



आईएफटीएम विश्वविद्यालय, मुरादाबाद, उत्तर प्रदेश
IFTM University, Moradabad, Uttar Pradesh
NAAC ACCREDITED

E-Content

IFTM University, Moradabad

Unit-1

PPC (Principles of Programming using C)

Program

A program is a set of instructions (in a specific programming language) that tell the computer to do various things. Computer programming is a way of giving computer instructions about what they should do next. These instructions are known as code, and computer programmers write code to solve problems or perform a task.

Algorithms

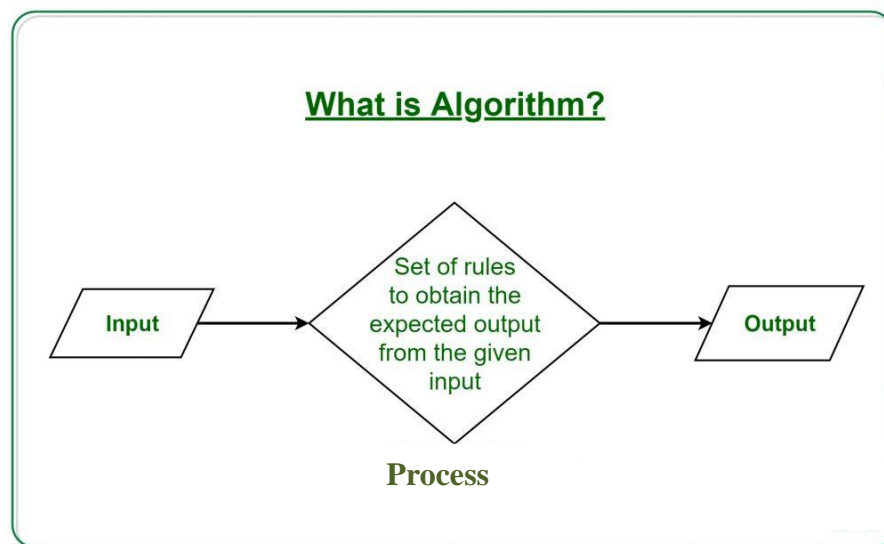
In programming, algorithm is a set of well defined instructions (in a general language) in sequence to solve the problem. In other words we can say an algorithm is a set of rules that must be followed when solving a particular problem. An algorithm becomes a program when it is written in the form of a programming language. Algorithm is a step-by-step solution.

Stages of defining an algorithm

To solve a problem we often need to take a step-by-steps approach to it. We can call this set of steps an algorithm. One way of thinking of an algorithm is as something taking an input, applying a process to it to produce the desired output. So we can say an algorithm has three stages-

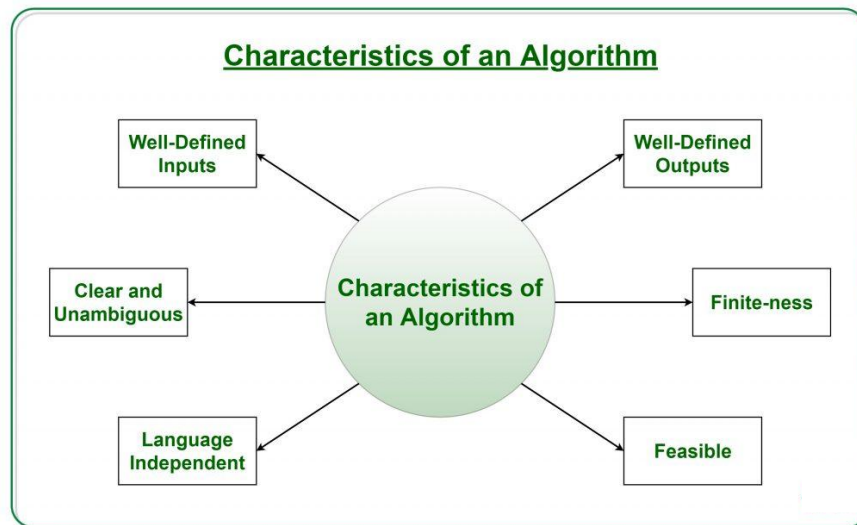
1. Input
2. Processing
3. Output

Input → Process → Output



Characteristics of an Algorithm

- **Clear and Unambiguous:** Algorithm should be clear and unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.
- **Well-Defined Inputs:** If an algorithm says to take inputs, it should be well-defined inputs.
- **Well-Defined Outputs:** The algorithm must clearly define what output will be yielded and it should be well-defined as well.
- **Finiteness:** The algorithm must be finite, i.e. it should not end up in an infinite loops or similar.
- **Feasible:** The algorithm must be simple, generic and practical; such that it can be executed upon will the available resources. It must not contain some future technology, or anything.
- **Language Independent:** The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be same, as expected.



Advantages of algorithm

1. It is easy to understand.
2. Algorithm is a step-wise representation of a solution to a given problem.
3. In Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.

Disadvantages of algorithm.

1. Writing algorithm takes a long time.
2. An Algorithm is not a computer program; it is rather a concept of how a program should be.
3. Branching and looping statements are difficult to show in Algorithms.

Examples of Algorithms in Programming

1. Write an algorithm to add two numbers entered by user.

Step 1: Start

Step 2: Declare variables n1, n2 and sum.

Step 3: Read values n1 and n2.

Step 4: Add n1 and n2 and assign the result to sum. $\text{sum} \leftarrow \text{n1} + \text{n2}$

Step 5: Display sum

Step 6: Stop

2. Write an algorithm to take square of a number entered by user.

Step 1: Start

Step 2: Declare variables n and sqr.

Step 3: Read value of n.

Step 4: Multiply n with n and assign the result to sqr. $\text{sqr} \leftarrow \text{n} * \text{n}$

Step 5: Display sqr

Step 6: Stop

Flowcharts

A flowchart is a diagram that describes a process, system or computer algorithm. A flowchart is simply a graphical representation of steps. It shows steps in sequential order and is widely used in presenting the flow of algorithms, workflow or processes. Typically, a flowchart shows the steps as boxes of various kinds, and their order by connecting them with arrows.

Flowchart Symbols






Rectangle Shape - Represents a process

Oval or Pill Shape - Represents the start or end

Diamond Shape - Represents a decision

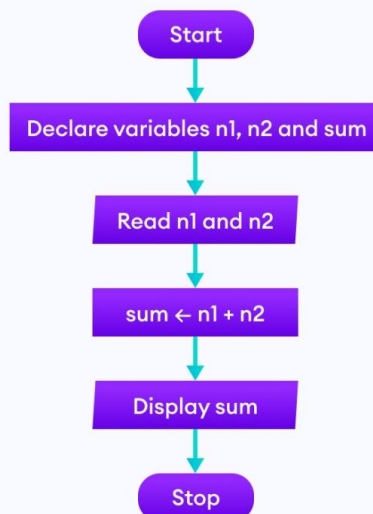
Parallelogram - Represents input/output

Arrow-Indicate Directional Flow

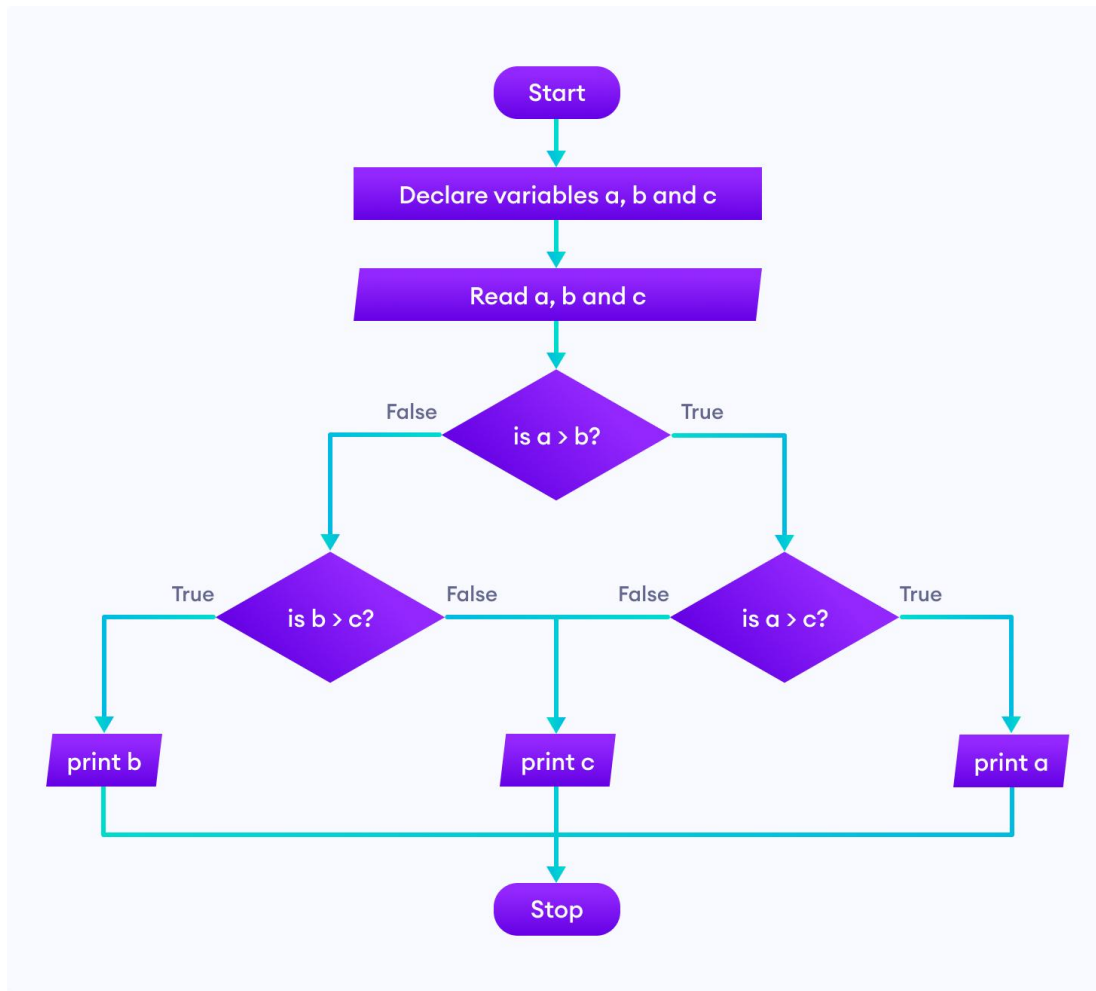
Symbol	Name	Function
	Start/end	An oval represents a start or end point
	Arrows	A line is a connector that shows relationships between the representative shapes
	Input/Output	A parallelogram represents input or output
	Process	A rectangle represents a process
	Decision	A diamond indicates a decision

Examples of flowcharts in programming

1. Add two numbers entered by the user.



2. Find the largest among three different numbers entered by the user.



Introduction to C Programming

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. C is a procedural programming language.

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

Initially, C language was developed to be used in **UNIX operating system**. It inherits many features of previous languages such as B and BCPL.

Let's see the programming languages that were developed before C language.

