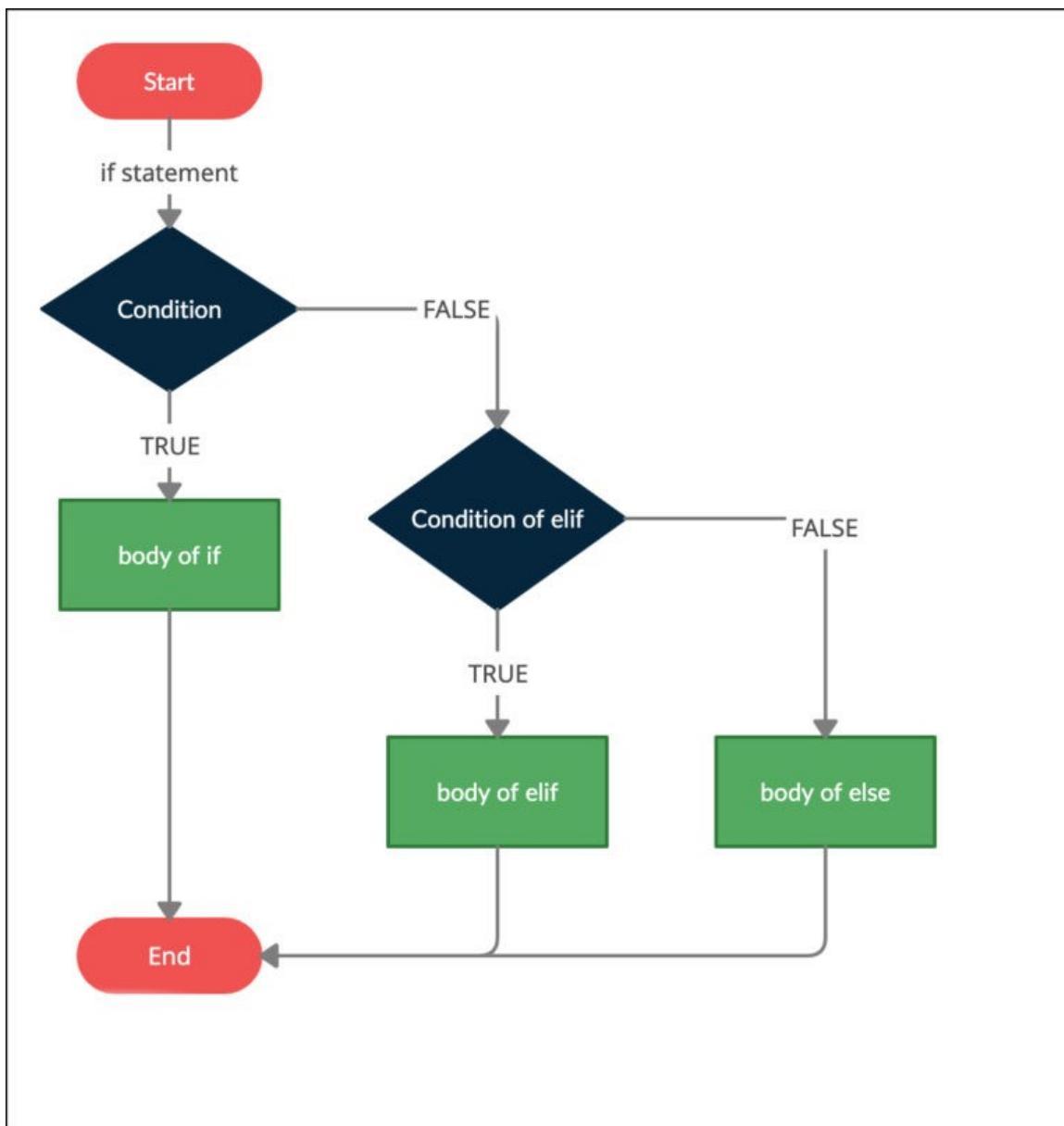
```
}

.

.

else
{
    //These statements would execute if all the conditions return false.
}
```

Flowchart:



Example:

```
#include<stdio.h>
int main()
{
int number=0;
printf("enter a number:");
scanf("%d",&number);
if(number==10)
{
printf("number is equals to 10");
}
else if(number==50)
{
printf("number is equal to 50");
}
else if(number==100)
{
printf("number is equal to 100");
}
else
{
printf("number is not equal to 10, 50 or 100");
}
return 0;
}
```

Output:

```
enter a number:4
number is not equal to 10, 50 or 100
enter a number:50
number is equal to 50
```

Example:

In this C else if program, the user is asked to enter their total six subject marks. Using this Else if statement, we will decide whether the person is qualified for scholarship or not

```
/* Example for Else If Statement in C Language */

#include <stdio.h>

int main()
{
    int Totalmarks;

    //Imagine you have 6 subjects and Grand total is 600
    printf("Please Enter your Total Marks\n");
    scanf( "%d", &Totalmarks );

    if (Totalmarks >= 540)
    {
        printf("You are eligible for Full Scholarship\n");
        printf("Congratulations\n");
    }
    else if (Totalmarks >= 480)
    {
        printf("You are eligible for 50 Percent Scholarship\n");
        printf("Congratulations\n");
    }
    else if (Totalmarks >= 400)
    {
        printf("You are eligible for 10 Percent Scholarship\n");
        printf("Congratulations\n");
    }
    else
    {
```

```
printf("You are Not eligible for Scholarship\n");
printf("We are really Sorry for You\n");
}
}
```

Example:

Print Grade Card

Here is another example of if else-if statement depicting a program to calculate the grade of the student according to the specified marks:

```
#include <stdio.h>
int main()
{
    int marks;
    printf("Enter your marks?");
    scanf("%d",&marks);
    if(marks > 85 && marks <= 100)
    {
        printf("Congrats ! you scored grade A ...");
    }
    else if (marks > 60 && marks <= 85)
    {
        printf("You scored grade B + ...");
    }
    else if (marks > 40 && marks <= 60)
    {
        printf("You scored grade B ...");
    }
    else if (marks > 30 && marks <= 40)
    {
        printf("You scored grade C ...");
    }
    else
```

```
{  
    printf("Sorry you are fail ...");  
}  
}
```

Output :

```
Enter your marks?10  
Sorry you are fail ...  
Enter your marks?40  
You scored grade C ...  
Enter your marks?90  
Congrats ! you scored grade A ...
```

Switch statement:

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

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

Syntax of switch...case

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

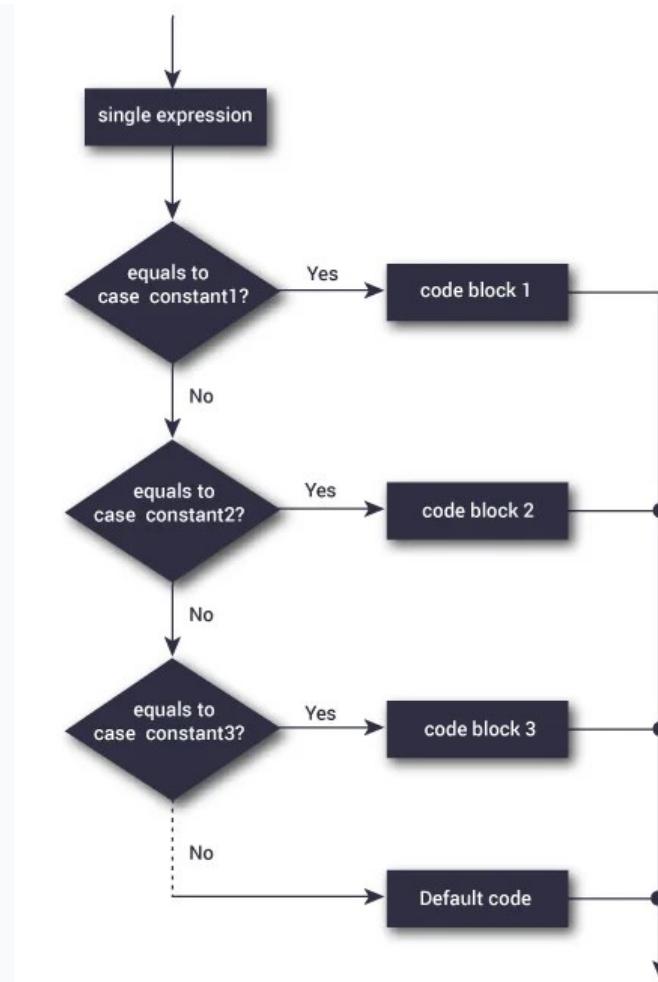
    case constant2:
        // statements
        break;
    .
    .
    .

    default:
        // default statements
}
```

How does the switch statement work?

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

- If there is a match, the corresponding statements after the matching label are executed. For example, if the value of the expression is equal to `constant2`, statements after `case constant2:` are executed until `break` is encountered.
- If there is no match, the default statements are executed.



Example:

```

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

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

    printf("Enter an operator (+, -, *, /): ");
    scanf("%c", &operation);
    printf("Enter two operands: ");
    scanf("%lf %lf", &n1, &n2);

    switch(operation)
    {
        case '+':
            printf("%.1lf + %.1lf = %.1lf", n1, n2, n1+n2);
            break;

        case '-':
            printf("%.1lf - %.1lf = %.1lf", n1, n2, n1-n2);
            break;

        case '*':
            printf("%.1lf * %.1lf = %.1lf", n1, n2, n1*n2);
            break;

        case '/':
            if(n2 != 0)
                printf("%.1lf / %.1lf = %.1lf", n1, n2, n1/n2);
            else
                printf("Error! Division by zero.");
            break;
    }
}
  
```

```

printf("%.1lf - %.1lf = %.1lf",n1, n2, n1-n2);
break;

case '*':
printf("%.1lf * %.1lf = %.1lf",n1, n2, n1*n2);
break;

case '/':
printf("%.1lf / %.1lf = %.1lf",n1, n2, n1/n2);
break;

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

return 0;
}

```

Output

Enter an operator (+, -, *, /): -

Enter two operands: 32.5

12.4

32.5 - 12.4 = 20.1

You can also use characters in switch case. for example –

```

#include <stdio.h>
int main()
{
    char ch='b';
    switch (ch)
    {
        case 'd':
            printf("CaseD ");
            break;
        case 'b':

```

```
    printf("CaseB");
    break;
case 'c':
    printf("CaseC");
    break;
case 'z':
    printf("CaseZ ");
    break;
default:
    printf("Default ");
}
return 0;
}
```

Output:

CaseB

Using Switch statement, write a program that displays the following menu for the food items available to take order from the customer:

- B= Burger
- F= French Fries
- P= Pizza
- S= Sandwiches

The program inputs the type of food and quantity. It finally displays the total charges for the order according to following criteria:

- Burger = Rs. 200
- French Fries= Rs. 50
- Pizza= Rs. 500
- Sandwiches= Rs. 150

```

#include <stdio.h>
int main()
{
    int b,f,p,s,Burger,French,Pizza,Sandwiches;
    char ch,B,F,P,S;
    printf("1=Burger\n2=French Fries\n3=pizza\n4=Sandwiches\n");
    printf("Enter your order \nplease Enter the choice 1,2,3,4\n");
    scanf("%d",&ch);
    switch(ch)
    {
        case 1:
            printf("your order is Burger\n");
            printf("please enter your quantity ");
            scanf("%d",&b);
            Burger=200*b;
            printf("your total charges is: %d",Burger);
            break;
        case 2:
            printf("your order is French \n");
            printf("please enter your quantity ");
            scanf("%d",&f);
            French=50*f;
            printf("your total charges is: %d",French);
            break;
        case 3:
            printf("your order is Pizza\n");
            printf("please enter your quantity ");
    }
}

```

```

scanf("%d",&p);
Pizza=500*p;

printf("your total charges is: %d",Pizza);
break;
case 4:
printf("your order is Sandwiches\n");
printf("please enter your quantity ");

scanf("%d",&s);
Sandwiches=150*s;

printf("your total charges is: %d",Sandwiches);
break;

default:
printf("invalid your choice");

break;
}
return 0;
}

```

Example 2:

```

#include <stdio.h>

int main()
{
printf("B=BURGER\nF=FRENCH FRY\nP=PIZZA\nS=SANDWICHES\n");
char ss;
int n,x;

```

```
scanf("%c",&ss);

switch(ss)

{

case 'B':

    scanf("%d",&n);

    x= n*200;

    printf("BURGER=Rs %d",x);

    break;

case 'F':

    scanf("%d",&n);

    x= n*50;

    printf("FRENCH FRY=Rs %d",x);

    break;

case 'P':

    scanf("%d",&n);

    x= n*500;

    printf("PIZZA=Rs %d",x);

    break;

case 'S':

    scanf("%d",&n);

    x= n*150;

    printf("SANDWICHES=Rs %d",x);

    break;

default:

    printf("invalid your choice");
```

```
}
```

```
return 0;
```

```
}
```

Example:

Write a c program using switch case statement to output the following

```
2*3=6
```

```
2+3=5
```

```
4+6=10
```

```
4/5=0.8
```

```
#include <stdio.h>

void main()

{
    char operator;
    int num1,num2;
    printf("\n Enter the operator (+, -, *, /):");
    scanf("%c",&operator);
    printf("\n Enter the Two numbers:");
    scanf("%d%d",&num1,&num2);
    switch (operator)
    {
        case '+':
            printf("%d+%d=%d",num1,num2,num1+num2);
            break;
    }
}
```

```

case '-':
    printf("%d-%d=%d",num1,num2,num1-num2);
    break;
case '*':
    printf("%d*%d=%d",num1,num2,num1*num2);
    break;
case '/':
    printf("%d / %d = %d",num1,num2,num1/num2);
    break;
default:
    printf("\n Enter the operator only");
    break;
}
}

```

goto Statement

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

Syntax of goto Statement

```
goto label;
```

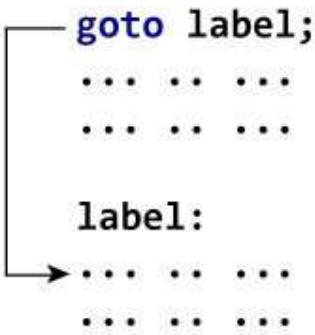
```
.... . . .
```

```
.... . . .
```

```
label:
```

```
statement;
```

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



Working of goto Statement

Example: goto Statement

```

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

#include <stdio.h>

int main() {

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

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

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

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

    return 0;
}
  
```

Output

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

Example:

```
//program demo using goto statement
#include <stdio.h>

int main() {
    int sum = 0;
    int i;
    for (i=0; i<=10; i++)
    {
        sum = sum + i; //sum += i;
        if(i==5)
        {
            goto addition;
        }
    }
    addition:
    printf("Sum = %d\n",sum);
    return 0;
}
```

Loop in C

Looping Statements in C execute the sequence of statements many times until the stated condition becomes false. A loop in C consists of two parts, a body of a loop and a control statement. The control statement is a combination of some conditions that direct the body of the loop to execute until the specified condition becomes false. The purpose of the C loop is to repeat the same code a number of times.

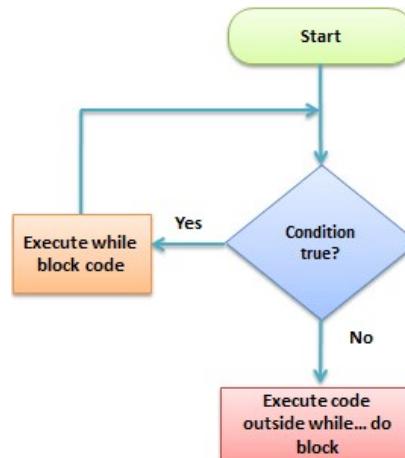
Types of Loops in C

Depending upon the position of a control statement in a program, looping statement in C is classified into two types:

1. Entry controlled loop
2. Exit controlled loop

In an **entry control loop in C**, a condition is checked before executing the body of a loop. It is also called as a pre-checking loop.

In an **exit-controlled loop**, a condition is checked after executing the body of a loop. It is also called as a post-checking loop.



The control conditions must be well defined and specified otherwise the loop will execute an infinite number of times. The loop that does not stop executing and processes the statements number of times is called as an **infinite loop**. An infinite loop

is also called as an “**Endless loop**.” Following are some characteristics of an infinite loop:

1. No termination condition is specified.
2. The specified conditions never meet.

The specified condition determines whether to execute the loop body or not.

‘C’ programming language provides us with three types of loop constructs:

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

Sr. No.	Loop Type	Description
1.	While Loop	In while loop, a condition is evaluated before processing a body of the loop. If a condition is true then and only then the body of a loop is executed.
2.	Do-While Loop	In a do...while loop, the condition is always executed after the body of a loop. It is also called an exit-controlled loop.
3.	For Loop	In a for loop, the initial value is performed only once, then the condition tests and compares the counter to a fixed value after each iteration, stopping the for loop when false is returned.

For loop

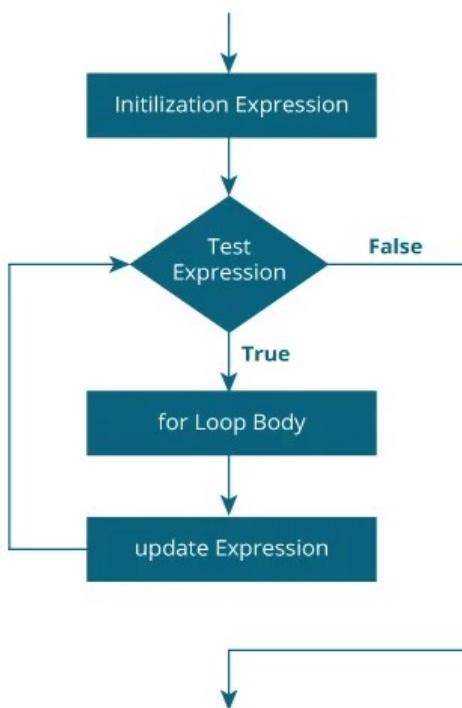
A for loop is a more efficient loop structure in ‘C’ programming. The general structure of for loop syntax in C is as follows:

Syntax of For Loop in C:

```
for (initial value; condition; increment or decrement)
{
    statements;
}
```

- The initial value of the for loop is performed only once.
- The condition is a Boolean expression that tests and compares the counter to a fixed value after each iteration, stopping the for loop when false is returned.
- The incrementation/decrementation increases (or decreases) the counter by a set value.

Flowchart:



Working of for loop

Following program illustrates the for loop in C programming example:

```
#include<stdio.h>
int main()
{
    int number;
    for(number=1;number<=10;number++) //for loop to print 1-10 numbers
    {
        printf("%d\n",number); //to print the number
    }
}
```

```
    return 0;  
}
```

Output:

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

```
// Program to calculate the sum of first n natural numbers  
// Positive integers 1,2,3...n are known as natural numbers  
  
#include <stdio.h>  
int main()  
{  
    int num, count, sum = 0;  
  
    printf("Enter a positive integer: ");  
    scanf("%d", &num);  
  
    // for loop terminates when num is less than count  
    for(count = 1; count <= num; ++count)  
    {  
        sum += count;  
    }  
  
    printf("Sum = %d", sum);  
  
    return 0;  
}
```

Output

```
Enter a positive integer: 10
```

Sum = 55

Nested For Loop

Nesting of loop is also possible. Let's take an example to understand this:

```
#include <stdio.h>
int main()
{
    for (int i=0; i<2; i++)
    {
        for (int j=0; j<4; j++)
        {
            printf("%d, %d\n",i ,j);
        }
    }
    return 0;
}
```

Output:

```
0, 0
0, 1
0, 2
0, 3
1, 0
1, 1
1, 2
1, 3
```

Consider the following example with multiple conditions in for loop, that uses nested for loop in C programming to output a multiplication table:

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

```

int i, j;
int table = 2;
int max = 5;
for (i = 1; i <= table; i++) { // outer loop
    for (j = 0; j <= max; j++) { // inner loop
        printf("%d x %d = %d\n", i, j, i*j);
    }
    printf("\n"); /* blank line between tables */
}
}

```

Output:

```

1 x 0 = 0
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5

```

```

2 x 0 = 0
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10

```

```

#include <stdio.h>
int main()
{
    int i,j,k;
    for(i=0,j=0,k=0;i<4,k<8,j<10;i++)
    {
        printf("%d %d %d\n",i,j,k);
    }
}

```

```
    j+=2;  
    k+=3;  
}  
}
```

Output:

```
0 0 0  
1 2 3  
2 4 6  
3 6 9  
4 8 12
```

```
#include<stdio.h>  
void main ()  
{  
    int i=0,j=2;  
    for(i = 0;i<5;i++,j=j+2)  
    {  
        printf("%d %d\n",i,j);  
    }  
}
```

Output:

```
0 2  
1 4  
2 6  
3 8  
4 10
```

Half Pyramid of *

```
*  
* *  
* * *  
* * * *  
* * * * *
```

C Program

```
#include <stdio.h>  
int main() {  
    int i, j, rows;  
    printf("Enter the number of rows: ");  
    scanf("%d", &rows);  
    for (i = 1; i <= rows; ++i) {  
        for (j = 1; j <= i; ++j) {  
            printf("* ");  
        }  
        printf("\n");  
    }  
    return 0;  
}
```

Half Pyramid of Numbers

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

C Program

```
#include <stdio.h>
int main() {
    int i, j, rows;
    printf("Enter the number of rows: ");
    scanf("%d", &rows);
    for (i = 1; i <= rows; ++i) {
        for (j = 1; j <= i; ++j) {
            printf("%d ", j);
        }
        printf("\n");
    }
    return 0;
}
```

Half Pyramid of Alphabets

```
A
B B
C C C
D D D D
E E E E E
```

C Program

```
#include <stdio.h>
int main() {
    int i, j;
    char input, alphabet = 'A';
    printf("Enter an uppercase character you want to print in the last row: ");
    scanf("%c", &input);
    for (i = 1; i <= (input - 'A' + 1); ++i) {
        for (j = 1; j <= i; ++j) {
```

```
    printf("%c ", alphabet);
}
++alphabet;
printf("\n");
}
return 0;
}
```

Inverted half pyramid of *

```
* * * * *
* * * *
* * *
* *
*
```

C Program

```
#include <stdio.h>
int main() {
    int i, j, rows;
    printf("Enter the number of rows: ");
    scanf("%d", &rows);
    for (i = rows; i >= 1; --i) {
        for (j = 1; j <= i; ++j) {
            printf("* ");
        }
        printf("\n");
    }
    return 0;
}
```

Inverted half pyramid of numbers

```
1 2 3 4 5
```

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

C Program

```
#include <stdio.h>  
  
int main() {  
    int i, j, rows;  
    printf("Enter the number of rows: ");  
    scanf("%d", &rows);  
    for (i = rows; i >= 1; --i) {  
        for (j = 1; j <= i; ++j) {  
            printf("%d ", j);  
        }  
        printf("\n");  
    }  
    return 0;  
}
```

Full Pyramid of *

```
*  
* * *  
* * * * *  
* * * * * * * *
```

C Program

```
#include <stdio.h>  
  
int main() {  
    int i, space, rows, k = 0;
```

```

printf("Enter the number of rows: ");
scanf("%d", &rows);
for (i = 1; i <= rows; ++i, k = 0) {
    for (space = 1; space <= rows - i; ++space) {
        printf(" ");
    }
    while (k != 2 * i - 1) {
        printf("*");
        ++k;
    }
    printf("\n");
}
return 0;
}

```

Full Pyramid of Numbers

```

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

```

C Program

```

#include <stdio.h>
int main() {
    int i, space, rows, k = 0, count = 0, count1 = 0;
    printf("Enter the number of rows: ");
    scanf("%d", &rows);
    for (i = 1; i <= rows; ++i) {
        for (space = 1; space <= rows - i; ++space) {
            printf(" ");
            ++count;
        }
        while (k != 2 * i - 1) {

```

```

if (count <= rows - 1) {
    printf("%d ", i + k);
    ++count;
} else {
    ++count1;
    printf("%d ", (i + k - 2 * count1));
}
++k;
}
count1 = count = k = 0;
printf("\n");
}
return 0;
}

```

Inverted full pyramid of *

```

* * * * * * *
* * * * * *
* * * *
* *
*

```

C Program

```

#include <stdio.h>
int main() {
    int rows, i, j, space;
    printf("Enter the number of rows: ");
    scanf("%d", &rows);
    for (i = rows; i >= 1; --i) {
        for (space = 0; space < rows - i; ++space)
            printf(" ");
        for (j = i; j <= 2 * i - 1; ++j)
            printf("* ");
        for (j = 0; j < i - 1; ++j)
            printf("* ");
        printf("\n");
    }
    return 0;
}

```

```
}
```

Pascal's Triangle

```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```

C Program

```
#include <stdio.h>

int main() {
    int rows, coef = 1, space, i, j;
    printf("Enter the number of rows: ");
    scanf("%d", &rows);
    for (i = 0; i < rows; i++) {
        for (space = 1; space <= rows - i; space++)
            printf(" ");
        for (j = 0; j <= i; j++) {
            if (j == 0 || i == 0)
                coef = 1;
            else
                coef = coef * (i - j + 1) / j;
            printf("%4d", coef);
        }
        printf("\n");
    }
    return 0;
}
```

Floyd's Triangle.