Language	Year	Developed By
Algol	1960	International Group
CPL	1963	Cambridge University, London University
BCPL	1967	Martin Richard

B	1970	Ken Thompson
Traditional C	1972	Dennis Ritchie
K & R C	1978	Kernighan & Dennis Ritchie
ANSI C	1989	ANSI Committee
ANSI/ISO C	1990	ISO Committee
C99	1999	Standardization Committee

Features of C Language

C is the widely used language. It provides many **features** that are given below.

1. Machine Independent or Portability

It refers to the usability of the same fragment of code in different environments. C programs are capable of being written on one platform and being run on another with or without any modification.

2. Modularity/Structured Language

This feature of C language allows the program to be splintered (broken) into smaller units and run individually with the help of functions. So it is easy to understand and modify.

3. Simple and Efficient

The **syntax style of C programming** is easy to understand. In high schools or colleges, C is generally taught as an introductory programming language as it is a well-established fact that it is easier to learn any other programming language in the long run if you are well familiar with C.

4. Faster

Since it is a compiler-based language, it is comparatively faster than other programming languages like **Java or Python**, which are interpreter based. A compiler considers the entire program as input and thereby generates an output file with the object code whereas an interpreter takes instruction by instruction as input and then generates an output but does not generate a file.

5. Popular

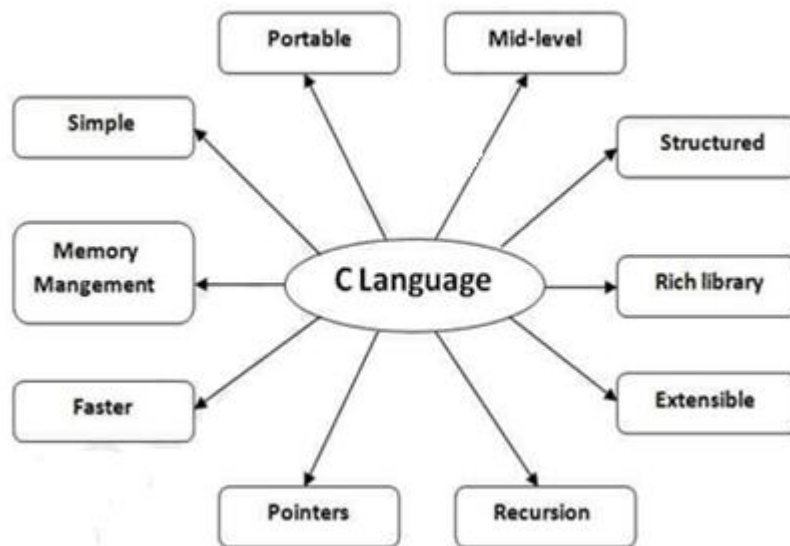
It is one of the most extensively used languages in the development of operating and embedded systems. Needless to mention how popular it is.

6. Rich Libraries

C language comprises of its library which has a wide range of built-in functions. Even the user-defined functions can be added to the C library. It gives the user a wide latitude of scope to develop his own functions for implementing problems for later use and implementation.

7. Memory Management

It supports the feature of DMA (Dynamic Memory Allocation), which helps in the utilization and management of memory. Among all the features of C, dynamism is unique. Using DMA, the size of a data structure can be changed during runtime using some predefined functions in the C library such as *malloc()*, *calloc()*, *free()* and *realloc()*.



8. Case Sensitive

It treats lowercase and uppercase characters differently. For instance, if we declare a variable 'x' of integer type, it would mean a different meaning altogether if we type 'X' rather than 'x'.

9. Mid-level programming language

C combines the features of both Assembly Level Languages (Low Level Languages) and Higher Level Languages. For this reason, C is referred to as a Middle Level Language. It is used to develop system applications such as kernel, driver, etc. It also supports the features of a high-level language. That is why it is known as mid-level language.

10. Recursion

In C, we can call the function within the function. It provides code reusability for every function. Recursion enables us to use the approach of backtracking.

11. Extensible

C language is extensible because it can easily adopt new features.

C Character Set

As every language contains a set of characters used to construct words, statements, etc., C language also has a set of characters which include **alphabets**, **digits**, and **special symbols**. C language supports a total of 256 characters.

Every C program contains statements. These statements are constructed using words and these words are constructed using characters from C character set. C language character set contains the following set of characters.

1. Alphabets
2. Digits
3. Special Symbols

Alphabets

C language supports all the alphabets from the English language. Lower and upper case letters together support 52 alphabets.

Lower case letters - **a to z**

UPPER CASE LETTERS - **A to Z**

Digits

C language supports 10 digits which are used to construct numerical values in C language.

Digits - **0, 1, 2, 3, 4, 5, 6, 7, 8, 9**

Special Symbols

C language supports a rich set of special symbols that include symbols to perform mathematical operations, to check conditions, white spaces, backspaces, and other special symbols.

~	tilde	%	percent sign		vertical bar	@	at symbol
+	plus sign	<	less than	_	underscore	-	minus sign
>	greater than	^	caret	#	number sign	=	equal to
&	ampersand	\$	dollar sign	/	slash	(left parenthesis
*	asterisk	\	back slash)	right parenthesis	:	colon
'	apostrophe	[left bracket	"	quotation mark	;	semicolon
]	right bracket	!	exclamation mark	,	comma	{	left flower brace
?	Question mark	.	dot operator	}	right flower brace		

C Keywords

Keywords are preserved words that have special meaning in C language. The meaning of C language keywords has already been described to the C compiler. These meaning cannot be changed. Thus, keywords cannot be used as variable names because that would try to change the existing meaning of the keyword, which is not allowed. There are total 32 keywords in C language. Keywords are written in lowercase letters.

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

Following table represents the keywords in 'C'-

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

Identifiers

In C language identifiers are the names given to variables, constants, functions and user-define data. These identifier are defined against a set of rules.

Rules for an Identifier

- The first character of an identifier should be either an alphabet or an underscore, and then it can be followed by any of the character, digit, or underscore.
- It should not begin with any numerical digit.
- In identifiers, both uppercase and lowercase letters are distinct. Therefore, we can say that identifiers are case sensitive.
- Commas or blank spaces cannot be specified within an identifier.
- Keywords cannot be represented as an identifier.
- Identifiers should be written in such a way that it is meaningful, short, and easy to read.

Example of valid identifiers

1. total, sum, average, _m_, sum_1, etc.

Example of invalid identifiers

1. 2sum (starts with a numerical digit)
2. **int** (reserved word)
3. **char** (reserved word)
4. m+n (special character, i.e., '+')

Differences between Keyword and Identifier

Keyword	Identifier
Keyword is a pre-defined word.	The identifier is a user-defined word
It must be written in a lowercase letter.	It can be written in both lowercase and uppercase letters.
Its meaning is pre-defined in the c compiler.	Its meaning is not defined in the c compiler.
It is a combination of alphabetical characters.	It is a combination of alphanumeric characters.
It does not contain the underscore character.	It can contain the underscore character.

Constant

As the name suggests the name constants is given to such variables or values in C programming language which cannot be modified once they are defined. They are fixed values in a program. C Constants is the most fundamental and essential part of the C programming language. Constants in C are the fixed values that are used in a program, and its value remains the same during the entire execution of the program.

Syntax:

```
const type constant_name;
```

Example:

```
const int pi=3.14;
```

Variable

When we want to store any information (data) on our computer/laptop, we store it in the computer's memory space. Instead of remembering the complex address of that memory space where we have stored our data, our operating system provides us with an option to create folders, name them, so that it becomes easier for us to find it and access it.

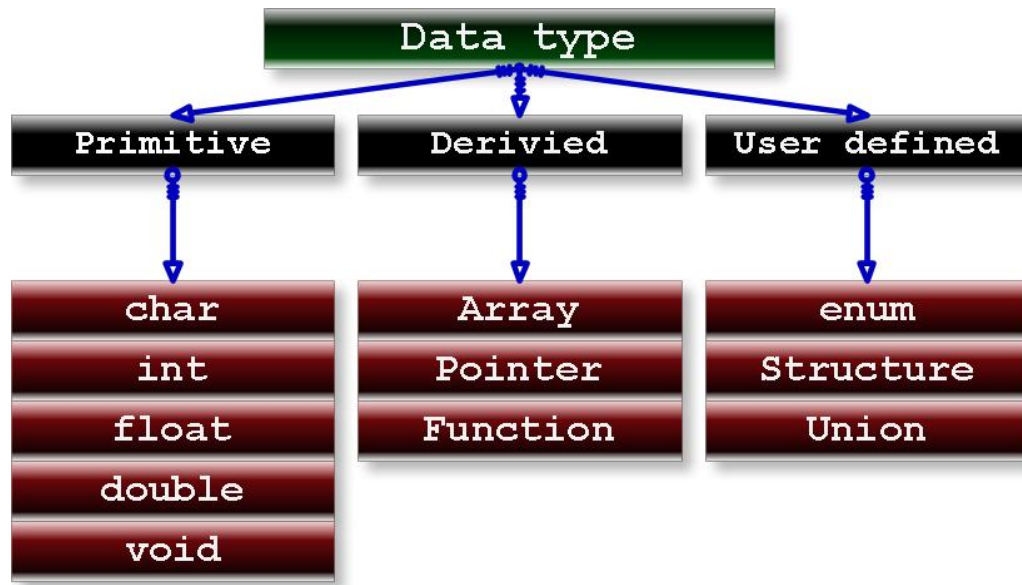
Similarly, in C language, when we want to use some data value in our program, we can store it in a memory space and name the memory space so that it becomes easier to access it. The naming of an address is known as variable. Variable is the name of memory location. Unlike constant, variables are changeable; we can change value of a variable during execution of a program. A programmer can choose a meaningful variable name. Example: name, height, age, total etc.

Data Types in C

Each variable in C has an associated data type. Each data type requires different amounts of memory and has some specific operations which can be performed over it. C provides various types of data-types which allow the programmer to select the appropriate type for the variable to set its value. The data-type in a programming language is the collection of data with values having fixed meaning as well as characteristics. Some of them are an integer, floating point, character, etc.

C Data Types are used to

- Identify the type of a variable when it declared.
- Identify the type of the return value of a function.
- Identify the type of a parameter expected by a function.



C provides three types of data types

- **Primary (Built-in) Data Types:**
int, char, double ,float and void.
- **Derived Data Types:**
Array, References, and Pointers.
- **User Defined Data Types:**
Structure, Union, and Enumeration.

int

Integers are whole numbers that can have both zero, positive and negative values but no decimal values.
For example, 0, -5, 10

We can use int for declaring an integer variable.

```
int id;          int rollno;
```

Here, id is a variable of type integer.

We can declare multiple variables at once in C programming. For example,

```
int id, age;
```

The size of int is usually 4 bytes. (1 byte= 8 bits)

char

Keyword `char` is used for declaring character type variables. It simply contains a single character enclosed within ' and ' (a pair of single quote). It is to be noted that the character '8' is not the same as 8.