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

C Program

```
#include <stdio.h>  
int main() {  
    int rows, i, j, number = 1;  
    printf("Enter the number of rows: ");  
    scanf("%d", &rows);  
    for (i = 1; i <= rows; i++) {  
        for (j = 1; j <= i; ++j) {  
            printf("%d ", number);  
            ++number;  
        }  
        printf("\n");  
    }  
    return 0;  
}
```

The Infinite Loop

A loop becomes an infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the 'for' loop are required, you can make an endless loop by leaving the conditional expression empty.

```
#include <stdio.h>  
  
int main () {  
  
    for( ; ; ) {  
        printf("This loop will run forever.\n");  
    }  
}
```

```
    return 0;  
}
```

while loop

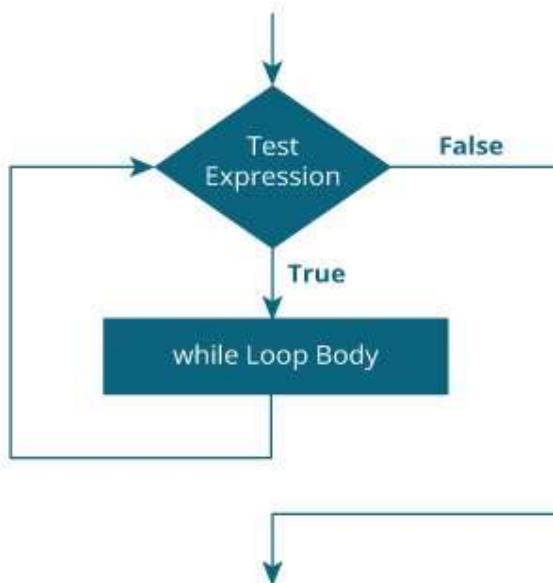
The syntax of the `while` loop is:

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

How while loop works?

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

Flowchart of while loop



Working of while loop

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

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

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

    return 0;
}
```

Output

```
1
2
3
4
5
```

Here, we have initialized `i` to 1.

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

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

```
{  
    int num=1;      //initializing the variable  
    while(num<=10)      //while loop with condition  
    {  
        printf("%d\n",num);  
        num++;          //incrementing operation  
    }  
    return 0;  
}
```

Output:

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

do...while loop

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

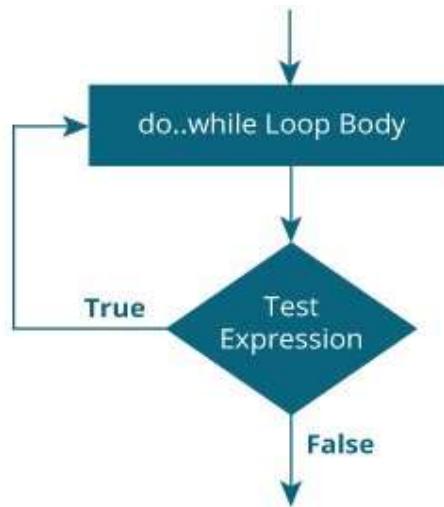
The syntax of the `do...while` loop is:

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

How do...while loop works?

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

Flowchart of do...while Loop



Working of do...while loop

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

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

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

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

    return 0;
}
```

Output

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

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

```
#include <stdio.h>
int main()
{
    int j=0;
    do
    {
        printf("Value of variable j is: %d\n", j);
        j++;
    }while (j<=3);
    return 0;
}
```

Output:

```
Value of variable j is: 0
Value of variable j is: 1
Value of variable j is: 2
Value of variable j is: 3
```

Below is a do-while loop in C example to print a table of number 2:

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

```
int main()
{
    int num=1;      //initializing the variable
    do      //do-while loop
    {
        printf("%d\n",2*num);
        num++;           //incrementing operation
    }while(num<=10);
    return 0;
}
```

Output:

```
2
4
6
8
10
12
14
16
18
20
```

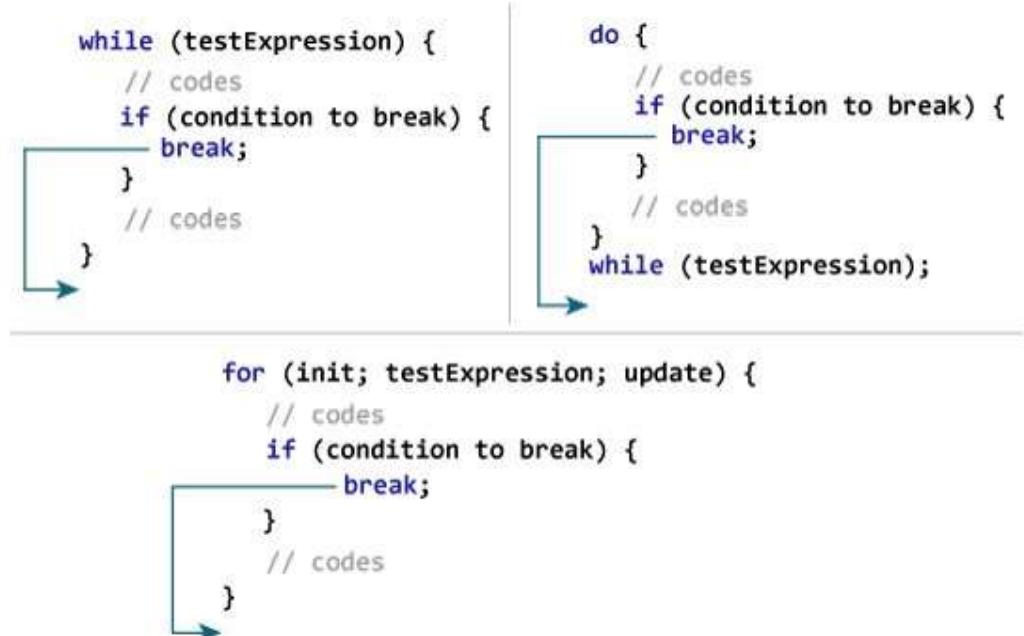
C break

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

```
break;
```

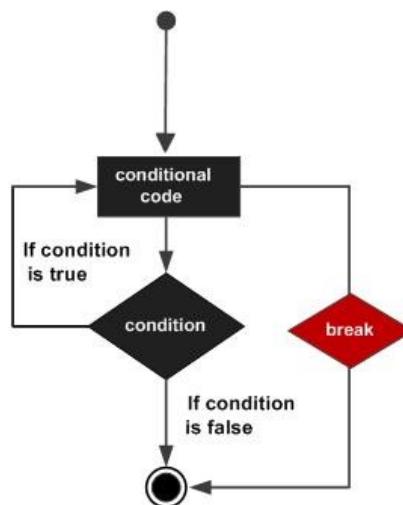
The break statement is almost always used with if...else statement inside the loop.

How break statement works?



Working of break in C

Flowchart:



Example 1: break statement

```

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

#include <stdio.h>

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

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

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

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

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

    return 0;
}

```

Output

```

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

```

Continue Statement:

The **continue** statement in C programming works somewhat like the **break** statement. Instead of forcing termination, it forces the next iteration of the loop to take place, skipping any code in between.

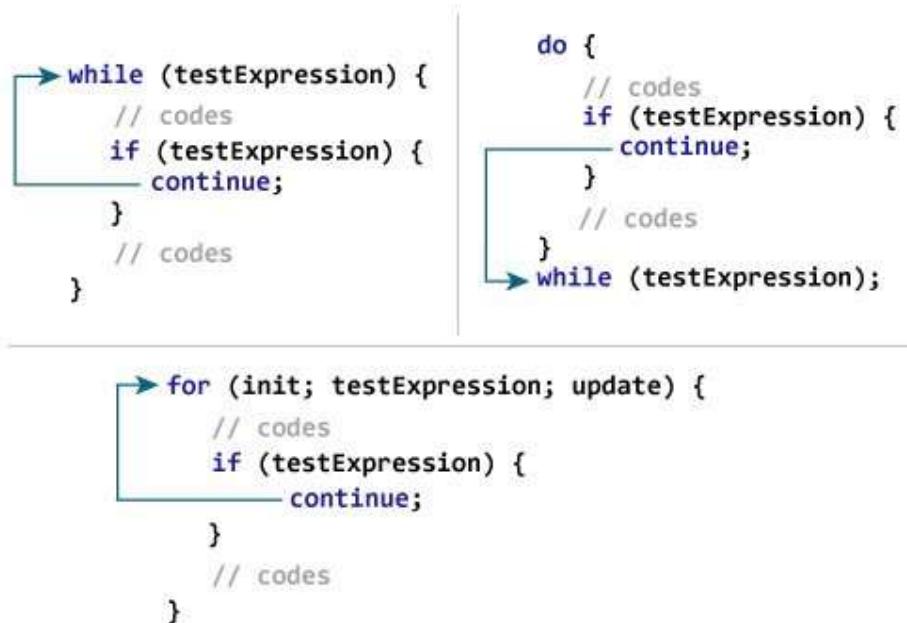
For the for loop, continue statement causes the conditional test and increment portions of the loop to execute. For the while and do...while loops, continue statement causes the program control to pass to the conditional tests.

Syntax

The syntax for a **continue** statement in C is as follows –

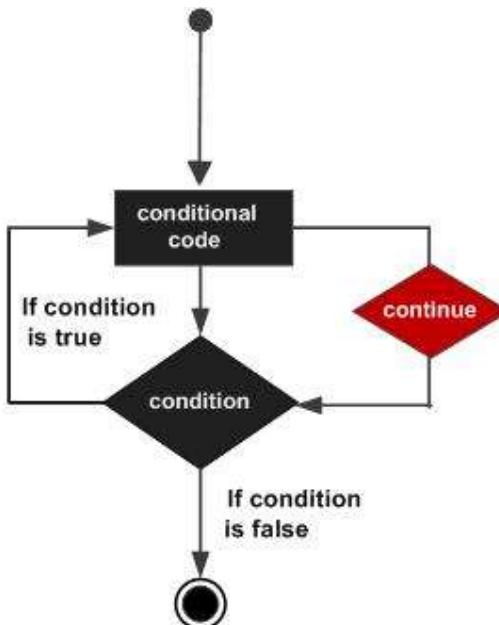
```
continue;
```

How continue statement works?



Working of Continue in C

Flowchart:



Example1:

```
//program demo using continue in for loop statement
```

```
#include <stdio.h>
int main()
{
    int i=0;
    for(i=0;i<10;i++)
    {
        if (i>=5 && i<=7)
        {
            continue;
        }
        printf("%d ", i);
    }

    return 0;
}
```

Output:

```
0 1 2 3 4 8 9
```

Example 2:

```
//program demo using continue in while loop statement
```

```
#include <stdio.h>
int main()
{
    int i=0;
    while(i<20)
    {
        if (i>=5 && i<=7)
        {
            i++;
            continue;
        }
    }
}
```

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

return 0;
}
```

Output:

```
0 1 2 3 4 8 9 10 11 12 13 14 15 16 17 18 19
```

Example 3:

```
//program demo using continue in do while statement
```

```
#include <stdio.h>
int main()
{
    int i=0;
    do
    {
        if (i>=5 && i<=7)
        {
            i++;
            continue;
        }
        printf("%d ", i);
        i++;
    } while(i<10);

    return 0;
}
```

Output:

0 1 2 3 4 8 9

```
#include <stdio.h>

int main () {

    /* local variable definition */
    int a = 10;

    /* do loop execution */
    do {

        if( a == 15) {
            /* skip the iteration */
            a = a + 1;
            continue;
        }

        printf("value of a: %d\n", a);
        a++;
    } while( a < 20 );

    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

```
// Program to calculate the sum of numbers (10 numbers max)
```

```
// If the user enters a negative number, it's not added to the result

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

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

        if (number < 0.0) {
            continue;
        }

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

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

    return 0;
}
```

Output

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

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

C Program using continue statement to find sum of all even numbers between 1 to N

```
#include <stdio.h>
int main(){
/*
 * This program calculate sum of even numbers
 */
int N, counter, sum=0;
printf("Enter a positive number\n");
scanf("%d", &N);

for(counter=1; counter <= N; counter++){
/*
 * Using continue statement to skip odd numbers
 */
if(counter%2 == 1){
    continue;
}
sum+= counter;
}
printf("Sum of all even numbers between 1 to %d = %d", N, sum);

return(0);
}
```

Above program find sum of all even numbers between 1 to N, using for loop. If counter is odd number, then continue statement will skip the sum statement.

Output

```
Enter a positive number
6
Sum of all even numbers between 1 to 6 = 12
```

C Functions

A function is a block of code that performs a specific task.

Suppose, you need to create a program to create a circle and color it. You can create two functions to solve this problem:

- create a circle function
- create a color function

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

Types of function

There are two types of function in C programming:

1. Standard library functions
2. User-defined functions

Standard library functions

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

These functions are defined in header files. For example,

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

Advantages of Using C library functions

1. They work

One of the most important reasons you should use library functions is simply because they work. These functions have gone through multiple rigorous testing and are easy to use.

2. The functions are optimized for performance

Since, the functions are "standard library" functions, a dedicated group of developers constantly make them better. In the process, they are able to create the most efficient code optimized for maximum performance.

3. It saves considerable development time

Since the general functions like printing to a screen, calculating the square root, and many more are already written. You shouldn't worry about creating them once again.

4. The functions are portable

With ever-changing real-world needs, your application is expected to work every time, everywhere. And, these library functions help you in that they do the same thing on every computer.

Square root using sqrt() function

Suppose, you want to find the square root of a number.

To compute the square root of a number, you can use the `sqrt()` library function. The function is defined in the `math.h` header file.

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

int main()
{
    float num, root;
    printf("Enter a number: ");
    scanf("%f", &num);

    // Computes the square root of num and stores in root.
    root = sqrt(num);

    printf("Square root of %.2f = %.2f", num, root);
    return 0;
}
```

```
}
```

When you run the program, the output will be:

```
Enter a number: 12
Square root of 12.00 = 3.46
```

User-defined function

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

C allows you to define functions according to your need. These functions are known as user-defined functions. For example:

Suppose, you need to create a circle and color it depending upon the radius and color. You can create two functions to solve this problem:

- `createCircle()` function
- `color()` function

How user-defined function works?

```
#include <stdio.h>

void functionName()

{

    . . .

    . . .

}

int main()
```

```
{  
    ...  
    ...  
    functionName();  
    ...  
    ...  
}
```

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

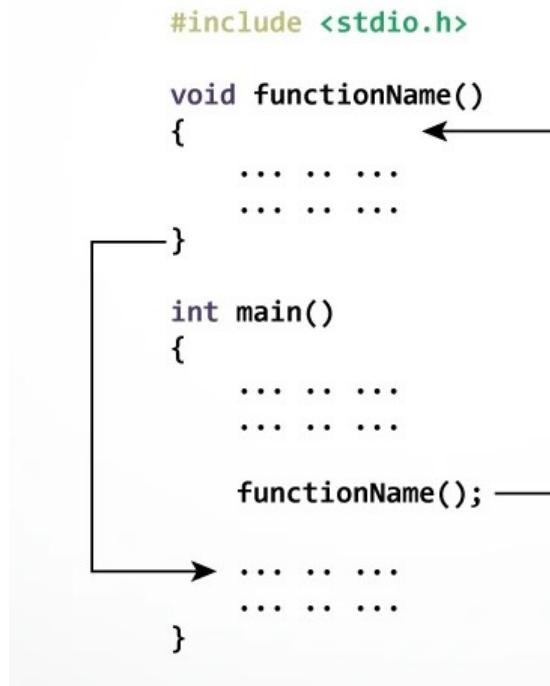
When the compiler encounters `functionName();`, control of the program jumps to

```
void functionName()
```

And, the compiler starts executing the codes inside `functionName()`.

The control of the program jumps back to the `main()` function once code inside the function definition is executed.

How function works in C programming?



Advantages of user-defined function

1. The program will be easier to understand, maintain and debug.
2. Reusable codes that can be used in other programs
3. A large program can be divided into smaller modules. Hence, a large project can be divided among many programmers.

Examples:

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

}

Output

Hello C Programming

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

Output

Going to calculate the sum of two numbers:

Enter two numbers 10

24

The sum is 34

Here is an example to add two integers. To perform this task, we have created an user-defined `addNumbers()`.

```
#include <stdio.h>
int addNumbers(int a, int b);      // function prototype

int main()
{
    int n1,n2,sum;
```

```

printf("Enters two numbers: ");
scanf("%d %d",&n1,&n2);

sum = addNumbers(n1, n2);      // function call
printf("sum = %d",sum);

return 0;
}

int addNumbers(int a, int b)    // function definition
{
    int result;
    result = a+b;
    return result;
}

```

Function prototype

A function prototype is simply the declaration of a function that specifies function's name, parameters and return type. It doesn't contain function body.

A function prototype gives information to the compiler that the function may later be used in the program.

Syntax of function prototype

```
returnType functionName(type1 argument1, type2 argument2, ...);
```

In the above example, `int addNumbers(int a, int b);` is the function prototype which provides the following information to the compiler:

1. name of the function is `addNumbers()`
2. return type of the function is `int`
3. two arguments of type `int` are passed to the function

The function prototype is not needed if the user-defined function is defined before the `main()` function.

Calling a function

Control of the program is transferred to the user-defined function by calling it.

Syntax of function call

```
functionName(argument1, argument2, ...);
```

In the above example, the function call is made using `addNumbers(n1, n2);` statement inside the `main()` function.

Function definition

Function definition contains the block of code to perform a specific task. In our example, adding two numbers and returning it.

Syntax of function definition

```
returnType functionName(type1 argument1, type2 argument2, ...)  
{  
    //body of the function  
}
```

When a function is called, the control of the program is transferred to the function definition. And, the compiler starts executing the codes inside the body of a function.

Passing arguments to a function

In programming, argument refers to the variable passed to the function. In the above example, two variables `n1` and `n2` are passed during the function call.

The parameters `a` and `b` accepts the passed arguments in the function definition. These arguments are called formal parameters of the function.

The type of arguments passed to a function and the formal parameters must match, otherwise, the compiler will throw an error.

How to pass arguments to a function?

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ...
    ...
    sum = addNumbers(n1, n2);
    ...
}

int addNumbers(int a, int b)
{
    ...
    ...
}
```

Passing Argument to Function

The type of arguments passed to a function and the formal parameters must match, otherwise, the compiler will throw an error.

If `n1` is of char type, `a` also should be of char type. If `n2` is of float type, variable `b` also should be of float type.

A function can also be called without passing an argument.

Return Statement

The return statement terminates the execution of a function and returns a value to the calling function. The program control is transferred to the calling function after the return statement.

In the above example, the value of the `result` variable is returned to the main function. The `sum` variable in the `main()` function is assigned this value.

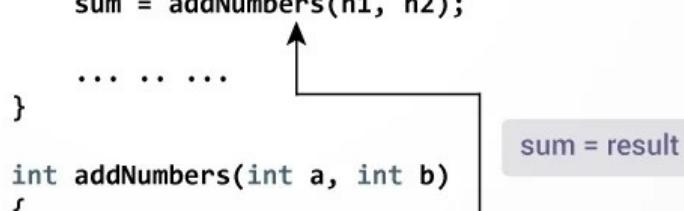
Return statement of a Function

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... ...
    sum = addNumbers(n1, n2);
    ...
}

int addNumbers(int a, int b)
{
    ...
    return result;
}
```



Return Statement of Function

Syntax of return statement

```
return (expression);
```

For example,

```
return a;  
return (a+b);
```

The type of value returned from the function and the return type specified in the function prototype and function definition must match.

Types of User-defined Functions in C Programming

Example 1: No Argument Passed and No Return Value

```

//program demo using user defined functions
//No Argument Passed and No Return Value
#include<stdio.h>
void sum(); //function prototype
int main()
{
    printf("\nGoing to calculate the sum of two numbers:");
    sum(); // Calling a function
    return 0;
}

void sum() // function definition
{
    int a,b;
    printf("\nEnter two numbers");
    scanf("%d %d",&a,&b);
    printf("The sum is %d",a+b);
}

```

```

#include <stdio.h>

void checkPrimeNumber();

int main() {
    checkPrimeNumber(); // argument is not passed
    return 0;
}

// return type is void meaning doesn't return any value
void checkPrimeNumber() {
    int n, i, flag = 0;

```

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

// 0 and 1 are not prime numbers
if (n == 0 || n == 1)
    flag = 1;

for(i = 2; i <= n/2; ++i) {
    if(n%i == 0) {
        flag = 1;
        break;
    }
}

if (flag == 1)
    printf("%d is not a prime number.", n);
else
    printf("%d is a prime number.", n);
}
```

The `checkPrimeNumber()` function takes input from the user, checks whether it is a prime number or not, and displays it on the screen.

The empty parentheses in `checkPrimeNumber();` inside the `main()` function indicates that no argument is passed to the function.

The return type of the function is `void`. Hence, no value is returned from the function.

Note: A prime number is a number that can only be divided by itself and 1 without remainders.

A prime number is a whole number greater than 1 with only two factors – themselves and 1.

A prime number cannot be divided by any other positive integers without leaving a remainder, decimal or fraction.

2 & 3 are Prime No's:

$2/2 = 1$ $2/1 = 2$

$3/3=1$ $3/1 = 3$

4 & 6 are not a Prime No's.

$4/2=2$

$6/3=2$

$6/2=3$

An example of a prime number is 13. Its only divisors are 1 and 13. Dividing a prime number by another natural number results in numbers leftover e.g. $13 \div 6 = 2$ remainder 1.

15 is not an example of a prime number because it can be divided by 5 and 3 as well as by itself and 1.

Why is 1 not a prime number?

1 is not a prime number because it has only one factor, namely 1. Prime numbers need to have exactly two factors.

Why is 2 a prime number?

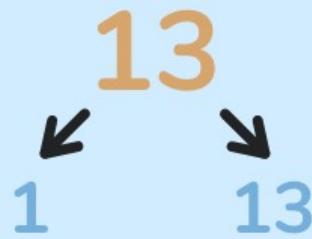
2 is a prime number because its only factors are 1 and itself.

Is 51 a prime number?

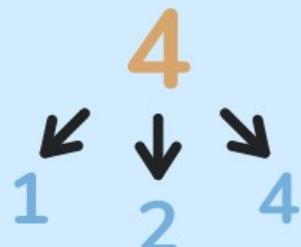
51 is not a prime number because it has 3 and 17 as divisors, as well as itself and 1.

In other words, 51 has four factors.

How do prime numbers work?



13 has **only two factors** - itself and **1**. So it is a prime number.



4 has **three factors** - itself, **1 and 2**. So it is NOT a prime number.

1. There are 8 prime numbers under 20: 2, 3, 5, 7, 11, 13, 17 and 19.
2. The first 10 prime numbers are 2, 3, 5, 7, 11, 13, 17, 19, 23, 29.
3. There are 25 prime numbers between 1 and 100.

List of prime numbers to 100

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97

Notice that this list of prime numbers contains only odd numbers, apart from 2.

Example 2: No Arguments Passed But Returns a Value

```
//program demo using user defined functions  
//No Arguments Passed But Returns a Value
```

```
#include<stdio.h>  
  
int sum(); //function prototype  
  
int main()  
{  
    int result;
```

```
printf("\nGoing to calculate the sum of two numbers:");

result = sum(); // Calling a function

printf("The sum is %d",result);

return 0;

}
```

```
int sum() // function definition

{
    int a,b,add;

    printf("\nEnter two numbers");

    scanf("%d %d",&a,&b);

    add = a + b;

    return add;
}
```

```
#include <stdio.h>
int getInteger();

int main() {

    int n, i, flag = 0;

    // no argument is passed
    n = getInteger();

    // 0 and 1 are not prime numbers
    if(n == 0 || n == 1)
        flag = 1;

    for(i = 2; i <= n/2; ++i) {
        if(n%i == 0){
            flag = 1;
            break;
        }
    }
}
```

```

if (flag == 1)
    printf("%d is not a prime number.", n);
else
    printf("%d is a prime number.", n);

return 0;
}

// returns integer entered by the user
int getInteger() {
    int n;

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

    return n;
}

```

The empty parentheses in the `n = getInteger();` statement indicates that no argument is passed to the function. And, the value returned from the function is assigned to `n`.

Here, the `getInteger()` function takes input from the user and returns it. The code to check whether a number is prime or not is inside the `main()` function.

Example 3: Argument Passed But No Return Value

```

//program demo using user defined functions
//with Arguments Passed and not Returns a Value
#include<stdio.h>
void sum(int a,int b); //function prototype
int main()
{
    int a,b;
    printf("\nGoing to calculate the sum of two numbers:");

    printf("\nEnter two numbers");
    scanf("%d %d",&a,&b);

    sum(a,b); // Calling a function
}

```

```

    return 0;
}

void sum(int a,int b) // function definition
{
    int add;
    add = a + b;
    printf("The sum is %d", add);
}

```

```

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

int main() {

    int n;

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

    // n is passed to the function
    checkPrimeAndDisplay(n);

    return 0;
}

// return type is void meaning doesn't return any value
void checkPrimeAndDisplay(int n) {
    int i, flag = 0;

    // 0 and 1 are not prime numbers
    if (n == 0 || n == 1)
        flag = 1;

```

```

for(i = 2; i <= n/2; ++i) {
    if(n%i == 0){
        flag = 1;
        break;
    }
}

if(flag == 1)
    printf("%d is not a prime number.",n);
else
    printf("%d is a prime number.", n);
}

```

The integer value entered by the user is passed to the `checkPrimeAndDisplay()` function.

Here, the `checkPrimeAndDisplay()` function checks whether the argument passed is a prime number or not and displays the appropriate message.

Example 4: Argument Passed and Returns a Value

```

//program demo using user defined functions
//with Arguments Passed and also Returns a Value
#include<stdio.h>
int sum(int a,int b); //function prototype
int main()
{
    int a, b, result;
    printf("\nGoing to calculate the sum of two numbers:");

    printf("\nEnter two numbers");
    scanf("%d %d",&a,&b);

```

```
result = sum(a,b); // Calling a function
printf("The sum is %d", result);
return 0;
}
```

```
int sum(int a,int b) // function definition
{
    int add;
    add = a + b;
    return add;
}
```

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

int main() {

    int n, flag;

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

    // n is passed to the checkPrimeNumber() function
    // the returned value is assigned to the flag variable
    flag = checkPrimeNumber(n);

    if(flag == 1)
        printf("%d is not a prime number",n);
    else
        printf("%d is a prime number",n);

    return 0;
}
```

```
// int is returned from the function
int checkPrimeNumber(int n) {

    // 0 and 1 are not prime numbers
    if (n == 0 || n == 1)
        return 1;

    int i;

    for(i=2; i <= n/2; ++i) {
        if(n%i == 0)
            return 1;
    }

    return 0;
}
```

The input from the user is passed to the `checkPrimeNumber()` function.

The `checkPrimeNumber()` function checks whether the passed argument is prime or not.

If the passed argument is a prime number, the function returns **0**. If the passed argument is a non-prime number, the function returns **1**. The return value is assigned to the `flag` variable.

Depending on whether `flag` is **0** or **1**, an appropriate message is printed from the `main()` function.

```
#include<stdio.h>
void average(int, int, int, int, int);
void main()
{
    int a,b,c,d,e;
    printf("\nGoing to calculate the average of five numbers:");
}
```

```

printf("\nEnter five numbers:");
scanf("%d %d %d %d %d",&a,&b,&c,&d,&e);
average(a,b,c,d,e);
}

void average(int a, int b, int c, int d, int e)
{
    float avg;
    avg = (a+b+c+d+e)/5;
    printf("The average of given five numbers : %f",avg);
}

```

Output

Going to calculate the average of five numbers:

Enter five numbers:10

20

30

40

50

The average of given five numbers : 30.000000

Program to check whether a number is even or odd

```

#include<stdio.h>
int even_odd(int);
void main()
{
    int n,flag=0;
    printf("\nGoing to check whether a number is even or odd");
    printf("\nEnter the number: ");
    scanf("%d",&n);
    flag = even_odd(n);
    if(flag == 0)
    {
        printf("\nThe number is odd");
    }
}

```

```

}

else

{
    printf("\nThe number is even");
}
}

int even_odd(int n)
{
    if(n%2 == 0)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

```

Output

Going to check whether a number is even or odd

Enter the number: 100

The number is even

Which approach is better?

Well, it depends on the problem you are trying to solve. In this case, passing an argument and returning a value from the function (example 4) is better.

A function should perform a specific task. The `checkPrimeNumber()` function doesn't take input from the user nor it displays the appropriate message. It only checks whether a number is prime or not.

Summary:

Example 1: No Argument Passed and No Return Value

```
void sum(); //function prototype
```

Example 2: No Arguments Passed But Returns a Value

```
int sum();
```

Example 3: Argument Passed But No Return Value

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

Example 4: Argument Passed and Returns a Value

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

C Recursion

A function that calls itself is known as a recursive function. And, this technique is known as recursion.

How recursion works?

```
void recurse()
```

```
{
```

```
.... . . .
```

```
recurse();
```

```
.... . . .
```

```
}
```

```
int main()
```

```
{
```

```
.... . . .
```

```
recurse();
```

```
.... . . .
```

```
}
```

How does recursion work?

```
void recurse()
{
    ... ...
    recurse(); ————— recursive call
    ...
}

int main()
{
    ... ...
    recurse(); —————
    ...
}
```

Working of Recursion

The recursion continues until some condition is met to prevent it.

To prevent infinite recursion, if...else statement (or similar approach) can be used where one branch makes the recursive call, and other doesn't.

Example: Sum of Natural Numbers Using Recursion

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

int main() {
    int number, result;

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

    result = sum(number);

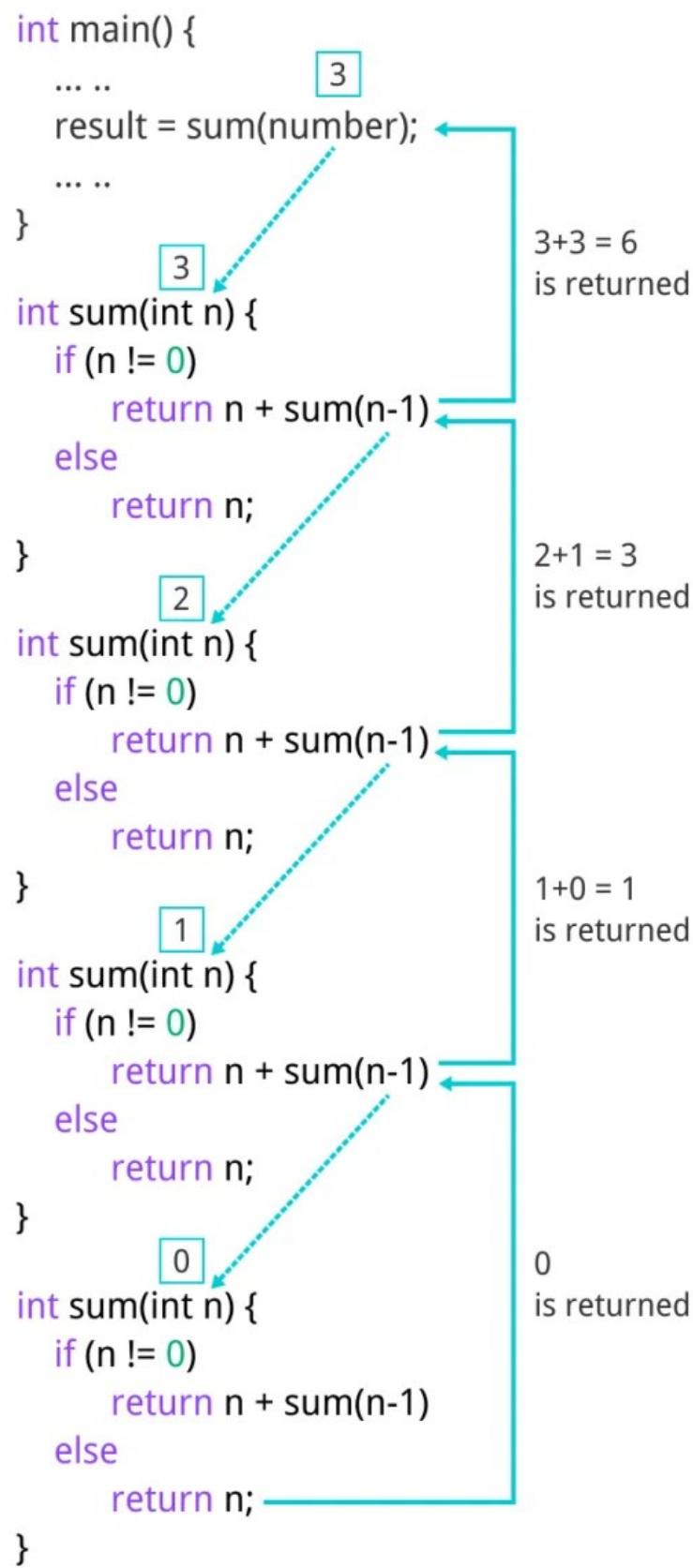
    printf("sum = %d", result);
    return 0;
}
```

```
int sum(int n) {  
    if (n != 0)  
        // sum() function calls itself  
        return n + sum(n-1);  
    else  
        return n;  
}
```

Initially, the `sum()` is called from the `main()` function with `number` passed as an argument.

Suppose, the value of `n` inside `sum()` is 3 initially. During the next function call, 2 is passed to the `sum()` function. This process continues until `n` is equal to 0.

When `n` is equal to 0, the `if` condition fails and the `else` part is executed returning the sum of integers ultimately to the `main()` function.



Advantages and Disadvantages of Recursion

Recursion makes program elegant. However, if performance is vital, use loops instead as recursion is usually much slower.

That being said, recursion is an important concept. It is frequently used in data structure and algorithms. For example, it is common to use recursion in problems such as tree traversal.

The factorial of a positive number n is given by:

factorial of n ($n!$) = $1 * 2 * 3 * 4 * \dots * n$

The factorial of a negative number doesn't exist. And the factorial of 0 is 1 .

Factorial of a Number Using Recursion

```
#include<stdio.h>
long int multiplyNumbers(int n);
int main() {
    int n;
    printf("Enter a positive integer: ");
    scanf("%d",&n);
    printf("Factorial of %d = %ld", n, multiplyNumbers(n));
    return 0;
}

long int multiplyNumbers(int n) {
    if (n>=1)
        return n*multiplyNumbers(n-1);
    else
        return 1;
}
```

Output

```
Enter a positive integer: 6
```

```
Factorial of 6 = 720
```

Suppose the user entered 6.

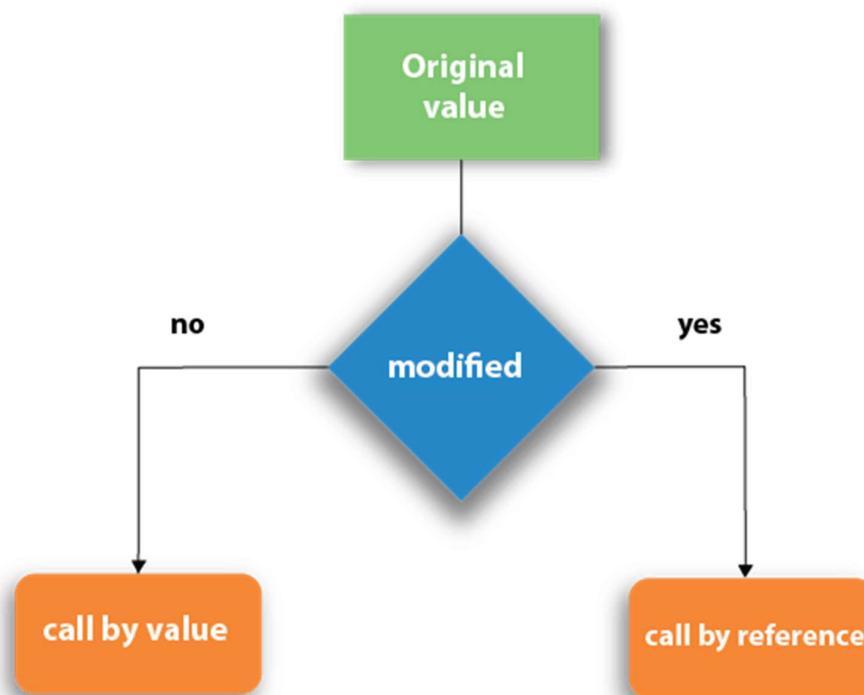
Initially, `multiplyNumbers()` is called from `main()` with 6 passed as an argument.

Then, 5 is passed to `multiplyNumbers()` from the same function (recursive call). In each recursive call, the value of argument `n` is decreased by 1.

When the value of `n` is less than 1, there is no recursive call and the factorial is returned ultimately to the `main()` function.

Call by value and Call by reference in C

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



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

Call by value

1. In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.
2. In call by value method, we can not modify the value of the actual parameter by the formal parameter.
3. In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
4. The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

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

```
#include<stdio.h>

void change(int num) {
    printf("Before adding value inside function num=%d \n",num);
    num=num+100;
    printf("After adding value inside function num=%d \n", num);
}

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

Output

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

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

```
#include <stdio.h>

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

Output

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

Call by reference

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

Example:

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

Output

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

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

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

```

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

```

Output

```

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

```

Difference between call by value and call by reference in c

No.	Call by value	Call by reference
1	A copy of the value is passed into the function	An address of value is passed into the function

2	Changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters.	Changes made inside the function validate outside of the function also. The values of the actual parameters do change by changing the formal parameters.
3	Actual and formal arguments are created at the different memory location	Actual and formal arguments are created at the same memory location

C - Scope Rules

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed. There are three places where variables can be declared in C programming language –

- Inside a function or a block which is called **local** variables.
- Outside of all functions which is called **global** variables.
- In the definition of function parameters which are called **formal** parameters.

Local Variables

Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. The following example shows how local variables are used. Here all the variables a, b, and c are local to main() function.

```
#include <stdio.h>
int main () {
    /* local variable declaration */
    int a, b;
    int c;
```

```

/* actual initialization */

a = 10;
b = 20;
c = a + b;

printf ("value of a = %d, b = %d and c = %d\n", a, b, c);

return 0;
}

```

```

#include <stdio.h>

int main(void) {

    for (int i = 0; i < 5; ++i) {
        printf("C programming");
    }

    // Error: i is not declared at this point
    printf("%d", i);
    return 0;
}

```

When you run the above program, you will get an error `undeclared identifier i`. It's because `i` is declared inside the `for` loop block. Outside of the block, it's undeclared. Let's take another example.

```

int main() {
    int n1; // n1 is a local variable to main()
}

```

```
void func() {  
    int n2; // n2 is a local variable to func()  
}
```

In the above example, `n1` is local to `main()` and `n2` is local to `func()`.

This means you cannot access the `n1` variable inside `func()` as it only exists inside `main()`. Similarly, you cannot access the `n2` variable inside `main()` as it only exists inside `func()`.

Global Variables

Global variables are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. The value of the global variable can be changed by any function. The following program show how global variables are used in a program.

```
#include <stdio.h>  
  
/* global variable declaration */  
int g;  
  
int main () {  
  
    /* local variable declaration */  
    int a, b;  
  
    /* actual initialization */  
    a = 10;  
    b = 20;  
    g = a + b;
```

```
printf ("value of a = %d, b = %d and g = %d\n", a, b, g);

return 0;
}
```

A program can have same name for local and global variables but the value of local variable inside a function will take preference. Here is an example –

```
#include <stdio.h>

/* global variable declaration */
int g = 20;

int main () {

/* local variable declaration */
int g = 10;

printf ("value of g = %d\n", g);

return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
value of g = 10
```

Example: Global Variable

```
#include <stdio.h>
void display();

int n = 5; // global variable

int main()
{
```

```

    ++n;
    display();
    return 0;
}

void display()
{
    ++n;
    printf("n = %d", n);
}

```

Output

```
n = 7
```

Suppose, a global variable is declared in file1. If you try to use that variable in a different file file2, the compiler will complain. To solve this problem, keyword `extern` is used in file2 to indicate that the external variable is declared in another file.

Formal Parameters

Formal parameters, are treated as local variables with-in a function and they take precedence over global variables. Following is an example –

```

#include <stdio.h>

/* global variable declaration */
int a = 20;

int main () {

/* local variable declaration in main function */
int a = 10;
int b = 20;
int c = 0;

printf ("value of a in main() = %d\n", a);
c = sum( a, b);
printf ("value of c in main() = %d\n", c);

```

```
    return 0;
}

/* function to add two integers */
int sum(int a, int b) {

    printf ("value of a in sum() = %d\n", a);
    printf ("value of b in sum() = %d\n", b);

    return a + b;
}
```

When the above code is compiled and executed, it produces the following result –

```
value of a in main() = 10
value of a in sum() = 10
value of b in sum() = 20
value of c in main() = 30
```

C Storage Classes

Every variable in C programming has two properties: type and storage class.

Type refers to the data type of a variable. And, storage class determines the scope, visibility and lifetime of a variable.

There are 4 types of storage class:

1. automatic
2. external
3. static
4. register

Automatic

The auto storage class is the default storage class for all local variables.

```
{  
    int mount;  
    auto int month;  
}
```

The example above defines two variables within the same storage class. 'auto' can only be used within functions, i.e., local variables.

1. Automatic variables are allocated memory automatically at runtime.
2. The visibility of the automatic variables is limited to the block in which they are defined.
 - The scope of the automatic variables is limited to the block in which they are defined.
3. The automatic variables are initialized to garbage by default.
4. The memory assigned to automatic variables gets freed upon exiting from the block.
5. The keyword used for defining automatic variables is auto.
6. Every local variable is automatic in C by default.