For example,

```
char test = 'r';           char ar = '8';
```

The size of the character variable is 1 byte.

float and double

`float` and `double` are used to hold real numbers means decimal values.

```
float salary;  
double price;
```

The size of float is usually 4 bytes and the size of double is usually 8 bytes.

Operator

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 –

- Arithmetic Operators
- Relational Operators
- Logical Operators

Arithmetic Operator

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

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

Relational Operator

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

Operator	Description	
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.
!=	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.
>	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.
<	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.
>=	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.
<=	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.

Logical Operator

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

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.

C – Special Operators

Special Operators in C:

Below are some of the special operators that the C programming language offers.

Operators	Description
&	This is used to get the address of the variable. Example : &a will give address of a.
*	This is used as pointer to a variable. Example : * a where, * is pointer to the variable a.
Sizeof ()	This gives the size of the variable. Example: size of (char) will give us 1.

Expressions

An expression is a combination of variables constants and operators written according to the syntax of C language. In C every expression evaluates to a value i.e., every expression results in some value of a certain type that can be assigned to a variable. Some examples of C expressions are shown in the table given below.

Algebraic Expression	C Expression
$a \times b - c$	$a * b - c$
$(m + n) (x + y)$	$(m + n) * (x + y)$
(ab / c)	$a * b / c$
$3x^2 + 2x + 1$	$3*x*x+2*x+1$
$(x / y) + c$	$x / y + c$

C - Program Structure

A C program basically consists of the following parts –

- Preprocessor Commands
- Functions
- Variables
- Statements & Expressions
- Comments

Let us look at a simple code that would print the words "Hello to C Language" –

```
#include <stdio.h>
int main()
{
    /* display below statement */
    printf("Hello to C Language \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 to C Language" to be displayed on the screen.
- The next line **return 0;** terminates the main() function and returns the value 0.

Input and output function in C

When we say **Input**, it means to feed some data into a program. 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.

All these built-in functions are present in C header files.

scanf() and printf() functions

The standard input-output header file, named **stdio.h** contains the definition of the functions **printf()** and **scanf()**, which are used to display output on screen and to take input from user respectively.

When we will compile the above code, it will ask us to enter a value. When we will enter the value, it will display the value we have entered on screen.

Format String	Meaning
%d	Scan or print an integer
%f	Scan or print a floating point number
%c	To scan or print a character
%s	To scan or print a character string. The scanning ends at whitespace.

```
#include<stdio.h>
void main()
{
    /*defining a variable*/
    int i;
    /* displaying message on the screen asking the user to input a value */
    printf("Please enter a value...");
    /*reading the value entered by the user */
    scanf("%d", &i);
    /*displaying the number as output*/
    printf( "\n Entered no is: %d", i);
}
```

Programming Language

A **programming language** is a formal computer language designed to communicate instructions to a machine, particularly a computer. Programming languages can be used to create programs to control the behavior of a machine or to express algorithms. A **programming language** is a special language programmers use to develop software programs, scripts, or other sets of instructions for computers to execute.

High-Level Language (HLL)

High-level language is any programming language that enables development of a program in much simpler programming context and is generally independent of the computer's hardware architecture. High-level language has a higher level of abstraction from the computer, and focuses more on the programming logic rather than the underlying hardware components such as memory addressing and register utilization. High level languages are designed to be used by the human operator or the programmer. They are referred to as "closer to humans." In other words, their programming style and context is easier to learn and implement, and the entire code generally focuses on the specific program to be created. High-level language doesn't require addressing hardware constraints to a greater extent when developing a program. However, every single program written in a high level language must be interpreted into machine language before being executed by the computer.

Low-Level Language

Low-level language is a programming language that deals with a computer's hardware components and constraints. It has no or a minute level of abstraction in reference to a computer and works to manage a computer's operational semantics. Low-level language may also be referred to as a computer's native language.

Low-level languages are designed to operate and handle the entire hardware and instructions set architecture of a computer directly. Low-level languages are considered to be closer to computers. In other words, their prime function is to operate, manage and manipulate the computing hardware and components. Programs and applications written in low-level language are directly executable on the computing hardware without any interpretation or translation. Machine language and assembly language are popular examples of low level languages.

Assembler, Compiler, Interpreter

Assembler A computer will not understand any program written in a language, other than its machine language. The programs written in other languages must be translated into the machine language. Such translation is performed with the help of software. A program which translates an assembly language program into a machine language program is called an assembler. If an assembler which runs on a computer and produces the machine codes for the same computer then it is called self assembler or resident assembler. If an assembler that runs on a computer and produces the machine codes for other computer then it is called Cross Assembler.

Assemblers are further divided into two types: One Pass Assembler and Two Pass Assembler. One pass assembler is the assembler which assigns the memory addresses to the variables and translates the source code into machine code in the first pass simultaneously. A Two Pass Assembler is the assembler which reads the source code twice. In the first pass, it reads all the variables and assigns them memory addresses. In the second pass, it reads the source code and translates the code into object code.

Compiler It is a program which translates a high level language program into a machine language program. A compiler is more intelligent than an assembler. It checks all kinds of limits, ranges, errors etc. But its program run time is more and occupies a larger part of the memory. It has slow speed. Because a compiler goes through the entire program and then translates the entire program into machine codes. If a compiler runs on a computer and produces the machine codes for the same computer then it is known as a self compiler or resident compiler. On the other hand, if a compiler runs on a computer and produces the machine codes for other computer then it is known as a cross compiler.

Interpreter An interpreter is a program which translates statements of a program into machine code. It translates only one statement of the program at a time. It reads only one statement of program, translates it and executes it. Then it reads the next statement of the program again translates it and executes it. In this way it proceeds further till all the statements are translated and executed. On the other hand, a compiler goes through the entire program and then translates the entire program into machine codes. A compiler is 5 to 25 times faster than an interpreter.

By the compiler, the machine codes are saved permanently for future reference. On the other hand, the machine codes produced by interpreter are not saved. An interpreter is a small program as compared to compiler. It occupies less memory space, so it can be used in a smaller system which has limited memory space.

Variable Declaration and Initialization in C Programming Language

C variables are names used for storing a data value to locations in memory. The value stored in the c variables may be changed during program execution.

C is a strongly typed language. Every variable must be declared, indicating its data type before it can be used.

Declaration of Variable

Declaration of variable in c can be done using following syntax:

```
data_type variable_name;
```

or

```
data_type variable1, variable2, ...,variablen;
```

where *data_type* is any valid c data type and *variable_name* is any valid identifier.

For example,

```
int a;
```

```
float variable;
```

```
float a, b;
```

Initialization of Variable

C variables declared can be initialized with the help of assignment operator '='.

Syntax

```
data_type variable_name=constant/literal/expression;
```

or

```
variable_name=constant/literal/expression;
```

Example:

```
int a=10;
```

```
int a=b+c;
```

```
a=10;
```

```
a=b+c;
```

Multiple variables can be initialized in a single statement by single value, for example, a=b=c=d=e=10;

Write a program in “C” to calculate the simple interest.

```
#include<stdio.h>

int main()
{
float p,r,t,si;
printf("Enter the principle amount");
scanf("%f", &p);
printf("Enter the rate of interest");
scanf("%f", &r);
printf("Enter time");
scanf("%f",&t);
si=p*r*t/100;
printf("The simple interest is %f", si);
return 0;
}
```

Write a program in “C” to calculate the sum of two integer numbers.

```
#include <stdio.h>
int main()
{
int num1, num2, sum;
printf("Enter first number: ");
scanf("%d", &num1);
printf("Enter second number: ");
scanf("%d", &num2);

sum = num1 + num2;
printf("Sum of the entered numbers: %d", sum);
return 0;
}
```

Unit-2

PPC (Principles of Programming using C)

C Statements

The bodies of C functions (including the main function) are made up of statements. These can either be **simple statements** that do not contain other statements, or **compound statements** that have other statements inside them. Control structures are compound statements like if/then/else, while, for, and do...while that control how or whether their component statements are executed.

1. Simple statements

A statement is a command given to the computer that instructs the computer to take a specific action, such as display to the screen, or collect input. A computer program is made up of a series of statements. The simplest kind of statement in C is an expression (followed by a semicolon, the terminator for all simple statements). Its value is computed and discarded. Examples:

Toggle line numbers

1. `x = 2;` `/* an assignment statement */`
2. `x = 2+3;` `/* another assignment statement */`
3. `2+3;` `/* has no effect---will be discarded by smart compilers */`

Most statements in a typical C program are simple statements of this form.

2. Compound statements

Statements that have other statements inside them are called compound statements. Compound statements come in two varieties:

- a) Conditionals (if, if else, if else if, switch)
- b) Loops (while, do while, for loop, Loops with break, continue, and goto).

C – If statement

When we need to execute a block of statements only when a given condition is true then we use if statement.

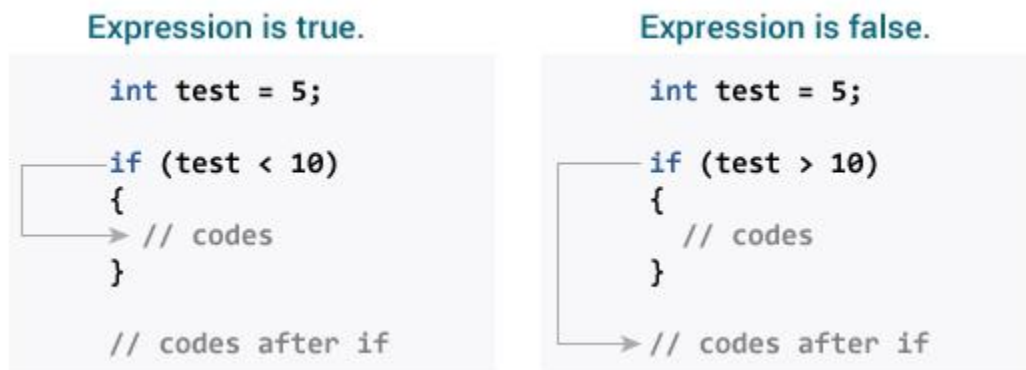
It is one of the powerful conditional statements. if statement is responsible for modifying the flow of execution of a program. if statement is always used with a condition. The condition is evaluated first before executing any statement inside the body of If. The syntax for if statement is as follows:

```
if (condition)
    instruction;
```

How if statement works?

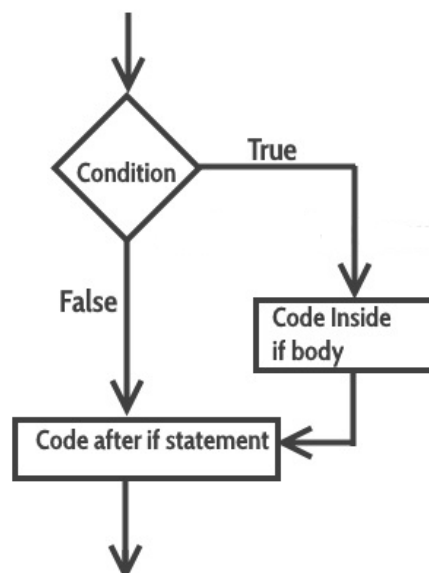
The if statement evaluates the test expression inside the parenthesis ().

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



The condition evaluates to either true or false. True is always a non-zero value, and false is a value that contains zero. Instructions can be a single instruction or a code block enclosed by curly braces { }.

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.



Example of if statement

```
#include <stdio.h>

int main()
{
    int x = 20;
    int y = 22;
    if (x<y)
    {
        printf("Variable x is less than y");
    }
    return 0;
}
```

Output:

Variable x is less than y

Explanation: The condition (x<y) specified in the “if” returns true for the value of x and y, so the statement inside the body of if is executed.

Example of if statement

```
#include<stdio.h>
void main()
{
    int a,b;
    printf("enter the value of a and b")
    scanf("%d %d", &a, &b);
    if(a>b)
        printf("%d is greater than %d", a, b)
}
```

Example of multiple if statements

We can use multiple if statements to check more than one conditions.