Example 1

```
#include <stdio.h>  
  
int main()  
{  
    int x=10;      //local variable (also automatic)  
    auto int y=20; //automatic variable  
    printf("x = %d\n",x);  
    printf("y = %d\n",y);  
    return 0;  
}
```

Register Variable

The `register` keyword is used to declare register variables. Register variables were supposed to be faster than local variables.

However, modern compilers are very good at code optimization, and there is a rare chance that using register variables will make your program faster.

The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{  
    register int miles;  
}
```

Example 1

```
#include <stdio.h>  
  
int main()  
{  
    register int a; // variable a is allocated memory in the CPU register (not in RAM).  
    a=100;  
    printf("a = %d\n",a);  
}
```

Output:

```
a = 100
```

Example 2

```
#include <stdio.h>  
int main()  
{  
    register int a = 0;  
    printf("%u",&a); // This will give a compile time error since we can not access the  
                    // address of a register variable.
```

```
}
```

Output:

```
main.c:5:5: error: address of register variable ?a? requested
printf("%u",&a);
^~~~~~
```

Static Variable

1. The variables defined as static specifier can hold their value between the multiple function calls.
2. Static local variables are visible only to the function or the block in which they are defined.
3. A same static variable can be declared many times but can be assigned at only one time.
4. Default initial value of the static integral variable is 0 otherwise null.
5. The visibility of the static global variable is limited to the file in which it has declared.
6. The keyword used to define static variable is static.

A static variable is declared by using the `static` keyword. For example;

```
static int i;
```

The value of a static variable persists until the end of the program.

Example 2: Static Variable

```
#include <stdio.h>
void display();

int main()
{
    display();
    display();
}
```

```
void display()
{
    static int c = 1;
    c += 5;
    printf("%d ",c);
}
```

Output

```
6 11
```

During the first function call, the value of `c` is initialized to 1. Its value is increased by 5. Now, the value of `c` is 6, which is printed on the screen.

During the second function call, `c` is not initialized to 1 again. It's because `c` is a static variable. The value `c` is increased by 5. Now, its value will be 11, which is printed on the screen.

```
#include<stdio.h>
void sum()
{
    static int a = 10;
    static int b = 24;
    printf("%d %d \n",a,b);
    a++;
    b++;
}
void main()
{
    int i;
    for(i = 0; i< 3; i++)
    {
        sum(); // The static variables holds their value between multiple function calls.
    }
}
```

Output:

```
10 24
11 25
12 26
```

```

#include <stdio.h>

/* function declaration */
void func(void);

static int count = 5; /* global variable */

main() {

    while(count--) {
        func();
    }

    return 0;
}

/* function definition */
void func( void ) {

    static int i = 5; /* local static variable */
    i++;

    printf("i is %d and count is %d\n", i, count);
}

```

When the above code is compiled and executed, it produces the following result –

```

i is 6 and count is 4
i is 7 and count is 3
i is 8 and count is 2
i is 9 and count is 1
i is 10 and count is 0

```

Working Example:

```

//program demo

#include<stdio.h>

void function1();

int main()

{
    function1();
    function1();
    function1();
    return 0;
}

```

```
}

void function1()
{
    int x=11;//local variable
    static int y=10;//static variable
    x=x+1;
    y=y+1;
    printf("x = %d , y = %d\n",x,y);
}
```

Output:

```
x = 12 , y = 11
x = 12 , y = 12
x = 12 , y = 13
```

External

1. The external storage class is used to tell the compiler that the variable defined as extern is declared with an external linkage elsewhere in the program.
2. The variables declared as extern are not allocated any memory. It is only declaration and intended to specify that the variable is declared elsewhere in the program.
3. The default initial value of external integral type is 0 otherwise null.
4. We can only initialize the extern variable globally, i.e., we can not initialize the external variable within any block or method.
5. An external variable can be declared many times but can be initialized at only once.
6. If a variable is declared as external then the compiler searches for that variable to be initialized somewhere in the program which may be extern or static. If it is not, then the compiler will show an error.

Example 1

```
#include <stdio.h>
int main()
{
extern int a;
printf("%d",a);
}
```

Output

```
main.c:(.text+0x6): undefined reference to `a'
collect2: error: ld returned 1 exit status
```

Example 2

```
#include <stdio.h>
int a;
int main()
{
extern int a; // variable a is defined globally, the memory will not be allocated to
a
printf("%d",a);
}
```

Output

```
0
```

Example 3

```
#include <stdio.h>
int a;
int main()
{
extern int a = 0; // this will show a compiler error since we can not use extern and
d initializer at same time
printf("%d",a);
}
```

Output

```
compile time error
main.c: In function ?main?:
```

```
main.c:5:16: error: ?a? has both ?extern? and initializer  
extern int a = 0;
```

Example 4

```
#include <stdio.h>  
int main()  
{  
extern int a; // Compiler will search here for a variable a defined and initialized so  
mewhere in the program or not.  
printf("%d",a);  
}  
int a = 20;
```

Output

```
20
```

Example 5

```
extern int a;  
int a = 10;  
<#include <stdio.h>  
int main()  
{  
printf("%d",a);  
}  
int a = 20; // compiler will show an error at this line
```

Output

Compile time error

Example:

First File: ext1.c

```
#include <stdio.h>  
  
int count ;  
extern void write_extern();
```

```
main() {
    count = 5;
    write_extern();
}
```

Second File: ext2.c

```
#include <stdio.h>

extern int count;

void write_extern(void) {
    printf("count is %d\n", count);
}
```

Here, `extern` is being used to declare `count` in the second file, where as it has its definition in the first file, `main.c`. Now, compile these two files as follows –

```
D:\> gcc ext1.c ext2.c
```

It will produce the executable program **a.exe**. When this program is executed, it produces the following result –

```
count is 5
```

Working Example:

myFile.h

```
extern int x=20;
```

m2.c

```
#include "myfile.h"
#include<stdio.h>
void printValue();
int main()
{
```

```
    printValue();
    return 0;
}

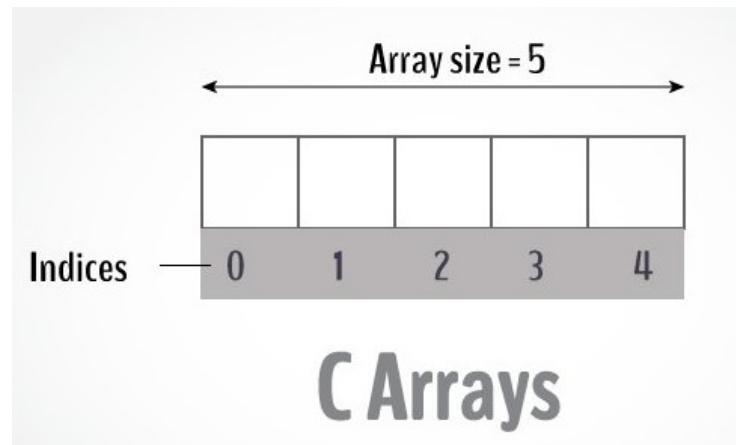
void printValue()
{
printf("Global variable: %d", x);
}
```

C ARRAYS

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

An array is defined as the collection of similar type of data items stored at contiguous memory locations. Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc. It also has the capability to store the collection of derived data types, such as pointers, structure, etc. The array is the simplest data structure where each data element can be randomly accessed by using its index number.

```
int data[5];
```



Advantage of C Array

- 1) Code Optimization:** Less code to access the data.
- 2) Ease of traversing:** By using the for loop, we can retrieve the elements of an array easily.
- 3) Ease of sorting:** To sort the elements of the array, we need a few lines of code only.
- 4) Random Access:** We can access any element randomly using the array.

Disadvantage of C Array

- 1) Fixed Size:** Whatever size, we define at the time of declaration of the array, we can't exceed the limit. So, it doesn't grow the size dynamically like LinkedList which we will learn later.

Declaration of C Array

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

For example,

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

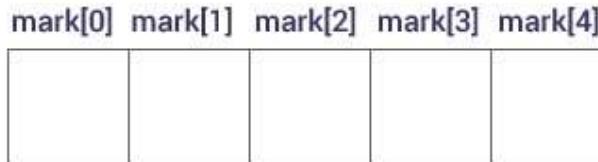
Here, we declared an array, `mark`, of floating-point type. And its size is 5. Meaning, it can hold 5 floating-point values.

It's important to note that the size and type of an array cannot be changed once it is declared.

Access Array Elements

You can access elements of an array by indices.

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



Declare an Array

Few keynotes:

- Arrays have 0 as the first index, not 1. In this example, `mark[0]` is the first element.
- If the size of an array is `n`, to access the last element, the `n-1` index is used. In this example, `mark[4]`
- Suppose the starting address of `mark[0]` is **2120d**. Then, the address of the `mark[1]` will be **2124d**. Similarly, the address of `mark[2]` will be **2128d** and so on.

This is because the size of a `float` is 4 bytes.

How to initialize an array?

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

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

You can also initialize an array like this.

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

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

mark[0]	mark[1]	mark[2]	mark[3]	mark[4]
19	10	8	17	9

Initialize an Array

Here,

mark[0] is equal to 19

mark[1] is equal to 10

mark[2] is equal to 8

mark[3] is equal to 17

mark[4] is equal to 9

Change Value of Array elements

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

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

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

Input and Output Array Elements

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

```
// take input and store it in the 3rd element  
scanf("%d", &mark[2]);  
  
// take input and store it in the ith element  
scanf("%d", &mark[i-1]);
```

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

```
// print the first element of the array  
printf("%d", mark[0]);  
  
// print the third element of the array  
printf("%d", mark[2]);  
  
// print ith element of the array  
printf("%d", mark[i-1]);
```

C array example

```
#include<stdio.h>  
  
int main()  
{  
    int i=0;  
    int marks[5];//declaration of array  
    marks[0]=80;//initialization of array  
    marks[1]=60;  
    marks[2]=70;  
    marks[3]=85;  
    marks[4]=75;  
    //traversal of array  
    for(i=0;i<5;i++){  
        printf("%d \n",marks[i]);  
    } //end of for loop  
  
return 0;
```

```
}
```

Output

```
80  
60  
70  
85  
75
```

C Array: Declaration with Initialization

We can initialize the c array at the time of declaration. Let's see the code.

```
int marks[5]={20,30,40,50,60};
```

In such case, there is **no requirement to define the size**. So it may also be written as the following code.

```
int marks[]={20,30,40,50,60};
```

```
#include<stdio.h>  
int main(){  
    int i=0;  
    int marks[5]={20,30,40,50,60}; //declaration and initialization of array  
    //traversal of array  
    for(i=0;i<5;i++){  
        printf("%d \n",marks[i]);  
    }  
    return 0;  
}
```

Output

```
20  
30  
40
```

50

60

C Array Example: Sorting an array

In the following program, we are using bubble sort method to sort the array in ascending order.

```
#include<stdio.h>
void main ()
{
    int i, j,temp;
    int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
    for(i = 0; i<10; i++)
    {
        for(j = i+1; j<10; j++)
        {
            if(a[j] > a[i])
            {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
    printf("Printing Sorted Element List ...\\n");
    for(i = 0; i<10; i++)
    {
        printf("%d\\n",a[i]);
    }
}
```

Program to print the largest and second largest element of the array.

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

```

{
    int arr[100],i,n,largest,sec_largest;
    printf("Enter the size of the array?");
    scanf("%d",&n);
    printf("Enter the elements of the array?");
    for(i = 0; i<n; i++)
    {
        scanf("%d",&arr[i]);
    }
    largest = arr[0];
    sec_largest = arr[1];
    for(i=0;i<n;i++)
    {
        if(arr[i]>largest)
        {
            sec_largest = largest;
            largest = arr[i];
        }
        else if (arr[i]>sec_largest && arr[i]!=largest)
        {
            sec_largest=arr[i];
        }
    }
    printf("largest = %d, second largest = %d",largest,sec_largest);
}

}

```

Example 1: Array Input/Output

```

// Program to take 5 values from the user and store them in an array
// Print the elements stored in the array
#include <stdio.h>

```

```

int main() {
    int values[5];
}

```

```

printf("Enter 5 integers: ");

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

printf("Displaying integers: ");

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

```

Output

```

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

```

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

Example 2: Calculate Average

```

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

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

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

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

```

```
for(i=0; i < n; ++i) {  
    printf("Enter number%d: ", i+1);  
    scanf("%d", &marks[i]);  
  
    // adding integers entered by the user to the sum variable  
    sum += marks[i];  
}  
  
average = sum / n;  
printf("Average = %d", average);  
  
return 0;  
}
```

Output

```
Enter n: 5  
Enter number1: 45  
Enter number2: 35  
Enter number3: 38  
Enter number4: 31  
Enter number5: 49  
Average = 39
```

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

Access elements out of its bound!

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

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

You can access the array elements from `testArray[0]` to `testArray[9]`.

Now let's say if you try to access `testArray[12]`. The element is not available. This may cause unexpected output (undefined behavior). Sometimes you might get an error and some other time your program may run correctly.

Hence, you should never access elements of an array outside of its bound.

C Multidimensional Arrays

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

The two-dimensional array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns. However, 2D arrays are created to implement a relational database lookalike data structure. It provides ease of holding the bulk of data at once which can be passed to any number of functions wherever required.

Declaration of two-dimensional Array in C

The syntax to declare the 2D array is given below.

```
data_type array_name[rows][columns];
```

For example,

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

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

	Column 1	Column 2	Column 3	Column 4
Row 1	<code>x[0][0]</code>	<code>x[0][1]</code>	<code>x[0][2]</code>	<code>x[0][3]</code>
Row 2	<code>x[1][0]</code>	<code>x[1][1]</code>	<code>x[1][2]</code>	<code>x[1][3]</code>
Row 3	<code>x[2][0]</code>	<code>x[2][1]</code>	<code>x[2][2]</code>	<code>x[2][3]</code>

Two dimensional Array

Two-dimensional array example in C

```
#include<stdio.h>

int main(){
    int i=0,j=0;
    int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};
}
```

```

//traversing 2D array
for(i=0;i<4;i++){
    for(j=0;j<3;j++){
        printf("arr[%d] [%d] = %d \n",i,j,arr[i][j]);
    } //end of j
} //end of i
return 0;
}

```

Output

```

arr[0][0] = 1
arr[0][1] = 2
arr[0][2] = 3
arr[1][0] = 2
arr[1][1] = 3
arr[1][2] = 4
arr[2][0] = 3
arr[2][1] = 4
arr[2][2] = 5
arr[3][0] = 4
arr[3][1] = 5
arr[3][2] = 6

```

C 2D array example: Storing elements in a matrix and printing it.

```

#include <stdio.h>

void main ()
{
    int arr[3][3],i,j;
    for (i=0;i<3;i++)
    {
        for (j=0;j<3;j++)
        {

```

```

        printf("Enter a[%d][%d]: ",i,j);
        scanf("%d",&arr[i][j]);
    }
}

printf("\n printing the elements ....\n");
for(i=0;i<3;i++)
{
    printf("\n");
    for (j=0;j<3;j++)
    {
        printf("%d\t",arr[i][j]);
    }
}

```

Output

```

Enter a[0][0]: 56
Enter a[0][1]: 10
Enter a[0][2]: 30
Enter a[1][0]: 34
Enter a[1][1]: 21
Enter a[1][2]: 34

```

```

Enter a[2][0]: 45
Enter a[2][1]: 56
Enter a[2][2]: 78

```

printing the elements

```

56    10    30
34    21    34
45    56    78

```

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

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

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

Initializing a multidimensional array

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

Initialization of a 2d array

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

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

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

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

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

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

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

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

Output

```
City 1, Day 1: 33
City 1, Day 2: 34
City 1, Day 3: 35
```

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

Displaying values:

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

Example 2: Sum of two matrices

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

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

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

```

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

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

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

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

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

```

Output

Enter elements of 1st matrix

Enter a11: 2;

Enter a12: 0.5;

Enter a21: -1.1;

Enter a22: 2;

Enter elements of 2nd matrix

Enter b11: 0.2;

Enter b12: 0;

Enter b21: 0.23;

Enter b22: 23;

Sum Of Matrix:

2.2 0.5

-0.9 25.0

Example:

```

//program demo to add two matrices
#include <stdio.h>
int main() {
    int r, c, a[10][10], b[10][10], sum[10][10], i, j;
    printf("Enter the number of rows (between 1 and 10): ");
    scanf("%d", &r);
    printf("Enter the number of columns (between 1 and 10): ");
    scanf("%d", &c);

    printf("\nEnter elements of 1st matrix:\n");
    for (i = 0; i < r; ++i)
        for (j = 0; j < c; ++j) {
            printf("Enter element a %d%d: ", i , j );
            scanf("%d", &a[i][j]);
        }

    printf("Enter elements of 2nd matrix:\n");
    for (i = 0; i < r; ++i)
        for (j = 0; j < c; ++j) {
            printf("Enter element b %d%d: ", i , j );
            scanf("%d", &b[i][j]);
        }

    // adding two matrices
    for (i = 0; i < r; ++i)
        for (j = 0; j < c; ++j) {
            sum[i][j] = a[i][j] + b[i][j];
        }

    // printing the result
    printf("\nSum of two matrices: \n");
    for (i = 0; i < r; ++i)
        for (j = 0; j < c; ++j) {
            printf("%d ", sum[i][j]);
        }
}

```

```

    if (j == c - 1) {
        printf("\n\n");
    }
}

return 0;
}

```

Example 3: Three-dimensional array

```

// C Program to store and print 12 values entered by the user

#include <stdio.h>
int main()
{
    int test[2][3][2];

    printf("Enter 12 values: \n");

    for (int i = 0; i < 2; ++i)
    {
        for (int j = 0; j < 3; ++j)
        {
            for (int k = 0; k < 2; ++k)
            {
                scanf("%d", &test[i][j][k]);
            }
        }
    }

    // Printing values with proper index.

    printf("\nDisplaying values:\n");
    for (int i = 0; i < 2; ++i)
    {
        for (int j = 0; j < 3; ++j)
        {
            for (int k = 0; k < 2; ++k)
            {
                printf("test[%d][%d][%d] = %d\n", i, j, k, test[i][j][k]);
            }
        }
    }

    return 0;
}

```

```
}
```

Output

```
Enter 12 values:
```

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12
```

```
Displaying Values:
```

```
test[0][0][0] = 1  
test[0][0][1] = 2  
test[0][1][0] = 3  
test[0][1][1] = 4  
test[0][2][0] = 5  
test[0][2][1] = 6  
test[1][0][0] = 7  
test[1][0][1] = 8  
test[1][1][0] = 9  
test[1][1][1] = 10  
test[1][2][0] = 11  
test[1][2][1] = 12
```

Pass arrays to a function in C

In C programming, you can pass an entire array to functions. Before we learn that, let's see how you can pass individual elements of an array to functions.

Pass Individual Array Elements

Passing array elements to a function is similar to passing variables to a function.

```
#include <stdio.h>  
void getarray(int arr[]){  
    printf("Elements of array are : ");
```

```

for(int i=0;i<5;i++)
{
    printf("%d ", arr[i]);
}
}

int main()
{
    int arr[5]={45,67,34,78,90};
    getarray(arr);
    return 0;
}

```

In the above program, we have first created the array **arr[]** and then we pass this array to the function **getarray()**. The **getarray()** function prints all the elements of the array **arr[]**.

Output

```

Elements of array are : 45 67 34 78 90
...Program finished with exit code 0
Press ENTER to exit console. []

```

Example 1: Pass Individual Array Elements

```

#include <stdio.h>
void display(int age1, int age2) {
    printf("%d\n", age1);
    printf("%d\n", age2);
}

int main() {
    int ageArray[] = {2, 8, 4, 12};

    // pass second and third elements to display()
    display(ageArray[1], ageArray[2]);
    return 0;
}

```

```
}
```

Output

```
8
```

```
4
```

Here, we have passed array parameters to the `display()` function in the same way we pass variables to a function.

```
// pass second and third elements to display()
display(ageArray[1], ageArray[2]);
```

We can see this in the function definition, where the function parameters are individual variables:

```
void display(int age1, int age2) {
    // code
}
```

Example 2: Pass Arrays to Functions

```
// Program to calculate the sum of array elements by passing to a function

#include <stdio.h>
float calculateSum(float num[]);

int main() {
    float result, num[] = {23.4, 55, 22.6, 3, 40.5, 18};

    // num array is passed to calculateSum()
    result = calculateSum(num);
    printf("Result = %.2f", result);
    return 0;
}

float calculateSum(float num[]) {
    float sum = 0.0;

    for (int i = 0; i < 6; ++i) {
        sum += num[i];
    }
}
```

```
    return sum;  
}
```

Output

Result = 162.50

To pass an entire array to a function, only the name of the array is passed as an argument.

```
result = calculateSum(num);
```

However, notice the use of `[]` in the function definition.

```
float calculateSum(float num[]) {  
... ...  
}
```

This informs the compiler that you are passing a one-dimensional array to the function.

C function to sort the array

```
#include<stdio.h>  
void Bubble_Sort(int());  
void main ()  
{  
    int arr[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};  
    Bubble_Sort(arr);  
}  
void Bubble_Sort(int a[]) //array a[] points to arr.  
{  
    int i, j,temp;  
    for(i = 0; i<10; i++)  
    {  
        for(j = i+1; j<10; j++)  
        {  
            if(a[j] < a[i])  
            {  
                temp = a[i];  
                a[i] = a[j];  
            }  
        }  
    }  
}
```

```

        a[j] = temp;
    }
}
printf("Printing Sorted Element List ...\\n");
for(i = 0; i<10; i++)
{
    printf("%d\\n",a[i]);
}
}

```

Pass Multidimensional Arrays to a Function

To pass multidimensional arrays to a function, only the name of the array is passed to the function (similar to one-dimensional arrays).

Example 3: Pass two-dimensional arrays

```

#include <stdio.h>
void displayNumbers(int num[2][2]);

int main() {
    int num[2][2];
    printf("Enter 4 numbers:\\n");
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 2; ++j) {
            scanf("%d", &num[i][j]);
        }
    }

    // pass multi-dimensional array to a function
    displayNumbers(num);

    return 0;
}

void displayNumbers(int num[2][2]) {

```

```
printf("Displaying:\n");
for (int i = 0; i < 2; ++i) {
    for (int j = 0; j < 2; ++j) {
        printf("%d\n", num[i][j]);
    }
}
```

Output

```
Enter 4 numbers:
```

```
2
3
4
5
```

```
Displaying:
```

```
2
3
4
5
```

Notice the parameter `int num[2][2]` in the function prototype and function definition:

```
// function prototype
void displayNumbers(int num[2][2]);
```

This signifies that the function takes a two-dimensional array as an argument. We can also pass arrays with more than 2 dimensions as a function argument.

When passing two-dimensional arrays, it is not mandatory to specify the number of rows in the array. However, the number of columns should always be specified.

For example,

```
void displayNumbers(int num[][][2]) {
    // code
}
```

Example:

```

//Passing array elements to a function

void readMatrix(int matrix[3][3]);

void printMatrix(int matrix[3][3]); //function declaration

int i,j;

#include <stdio.h>

int main()

{

    int matrix[3][3];

    readMatrix(matrix); //calling a function

    printMatrix(matrix);

    return 0;

}

void readMatrix(int matrix[3][3]) //function definition

{

    printf("Enter matrix elements : \n");

    for(i=0;i<3;i++)

    {

        for(j=0;j<3;j++)

        {

            scanf("%d", &matrix[i][j]);

        }

    }

}

void printMatrix(int matrix[3][3]) //function definition

{

```

```
printf("The given matrix is : \n");
for(i=0;i<3;i++)
{
    for(j=0;j<3;j++)
    {
        printf("%d ", matrix[i][j]);
    }
    printf("\n");
}
}
```

Example:

```
//Program to add two matrices using passing array elements to a function
#include <stdio.h>
#define MAXROWS 3
#define MAXCOLS 3

void readMatrix(int matrix[][][MAXCOLS]);
void printMatrix(int matrix[][][MAXCOLS]); //function declaration
void sumMatrix(int matA[][][MAXCOLS], int matB[][][MAXCOLS], int
matC[][][MAXCOLS]);

int i,j;

int main()
{
    int matA[3][3],matB[3][3],matC[3][3];
    readMatrix(matA); //calling a function
    printf("The given matrix A is : \n");
    printMatrix(matA);
    readMatrix(matB); //calling a function
    printf("The given matrix B is : \n");
    printMatrix(matB);

    sumMatrix(matA,matB,matC);

    printf ("The sum of matrices is\n");
    printMatrix(matC);

    return 0;
}

void readMatrix(int matrix[][][MAXCOLS]) //function definition
{
    printf("Enter matrix elements: \n");
}
```

```

for(i=0;i<3;i++)
{
    for(j=0;j<3;j++)
    {
        scanf("%d", &matrix[i][j]);
    }
}
void printMatrix(int matrix[][MAXCOLS]) //function definition
{
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}

/* Addition of matrices */
void sumMatrix(int matA[][MAXCOLS], int matB[][MAXCOLS], int matC[][MAXCOLS])
{
    for(i=0; i< 3 ; i++)
    {
        for ( j=0; j < 3; j++)
        {
            matC[i][j] = matA[i][j] + matB[i][j];
        }
    }
}

```

Output:

Enter matrix elements:

1 0 1 2 1 1 3 0 1

The given matrix A is:

1 0 1

2 1 1

3 0 1

Enter matrix elements:

1 1 1 2 2 2 3 3 3

The given matrix B is:

1 1 1

2 2 2

3 3 3

The sum of matrices is

2 1 2

4 3 3

6 3 4

Process exited after 41.28 seconds with return value 0

Press any key to continue . . .

C Pointers

Pointers are powerful features of C and C++ programming. Pointer is an important feature of C programming language. It is considered as the beauty of C programming language making this language more powerful and robust.

A pointer is a special variable in C programming language which stores the memory address of other variables of the same data type. As a pointer is variable, it is also created in some memory location.

Consider the following example to define a pointer which stores the address of an integer.

```
int n = 10;  
int* p = &n; // Variable p of type pointer is pointing to the address of the variable  
n of type integer.
```

Declaring a pointer

The pointer in c language can be declared using * (asterisk symbol). It is also known as indirection pointer used to dereference a pointer.

Like normal variables, pointer variables must be declared before using them. General syntax for declaring pointer variable is:

```
data_type * pointer_name;
```

Here, data_type can be any valid C data types and pointer_name can be any valid C identifier.

```
int *a;//pointer to int  
char *c;//pointer to char
```

Examples of Declaration of Pointer:

1. **int *ptr;**

Here **ptr** is a pointer variable and it is read as a **pointer to integer** since it can point to integer variables.

2. **float *fptr;**

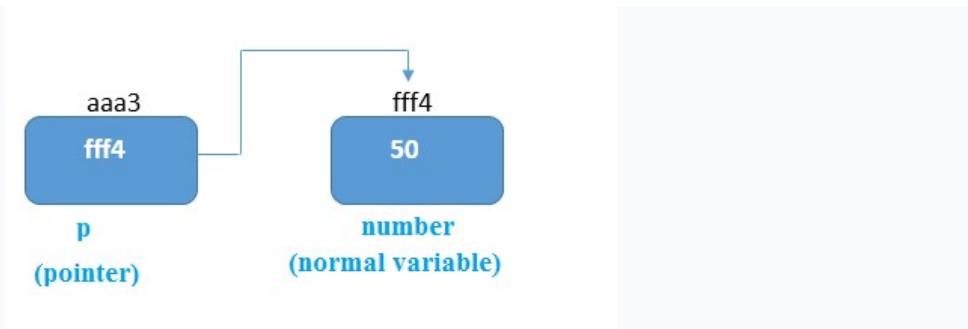
Here **fptr** is a pointer variable and it is read as a **pointer to float** since it can point to float variables.

3. **char *cp;**

Here **cp** is a pointer variable and it is read as a **pointer to character** since it can point to character variables.

Pointer Example

An example of using pointers to print the address and value is given below.



As you can see in the above figure, pointer variable stores the address of number variable, i.e., fff4. The value of number variable is 50. But the address of pointer variable p is aaa3.

By the help of * (**indirection operator**), we can print the value of pointer variable p.

Let's see the pointer example as explained for the above figure.

```
#include<stdio.h>
int main(){
    int number=50;
    int *p;
    p=&number;//stores the address of number variable
```

```
printf("Address of p variable is %x \n",p); // p contains the address of the number  
therefore printing p gives the address of number.  
printf("Value of p variable is %d \n",*p); // As we know that * is used to dereference a pointer therefore if we print *p, we will get the value stored at the address contained by p.  
return 0;  
}
```

Output

```
Address of number variable is fff4  
Address of p variable is fff4  
Value of p variable is 50
```

Pointer to array

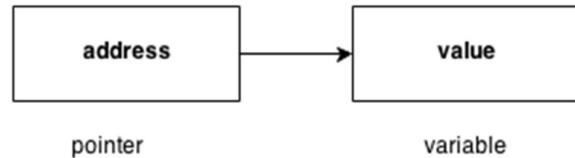
```
int arr[10];  
int *p[10]=&arr; // Variable p of type pointer is pointing to the address of an integer array arr.
```

Pointer to a function

```
void show (int);  
void(*p)(int) = &display; // Pointer p is pointing to the address of a function
```

Pointer to structure

```
struct st {  
    int i;  
    float f;  
}ref;  
struct st *p = &ref;
```



Advantages of pointer

1. Pointer **reduces the code** and **improves the performance**, it is used to retrieving strings, trees, etc. and used with arrays, structures, and functions.
2. We can **return multiple values from a function** using the pointer.
3. It makes you able to **access any memory location** in the computer's memory.
Pointers provide direct access to memory.
4. Pointers reduce the storage space and complexity of programs.
5. Pointers reduce the execution time of programs.
6. Pointers provide an alternate way to access individual array elements.
7. Pointers allow us to perform dynamic memory allocation and deallocation.
8. Pointers allow us to create more complex data structures like linked lists, trees, graphs, stacks etc.

Disadvantages of Pointer

1. If pointers are referenced with incorrect values, then it affects the whole program.
2. Memory leak occurs if dynamically allocated memory is not freed.
3. Segmentation fault can occur due to uninitialized pointer.

Usage of pointer

There are many applications of pointers in c language.

1) Dynamic memory allocation

In c language, we can dynamically allocate memory using malloc() and calloc() functions where the pointer is used.

2) Arrays, Functions, and Structures

Pointers in c language are widely used in arrays, functions, and structures. It reduces the code and improves the performance.

Address in C

If you have a variable `var` in your program, `&var` will give you its address in the memory.

We have used address numerous times while using the `scanf()` function.

```
scanf("%d", &var);
```

Here, the value entered by the user is stored in the address of `var` variable. Let's take a working example.

```
#include <stdio.h>
int main()
{
    int var = 5;
    printf("var: %d\n", var);

    // Notice the use of & before var
    printf("address of var: %p", &var);
    return 0;
}
```

Output

```
var: 5
address of var: 2686778
```

Note: You will probably get a different address when you run the above code.

```
#include<stdio.h>
int main(){
    int number=50;
    printf("value of number is %d, address of number is %u",number,&number);
    return 0;
}
```

Output

```
value of number is 50, address of number is fff4
```

C Pointers

Pointers (pointer variables) are special variables that are used to store addresses rather than values.

Pointer Syntax

Here is how we can declare pointers.

```
int* p;
```

Here, we have declared a pointer `p` of `int` type.

You can also declare pointers in these ways.

```
int *p1;  
int * p2;
```

Let's take another example of declaring pointers.

```
int* p1, p2;
```

Here, we have declared a pointer `p1` and a normal variable `p2`.

Assigning addresses to Pointers

Let's take an example.

```
int* pc, c;  
c = 5;  
pc = &c;
```

Here, `5` is assigned to the `c` variable. And, the address of `c` is assigned to the `pc` pointer.

Get Value of Thing Pointed by Pointers

To get the value of the thing pointed by the pointers, we use the `*` operator. For example:

```
int* pc, c;  
c = 5;  
pc = &c;  
printf("%d", *pc); // Output: 5
```

Here, the address of `c` is assigned to the `pc` pointer. To get the value stored in that address, we used `*pc`.

Note: In the above example, `pc` is a pointer, not `*pc`. You cannot and should not do something like `*pc = &c;`

By the way, `*` is called the *dereference operator* (when working with pointers). It operates on a pointer and gives the value stored in that pointer.

Changing Value Pointed by Pointers

Let's take an example.

```
int* pc, c;  
c = 5;  
pc = &c;  
c = 1;  
printf("%d", c); // Output: 1  
printf("%d", *pc); // Output: 1
```

We have assigned the address of `c` to the `pc` pointer.

Then, we changed the value of `c` to 1. Since `pc` and the address of `c` is the same, `*pc` gives us 1.

Let's take another example.

```
int* pc, c;  
c = 5;  
pc = &c;  
*pc = 1;  
printf("%d", *pc); // Output: 1  
printf("%d", c); // Output: 1
```

We have assigned the address of `c` to the `pc` pointer.

Then, we changed `*pc` to 1 using `*pc = 1;`. Since `pc` and the address of `c` is the same, `c` will be equal to 1.

Let's take one more example.

```
int* pc, c, d;  
c = 5;  
d = -15;  
  
pc = &c; printf("%d", *pc); // Output: 5  
pc = &d; printf("%d", *pc); // Output: -15
```

Initially, the address of `c` is assigned to the `pc` pointer using `pc = &c;`. Since `c` is 5, `*pc` gives us 5.

Then, the address of `d` is assigned to the `pc` pointer using `pc = &d;`. Since `d` is -15, `*pc` gives us -15.

Example: Working of Pointers

Let's take a working example.

```
#include <stdio.h>  
int main()  
{  
    int* pc, c;
```

```

c = 22;
printf("Address of c: %p\n", &c);
printf("Value of c: %d\n\n", c); // 22

pc = &c;
printf("Address of pointer pc: %p\n", pc);
printf("Content of pointer pc: %d\n\n", *pc); // 22

c = 11;
printf("Address of pointer pc: %p\n", pc);
printf("Content of pointer pc: %d\n\n", *pc); // 11

*pc = 2;
printf("Address of c: %p\n", &c);
printf("Value of c: %d\n\n", c); // 2
return 0;
}

```

Output

Address of c: 2686784

Value of c: 22

Address of pointer pc: 2686784

Content of pointer pc: 22

Address of pointer pc: 2686784

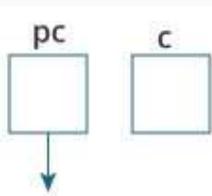
Content of pointer pc: 11

Address of c: 2686784

Value of c: 2

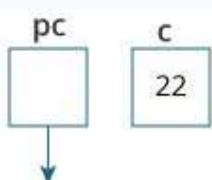
Explanation of the program

1. `int* pc, c;`



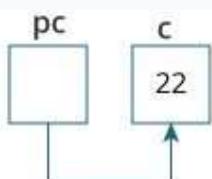
Here, a pointer `pc` and a normal variable `c`, both of type `int`, is created. Since `pc` and `c` are not initialized at initially, pointer `pc` points to either no address or a random address. And, variable `c` has an address but contains random garbage value.

2. `c = 22;`



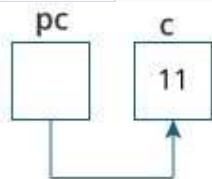
This assigns 22 to the variable `c`. That is, 22 is stored in the memory location of variable `c`.

3. `pc = &c;`



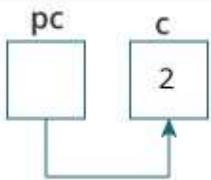
This assigns the address of variable `c` to the pointer `pc`.

4. `c = 11;`



This assigns 11 to variable `c`.

5. `*pc = 2;`



This changes the value at the memory location pointed by the pointer `pc` to 2.

Referencing of Pointer (Initialization of Pointer)

Making a pointer variable to point other variables by providing address of that variable to the pointer is known as **referencing of pointer**.

It is also known as **initialization of pointers**. For proper use of pointer, pointer variables must point to some valid address and it is important to note that without referencing, pointer variables are meaningless.

General syntax for referencing of pointer is:

```
pointer_variable = &normal_variable;
```

Here `pointer_variable` and `normal_variable` must be of the same data types.

Examples of Referencing of Pointer:

- 1.
2. `int a=5;`
3. `int *ptr;`
4. `ptr = &a;`
- 5.

6. Here pointer **ptr** got address of variable **a** so, pointer **ptr** is now pointing to variable **a**.

7.

8. **float val=5.5;**

9. **float *p;**

10. **p = &val;**

11.

12. Here pointer **p** got address of variable **val** so, pointer **p** is now pointing to variable **val**.

But!

```
float x=30.4;  
int *iptr;  
iptr = &x;
```

is **invalid!!!** Because pointer **iptr** cannot store address of float variable.

Dereferencing of Pointer

So far, the operator ***** (star) used in front of the name of the pointer variable is known as **pointer or dereferencing or indirection operator**. After valid referencing of pointer variable, *** pointer_variable** gives the value of the variable pointed by pointer variable and this is known as dereferencing of pointer. For the sake of simplicity, ***pointer_variable** after referencing instructs compilers that go to the memory address stored by **pointer_variable** and get value from that memory address.

Examples of Dereferencing of Pointer:

```
int a=5;
int *ptr;
ptr = &a;
printf("a = %d", *ptr);
```

This prints a = 5 which is accessed through pointer.

```
float val=5.5;
float *p;
p = &val;
printf("val = %f", *p);
```

This prints val = 5.5 which is accessed through pointer. .

Pointer Examples

1. Program to illustrate concept of declaration, referencing, dereferencing of pointer and displaying address

C Source Code: Pointer Example

```
#include<stdio.h>

int main()
{
    int a=50, *p;
    /* Here a is normal variable and *p is pointer variable */

    p = &a; /* Referencing */
    /* After above statement i.e. referencing, now pointer points to a */

    printf("Value of a = %d\n", a);
    printf("Address of a = %u\n", &a);
    printf("Value of a using pointer = %d\n", *p);
    printf("Address of a using pointer = %u\n", p);
    return 0;
```

}

Output

```
Value of a = 50
Address of a = 65524
Value of a using pointer = 50
Address of a using pointer = 65524
```

Above address will be different on your system.

2. Program to illustrate concept of declaration, referencing and dereferencing of pointer:

C Source Code: Pointer Example

```
#include<stdio.h>

int main()
{
    int a=50, *p;
    float val = 50.5, *f;

    p = &a;
    f = &val;
    printf("Value of a = %d\n", *p);
    printf("Value of val = %f\n", *f)

    return 0;
}
```

Output

```
Value of a = 50
Value of val = 50.5
```

NULL Pointers

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries.

Consider the following program –

```
int *p=NULL;
```

In the most libraries, the value of the pointer is 0 (zero).

Example:

```
#include <stdio.h>

int main () {
    int *ptr = NULL;
    printf("The value of ptr is : %x\n", ptr );
    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

The value of ptr is 0

Pointer Program to swap two numbers without using the 3rd variable.

```
#include<stdio.h>
int main(){
    int a=10,b=20,*p1=&a,*p2=&b;

    printf("Before swap: *p1=%d *p2=%d",*p1,*p2);
    *p1=*p1+*p2;
    *p2=*p1-*p2;
    *p1=*p1-*p2;
    printf("\nAfter swap: *p1=%d *p2=%d",*p1,*p2);

    return 0;
}
```

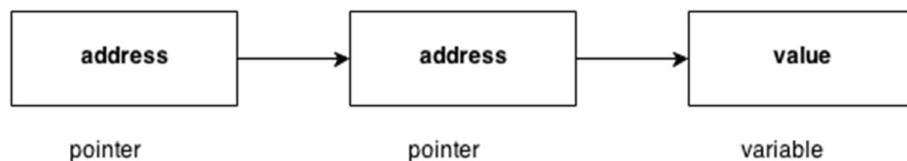
Output

```
Before swap: *p1=10 *p2=20
```

```
After swap: *p1=20 *p2=10
```

C Double Pointer (Pointer to Pointer)

As we know that, a pointer is used to store the address of a variable in C. Pointer reduces the access time of a variable. However, In C, we can also define a pointer to store the address of another pointer. Such pointer is known as a double pointer (pointer to pointer). The first pointer is used to store the address of a variable whereas the second pointer is used to store the address of the first pointer. Let's understand it by the diagram given below.



C allows the use of pointers that points to other pointers. For making pointer to point another pointer we need to add extra * to a pointer variable.

The syntax of declaring a double pointer is given below.

```
int **p; // pointer to a pointer which is pointing to an integer.
```

Pointer to Pointer Examples

```
1. int p=10, *ptr, **pptr;
```

In this declaration p is normal variable, ptr is pointer to integer and pptr is pointer to pointer.

```
2. float f=50.5, *fp, **fpt, ***fptr;
```

In this declaration **f** is normal variable, **fp** is **pointer to float**, **fpt** is **pointer to pointer** and **fptr** is **pointer to pointer to pointer**.

3. **char c, *d, **e, ***f, ****g, *****h;**

Here **e, f, g** and **h** are all **pointer to pointer**.

Theoretically there is no limit on how many stars that can be added before a pointer variable but there will be some limit and this limit comes from how the compiler is designed and what system you are using!

Pointer to Pointer Example

```
#include<stdio.h>

int main()
{
    int p=10, *q, **r, ***s;

    /* Referencing */
    q = &p;
    r = &q;
    s = &r;

    /* Some Results */
    printf("\nValue of p = %d", p);
    printf("\nValue of p using pointer q = %d", *q);
    printf("\nValue of p using pointer r = %d", **r);
    printf("\nValue of p using pointer s = %d", ***s);
}
```

```
return 0;
```

```
}
```

Output

```
Value of p = 10  
Value of p using pointer q = 10  
Value of p using pointer r = 10  
Value of p using pointer s = 10
```

Explanation: Pointer to Pointer Example

In this program is **p** is **normal variable**, **q** is **pointer to integer** which means it can point to some **integer variable**. When statement **q = &p;** is executed then pointer **q** is storing address of **p** so value of **p** can be dereferenced using **q**.

Remember dereferencing? After referencing, ***q** is ***&p** and **to remember if & comes after * i.e. *& (not &*) then you can cancel that out :).** So after canceling ***&** from ***&p** we get **p** right?

Let's apply same logic for other statement!! Here **r** is pointer to pointer so it can store address of any pointer variable of type integer. And this is exactly what we are doing using **r = &q;** After this assignment, pointer to pointer variable **r** is storing address of another pointer variable **q**. So ****r** is ***(*&)q=>*q=>(*&)p=>p :**

Similarly, **s = &r;** *****s=>**(*&)r=>**r=> *(*&)q=>*q=>(*&)p=>p :** So form all statements we get value of **p** which is equal to 10.

Consider the following example.

```
#include<stdio.h>  
void main ()  
{  
    int a = 10;  
    int *p;
```

```

int **pp;
p = &a; // pointer p is pointing to the address of a
pp = &p; // pointer pp is a double pointer pointing to the address of pointer p
printf("address of a: %x\n",p); // Address of a will be printed
printf("address of p: %x\n",pp); // Address of p will be printed
printf("value stored at p: %d\n",*p); // value stored at the address contained b
y p i.e. 10 will be printed
printf("value stored at pp: %d\n",**pp); // value stored at the address containe
d by the pointer stoyred at pp
}

```

Output

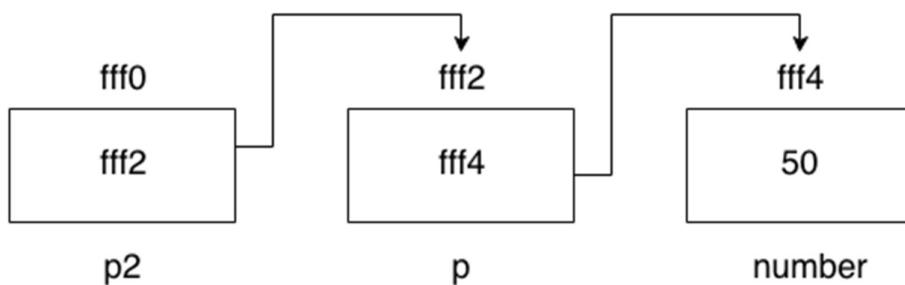
```

address of a: d26a8734
address of p: d26a8738
value stored at p: 10
value stored at pp: 10

```

C double pointer example

Let's see an example where one pointer points to the address of another pointer.



As you can see in the above figure, p2 contains the address of p (fff2), and p contains the address of number variable (fff4).