```
#include <stdio.h>

int main()
{
    int x, y;
    printf("enter the value of x:");
    scanf("%d", &x);
    printf("enter the value of y:");
    scanf("%d", &y);
    if (x>y)
    {
        printf("x is greater than y\n");
    }
    if (x<y)
    {
        printf("x is less than y\n");
    }
    if (x==y)
    {
        printf("x is equal to y\n");
    }
    return 0;
}
```

In the above example the output depends on the user input.

Output:

enter the value of x:20

enter the value of y:20

x is equal to y

Write a program to find the largest number among the three using multiple if statements.

```
#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
```

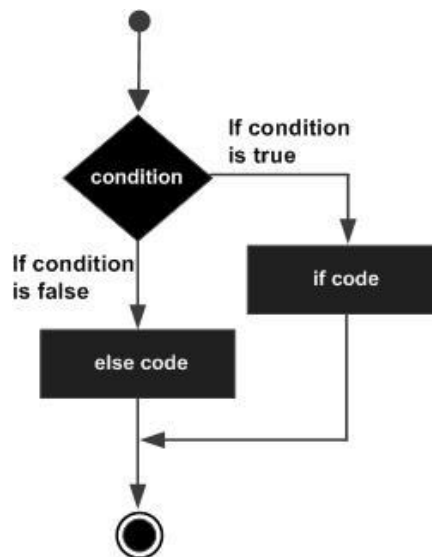
C-If else statement

The if-else statement is used to perform two operations for a single condition. The if-else statement is an extension to the if statement using which, we can perform two different operations, i.e., one is for the correctness of that condition, and the other is for the incorrectness of the condition.

The syntax of the if..else statement is:

```
if (test expression)
{
    // statements to be executed if the test expression is true
}
else
{
    // statements to be executed if the test expression is false
}
```

Flow Diagram of if-else



Write a program to check whether a person is eligible to vote or not using if-else.

```
#include <stdio.h>
int main()
{
    int age;
    printf("Enter your age?");
    scanf("%d", &age);
    if(age>=18)
    {
        printf("You are eligible to vote...");
    }
    else
    {
        printf("Sorry ... you can't vote");
    }
}
```

Example

```
#include <stdio.h>
int main()
{
    int x, y;
    printf("enter the value of x:");
    scanf("%d", &x);
    printf("enter the value of y:");
    scanf("%d", &y);
    if (x>y)
    {
        printf("x is greater than y\n");
    }
    else
    {
        printf("x is less than y\n");
    }
    return 0;
}
```

Write a program to check whether an integer is even or odd using if else statement.

```
#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;
}
```

C-Nested If Statement

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

Syntax:

```
if (condition1)
{
    // executes when condition1 is true
    if (condition2)
    {
        // executes when condition2 is true
    }
}
```

Example

```
#include <stdio.h>
int main()
{
    int i;
    printf("Enter value of i");
    scanf("%d", &i);
    if (i < 10)
    {
        printf("i is smaller than 10\n");
        if (i < 15)
            printf("i is smaller than 15 too\n");
        if (i < 20)
            printf("i is smaller than 20 too\n");
    }
    return 0;
}
```

C 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"
}

```

Example of nested if..else

```

#include <stdio.h>
int main()
{
    int var1, var2;
    printf("Input the value of var1:");
    scanf("%d", &var1);
    printf("Input the value of var2:");
    scanf("%d",&var2);
    if (var1 != var2)
    {
        printf("var1 is not equal to var2\n");
        //Nested if else
        if (var1 > var2)
        {
            printf("var1 is greater than var2\n");
        }
        else
        {
            printf("var2 is greater than var1\n");
        }
    }
    else
    {
        printf("var1 is equal to var2\n");
    }
}

```

```
return 0;
```

```
}
```

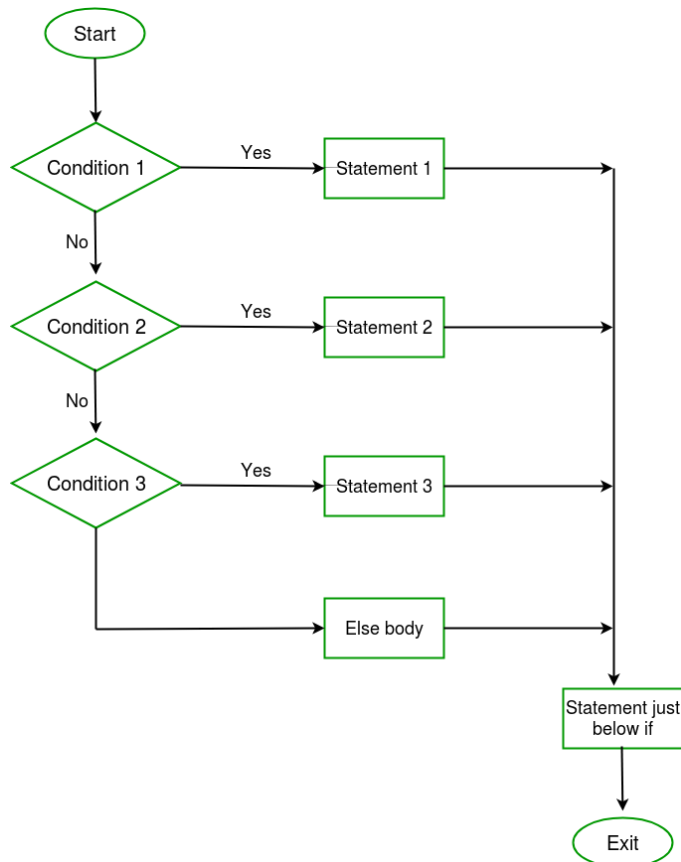
If-else-if ladder in C

The C if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the C else-if ladder is bypassed.

If none of the conditions are true, then the final else statement will be executed.

Syntax:

```
if(condition1)
{
//code to be executed if condition1 is true
}
else if(condition2)
{
//code to be executed if condition2 is true
}
else if(condition3)
{
//code to be executed if condition3 is true
}
...
else
{
//code to be executed if all the conditions are false
}
```



Example

```
#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 ...");
    }
}
```

Conditional Operator in C

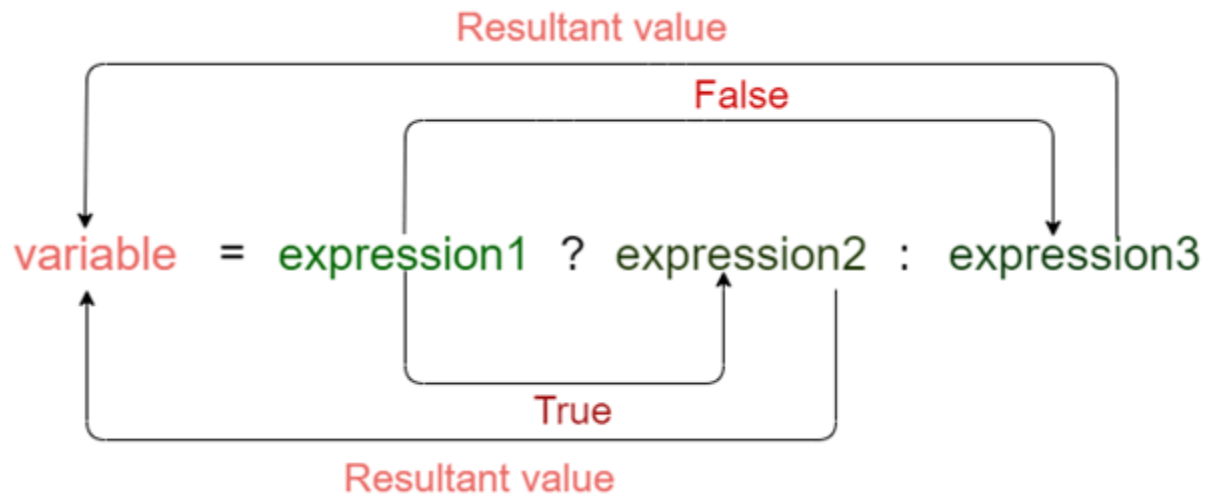
The conditional operator is also known as a **ternary operator**. The conditional statements are the decision-making statements which depend upon the output of the expression. It is represented by two symbols, i.e., '?' and ':'

As conditional operator works on three operands, so it is also known as the ternary operator.

The behavior of the conditional operator is similar to the 'if-else' statement as 'if-else' statement is also a decision-making statement.

Syntax of a conditional operator

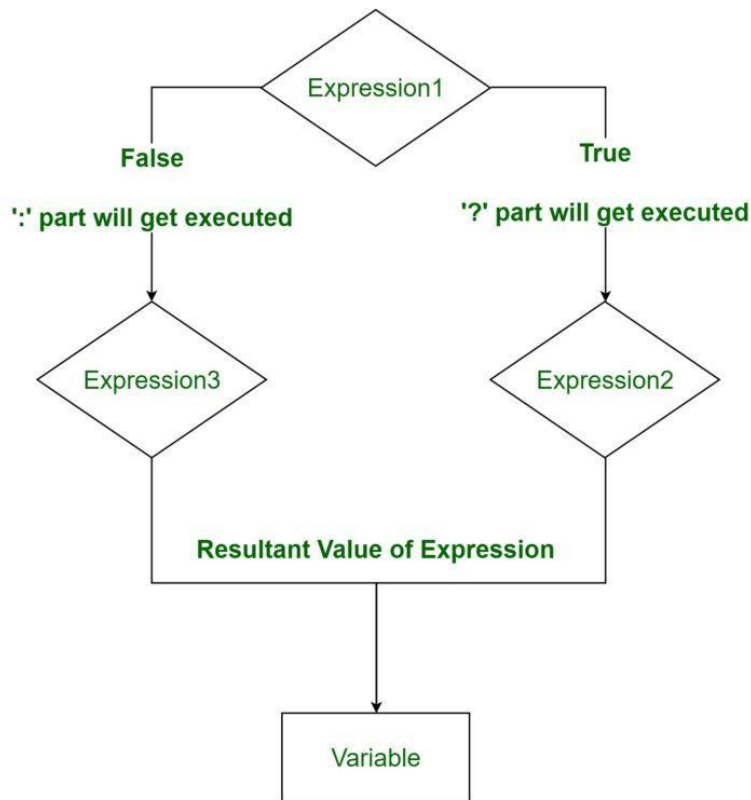
Expression1? expression2: expression3;



Meaning of above syntax

- In the above syntax, the expression1 is a Boolean condition that can be either true or false value.
- If the expression1 results into a true value, then the expression2 will execute.
- The expression2 is said to be true only when it returns a non-zero value.
- If the expression1 returns false value then the expression3 will execute.
- The expression3 is said to be false only when it returns zero value.