```

#include<stdio.h>
int main(){
int number=50;
int *p;//pointer to int
int **p2;//pointer to pointer

```

```

p=&number;//stores the address of number variable
p2=&p;
printf("Address of number variable is %x \n",&number);
printf("Address of p variable is %x \n",p);
printf("Value of *p variable is %d \n",*p);
printf("Address of p2 variable is %x \n",p2);
printf("Value of **p2 variable is %d \n",*p);
return 0;
}

```

Output

```

Address of number variable is fff4
Address of p variable is fff4
Value of *p variable is 50
Address of p2 variable is fff2
Value of **p variable is 50

```

Note:

The `%x` is the `printf` format that indicates that the `int` value should be displayed in the hexadecimal.

Pointer Arithmetic in C

We can perform arithmetic operations on the pointers like addition, subtraction, etc. However, as we know that pointer contains the address, the result of an arithmetic operation performed on the pointer will also be a pointer if the other operand is of type integer. In pointer-from-pointer subtraction, the result will be an integer value. Following arithmetic operations are possible on the pointer in C language:

1. Increment
2. Decrement
3. Addition
4. Subtraction
5. Comparison

Incrementing Pointer in C

If we increment a pointer by 1, the pointer will start pointing to the immediate next location. This is somewhat different from the general arithmetic since the value of the pointer will get increased by the size of the data type to which the pointer is pointing.

We can traverse an array by using the increment operation on a pointer which will keep pointing to every element of the array, perform some operation on that, and update itself in a loop.

The Rule to increment the pointer is given below:

1. new_address= current_address + i * size_of(data type)

Where i is the number by which the pointer get increased.

32-bit

For 32-bit int variable, it will be incremented by 2 bytes.

64-bit

For 64-bit int variable, it will be incremented by 4 bytes.

Let's see the example of incrementing pointer variable on 64-bit architecture.

```
#include<stdio.h>
int main(){
    int number=50;
    int *p;//pointer to int
    p=&number;//stores the address of number variable
    printf("Address of p variable is %u \n",p);
    p=p+1;
    printf("After increment: Address of p variable is %u \n",p); // in our case, p will get incremented by 4 bytes.
    return 0;
}
```

Note:

The %u format specifiers implemented for fetching values form the address of variable having unsigned decimal integer stored in the memory.

Output

```
Address of p variable is 3214864300  
After increment: Address of p variable is 3214864304
```

Traversing an array by using pointer

```
#include<stdio.h>  
void main ()  
{  
    int arr[5] = {1, 2, 3, 4, 5};  
    int *p = arr;  
    int i;  
    printf("printing array elements...\n");  
    for(i = 0; i < 5; i++)  
    {  
        printf("%d ",*(p+i));  
    }  
}
```

Output

```
printing array elements...  
1 2 3 4 5
```

Decrementing Pointer in C

Like increment, we can decrement a pointer variable. If we decrement a pointer, it will start pointing to the previous location. The formula of decrementing the pointer is given below:

$$\text{new_address} = \text{current_address} - i * \text{size_of(data type)}$$

32-bit

For 32-bit int variable, it will be decremented by 2 bytes.

64-bit

For 64-bit int variable, it will be decremented by 4 bytes.

Let's see the example of decrementing pointer variable on 64-bit OS.

```
#include <stdio.h>
void main(){
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p-1;
printf("After decrement: Address of p variable is %u \n",p); // P will now point to t
he immidiate previous location.
}
```

Output

```
Address of p variable is 3214864300
After decrement: Address of p variable is 3214864296
```

C Pointer Addition

We can add a value to the pointer variable. The formula of adding value to pointer is given below:

$$\text{new_address} = \text{current_address} + (\text{number} * \text{size_of(data type)})$$

32-bit

For 32-bit int variable, it will add $2 * \text{number}$.

64-bit

For 64-bit int variable, it will add $4 * \text{number}$.

Let's see the example of adding value to pointer variable on 64-bit architecture.

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

```

int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p+3; //adding 3 to pointer variable
printf("After adding 3: Address of p variable is %u \n",p);
return 0;
}

```

Output

```

Address of p variable is 3214864300
After adding 3: Address of p variable is 3214864312

```

As you can see, the address of p is 3214864300. But after adding 3 with p variable, it is 3214864312, i.e., $4*3=12$ increment. Since we are using 64-bit architecture, it increments 12. But if we were using 32-bit architecture, it was incrementing to 6 only, i.e., $2*3=6$. As integer value occupies 2-byte memory in 32-bit OS.

C Pointer Subtraction

Like pointer addition, we can subtract a value from the pointer variable. Subtracting any number from a pointer will give an address. The formula of subtracting value from the pointer variable is given below:

$$\text{new_address} = \text{current_address} - (\text{number} * \text{size_of(data type)})$$

32-bit

For 32-bit int variable, it will subtract $2 * \text{number}$.

64-bit

For 64-bit int variable, it will subtract $4 * \text{number}$.

Let's see the example of subtracting value from the pointer variable on 64-bit architecture.

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

```

int main(){
    int number=50;
    int *p;//pointer to int
    p=&number;//stores the address of number variable
    printf("Address of p variable is %u \n",p);
    p=p-3; //subtracting 3 from pointer variable
    printf("After subtracting 3: Address of p variable is %u \n",p);
    return 0;
}

```

Output

```

Address of p variable is 3214864300
After subtracting 3: Address of p variable is 3214864288

```

You can see after subtracting 3 from the pointer variable, it is 12 (4*3) less than the previous address value.

However, instead of subtracting a number, we can also subtract an address from another address (pointer). This will result in a number. It will not be a simple arithmetic operation, but it will follow the following rule.

If two pointers are of the same type,

1. $\text{Address2} - \text{Address1} = (\text{Subtraction of two addresses})/\text{size of data type which pointer point to}$

Consider the following example to subtract one pointer from another.

```

#include<stdio.h>
void main ()
{
    int i = 100;
    int *p = &i;
    int *temp;
    temp = p;
    p = p + 3;
    printf("Pointer Subtraction: %d - %d = %d",p, temp, p-temp);
}

```

Output

Pointer Subtraction: 1030585080 - 1030585068 = 3

Illegal arithmetic with pointers

There are various operations which can not be performed on pointers. Since, pointer stores address hence we must ignore the operations which may lead to an illegal address, for example, addition, and multiplication. A list of such operations is given below.

1. Address + Address = illegal
2. Address * Address = illegal
3. Address % Address = illegal
4. Address / Address = illegal
5. Address & Address = illegal
6. Address ^ Address = illegal
7. Address | Address = illegal
8. ~Address = illegal

Pointer to function in C

As we discussed in the previous chapter, a pointer can point to a function in C. However, the declaration of the pointer variable must be the same as the function. Consider the following example to make a pointer pointing to the function.

```
#include<stdio.h>
int addition ();
int main ()
{
    int result;
    int (*ptr)();
    ptr = &addition;
    result = (*ptr)();
    printf("The sum is %d",result);
}
int addition()
{
```

```

int a, b;
printf("Enter two numbers?");
scanf("%d %d",&a,&b);
return a+b;
}

```

Output

```
Enter two numbers?10 15
```

```
The sum is 25
```

Pointer to Array of functions in C

To understand the concept of an array of functions, we must understand the array of function. Basically, an array of the function is an array which contains the addresses of functions. In other words, the pointer to an array of functions is a pointer pointing to an array which contains the pointers to the functions. Consider the following example.

```

#include<stdio.h>
int show();
int showadd(int);
int (*arr[3])();
int (*(*ptr)[3])();

int main ()
{
    int result1;
    arr[0] = show;
    arr[1] = showadd;
    ptr = &arr;
    result1 = (**ptr)();
    printf("printing the value returned by show : %d",result1);
    (*(*ptr+1))(result1);
}

int show()
{

```

```

int a = 65;
return a++;
}

int showadd(int b)
{
    printf("\nAdding 90 to the value returned by show: %d",b+90);
}

```

Output

```

printing the value returned by show : 65
Adding 90 to the value returned by show: 155

```

C Pointers & Functions

C Pass Addresses and Pointers

In C programming, it is also possible to pass addresses as arguments to functions.

To accept these addresses in the function definition, we can use pointers. It's because pointers are used to store addresses. Let's take an example:

Example: Pass Addresses to Functions

```

#include <stdio.h>
void swap(int *n1, int *n2);

int main()
{
    int num1 = 5, num2 = 10;

    // address of num1 and num2 is passed
    swap( &num1, &num2);

    printf("num1 = %d\n", num1);
    printf("num2 = %d", num2);
    return 0;
}

void swap(int* n1, int* n2)
{
    int temp;
    temp = *n1;
}

```

```
*n1 = *n2;
*n2 = temp;
}
```

When you run the program, the output will be:

```
num1 = 10
num2 = 5
```

The address of `num1` and `num2` are passed to the `swap()` function using `swap(&num1, &num2);`.

Pointers `n1` and `n2` accept these arguments in the function definition.

```
void swap(int* n1, int* n2) {
    ...
}
```

When `*n1` and `*n2` are changed inside the `swap()` function, `num1` and `num2` inside the `main()` function are also changed.

Inside the `swap()` function, `*n1` and `*n2` swapped. Hence, `num1` and `num2` are also swapped.

Notice that `swap()` is not returning anything; its return type is `void`.

Example 2: Passing Pointers to Functions

```
#include <stdio.h>

void addOne(int* ptr) {
    (*ptr)++; // adding 1 to *ptr
}

int main()
{
    int* p, i = 10;
    p = &i;
    addOne(p);

    printf("%d", *p); // 11
    return 0;
}
```

```
}
```

Here, the value stored at `p`, `*p`, is 10 initially.

We then passed the pointer `p` to the `addOne()` function. The `ptr` pointer gets this address in the `addOne()` function.

Inside the function, we increased the value stored at `ptr` by 1 using `(*ptr)++`.

Since `ptr` and `p` pointers both have the same address, `*p` inside `main()` is also 11.

Relationship Between Arrays and Pointers

An array is a block of sequential data. Let's write a program to print addresses of array elements.

```
#include <stdio.h>
int main() {
    int x[4];
    int i;

    for(i = 0; i < 4; ++i) {
        printf("&x[%d] = %p\n", i, &x[i]);
    }

    printf("Address of array x: %p", x);

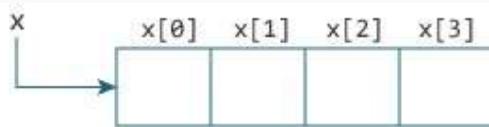
    return 0;
}
```

Output

```
&x[0] = 1450734448
&x[1] = 1450734452
&x[2] = 1450734456
&x[3] = 1450734460
Address of array x: 1450734448
```

There is a difference of 4 bytes between two consecutive elements of array `x`. It is because the size of `int` is 4 bytes (on our compiler).

Notice that, the address of `&x[0]` and `x` is the same. It's because the variable name `x` points to the first element of the array.



Relation between Arrays and Pointers

From the above example, it is clear that `&x[0]` is equivalent to `x`. And, `x[0]` is equivalent to `*x`.

Similarly,

- `&x[1]` is equivalent to `x+1` and `x[1]` is equivalent to `*(x+1)`.
- `&x[2]` is equivalent to `x+2` and `x[2]` is equivalent to `*(x+2)`.
- ...
- Basically, `&x[i]` is equivalent to `x+i` and `x[i]` is equivalent to `*(x+i)`.

Example 1: Pointers and Arrays

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

    int i, x[6], sum = 0;

    printf("Enter 6 numbers: ");

    for(i = 0; i < 6; ++i) {
        // Equivalent to scanf("%d", &x[i]);
        scanf("%d", x+i);

        // Equivalent to sum += x[i]
        sum += *(x+i);
    }

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

    return 0;
}
```

When you run the program, the output will be:

```
Enter 6 numbers: 2
```

```
3
```

```
4
```

```
4
```

```
12
```

```
4
```

```
Sum = 29
```

Here, we have declared an array `x` of 6 elements. To access elements of the array, we have used pointers.

Example 2: Arrays and Pointers

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

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

    // ptr is assigned the address of the third element
    ptr = &x[2];

    printf("*ptr = %d \n", *ptr); // 3
    printf("*(ptr+1) = %d \n", *(ptr+1)); // 4
    printf("*(ptr-1) = %d", *(ptr-1)); // 2

    return 0;
}
```

When you run the program, the output will be:

```
*ptr = 3
*(ptr+1) = 4
*(ptr-1) = 2
```

In this example, `&x[2]`, the address of the third element, is assigned to the `ptr` pointer.

Hence, `3` was displayed when we printed `*ptr`.

And, printing `*(ptr+1)` gives us the fourth element. Similarly, printing `*(ptr-1)` gives us the second element.

C Dynamic Memory Allocation

Here, you'll learn to dynamically allocate memory in your C program using standard library functions: `malloc()`, `calloc()`, `free()` and `realloc()`.

As you know, an array is a collection of a fixed number of values. Once the size of an array is declared, you cannot change it.

Sometimes the size of the array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming.

To allocate memory dynamically, library functions are `malloc()`, `calloc()`, `realloc()` and `free()` are used. These functions are defined in the `<stdlib.h>` header file.

C `malloc()`

The name "malloc" stands for memory allocation.

The `malloc()` function reserves a block of memory of the specified number of bytes. And, it returns a pointer of `void` which can be casted into pointers of any form.

Syntax of `malloc()`

```
ptr = (castType*) malloc(size);
```

Example

```
ptr = (float*) malloc(100 * sizeof(float));
```

The above statement allocates 400 bytes of memory. It's because the size of `float` is 4 bytes. And, the pointer `ptr` holds the address of the first byte in the allocated memory.

The expression results in a `NULL` pointer if the memory cannot be allocated.

C `calloc()`

The name "calloc" stands for contiguous allocation.

The `malloc()` function allocates memory and leaves the memory uninitialized, whereas the `calloc()` function allocates memory and initializes all bits to zero.

Syntax of calloc()

```
ptr = (castType*)calloc(n, size);
```

Example:

```
ptr = (float*) calloc(25, sizeof(float));
```

The above statement allocates contiguous space in memory for 25 elements of type float.

C free()

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on their own. You must explicitly use free() to release the space.

Syntax of free()

```
free(ptr);
```

This statement frees the space allocated in the memory pointed by ptr.

Example 1: malloc() and free()

```
// Program to calculate the sum of n numbers entered by the user
```

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

int main() {
    int n, i, *ptr, sum = 0;

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

    ptr = (int*) malloc(n * sizeof(int));

    // if memory cannot be allocated
```

```

if(ptr == NULL) {
    printf("Error! memory not allocated.");
    exit(0);
}

printf("Enter elements: ");
for(i = 0; i < n; ++i) {
    scanf("%d", ptr + i);
    sum += *(ptr + i);
}

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

// deallocated the memory
free(ptr);

return 0;
}

```

Output

```

Enter number of elements: 3
Enter elements: 100
20
36
Sum = 156

```

Here, we have dynamically allocated the memory for `n` number of `int`.

Example 2: `calloc()` and `free()`

```

// Program to calculate the sum of n numbers entered by the user

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

```

```
int main() {
    int n, i, *ptr, sum = 0;
    printf("Enter number of elements: ");
    scanf("%d", &n);

    ptr = (int*) calloc(n, sizeof(int));
    if(ptr == NULL) {
        printf("Error! memory not allocated.");
        exit(0);
    }

    printf("Enter elements: ");
    for(i = 0; i < n; ++i) {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }

    printf("Sum = %d", sum);
    free(ptr);
    return 0;
}
```

Output

```
Enter number of elements: 3
```

```
Enter elements: 100
```

```
20
```

```
36
```

```
Sum = 156
```

C realloc()

If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using the `realloc()` function.

Syntax of realloc()

```
ptr = realloc(ptr, x);
```

Here, `ptr` is reallocated with a new size `x`.

Example 3: realloc()

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

int main() {
    int *ptr, i, n1, n2;
    printf("Enter size: ");
    scanf("%d", &n1);

    ptr = (int*) malloc(n1 * sizeof(int));

    printf("Addresses of previously allocated memory:\n");
    for(i = 0; i < n1; ++i)
        printf("%pc\n", ptr + i);

    printf("\nEnter the new size: ");
    scanf("%d", &n2);

    // relocating the memory
    ptr = realloc(ptr, n2 * sizeof(int));

    printf("Addresses of newly allocated memory:\n");
```

```
for(i = 0; i < n2; ++i)
    printf("%pc\n", ptr + i);

free(ptr);

return 0;
}
```

Output

Enter size: 2

Addresses of previously allocated memory:

26855472

26855476

Enter the new size: 4

Addresses of newly allocated memory:

26855472

26855476

26855480

26855484

STRINGS:

In C programming, a string is a sequence of characters terminated with a null character `\0`. For example:

```
char c[] = "c string";
```

When the compiler encounters a sequence of characters enclosed in the double quotation marks, it appends a null character `\0` at the end by default.

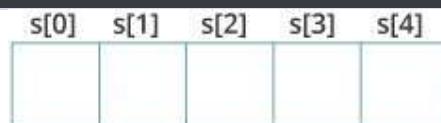


Memory Diagram

How to declare a string?

Here's how you can declare strings:

```
char s[5];
```



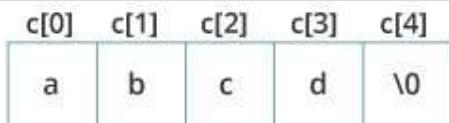
String Declaration in C

Here, we have declared a string of 5 characters.

How to initialize strings?

You can initialize strings in a number of ways.

```
char c[] = "abcd";  
char c[50] = "abcd";  
char c[] = {'a', 'b', 'c', 'd', '\0'};  
char c[5] = {'a', 'b', 'c', 'd', '\0'};
```



String Initialization in C

Let's take another example:

```
char c[5] = "abcde";
```

Here, we are trying to assign 6 characters (the last character is '\0') to a `char` array having 5 characters. This is bad and you should never do this.

Assigning Values to Strings

Arrays and strings are second-class citizens in C; they do not support the assignment operator once it is declared. For example,

```
char c[100];
c = "C programming"; // Error! array type is not assignable.
```

Note: Use the `strcpy()` function to copy the string instead.

Read String from the user

You can use the `scanf()` function to read a string.

The `scanf()` function reads the sequence of characters until it encounters whitespace (space, newline, tab, etc.).

Example 1: `scanf()` to read a string

```
#include <stdio.h>
int main()
{
    char name[20];
    printf("Enter name: ");
    scanf("%s", name);
    printf("Your name is %s.", name);
    return 0;
}
```

Output

```
Enter name: Dennis Ritchie
Your name is Dennis.
```

Even though Dennis Ritchie was entered in the above program, only "Dennis" was stored in the name string. It's because there was a space after Dennis.
Also notice that we have used the code name instead of &name with scanf().

```
scanf("%s", name);
```

This is because name is a char array, and we know that array names decay to pointers in C.

Thus, the name in scanf() already points to the address of the first element in the string, which is why we don't need to use &.

How to read a line of text?

You can use the fgets() function to read a line of string. And, you can use puts() to display the string.

Example 2: fgets() and puts()

```
#include <stdio.h>
int main()
{
    char name[30];
    printf("Enter name: ");
    fgets(name, sizeof(name), stdin); // read string
    printf("Name: ");
    puts(name); // display string
    return 0;
}
```

Output

```
Enter name: Tom Hanks
Name: Tom Hanks
```

Here, we have used fgets() function to read a string from the user.

```
fgets(name, sizeof(name), stdin); // read string
```

The `sizeof(name)` results to 30. Hence, we can take a maximum of 30 characters as input which is the size of the `name` string.

To print the string, we have used `puts(name);`

Note: The `gets()` function can also be used to take input from the user. However, it is removed from the C standard.

It's because `gets()` allows you to input any length of characters. Hence, there might be a buffer overflow.

Example:

In order to read a string containing spaces, we use the `gets()` function. `Gets` ignores the whitespaces. It stops reading when a newline is reached (the Enter key is pressed).

For example:

```
#include <stdio.h>
int main() {
    char full_name[25];
    printf("Enter your full name: ");
    gets(full_name);
    printf("My full name is %s ",full_name);
    return 0;
}
```

Output:

Enter your full name: Dennis Ritchie

My full name is Dennis Ritchie

Passing Strings to Functions

Strings can be passed to a function in a similar way as arrays. Learn more about passing arrays to a function.

Example 3: Passing string to a Function

```

#include <stdio.h>
void displayString(char str[]);

int main()
{
    char str[50];
    printf("Enter string: ");
    fgets(str, sizeof(str), stdin);
    displayString(str); // Passing string to a function.
    return 0;
}

void displayString(char str[])
{
    printf("String Output: ");
    puts(str);
}

```

Strings and Pointers

Similar like arrays, string names are "decayed" to pointers. Hence, you can use pointers to manipulate elements of the string. We recommended you to check C Arrays and Pointers before you check this example.

Example 4: Strings and Pointers

```

#include <stdio.h>

int main(void) {
    char name[] = "Harry Potter";

    printf("%c", *name); // Output: H
    printf("%c", *(name+1)); // Output: a
    printf("%c", *(name+7)); // Output: o

```

```

char *namePtr;

namePtr = name;
printf("%c", *namePtr);    // Output: H
printf("%c", *(namePtr+1)); // Output: a
printf("%c", *(namePtr+7)); // Output: o
}

```

Commonly Used String Functions

- **strlen()** - calculates the length of a string
- **strcpy()** - copies a string to another
- **strcmp()** - compares two strings
- **strcat()** - concatenates two strings

String Manipulations In C Programming Using Library Functions

You need to often manipulate strings according to the need of a problem. Most, if not all, of the time string manipulation can be done manually but, this makes programming complex and large.

To solve this, C supports a large number of string handling functions in the standard library "string.h".

Few commonly used string handling functions are discussed below:

Function	Work of Function
<u>strlen()</u>	computes string's length
<u>strcpy()</u>	copies a string to another
<u>strcat()</u>	concatenates(joins) two strings

Function	Work of Function
<u>strcmp()</u>	compares two strings
<u>strlwr()</u>	converts string to lowercase
<u>strupr()</u>	converts string to uppercase

Strings handling functions are defined under "string.h" header file.

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

Note: You have to include the code below to run string handling functions.

[gets\(\)](#) and [puts\(\)](#)

Functions gets() and puts() are two string functions to take string input from the user and display it respectively.