Flow Chart of Conditional or Ternary Operator



Example 1

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

```
int main()
```

```
{
```

```
    int age; // variable declaration
```

```
    printf("Enter your age");
```

```
    scanf("%d",&age); // taking user input for age variable
```

```
    (age>=18)? (printf("eligible for voting")) : (printf("not eligible for voting")); // conditional oper
```

```
ator
```

```
    return 0;
```

```
}
```

Example 2

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

```
int main()
```

```
{
```

```
int a=5,b; // variable declaration
b=((a==5)?(3):(2)); // conditional operator
printf("The value of 'b' variable is : %d",b);
return 0;
}
```

Program to store the greatest of the two number using conditional operator.

```
#include <stdio.h>
int main()
{
    int m,n;
    printf("Enter values of m and n");
    scanf("%d %d", &m,&n);

    (m > n) ? printf("m is greater than n that is %d > %d",
                  m, n)
            : printf("n is greater than m that is %d > %d",
                  n, m);

    return 0;
}
```

Switch statement in C

The switch statement in C is an alternate to if-else-if ladder statement which allows us to execute multiple operations for the different possible values of a single variable called switch variable. Here, we can define various statements in the multiple cases for the different values of a single variable.

The syntax of switch statement in [c language](#) is given below:

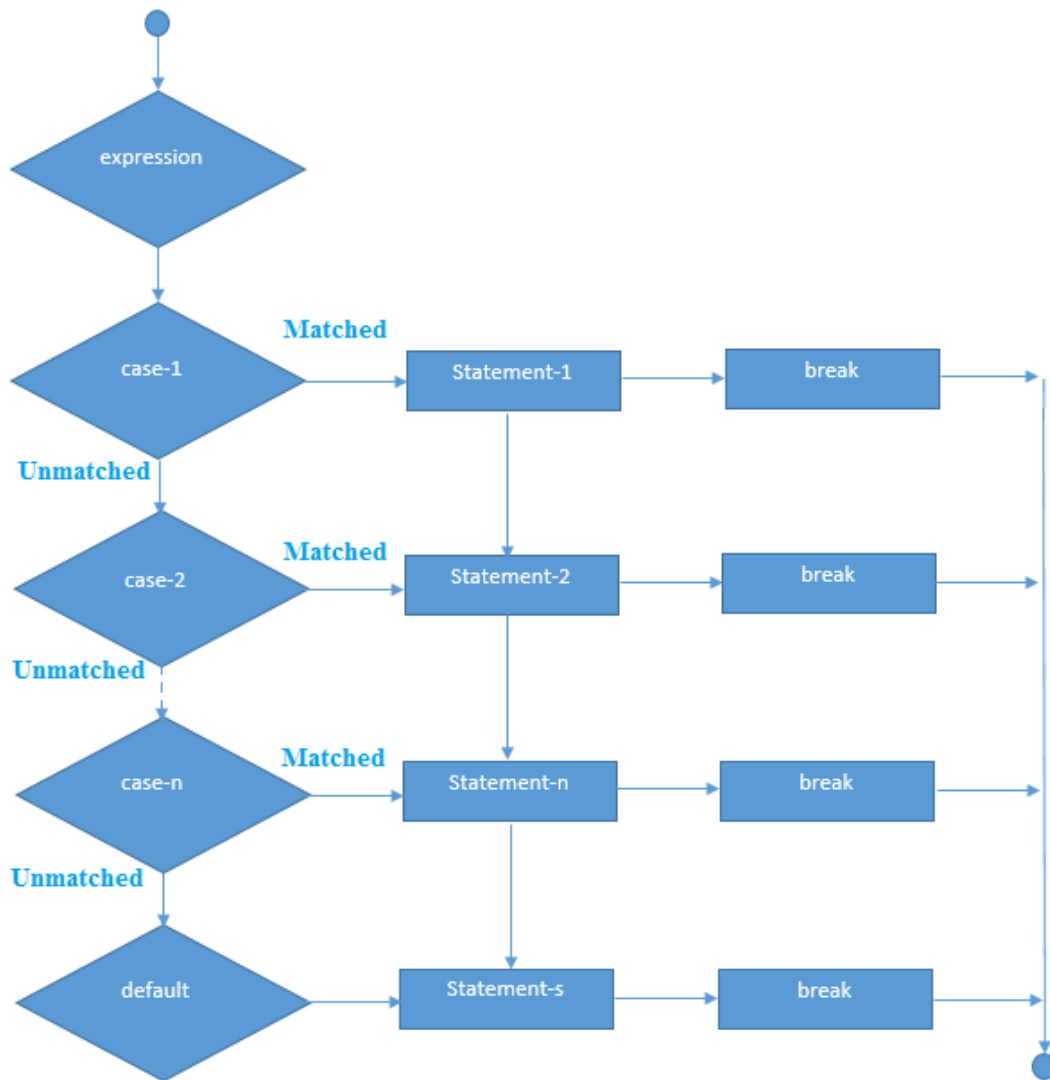
```
switch (expression)
{
    case value1:
        //code to be executed;
        break; //optional
    case value2:
        //code to be executed;
        break; //optional
    .....
    case value n:
        //code to be executed;
        break; //optional
    default:
        code to be executed if all cases are not matched;
}
```

Switch statement in C tests the value of a variable and compares it with multiple cases. Once the case match is found, a block of statements associated with that particular case is executed.

Each case in a block of a switch has a different name/number which is referred to as an identifier. The value provided by the user is compared with all the cases inside the switch block until the match is found.

Rules for switch statement in C language

- 1) The switch expression must be of an integer or character type.
- 2) The case value must be an integer or character constant.
- 3) The case value can be used only inside the switch statement.
- 4) The break statement in switch case is not must. It is optional. If there is no break statement found in the case, all the cases will be executed present after the matched case. It is known as fall through the state of C switch statement.



Functioning of switch case statement

First, the integer expression specified in the switch statement is evaluated. This value is then matched one by one with the constant values given in the different cases. If a match is found, then all the statements specified in that case are executed along with the all the cases present after that case including the default statement. No two cases can have similar values. If the matched case contains a break statement, then all the cases present after that will be skipped, and the control comes out of the switch. Otherwise, all the cases following the matched case will be executed.

Let's see a simple example of c language switch statement.

```
#include<stdio.h>
int main()
{
    int number=0;
    printf("enter a number:");
    scanf("%d",&number);
```

```

switch(number)
{
case 10:
printf("number is equals to 10");
break;
case 50:
printf("number is equal to 50");
break;
case 100:
printf("number is equal to 100");
break;
default:
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

```

Switch case example 2

```

#include <stdio.h>
int main()
{
int x = 10, y = 5;
switch(x>y && x+y>0)
{
case 1:
printf("hi");
break;
case 0:
printf("bye");
break;
default:
printf(" Hello bye ");
}
}

```

Output

```

hi

```

C Switch statement is fall-through

In C language, the switch statement is fall through; it means if you don't use a break statement in the switch case, all the cases after the matching case will be executed.

Let's try to understand the fall through state of switch statement by the example given below.

```
#include<stdio.h>

int main()
{
    int number=0;
    printf("enter a number:");
    scanf("%d",&number);
    switch(number){
        case 10:
            printf("number is equal to 10\n");
        case 50:
            printf("number is equal to 50\n");
        case 100:
            printf("number is equal to 100\n");
        default:
            printf("number is not equal to 10, 50 or 100");
    }
    return 0;
}
```

Output

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

Output

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


Nested-Switch Statement:

Nested-Switch statements refers to Switch statements inside of another Switch Statements.

Syntax:

```
switch(n)
{
    // code to be executed if n = 1;
    case 1:

        // Nested switch
        switch(num)
        {
            // code to be executed if num = 10
            case 10:
                statement 1;
                break;
            // code to be executed if num = 20
            case 20:
                statement 2;
                break;
            // code to be executed if num = 30
            case 30:
                statement 3;
                break;
            // code to be executed if num
            // doesn't match any cases
            default:
        }
        break;
    // code to be executed if n = 2;
    case 2:
        statement 2;
        break;
    // code to be executed if n = 3;
    case 3:
        statement 3;
        break;
```

```
// code to be executed if n doesn't match any cases
default:
}
```

Example#1

```
#include <stdio.h>
int main()
{
    int x = 1, y = 2;
    // Outer Switch
    switch (x)
    {
        // If x == 1
        case 1:
            // Nested Switch
            switch (y) {
                // If y == 2
                case 2:
                    printf( "Choice is 2");
                    break;
                // If y == 3
                case 3:
                    printf( "Choice is 3");
                    break;
            }
            break;
        // If x == 4
        case 4:
            printf( "Choice is 4");
            break;
        // If x == 5
        case 5:
            printf( "Choice is 5");
            break;
        default:
```

```
    printf( "Choice is other than 1, 2 3, 4, or 5");  
    break;  
}  
return 0;  
}
```

Output:

Choice is 2

Example #2

```
#include <stdio.h>  
  
int main () {  
    /* local variable definition */  
    int a = 100;  
    int b = 200;  
    switch(a) {  
        case 100:  
            printf("This is part of outer switch\n", a );  
            switch(b) {  
                case 200:  
                    printf("This is part of inner switch\n", a );  
            }  
        }  
    printf("Exact value of a is : %d\n", a );  
    printf("Exact value of b is : %d\n", b );  
    return 0;  
}
```

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

This is part of outer switch

This is part of inner switch

Exact value of a is : 100

Exact value of b is : 200

Write a C program to print day of week name using switch case.

Iteration Statements in C

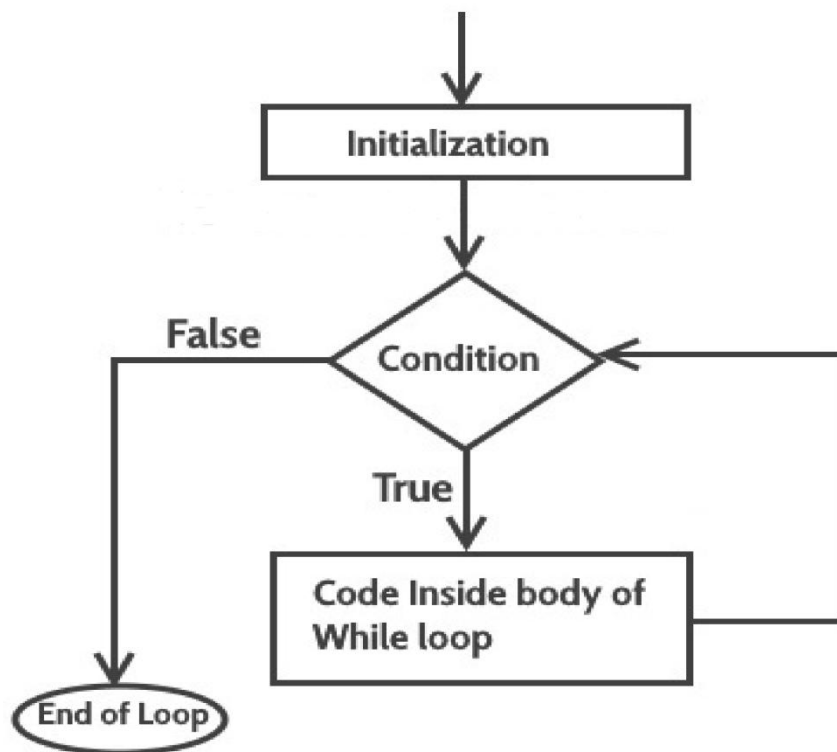
Iteration is when the same procedure is repeated multiple times. Iteration statements create loops in the program. A loop is used for executing a block of statements repeatedly until a given condition returns false. In other words, it repeats the set of statements until the condition for termination is met. A **Loop** executes the sequence of statements many times until the stated condition becomes false. A loop consists of two parts, a body of a loop and a control statement. Iteration statements in C are *while, do-while and for loop*.

1. while Loop

The while loop in C is most fundamental loop statement. It repeats a statement or block while its controlling expression is true. Here is its general form:

```
while (condition)
{
    // body of loop (code to be executed)
}
```

The condition can be any boolean expression. The body of the loop will be executed as long as the conditional expression is true.



How while loop works?

The while loop evaluates the test expression inside the parenthesis ().

If the test expression is true, statements inside the body of while loop are executed. Then, the test expression is evaluated again.

The process goes on until the test expression is evaluated to false.

If the test expression is false, the loop terminates (ends).

Example

```
#include <stdio.h>

void main ()
{
    int count=1;
    while (count <= 4)
    {
        printf("%d ", count);
        count++;
    }
}
```

The output of the program is:

1 2 3 4

Example

```
#include <stdio.h>

int main()
{
    int var=1;
    while (var <=2)
    {
        printf("%d ", var);
    }
}
```

The program is an example of **infinite while loop**. Since the value of the variable var is same (there is no ++ or -- operator used on this variable, inside the body of loop) the condition var<=2 will be true forever and the loop would never terminate.

Example

```
#include <stdio.h>

int main()
{
    int var = 6;
    while (var >=5)
    {
        printf("%d", var);
        var++;
    }
    return 0;
}
```

Infinite loop: var will always have value ≥ 5 so the loop would never end.

Example

```
#include <stdio.h>

int main ()
{
    int a = 10;
    while (a < 20 )
    {
        printf("value of a: %d\n", a);
        a++;
    }
    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: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19

Program to print table for the given number using while loop in C

```
#include<stdio.h>

int main()
{
    int i=1,number;
    printf("Enter a number: ");
    scanf("%d",&number);
    while(i<=10)
    {
        printf("%d \n",(number*i));
        i++;
    }
    return 0;
}
```

Output

Enter a number: 50

50
100
150
200
250
300
350
400
450
500

2. do-while statement

A do...while loop in C is similar to the while loop except that the condition is always executed after the body of a loop. The *do-while loop always executes its body at least once*, because its conditional expression is at the bottom of the loop. Its general form is:

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

Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates. As with all of C's loops, condition must be a Boolean expression.

```
#include <stdio.h>
int main ()
{
    int n = 10;
    do
    {
        printf("tick %d", n);
        printf("\n");

        n--;
    }while (n > 0);
}
```

Example

```
#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
```

While vs do..while loop in C**Using while loop:**

```
#include <stdio.h>

int main()
{
    int i=0;
    while(i==1)
    {
        printf("while vs do-while");
    }
    printf("Out of loop");
}
```

Output:

```
Out of loop
```

Same example using do-while loop

```
#include <stdio.h>

int main()
{
    int i=0;
    do
    {
        printf("hello\n");
    } while(i==1);
    printf("Out of loop");
}
```

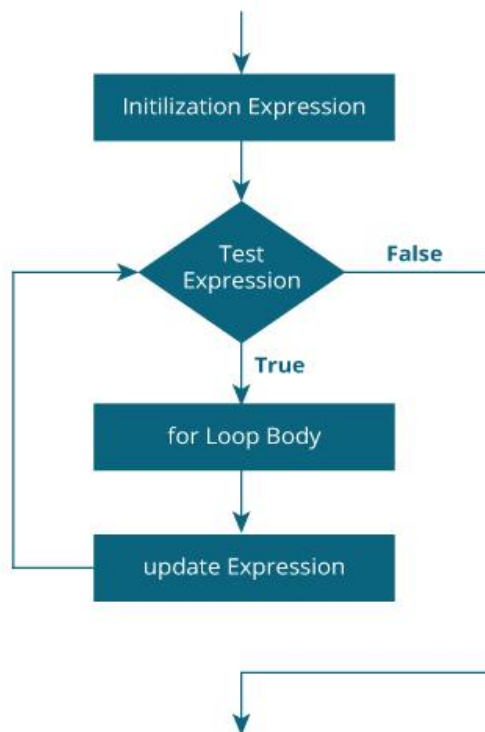
Output:

```
hello
Out of loop
```

3. for loop

Here is the general form of the traditional for statement:

```
For (initialization; condition; iteration)
{
    // body
}
```



Example:

```
//example program to illustrate for looping
#include <stdio.h>
int main ()
{
    int n;
    for(n = 10; n>0 ; n--)
    {
        printf("tick %d",n);
        printf("\n");
    }
}
```

There can be more than one statement in initialization and iteration section. They must be separated with comma.

Example:

```
#include <stdio.h>

int main ()
{
    int a, b;
    for (a = 1, b = 4; a < b; a++, b--)
    {
        printf("a = %d \n", a);
        printf("b = %d \n", b);
    }
}
```

The output of the program is:

```
a = 1
b = 4
a = 2
b = 3
```

Here, the initialization portion sets the values of both a and b. The two comma separated statements in the iteration portion are executed each time the loop repeats.

Nested Loops

Loops can be nested as per requirement. Here is example of nesting the loops.

```
// nested loops
#include <stdio.h>

int main ()
{
    int i, j;
    for (i = 0; i < 8; i++)
    {
        for (j = i; j < 8; j++)
            printf("*");
        printf("\n");
    }
}
```

```
}
```

Here, two for loops are nested. The number times inner loop iterates depends on the value of i in outer loop.

The output of the program is:

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

#Write a C program to display first 10 natural numbers using for loop.

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

```
void main()
```

```
{
```

```
    int i;
```

```
        printf("The first 10 natural numbers are:\n");
```

```
        for (i=1;i<=10;i++)
```

```
        {
```

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

```
        }
```

```
printf("\n");
```

```
}
```

Output

The first 10 natural numbers are:

1 2 3 4 5 6 7 8 9 10

#Write a C program to display the sum of first 10 natural numbers

```

#include <stdio.h>

void main()
{
    int j, sum = 0;
    printf("The first 10 natural number is:\n");
    for (j = 1; j <= 10; j++)
    {
        sum = sum + j;
        printf("%d ",j);
    }
    printf("\nThe Sum is: %d\n", sum);
}

```

Output

The first 10 natural number is:

1 2 3 4 5 6 7 8 9 10

The Sum is: 55

For Loop variations

We can use 2 or more loop control variables

```

#include<stdio.h>

int main()
{
    int i,j;
    for(i=0, j=10; i<10; i++, j--)
    {
        printf("%d + %d = %d\n",i,j,i+j);
    }

    printf("The value of i is %d",i);
    return 0;
}

```

In the example above, i and j both are initialized, separated by a comma operator. In the increment expression, i is incremented and j is decremented.

Jump Statements In C

Jump statements are used to interrupt the normal flow of program. Jump Statement makes the control jump to another section of the program unconditionally when encountered. It is usually used to terminate the loop or switch-case instantly. It is also used to escape the execution of a section of the program.

Types of Jump Statements

- a) Break
- b) Continue
- c) GoTo
- d) Return
- e) Exit

Break Statement

A **break** statement is used to terminate the execution of the rest of the block where it is present and takes the control out of the block to the next statement. When compiler finds the break statement inside a loop or switch, compiler will abort the loop and continue to execute statements followed by loop. The break statement in C can be used in the following two scenarios:

- 1. With switch case
- 2. With loop

Examples of break statement

Example 1

```
#include<stdio.h>
#include<conio.h>
void main ()
{
    int i;
    for(i = 1; i<=10; i++)
    {
        if(i == 5)
            break;
        printf("%d ",i);
    }
}
```

Output

1 2 3 4

Example 2

```
#include<stdio.h>

void main()
{
    int a=1;
    while(a<=10)
    {
        if(a==5)
            break;
        printf("%d" ,a);
        a++;
    }
}
```

Output :

1 2 3 4

Example 3

```
#include<stdio.h>

void main ()
{
    int i = 0;
    while(1)
    {
        printf("%d ",i);
        i++;
        if(i == 10)
            break;
    }
    printf("came out of while loop");
}
```

Output

0 1 2 3 4 5 6 7 8 9 came out of while loop

Continue Statement

The continue statement is also used inside loop. Continue statement is opposite of break statement, instead of terminating the loop, it forces to execute the next iteration of the loop. As the name suggest the continue statement forces the loop to continue or execute the next iteration. When the continue statement is executed in the loop, the code inside the loop following the continue statement will be skipped and next iteration of the loop will begin.

Example 1

```
#include<stdio.h>
#include<conio.h>
void main ()
{
    int i;
    for(i = 1; i<=10; i++)
    {
        if(i == 5)
            continue;
        printf("%d ",i);
    }
}
```

Output

1 2 3 4 6 7 8 9 10

Example 2

```
#include<stdio.h>
void main()
{
    int a=0;
    while(a<10)
    {
        a++;
        if(a==5)
            continue;
        printf("\nStatement %d.",a);
    }
    printf("\nEnd of Program.");
}
```


Output:

Statement 1.
Statement 2.
Statement 3.
Statement 4.
Statement 6.
Statement 7.
Statement 8.
Statement 9.
Statement 10.
End of Program.

Goto Statement

The goto statement is a jump statement which jumps from one point to another point within a function.

Syntax of goto statement

```
goto label;  
-----  
-----  
label:  
-----  
-----
```

In the above syntax, label is an identifier. When, the control of program reaches to goto statement, the control of the program will jump to the label: and executes the code after it.

Example of goto statement

```
#include<stdio.h>  
void main()  
{  
    printf("\nStatement 1.");  
    printf("\nStatement 2.");  
    printf("\nStatement 3.");  
    goto last;  
  
    printf("\nStatement 4.");
```

```

        printf("\nStatement 5.");
        last:
        printf("\nEnd of Program.");
    }

```

Output :