```
#include<stdio.h>

int main()
{
    char name[30];
    printf("Enter name: ");
    gets(name);    //Function to read string from user.
    printf("Name: ");
    puts(name);   //Function to display string.
    return 0;
}
```

Note: Though, [gets\(\)](#) and [puts\(\)](#) function handle strings, both these functions are defined in "stdio.h" header file.

C strlen()

The strlen() function calculates the length of a given string.

The strlen() function takes a string as an argument and returns its length. The returned value is of type size_t (the unsigned integer type).

It is defined in the <string.h> header file.

Example: C strlen() function

```
#include <stdio.h>
#include <string.h>
int main()
{
    char a[20] = "Program";
    char b[20] = {'P', 'r', 'o', 'g', 'r', 'a', 'm', '\0'};

    // using the %zu format specifier to print size_t
    printf("Length of string a = %zu \n", strlen(a));
    printf("Length of string b = %zu \n", strlen(b));

    return 0;
}
```

Output

```
Length of string a = 7
Length of string b = 7
```

Note that the `strlen()` function doesn't count the null character `\0` while calculating the length.

C strcpy()

Here we see the use of the `strcpy()` function in C programming to copy strings (with the help of an example).

C strcpy()

The function prototype of `strcpy()` is:

```
char* strcpy(char* destination, const char* source);
```

- The `strcpy()` function copies the string pointed by `source` (including the null character) to the `destination`.
- The `strcpy()` function also returns the copied string.

The `strcpy()` function is defined in the `string.h` header file.

Example: C strcpy()

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

int main() {
    char str1[20] = "C programming";
    char str2[20];

    // copying str1 to str2
    strcpy(str2, str1);

    puts(str2); // C programming

    return 0;
}
```

Output

C programming

Note: When you use `strcpy()`, the size of the destination string should be large enough to store the copied string. Otherwise, it may result in **undefined behavior**.

C strcat()

In C programming, the `strcat()` function concatenates (joins) two strings.

The function definition of `strcat()` is:

```
char *strcat(char *destination, const char *source)
```

It is defined in the `string.h` header file.

strcat() arguments

As you can see, the `strcat()` function takes two arguments:

destination - destination string

source - source string

The `strcat()` function concatenates the `destination` string and the `source` string, and the result is stored in the `destination` string.

Example: C `strcat()` function

```
#include <stdio.h>
#include <string.h>
int main() {
    char str1[100] = "This is ", str2[] = "google.com";

    // concatenates str1 and str2
    // the resultant string is stored in str1.
    strcat(str1, str2);

    puts(str1);
    puts(str2);

    return 0;
}
```

Output

```
This is google.com
google.com
```

Note: When we use `strcat()`, the size of the destination string should be large enough to store the resultant string. If not, we will get the segmentation fault error.

C `strcmp()`

We will see how to compare two strings using the `strcmp()` function.

The `strcmp()` compares two strings character by character. If the strings are equal, the function returns 0.

C `strcmp()` Prototype

The function prototype of `strcmp()` is:

```
int strcmp (const char* str1, const char* str2);
```

strcmp() Parameters

The function takes two parameters:

- **str1** - a string
- **str2** - a string

Return Value from strcmp()

Return Value	Remarks
0	if strings are equal
>0	if the first non-matching character in str1 is greater (in ASCII) than that of str2.
<0	if the first non-matching character in str1 is lower (in ASCII) than that of str2.

The `strcmp()` function is defined in the `string.h` header file.

Example: C strcmp() function

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

int main() {
    char str1[] = "abcd", str2[] = "abCd", str3[] = "abcd";
    int result;

    // comparing strings str1 and str2
    result = strcmp(str1, str2);
    printf("strcmp(str1, str2) = %d\n", result);

    // comparing strings str1 and str3
}
```

```
result = strcmp(str1, str3);
printf("strcmp(str1, str3) = %d\n", result);

return 0;
}
```

Output

```
strcmp(str1, str2) = 1
strcmp(str1, str3) = 0
```

In the program,

- strings `str1` and `str2` are not equal. Hence, the result is a non-zero integer.
- strings `str1` and `str3` are equal. Hence, the result is 0.

C Struct

Arrays are used to store large set of data and manipulate them but the disadvantage is that all the elements stored in an array are to be of the same data type. If we need to use a collection of different data type items it is not possible using an array. When we require using a collection of different data items of different data types, we can use a structure. Structure is a method of packing data of different types. A structure is a convenient method of handling a group of related data items of different data types.

In C programming, a struct (or structure) is a collection of variables (can be of different types) under a single name.

Define Structures

Before you can create structure variables, you need to define its data type. To define a struct, the `struct` keyword is used.

Syntax of struct

```
struct structureName {  
    dataType member1;  
    dataType member2;  
    ...  
};
```

For example,

```
struct Person {  
    char name[50];  
    int citNo; //Citizenship No.  
    float salary;  
};
```

Here, a derived type `struct Person` is defined. Now, you can create variables of this type.

```
struct Books {
```

```
char title[50];
char author[50];
char subject[100];
int book_id;
} book;
```

Example:

```
struct lib_books {
    char title[20];
    char author[15];
    int pages;
    float price;
};
```

the keyword struct declares a structure to holds the details of four fields namely title, author pages and price. These are members of the structures. Each member may belong to different or same data type. The tag name can be used to define objects that have the tag name's structure. The structure we just declared is not a variable by itself but a template for the structure.

We can declare structure variables using the tag name anywhere in the program. For example, the statement,

```
struct lib_books book1, book2, book3;
```

declares book1, book2, book3 as variables of type struct lib_books each declaration has four elements of the structure lib_books. The complete structure declaration might look like this

```
struct lib_books {
    char title[20];
    char author[15];
    int pages;
    float price;
};
```

```
struct lib_books, book1, book2, book3;
```

structures do not occupy any memory until it is associated with the structure variable such as book1.

Create struct Variables

When a `struct` type is declared, no storage or memory is allocated. To allocate memory of a given structure type and work with it, we need to create variables.

Here's how we create structure variables:

```
struct Person {  
    // code  
};  
  
int main() {  
    struct Person person1, person2, p[20];  
    return 0;  
}
```

Another way of creating a `struct` variable is:

```
struct Person {  
    // code  
} person1, person2, p[20];
```

In both cases,

- `person1` and `person2` are `struct Person` variables
- `p[]` is a `struct Person` array of size 20.

Access Members of a Structure

There are two types of operators used for accessing members of a structure.

1. `.` - Member operator
2. `->` - Structure pointer operator

Suppose, you want to access the `salary` of `person2`. Here's how you can do it.

```
person2.salary
```

Example 1: C structs

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

// create struct with person1 variable
struct Person {
    char name[50];
    int citNo;
    float salary;
} person1;

int main() {

    // assign value to name of person1
    strcpy(person1.name, "George Orwell");

    // assign values to other person1 variables
    person1.citNo = 1984;
    person1.salary = 2500;

    // print struct variables
    printf("Name: %s\n", person1.name);
    printf("Citizenship No.: %d\n", person1.citNo);
    printf("Salary: %.2f", person1.salary);

    return 0;
}
```

Output

```
Name: George Orwell
Citizenship No.: 1984
Salary: 2500.00
```

In this program, we have created a `struct` named `Person`. We have also created a variable of `Person` named `person1`.

In `main()`, we have assigned values to the variables defined in `Person` for the `person1` object.

```
strcpy(person1.name, "George Orwell");
person1.citNo = 1984;
```

```
person1.salary = 2500;
```

Notice that we have used `strcpy()` function to assign the value to `person1.name`.

This is because `name` is a `char` array (C-string) and we cannot use the assignment operator `=` with it after we have declared the string.

Finally, we printed the data of `person1`.

Example 2:

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

struct Books {
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
};

int main( ) {

    struct Books Book1;      /* Declare Book1 of type Book */
    struct Books Book2;      /* Declare Book2 of type Book */

    /* book 1 specification */
    strcpy( Book1.title, "C Programming");
    strcpy( Book1.author, "Nuha Ali");
    strcpy( Book1.subject, "C Programming Tutorial");
    Book1.book_id = 6495407;

    /* book 2 specification */
    strcpy( Book2.title, "Telecom Billing");
    strcpy( Book2.author, "Zara Ali");
    strcpy( Book2.subject, "Telecom Billing Tutorial");
    Book2.book_id = 6495700;
```

```

/* print Book1 info */
printf( "Book 1 title : %s\n", Book1.title);
printf( "Book 1 author : %s\n", Book1.author);
printf( "Book 1 subject : %s\n", Book1.subject);
printf( "Book 1 book_id : %d\n", Book1.book_id);

/* print Book2 info */
printf( "Book 2 title : %s\n", Book2.title);
printf( "Book 2 author : %s\n", Book2.author);
printf( "Book 2 subject : %s\n", Book2.subject);
printf( "Book 2 book_id : %d\n", Book2.book_id);

return 0;
}

```

When the above code is compiled and executed, it produces the following result –

```

Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700

```

Keyword `typedef`

We use the `typedef` keyword to create an alias name for data types. It is commonly used with structures to simplify the syntax of declaring variables.

For example, let us look at the following code:

```

struct Distance{
    int feet;
    float inch;
};

```

```
int main() {
    struct Distance d1, d2;
}
```

We can use `typedef` to write an equivalent code with a simplified syntax:

```
typedef struct Distance {
    int feet;
    float inch;
} distances;

int main() {
    distances d1, d2;
}
```

Example 2: C++ typedef

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

// struct with typedef person
typedef struct Person {
    char name[50];
    int citNo;
    float salary;
} person;

int main() {

    // create Person variable
    person p1;

    // assign value to name of p1
    strcpy(p1.name, "George Orwell");

    // assign values to other p1 variables
    p1.citNo = 1984;
    p1.salary = 2500;

    // print struct variables
    printf("Name: %s\n", p1.name);
    printf("Citizenship No.: %d\n", p1.citNo);
    printf("Salary: %.2f", p1.salary);

    return 0;
}
```

```
}
```

Output

```
Name: George Orwell  
Citizenship No.: 1984  
Salary: 2500.00
```

Here, we have used `typedef` with the `Person` structure to create an alias `person`.

```
// struct with typedef person  
typedef struct Person {  
    // code  
} person;
```

Now, we can simply declare a `Person` variable using the `person` alias:

```
// equivalent to struct Person p1  
person p1;
```

Example:

```
//program demo using typedef structures  
#include <stdio.h>  
#include <string.h>
```

```
typedef struct Books {  
    char title[50];  
    char author[50];  
    char subject[100];  
    int book_id;  
} Book1;
```

```
int main( ) {  
    Book1 b1;  
  
    /* book 1 specification */
```

```

strcpy( b1.title, "C Programming");
strcpy( b1.author, "Nuha Ali");
strcpy( b1.subject, "C Programming Tutorial");
b1.book_id = 6495407;

/* print Book1 info */

printf( "Book 1 title : %s\n", b1.title);
printf( "Book 1 author : %s\n", b1.author);
printf( "Book 1 subject : %s\n", b1.subject);
printf( "Book 1 book_id : %d\n", b1.book_id);

return 0;
}

```

Nested Structures

You can create structures within a structure in C programming. For example,

```

struct complex {
    int imag;
    float real;
};

struct number {
    struct complex comp;
    int integers;
} num1, num2;

```

Suppose, you want to set `imag` of `num2` variable to **11**. Here's how you can do it:

```
num2.comp.imag = 11;
```

Example 3: C++ Nested Structures

```
#include <stdio.h>

struct complex {
    int imag;
    float real;
};

struct number {
    struct complex comp;
    int integer;
} num1;

int main() {

    // initialize complex variables
    num1.comp.imag = 11;
    num1.comp.real = 5.25;

    // initialize number variable
    num1.integer = 6;

    // print struct variables
    printf("Imaginary Part: %d\n", num1.comp.imag);
    printf("Real Part: %.2f\n", num1.comp.real);
    printf("Integer: %d", num1.integer);

    return 0;
}
```

Output

```
Imaginary Part: 11
Real Part: 5.25
```

Integer: 6

Why structs in C?

Suppose, you want to store information about a person: his/her name, citizenship number, and salary. You can create different variables `name`, `citNo` and `salary` to store this information.

What if you need to store information of more than one person? Now, you need to create different variables for each information per person: `name1`, `citNo1`, `salary1`, `name2`, `citNo2`, `salary2`, etc.

A better approach would be to have a collection of all related information under a single name `Person` structure and use it for every person.

Structures as Function Arguments

You can pass a structure as a function argument in the same way as you pass any other variable or pointer.

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

struct Books {
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
};

/* function declaration */
void printBook( struct Books book );

int main( ) {
    struct Books Book1; /* Declare Book1 of type Book */
}
```

```

struct Books Book2;      /* Declare Book2 of type Book */

/* book 1 specification */
strcpy( Book1.title, "C Programming");
strcpy( Book1.author, "Nuha Ali");
strcpy( Book1.subject, "C Programming Tutorial");
Book1.book_id = 6495407;

/* book 2 specification */
strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Zara Ali");
strcpy( Book2.subject, "Telecom Billing Tutorial");
Book2.book_id = 6495700;

/* print Book1 info */
printBook( Book1 );

/* Print Book2 info */
printBook( Book2 );

return 0;
}

void printBook( struct Books book ) {

    printf( "Book title : %s\n", book.title);
    printf( "Book author : %s\n", book.author);
    printf( "Book subject : %s\n", book.subject);
    printf( "Book book_id : %d\n", book.book_id);
}

```

When the above code is compiled and executed, it produces the following result –

Book title : C Programming

Book author : Nuha Ali

```
Book subject : C Programming Tutorial  
Book book_id : 6495407  
Book title : Telecom Billing  
Book author : Zara Ali  
Book subject : Telecom Billing Tutorial  
Book book_id : 6495700
```

C Pointers to struct

Here's how you can create pointers to structs.

```
struct name {  
    member1;  
    member2;  
    .  
    .  
};  
  
int main()  
{  
    struct name *ptr, Harry;  
}
```

Here, `ptr` is a pointer to `struct`.

Example: Access members using Pointer

To access members of a structure using pointers, we use the `->` operator.

```
#include <stdio.h>  
struct person  
{  
    int age;  
    float weight;  
};  
  
int main()  
{
```

```

struct person *personPtr, person1;
personPtr = &person1;

printf("Enter age: ");
scanf("%d", &personPtr->age);

printf("Enter weight: ");
scanf("%f", &personPtr->weight);

printf("Displaying:\n");
printf("Age: %d\n", personPtr->age);
printf("weight: %f", personPtr->weight);

return 0;
}

```

In this example, the address of `person1` is stored in the `personPtr` pointer using `personPtr = &person1;`.

Now, you can access the members of `person1` using the `personPtr` pointer. By the way,

- `personPtr->age` is equivalent to `(*personPtr).age`
- `personPtr->weight` is equivalent to `(*personPtr).weight`

Passing structs to functions

Here's how you can pass structures to a function

```

#include <stdio.h>
struct student {
    char name[50];
    int age;
};

```

```
// function prototype  
void display(struct student s);  
  
int main() {  
    struct student s1;  
  
    printf("Enter name: ");  
  
    // read string input from the user until \n is entered  
    // \n is discarded  
    scanf("%[^\\n]*c", s1.name);  
  
    printf("Enter age: ");  
    scanf("%d", &s1.age);  
  
    display(s1); // passing struct as an argument  
  
    return 0;  
}
```

```
void display(struct student s) {  
    printf("\nDisplaying information\n");  
    printf("Name: %s", s.name);  
    printf("\nAge: %d", s.age);  
}
```

Output

```
Enter name: Bond
```

```
Enter age: 13
```

```
Displaying information
```

```
Name: Bond
```

```
Age: 13
```

Here, a struct variable `s1` of type `struct student` is created. The variable is passed to the `display()` function using `display(s1);` statement.

Return struct from a function

Here's how you can return structure from a function:

```
#include <stdio.h>
struct student
{
    char name[50];
    int age;
};

// function prototype
struct student getInformation();

int main()
{
    struct student s;

    s = getInformation();

    printf("\nDisplaying information\n");
    printf("Name: %s", s.name);
    printf("\nRoll: %d", s.age);

    return 0;
}
struct student getInformation()
{
    struct student s1;

    printf("Enter name: ");
    scanf ("%[^\\n]*c", s1.name);

    printf("Enter age: ");
    scanf("%d", &s1.age);

    return s1;
}
```

Here, the `getInformation()` function is called using `s = getInformation();` statement. The function returns a structure of type `struct student`. The returned structure is displayed from the `main()` function.

Notice that, the return type of `getInformation()` is also `struct student`.

C Union:

A union is a user-defined type similar to structs in C except for one key difference. Structures allocate enough space to store all their members, whereas **unions can only hold one member value at a time.**

Unions like structure contain members whose individual data types may differ from one another. However the members that compose a union all share the same storage area within the computer's memory whereas each member within a structure is assigned its own unique storage area. Thus, unions are used to observe memory. They are useful for application involving multiple members. Where values need not be assigned to all the members at any one time.

How to define a union?

We use the `union` keyword to define unions. Here's an example:

```
union car
{
    char name[50];
    int price;
};
```

The above code defines a derived type `union car`.

Create union variables

When a union is defined, it creates a user-defined type. However, no memory is allocated. To allocate memory for a given union type and work with it, we need to create variables.

Here's how we create union variables.

```
union car
{
    char name[50];
    int price;
};

int main()
{
    union car car1, car2, *car3;
    return 0;
}
```

Another way of creating union variables is:

```
union car
{
    char name[50];
    int price;
} car1, car2, *car3;
```

In both cases, union variables `car1`, `car2`, and a union pointer `car3` of `union car` type are created.

Access members of a union

We use the `.` operator to access members of a union. And to access pointer variables, we use the `->` operator.

In the above example,

- To access `price` for `car1`, `car1.price` is used.
- To access `price` using `car3`, either `(*car3).price` or `car3->price` can be used.

Difference between unions and structures

Let's take an example to demonstrate the difference between unions and structures:

```
#include <stdio.h>
union unionJob
{
    //defining a union
    char name[32];
    float salary;
    int workerNo;
} uJob;

struct structJob
{
    char name[32];
    float salary;
    int workerNo;
} sJob;

int main()
{
    printf("size of union = %d bytes", sizeof(uJob));
    printf("\nsize of structure = %d bytes", sizeof(sJob));
    return 0;
}
```

Output

```
size of union = 32
size of structure = 40
```

With a union, all members share **the same memory**.

Example: Accessing Union Members

```

#include <stdio.h>
union Job {
    float salary;
    int workerNo;
} j;

int main() {
    j.salary = 12.3;

    // when j.workerNo is assigned a value,
    // j.salary will no longer hold 12.3
    j.workerNo = 100;

    printf("Salary = %.1f\n", j.salary);
    printf("Number of workers = %d", j.workerNo);
    return 0;
}

```

Output

```

Salary = 0.0
Number of workers = 100

```

Differences between structure and union:

Structure	Union
You can use a struct keyword to define a structure.	You can use a union keyword to define a union.
Every member within structure is assigned a unique memory location.	In union, a memory location is shared by all the data members.
Changing the value of one data member will not affect other data members in structure.	Changing the value of one data member will change the value of other data members in union.

It enables you to initialize several members at once.	It enables you to initialize only the first member of union.
The total size of the structure is the sum of the size of every data member.	The total size of the union is the size of the largest data member.
It is mainly used for storing various data types.	It is mainly used for storing one of the many data types that are available.
It occupies space for each and every member written in inner parameters.	It occupies space for a member having the highest size written in inner parameters.
You can retrieve any member at a time.	You can access one member at a time in the union.
It supports flexible array.	It does not support a flexible array.

Advantages of structure

Here are pros/benefits for using structure:

- Structures gather more than one piece of data about the same subject together in the same place.
- It is helpful when you want to gather the data of similar data types and parameters like first name, last name, etc.
- It is very easy to maintain as we can represent the whole record by using a single name.
- In structure, we can pass complete set of records to any function using a single parameter.
- You can use an array of structure to store more records with similar types.

Advantages of union

Here, are pros/benefits for using union:

- It occupies less memory compared to structure.
- When you use union, only the last variable can be directly accessed.
- Union is used when you have to use the same memory location for two or more data members.
- It enables you to hold data of only one data member.
- Its allocated space is equal to maximum size of the data member.

Disadvantages of structure

Here are cons/drawbacks for using structure:

- If the complexity of IT project goes beyond the limit, it becomes hard to manage.
- Change of one data structure in a code necessitates changes at many other places. Therefore, the changes become hard to track.
- Structure is slower because it requires storage space for all the data.
- You can retrieve any member at a time in structure whereas you can access one member at a time in the union.
- Structure occupies space for each and every member written in inner parameters while union occupies space for a member having the highest size written in inner parameters.
- Structure supports flexible array. Union does not support a flexible array.

Disadvantages of union

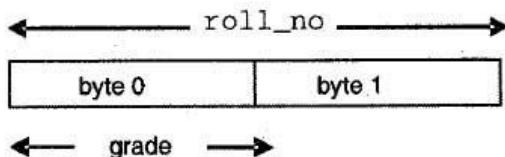
Here, are cons/drawbacks for using union:

- You can use only one union member at a time.
- All the union variables cannot be initialized or used with varying values at a time.
- Union assigns one common storage space for all its members.

Both the structures and unions are syntactically and functionally same, however, they differ in the way memory is allocated to their members. While declaring structure variables, the different members are stored in different, although, adjacent memory locations whereas different members of a union variable share the same memory location. The amount of memory sufficient to hold the largest member of a union is allocated to a union variable. Thus, a union enables the same block of memory to store variable of one type at one time and of different type at another time. To understand the concept of memory allocation to a union, consider these statements.

```
union stu //union definition {  
    int roll_no;      //occupies 2 bytes  
    char grade;      // occupies 1 byte  
}stu1;           //union variable declaration
```

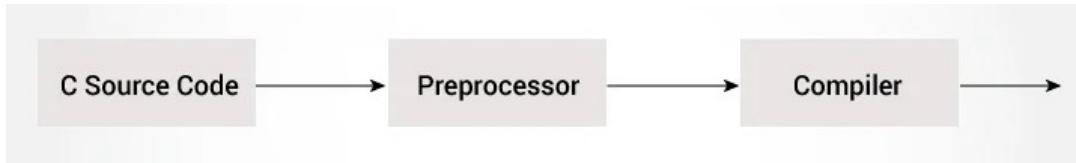
In these statements, a union stu is defined, which enables members of type int and char to share same space in the memory. However, if roll_no and grade belong to a structure, then they are stored in separate memory locations. For example, in this case, the compiler allocates 2 bytes to stu1, while in a structure 3 bytes will be allocated. The memory allocation of union members is shown in Figure



Memory Representation of stu1

As shown in Figure, same memory location is used for roll_no and grade members of union variable stu1. This means the same memory location can represent different values at different points of time. Hence, unions help in conserving memory space and are useful in applications where values need not be assigned to all the members at one time.

C Preprocessors



The C preprocessor is a macro preprocessor (allows you to define macros) that transforms your program before it is compiled. These transformations can be the inclusion of header file, macro expansions etc.

All preprocessing directives begin with a `#` symbol. For example,

```
#define PI 3.14
```

Some of the common uses of C preprocessor are:

Including Header Files: `#include`

The `#include` preprocessor is used to include header files to C programs. For example,

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

Here, `stdio.h` is a header file. The `#include` preprocessor directive replaces the above line with the contents of `stdio.h` header file.

That's the reason why you need to use `#include <stdio.h>` before you can use functions like `scanf()` and `printf()`.

You can also create your own header file containing function declaration and include it in your program using this preprocessor directive.