```

Statement 1.
Statement 2.
Statement 3.
End of Program.

```

Example 2

```

#include <stdio.h>
#include <conio.h>
int main()
{
    int english, chemistry, computers, physics, maths;
    float Total, Average, Percentage;
    printf("Please Enter the marks of five subjects: \n");
    scanf("%d%d%d%d%d", &english, &chemistry, &computers, &physics, &maths);
    Total = english + chemistry + computers + physics + maths;
    Average = Total / 5;
    goto label;
    Percentage = (Total / 500) * 100;
    label:
    printf("Total Marks = %.2f\n", Total);
    printf("Average Marks = %.2f\n", Average);
    printf("Marks Percentage = %.2f", Percentage);
    getch();
    return 0;
}

```

Return statement

The return jump statement is used to return some value or simply pass the control to the calling function. This statement does not mandatorily need any conditional statements. As soon as the statement is executed, the **flow of the program stops immediately** and returns the control from where it was called. An important point to be taken into consideration is that **return** can only be used in functions.

Example:

```
#include<stdio.h>

int sum(int , int);

void main()
{
    int num1, num2, res;
    printf("\nEnter the two numbers : ");
    scanf("%d %d", &num1, &num2);
    res = sum(num1, num2);
    printf("\n Addition of two number is : %d ", res);
}

int sum(int num1, int num2)
{
    int num3;
    num3 = num1 + num2;
    return (num3);
}
```

Exit Statement

An exit statement is used to terminate the current execution flow. As soon as exit statement is found, it will terminate the program. It has single argument of zero. For example: exit(0);

In the C Language, the required header for the exit function is: #include <stdlib.h>

Example

```
#include<stdio.h>

int main()
{
    int i,j;
    for (i=1;i<=10;i++)
    {
        printf("%d", i);
    }
}
```

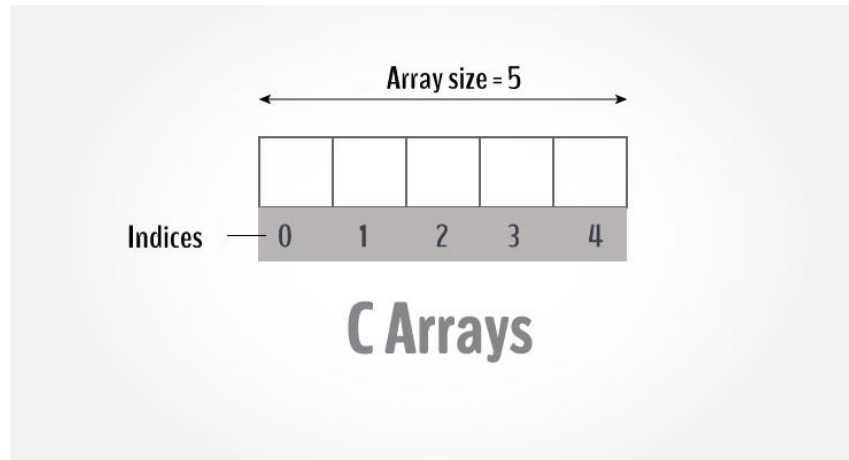
```
    }  
    printf("\n");  
    exit(0);  
    for (j=1;j<=10;j++)  
    {  
        printf("%d", j);  
    }  
    return 0;  
}
```

Unit-3

Principles of Programming using C (PPC)

Array

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. The array is the simplest data structure where each data element can be randomly accessed by using its index number.



In other words we can say that an array is a variable that can store multiple values. For example, if we want to store 100 integers, we can create an array for it.

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

Declaration of C Array

We can declare an array in the C language in the following way.

```
data_type array_name[array_size];
```

EX:

```
int marks[5];
```

Initialization of C Array

The simplest way to initialize an array is by using the index of each element. We can initialize each element of the array by using the index. Consider the following example.

```
marks[0]=80; //initialization of array
```

```
marks[1]=60;
```

```
marks[2]=70;
```

```
marks[3]=85;
```

```
marks[4]=75;
```

80	60	70	85	75
marks[0]	marks[1]	marks[2]	marks[3]	marks[4]

Initialization of Array

C array example

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

```
int main()
```

```
{
```

```
int i;
```

```
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 also initialize the c array at the time of declaration.

```
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};
```

Accessing individual elements of an array

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

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

```
void main()
{
int a[]={2,4,10};
printf(" %d \n %d", a[0], a[1]);
printf("\n %d", a[2]);
getch();
}
```

Output

```
2
4
10
```

Write a program to insert elements into an array and print the elements of the array.

```
#include <stdio.h>
int main()
{
    int a[1000],i,n;
    printf("Enter size of array: ");
    scanf("%d",&n);
    printf("Enter %d elements in the array : ", n);
    for(i=0; i<=n-; i++)
    {
        scanf("%d", &a[i]);
    }
    printf("\nElements in array are: ");
    for(i=0;i<n;i++)
    {
        printf("%d ", a[i]);
    }
    return 0;
}
```

Output

```
Enter size of array: 5
Enter 5 elements in the array: 1
2
3
4
5

Elements in array are: 1 2 3 4 5
```

Write a program in C to find the sum of all elements of an array.

```
#include <stdio.h>
#include <conio.h>
int main()
{
    int a[1000],i,n,sum=0;
    printf("Enter size of the array : ");
    scanf("%d",&n);
    printf("Enter elements in array : ");
    for(i=0; i<n; i++)
    {
        scanf("%d",&a[i]);
    }
    for(i=0; i<n; i++)
    {
        sum =sum+ a[i];
    }
    printf("sum of array is : %d",sum);
    return 0;
}
```


Multi Dimensional Array

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

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

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

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

A two-dimensional (2D) array is an array of arrays. A three-dimensional (3D) array is an array of arrays of arrays. In C programming an array can have two, three, or even ten or more dimensions. The maximum dimensions of an array in a C program depend on which compiler is being used. More dimensions in an array mean more data to be held, but also means greater difficulty in managing and understanding arrays.

A multidimensional array is declared using the following syntax:

```
data_type array_name[d1][d2][d3][d4].....[dn];
```

Two-dimensional Arrays

The simplest form of multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size [x][y], we would write something as follows –

```
data_type arrayName [ x ][ y ];
```

Where **datatype** can be any valid C data type and **arrayName** will be a valid C identifier. A two-dimensional array can be considered as a table which will have x number of rows and y number of columns. A two-dimensional array **a**, which contains three rows and four columns can be shown as follows-

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in the array **a** is identified by an element name of the form **a[i][j]**, where 'a' is the name of the array, and 'i' and 'j' are the subscripts that uniquely identify each element in 'a'.

Initializing a multidimensional array

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

Initialization of a 2d array

// Different ways to initialize two-dimensional array

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

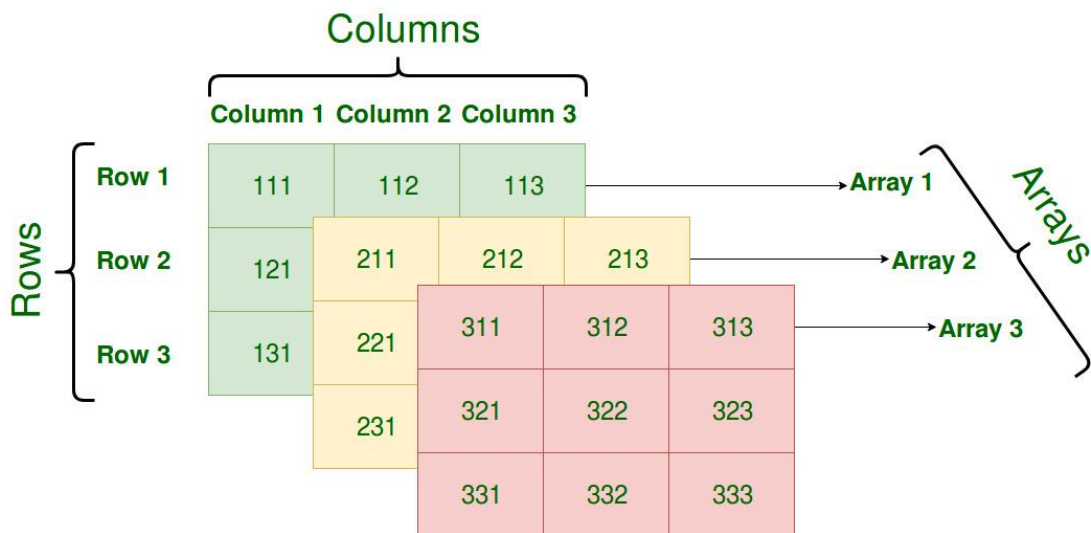
```
int c[][3] = {{1, 3, 0}, {-1, 5, 9}};
```

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

Initialization of a 3d array

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

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



Example: 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 2nd matrix\n");
    for (int i = 0; i < 2; ++i)
        for (int j = 0; j < 2; ++j)
        {
            printf("Enter b%d%d: ", i + 1, j + 1);
            scanf("%f", &b[i][j]);
        }

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

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

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

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

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
```

Passing an array element to a function

Passing array elements to a function is similar to passing variables to a function.

Example : Passing an array

```
#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};
    // Passing second and third elements to display()
    display(ageArray[1], ageArray[2]);
    return 0;
}
```

Output

```
8
4
```

C - Strings

Strings are defined as an array of characters. In C programming, a string is a sequence of characters terminated with a null character `\0`. For example:

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

```
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.

c		s	t	r	i	n	g	\0
---	--	---	---	---	---	---	---	----

How to declare a string?

Here's how we can declare strings:

```
char s[5];
```

s[0]	s[1]	s[2]	s[3]	s[4]

How to initialize strings?

We can initialize strings in a number of ways as:-

```
char c[] = "abcd";
```

```
char c[5] = "abcd";
```

```
char c[] = {'a', 'b', 'c', 'd', '\0'};
```

```
char c[5] = {'a', 'b', 'c', 'd', '\0'};
```

c[0]	c[1]	c[2]	c[3]	c[4]
a	b	c	d	\0

String Input: Read a String

We 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.

In order to read a string contains spaces, we use the **gets()** function. Gets ignores the whitespaces. It stops reading when a newline is reached (the Enter key is pressed).

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

String Output: Print/Display a String

The standard **printf** function is used for printing or displaying Strings in C on an output device. The format specifier used is %s

Example,

```
printf("%s", name);
```

String output is also done with the puts() functions.

Example

```
#include <stdio.h>

int main()
{
    char ch[100];
    printf("Enter any string\n");
    gets(ch);
    printf("\nThe string is - ");
    puts(ch);
    return 0;
}
```

C supports a wide range of functions that manipulate null-terminated strings –

Sr.No.	Function & Purpose
1	strcpy(s1, s2); Copies string s2 into string s1.
2	strcat(s1, s2); Concatenates string s2 onto the end of string s1.
3	strlen(s1); Returns the length of string s1.
4	strcmp(s1, s2); Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	strchr(s1, ch); Returns a pointer to the first occurrence of character ch in string s1.
6	strstr(s1, s2); Returns a pointer to the first occurrence of string s2 in string s1.