```
#include "my_header.h"
```

The following section lists down all the important preprocessor directives –

Sr.No.	Directive & Description
1	#define Substitutes a preprocessor macro.
2	#include Inserts a particular header from another file.
3	#undef Undefines a preprocessor macro.
4	#ifdef Returns true if this macro is defined.
5	#ifndef Returns true if this macro is not defined.
6	#if Tests if a compile time condition is true.
7	#else The alternative for #if.
8	#elif #else and #if in one statement.
9	#endif Ends preprocessor conditional.
10	#error Prints error message on stderr.
11	#pragma Issues special commands to the compiler, using a standardized method.

Macros using #define

A macro is a fragment of code that is given a name. You can define a macro in C using the `#define` preprocessor directive.

Here's an example.

```
#define c 299792458 // speed of light
```

Here, when we use `c` in our program, it is replaced with `299792458`.

Example 1: #define preprocessor

```
#include <stdio.h>
#define PI 3.1415

int main()
{
    float radius, area;
    printf("Enter the radius: ");
    scanf("%f", &radius);

    // Notice, the use of PI
    area = PI*radius*radius;

    printf("Area=%.2f",area);
    return 0;
}
```

Function like Macros

You can also define macros that work in a similar way like a function call. This is known as function-like macros. For example,

```
#define circleArea(r) (3.1415*(r)*(r))
```

Every time the program encounters `circleArea(argument)`, it is replaced by `(3.1415*(argument)*(argument))`.

Suppose, we passed 5 as an argument then, it expands as below:

circleArea(5) expands to (3.1415*5*5)

Example 2: Using #define preprocessor

```
#include <stdio.h>
#define PI 3.1415
#define circleArea(r) (PI*r*r)

int main() {
    float radius, area;

    printf("Enter the radius: ");
    scanf("%f", &radius);
    area = circleArea(radius);
    printf("Area = %.2f", area);

    return 0;
}
```

Conditional Compilation

In C programming, you can instruct preprocessor whether to include a block of code or not. To do so, conditional directives can be used.

It's similar to a if statement with one major difference.

The if statement is tested during the execution time to check whether a block of code should be executed or not whereas, the conditionals are used to include (or skip) a block of code in your program before execution.

Uses of Conditional

use different code depending on the machine, operating system

compile same source file in two different programs

to exclude certain code from the program but to keep it as reference for future purpose

How to use conditional?

To use conditional, `#ifdef`, `#if`, `#defined`, `#else` and `#elif` directives are used.

Preprocessors Examples

Analyze the following examples to understand various directives.

```
#define MAX_ARRAY_LENGTH 20
```

This directive tells the CPP to replace instances of MAX_ARRAY_LENGTH with 20.

Use `#define` for constants to increase readability.

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

These directives tell the CPP to get stdio.h from **System Libraries** and add the text to the current source file. The next line tells CPP to get **myheader.h** from the local directory and add the content to the current source file.

```
#undef FILE_SIZE
#define FILE_SIZE 42
```

It tells the CPP to undefine existing FILE_SIZE and define it as 42.

```
#ifndef MESSAGE
#define MESSAGE "You wish!"
#endif
```

It tells the CPP to define MESSAGE only if MESSAGE isn't already defined.

```
#ifdef DEBUG
/* Your debugging statements here */
#endif
```

It tells the CPP to process the statements enclosed if DEBUG is defined. This is useful if you pass the `-DDEBUG` flag to the gcc compiler at the time of compilation. This will define DEBUG, so you can turn debugging on and off on the fly during compilation.

#ifdef Directive

```
#ifdef MACRO
    // conditional codes
#endif
```

Here, the conditional codes are included in the program only if `MACRO` is defined.

#if, #elif and #else Directive

```
#if expression
    // conditional codes
#endif
```

Here, `expression` is an expression of integer type (can be integers, characters, arithmetic expression, macros and so on).

The conditional codes are included in the program only if the `expression` is evaluated to a non-zero value.

The optional `#else` directive can be used with `#if` directive.

```
#if expression
    conditional codes if expression is non-zero
#else
    conditional if expression is 0
#endif
```

You can also add nested conditional to your `#if...#else` using `#elif`

```
#if expression
    // conditional codes if expression is non-zero
#elif expression1
    // conditional codes if expression is non-zero
#elif expression2
    // conditional codes if expression is non-zero
#else
    // conditional if all expressions are 0
#endif
```

#defined

The special operator `#defined` is used to test whether a certain macro is defined or not. It's often used with `#if` directive.

```
#if defined BUFFER_SIZE && BUFFER_SIZE >= 2048  
// codes
```

Predefined Macros

Here are some predefined macros in C programming.

Macro	Value
<code>__DATE__</code>	A string containing the current date
<code>__FILE__</code>	A string containing the file name
<code>__LINE__</code>	An integer representing the current line number
<code>__STDC__</code>	If follows ANSI standard C, then the value is a nonzero integer Defined as 1 when the compiler complies with the ANSI standard.
<code>__TIME__</code>	A string containing the current date.

Example 3: Get current time using `__TIME__`

The following program outputs the current time using `__TIME__` macro.

```
#include <stdio.h>  
int main()  
{  
    printf("Current time: %s", __TIME__);  
}
```

Output

Current time: 19:54:39

```
#include <stdio.h>

int main() {

    printf("File :%s\n", __FILE__ );
    printf("Date :%s\n", __DATE__ );
    printf("Time :%s\n", __TIME__ );
    printf("Line :%d\n", __LINE__ );
    printf("ANSI :%d\n", __STDC__ );

}
```

When the above code in a file **test.c** is compiled and executed, it produces the following result –

```
File :test.c
Date :Jun 2 2012
Time :03:36:24
Line :8
ANSI :1
```

Nesting of Macros in C

A macro may be used in the definition of another macro as illustrated below.

```
#define A(x) x*x

#define cost(A,y) A*y
```

Program illustrates this concept. In this program, Area is a function of x, and the cost of painting the area is defined as a function of area in the macro definition.

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

```
#define Area(x) x*x
#define Costpaint(x,y,z) (z*y + Area (x))
void main()
{
    int A = 8, B= 6, C = 4;
    clrscr();
    printf("The area of square= %d\n", Area(A));
    printf("Cost of paint= %d\n", Costpaint(A,B,C));
}
```

Command Line Arguments in C

The arguments passed from command line are called command line arguments. These arguments are handled by main() function.

To support command line argument, you need to change the structure of main() function as given below.

```
int main(int argc, char *argv[] )
```

Here, **argc** counts the number of arguments. It counts the file name as the first argument.

The **argv[]** contains the total number of arguments. The first argument is the file name always.

Remember that **argv[0]** holds the name of the program and **argv[1]** points to the first command line argument and **argv[n]** gives the last argument. If no argument is supplied, **argc** will be 1.

Example

```
#include <stdio.h>

#include <conio.h>

int main(int argc, char *argv[])
{
    int i;
    if( argc >= 2 )
    {
        printf("The arguments supplied are:\n");
        for(i = 1; i < argc; i++)
        {
            printf("%s\t", argv[i]);
        }
    }
}
```

```

}

else

{

    printf("argument list is empty.\n");

}

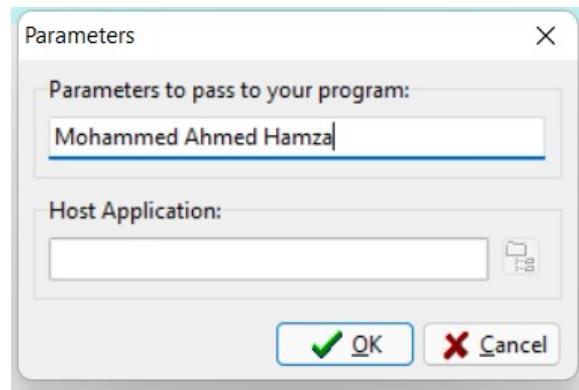
return 0;

}

```

Steps:

1. Compile
2. Execute -> Click on Parameters



Example 2:

```

// C program to illustrate
// command line arguments
#include<stdio.h>

int main(int argc,char* argv[])
{
    int counter;

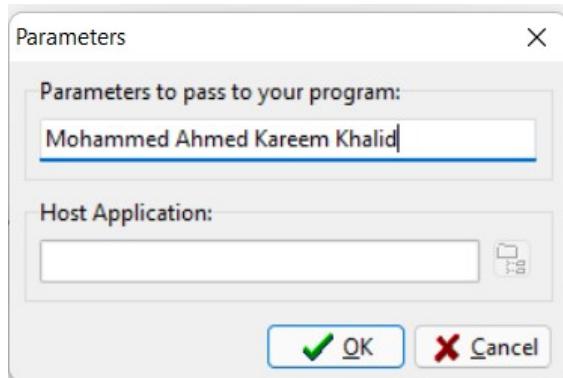
```

```

printf("Program Name Is: %s",argv[0]);
if(argc==1)
    printf("\nNo Extra Command Line Argument Passed Other Than Program
Name");
if(argc>=2)
{
    printf("\nNumber Of Arguments Passed: %d",argc);
    printf("\n----Following Are The Command Line Arguments Passed----");
    for(counter=0;counter<argc;counter++)
        printf("\nargv[%d]: %s",counter,argv[counter]);
}
return 0;
}

```

Save the above program as cmdline.c



Output:

Program Name Is: D:\cmdline.exe

Number Of Arguments Passed: 5

----Following Are The Command Line Arguments Passed----

argv[0]: D:\cmdline.exe

argv[1]: Mohammed

argv[2]: Ahmed

argv[3]: Kareem

argv[4]: Khalid

Process exited after 0.03448 seconds with return value 0

Press any key to continue . . .

Let's see the example of command line arguments where we are passing one argument with file name.

```
//program demo reading only first argument
```

```
#include <stdio.h>
void main(int argc, char *argv[] ) {

    printf("Program name is: %s\n", argv[0]);

    if(argc < 2){
        printf("No argument passed through command line.\n");
    }
    else{
        printf("First argument is: %s\n", argv[1]);
    }
}
```

Run this program as follows in Linux:

```
./program hello
```

Run this program as follows in Windows from command line:

```
program.exe hello
```

Output:

```
Program name is: program
```

```
First argument is: hello
```

If you pass many arguments, it will print only one.

```
./program hello c how r u
```

Output:

```
Program name is: program  
First argument is: hello
```

But if you pass many arguments within double quote, all arguments will be treated as a single argument only.

```
./program "hello c how r u"
```

Output:

```
Program name is: program  
First argument is: hello c how r u
```

You can write your program to print all the arguments. In this program, we are printing only argv[1], that is why it is printing only one argument.

```
#include <stdio.h>  
  
int main( int argc, char *argv[] ) {  
  
    if( argc == 2 ) {  
        printf("The argument supplied is %s\n", argv[1]);  
    }  
    else if( argc > 2 ) {  
        printf("Too many arguments supplied.\n");  
    }  
    else {  
        printf("One argument expected.\n");  
    }  
}
```

When the above code is compiled and executed with single argument, it produces the following result.

```
$ ./a.out testing  
The argument supplied is testing
```

When the above code is compiled and executed with a two arguments, it produces the following result.

```
$./a.out testing1 testing2  
Too many arguments supplied.
```

When the above code is compiled and executed without passing any argument, it produces the following result.

```
$./a.out  
One argument expected
```

```
#include <stdio.h>

int main( int argc, char *argv[] ) {

    printf("Program name %s\n", argv[0]);

    if( argc == 2 ) {
        printf("The argument supplied is %s\n", argv[1]);
    }
    else if( argc > 2 ) {
        printf("Too many arguments supplied.\n");
    }
    else {
        printf("One argument expected.\n");
    }
}
```

When the above code is compiled and executed with a single argument separated by space but inside double quotes, it produces the following result.

```
./a.out "testing1 testing2"
```

```
Program name ./a.out
The argument supplied is testing1 testing2
```


File Handling in C

In this lesson, you will learn about file handling in C. You will learn to handle standard I/O in C using `fprintf()`, `fscanf()`, `fread()`, `fwrite()`, `fseek()` etc. with the help of examples.

A file is a container in computer storage devices used for storing data.

Why files are needed?

- When a program is terminated, the entire data is lost. Storing in a file will preserve your data even if the program terminates.
- If you have to enter a large number of data, it will take a lot of time to enter them all.
However, if you have a file containing all the data, you can easily access the contents of the file using a few commands in C.
- You can easily move your data from one computer to another without any changes.

Types of Files

When dealing with files, there are two types of files you should know about:

1. Text files
2. Binary files

1. Text files

Text files are the normal **.txt** files. You can easily create text files using any simple text editors such as Notepad.

When you open those files, you'll see all the contents within the file as plain text. You can easily edit or delete the contents.

They take minimum effort to maintain, are easily readable, and provide the least security and takes bigger storage space.

2. Binary files

Binary files are mostly the **.bin** files in your computer.

Instead of storing data in plain text, they store it in the binary form (0's and 1's).

They can hold a higher amount of data, are not readable easily, and provides better security than text files.

File Operations

In C, you can perform four major operations on files, either text or binary:

1. Creating a new file
2. Opening an existing file
3. Closing a file
4. Reading from and writing information to a file

Working with files

When working with files, you need to declare a pointer of type file. This declaration is needed for communication between the file and the program.

```
FILE *fptr;
```

Opening a file - for creation and edit

Opening a file is performed using the `fopen()` function defined in the `stdio.h` header file.

The syntax for opening a file in standard I/O is:

```
ptr = fopen("fileopen","mode");
```

For example,

```
fopen("E:\\cprogram\\newprogram.txt","w");
```

```
fopen("E:\\cprogram\\oldprogram.bin","rb");
```

- Let's suppose the file `newprogram.txt` doesn't exist in the location `E:\\cprogram`.
The first function creates a new file named `newprogram.txt` and opens it for

writing as per the mode '**w**'.

The writing mode allows you to create and edit (overwrite) the contents of the file.

- Now let's suppose the second binary file `oldprogram.bin` exists in the location `E:\cprogram`. The second function opens the existing file for reading in binary mode '**rb**'.

The reading mode only allows you to read the file, you cannot write into the file.

Opening Modes in Standard I/O		
Mode	Meaning of Mode	During Inexistence of file
r	Open for reading.	If the file does not exist, <code>fopen()</code> returns NULL.
rb	Open for reading in binary mode.	If the file does not exist, <code>fopen()</code> returns NULL.
w	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb	Open for writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a	Open for append. Data is added to the end of the file.	If the file does not exist, it will be created.
ab	Open for append in binary mode. Data is added to the end of the file.	If the file does not exist, it will be created.

Opening Modes in Standard I/O		
Mode	Meaning of Mode	During Inexistence of file
r+	Open for both reading and writing.	If the file does not exist, <code>fopen()</code> returns NULL.
rb+	Open for both reading and writing in binary mode.	If the file does not exist, <code>fopen()</code> returns NULL.
w+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb+	Open for both reading and writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Open for both reading and appending.	If the file does not exist, it will be created.
ab+	Open for both reading and appending in binary mode.	If the file does not exist, it will be created.

Closing a File

The file (both text and binary) should be closed after reading/writing.

Closing a file is performed using the `fclose()` function.

```
fclose(fp);
```

Here, `fp` is a file pointer associated with the file to be closed.

Reading and writing to a text file

For reading and writing to a text file, we use the functions `fprintf()` and `fscanf()`.

They are just the file versions of `printf()` and `scanf()`. The only difference is that `fprintf()` and `fscanf()` expects a pointer to the structure `FILE`.

Example 1: Write to a text file

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

int main()
{
    int num;
    FILE *fptr;

    // use appropriate location if you are using MacOS or Linux
    fptr = fopen("C:\\program.txt","w");

    if(fptr == NULL)
    {
        printf("Error!");
        exit(1);
    }

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

    fprintf(fptr,"%d",num);
    fclose(fptr);

    return 0;
}
```

This program takes a number from the user and stores in the file `program.txt`.

After you compile and run this program, you can see a text file `program.txt` created in C drive of your computer. When you open the file, you can see the integer you entered.

Example 2: Read from a text file

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

int main()
{
    int num;
    FILE *fptr;
```

```

if ((fptr = fopen("C:\\program.txt","r")) == NULL){
    printf("Error! opening file");

    // Program exits if the file pointer returns NULL.
    exit(1);
}

fscanf(fptr,"%d", &num);

printf("Value of n=%d", num);
fclose(fptr);

return 0;
}

```

This program reads the integer present in the `program.txt` file and prints it onto the screen.

If you successfully created the file from **Example 1**, running this program will get you the integer you entered.

Other functions like `fgetchar()`, `fputc()` etc. can be used in a similar way.

Reading and writing to a binary file

Functions `fread()` and `fwrite()` are used for reading from and writing to a file on the disk respectively in case of binary files.

Writing to a binary file

To write into a binary file, you need to use the `fwrite()` function. The functions take four arguments:

1. address of data to be written in the disk
2. size of data to be written in the disk
3. number of such type of data
4. pointer to the file where you want to write.

```
fwrite(addressData, sizeData, numbersData, pointerToFile);
```

Example 3: Write to a binary file using fwrite()

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

struct threeNum
{
    int n1, n2, n3;
};

int main()
{
    int n;
    struct threeNum num;
    FILE *fptr;

    if ((fptr = fopen("C:\\program.bin","wb")) == NULL){
        printf("Error! opening file");

        // Program exits if the file pointer returns NULL.
        exit(1);
    }

    for(n = 1; n < 5; ++n)
    {
        num.n1 = n;
        num.n2 = 5*n;
        num.n3 = 5*n + 1;
        fwrite(&num, sizeof(struct threeNum), 1, fptr);
    }
    fclose(fptr);

    return 0;
}
```

In this program, we create a new file `program.bin` in the C drive.

We declare a structure `threeNum` with three numbers - `n1`, `n2` and `n3`, and define it in the main function as `num`.

Now, inside the for loop, we store the value into the file using `fwrite()`.

The first parameter takes the address of `num` and the second parameter takes the size of the structure `threeNum`.

Since we're only inserting one instance of `num`, the third parameter is `1`. And, the last parameter `*fptr` points to the file we're storing the data.

Finally, we close the file.

Reading from a binary file

Function `fread()` also take 4 arguments similar to the `fwrite()` function as above.

```
fread(addressData, sizeData, numbersData, pointerToFile);
```

Example 4: Read from a binary file using `fread()`

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

struct threeNum
{
    int n1, n2, n3;
};

int main()
{
    int n;
    struct threeNum num;
    FILE *fptr;

    if ((fptr = fopen("C:\\program.bin","rb")) == NULL){
        printf("Error! opening file");

        // Program exits if the file pointer returns NULL.
        exit(1);
    }

    for(n = 1; n < 5; ++n)
    {
        fread(&num, sizeof(struct threeNum), 1, fptr);
        printf("n1: %d\\tn2: %d\\tn3: %d\\n", num.n1, num.n2, num.n3);
    }
    fclose(fptr);

    return 0;
}
```

```
}
```

In this program, you read the same file `program.bin` and loop through the records one by one.

In simple terms, you read one `threeNum` record of `threeNum` size from the file pointed by `*fptr` into the structure `num`.

You'll get the same records you inserted in **Example 3**.

Getting data using `fseek()`

If you have many records inside a file and need to access a record at a specific position, you need to loop through all the records before it to get the record.

This will waste a lot of memory and operation time. An easier way to get to the required data can be achieved using `fseek()`.

As the name suggests, `fseek()` seeks the cursor to the given record in the file.

Syntax of `fseek()`

```
fseek(FILE * stream, long int offset, int whence);
```

The first parameter `stream` is the pointer to the file. The second parameter is the position of the record to be found, and the third parameter specifies the location where the offset starts.

Different whence in <code>fseek()</code>	
Whence	Meaning
<code>SEEK_SET</code>	Starts the offset from the beginning of the file.
<code>SEEK_END</code>	Starts the offset from the end of the file.
<code>SEEK_CUR</code>	Starts the offset from the current location of the cursor in the file.

Example 5: fseek()

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

struct threeNum
{
    int n1, n2, n3;
};

int main()
{
    int n;
    struct threeNum num;
    FILE *fptr;

    if ((fptr = fopen("C:\\program.bin","rb")) == NULL){
        printf("Error! opening file");

        // Program exits if the file pointer returns NULL.
        exit(1);
    }

    // Moves the cursor to the end of the file
    fseek(fptr, -sizeof(struct threeNum), SEEK_END);

    for(n = 1; n < 5; ++n)
    {
        fread(&num, sizeof(struct threeNum), 1, fptr);
        printf("n1: %d\\tn2: %d\\tn3: %d\\n", num.n1, num.n2, num.n3);
        fseek(fptr, -2*sizeof(struct threeNum), SEEK_CUR);
    }
    fclose(fptr);

    return 0;
}
```

This program will start reading the records from the file `program.bin` in the reverse order (last to first) and prints it.

C program to read name and marks of n number of students and store them in a file.

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

```
int main()
{
    char name[50];
    int marks, i, num;

    printf("Enter number of students: ");
    scanf("%d", &num);

    FILE *fptr;
    fptr = (fopen("C:\\student.txt", "w"));
    if(fptr == NULL)
    {
        printf("Error!");
        exit(1);
    }

    for(i = 0; i < num; ++i)
    {
        printf("For student%d\\nEnter name: ", i+1);
        scanf("%s", name);

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

        fprintf(fptr, "\\nName: %s \\nMarks=%d \\n", name, marks);
    }

    fclose(fptr);
    return 0;
}
```