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);
    return 0;
}
```

Output

C programming

Example: C strcat() function

```
#include <stdio.h>
#include <string.h>
int main()
{
    char str1[100] = "This is ";
    char str2[] = " a program";
    // concatenates str1 and str2 and the resultant string is stored in str1.
    strcat(str1, str2);
    puts(str1);
    puts(str2);
    return 0;
}
```

Output

This is a program
a program

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
```

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) = 32
strcmp(str1, str3) = 0
```

Unit-4

Principles of Programming using C (PPC)

A function is a block of code that performs a specific task. In c, we can divide a large program into the basic building blocks known as function. The function contains the set of programming statements enclosed by {}. A function can be called multiple times to provide reusability and modularity to the C program.

Need of functions in C

Functions are used because of following reasons –

- a) To improve the readability of code.
- b) Improves the reusability of the code, same function can be used in any program rather than writing the same code from scratch.
- c) Debugging of the code would be easier if we use functions, as errors are easy to be traced.
- d) Reduces the size of the code, duplicate set of statements are replaced by function calls.

Types of function

There are two types of function in C programming:

- Standard library functions
- User-defined functions

Standard library functions

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

These functions are defined in header files. For example,

The printf() is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in the stdio.h header file.

Hence, to use the printf()function, we need to include the stdio.h header file using #include <stdio.h>.

The sqrt() function calculates the square root of a number. The function is defined in the math.h header file.

User-defined function

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

Advantage of functions in C

There are the following advantages of C functions.

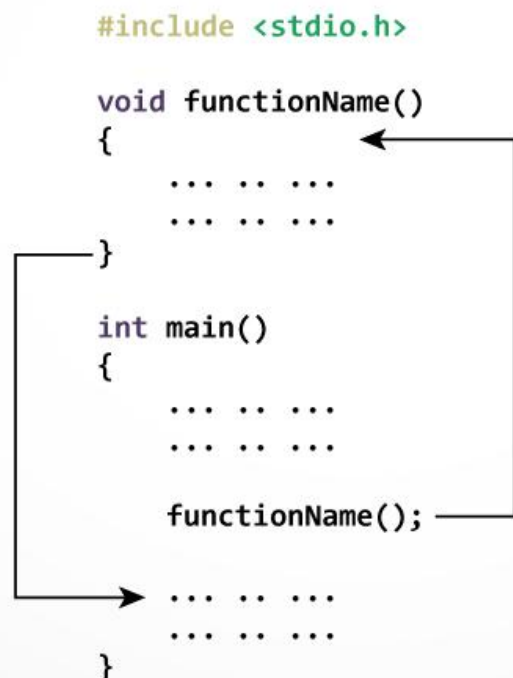
- By using functions, we can avoid rewriting same logic/code again and again in a program.
- We can call C functions any number of times in a program and from any place in a program.
- We can track a large C program easily when it is divided into multiple functions.
- Reusability is the main achievement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

Function Aspects

There are three aspects of a C function.

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

How function works in C programming?



Function Declarations

A function declaration has the following parts –

return_type function_name(parameter list);

Example of function declaration is as follows –

int max(int num1, int num2);

Parameter names are not important in function declaration only their type is required, so the following is also a valid declaration –

```
int max(int, int);
```

Function declaration is required when we define a function in one source file and call that function in another file. In such case, we should declare the function at the top of the file calling the function.

Defining a Function

The general form of a function definition in C programming language is as follows –

```
return_type function_name( parameter list )
```

```
{  
    body of the function  
}
```

A function definition in C programming consists of a function header and a function body. Here are all the parts of a function –

Return Type – A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.

Function Name – This is the actual name of the function. The function name and the parameter list together constitute the function signature.

Parameters – A parameter is like a placeholder. When a function is invoked, we pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

Function Body – The function body contains a collection of statements that define what the function does.

Example

```
int max(int x, int y)  
{  
    if (x > y)  
        return x;  
    else  
        return y;  
}
```

Calling a Function

While creating a C function, we give a definition of what the function has to do. To use a function, we will have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function, we simply need to pass the required parameters along with the function name, and if the function returns a value, then we can store the returned value.

For example –

```
#include <stdio.h>

int max(int x, int y)
{
    if (x > y)
        return x;
    else
        return y;
}

int main
{
    int a = 10, b = 20;
    int m = max(a, b);
    printf("m is %d", m);
    return 0;
}
```

Example 2

```
#include<stdio.h>

float square ( float x );

int main( )
{
    float m, n ;
    printf ( "\nEnter some number for finding square \n");
    scanf ( "%f", &m ) ;
    n = square ( m ) ;
}
```

```

printf ( "\nSquare of the given number %f is %f",m,n );
}
float square ( float x ) // function definition
{
    float p ;
    p = x * x ;
    return ( p ) ;
}

```

OUTPUT:

```

Enter some number for finding square
2
Square of the given number 2.000000 is 4.000000

```

C – math.h library functions

LIST OF INBUILT C FUNCTIONS IN MATH.H FILE:

- “math.h” header file supports all the mathematical related functions in C language. All the arithmetic functions used in C language are given below.

No.	Function	Description
1)	ceil(number)	rounds up the given number. It returns the integer value which is greater than or equal to given number.
2)	floor(number)	rounds down the given number. It returns the integer value which is less than or equal to given number.
3)	sqrt(number)	returns the square root of given number.
4)	pow(base, exponent)	returns the power of given number.
5)	abs(number)	returns the absolute value of given number.

C Math Example

Let's see a simple example of math functions found in math.h header file.

```
#include<stdio.h>
#include <math.h>
int main(){
printf("\n%f",ceil(3.6));
printf("\n%f",ceil(3.3));
printf("\n%f",floor(3.6));
printf("\n%f",floor(3.2));
printf("\n%f",sqrt(16));
printf("\n%f",sqrt(7));
printf("\n%f",pow(2,4));
printf("\n%f",pow(3,3));
printf("\n%d",abs(-12));
return 0;
}
```

Output:

```
4.000000
4.000000
3.000000
3.000000
4.000000
2.645751
16.000000
27.000000
12
```

Function Arguments

C programming function arguments also known as parameters are the variables that will receive the data sent by the calling program. These arguments serve as input data to the function to carry out the specified task.

Basically, there are two types of arguments:

- **Actual arguments**
- **Formal arguments**

```

#include <stdio.h>
return_type tunc_name(arguments);
{
    .....
    .....
}

Int main()
{
    .....
    tunc_name(arguments_value);
    .....
    return 0;
}

```

↑
formal arguments

↓
actual arguments

The variables declared in the function prototype or definition are known as **Formal arguments** and the values that are passed to the called function from the main function are known as **Actual arguments**.

The actual arguments and formal arguments must match in number, type, and order.

Following are the two ways to pass arguments to the function:

- **Call by value**
- **Call by reference**

Type	Description
Call by Value	-> The actual parameter is passed to a function. -> New memory area created for the passed parameters, can be used only within the function. -> The actual parameters cannot be modified here.
Call by Reference	-> Instead of copying variable; an address is passed to function as parameters. -> Address operator(&) is used in the parameter of the called function. -> Changes in function reflect the change of the original variables.

Example- Let us call the function swap() by passing actual values as in the following example –

```

#include <stdio.h>
/* function declaration */
void swap(int x, int y);
int main ()
{
    /* local variable definition */

```



```

int a = 100;
int b = 200;

printf("Before swap, value of a : %d\n", a );
printf("Before swap, value of b : %d\n", b );

/* calling a function to swap the values */
swap(a, b);

printf("After swap, value of a : %d\n", a );
printf("After swap, value of b : %d\n", b );

return 0;
}

void swap(int x, int y) {
    int temp;

    temp = x; /* save the value of x */
    x = y;    /* put y into x */
    y = temp; /* put temp into y */
}

```

Output

Before swap, value of a : 100

Before swap, value of b : 200

After swap, value of a : 200

After swap, value of b : 100

Example-Let us now call the function **swap()** by passing values by reference as in the following example –

```

include <stdio.h>
void swap(int *x, int *y);
int main ()
{
    /* local variable definition */
    int a = 100;
    int b = 200;
    printf("Before swap, value of a : %d\n", a );
    printf("Before swap, value of b : %d\n", b );
    /* calling a function to swap the values */
    swap(&a, &b);
    printf("After swap, value of a : %d\n", a );
    printf("After swap, value of b : %d\n", b );
    return 0;
}

```

```

}
void swap(int *x, int *y) {
    int temp;
    temp = *x; /* save the value of x */
    *x = *y;   /* put y into x */
    *y = temp; /* put temp into y */
}

```

Output

Before swap, value of a : 100

Before swap, value of b : 200

After swap, value of a : 200

After swap, value of b : 100

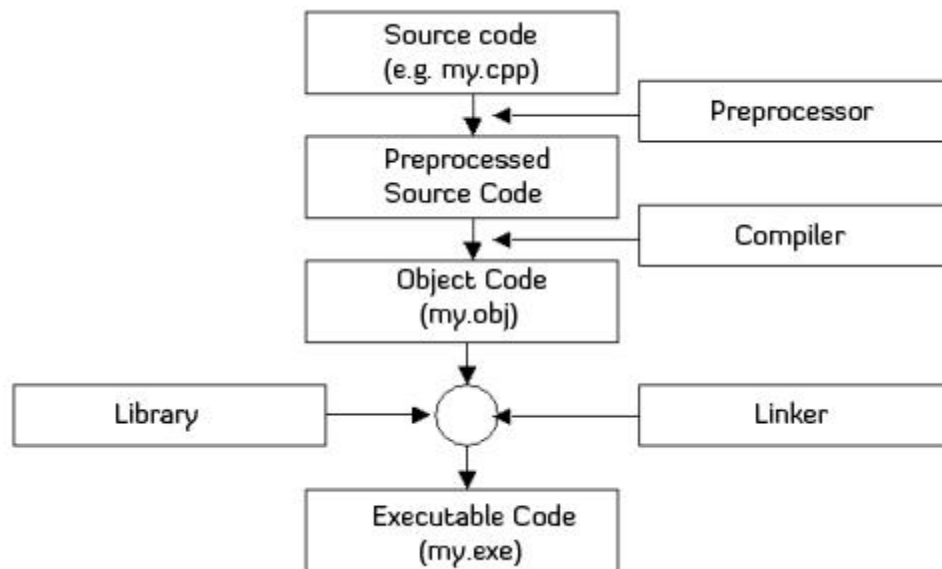
Some Point of Difference between Call by Value and Call by Reference

Parameters	Call by value	Call by reference
Definition	While calling a function, when you pass values by copying variables, it is known as "Call By Values."	While calling a function, in programming language instead of copying the values of variables, the address of the variables is used it is known as "Call By References."
Arguments	In this method, a copy of the variable is passed.	In this method, a variable itself is passed.
Effect	Changes made in a copy of variable never modify the value of variable outside the function.	Change in the variable also affects the value of the variable outside the function.
Alteration of value	Does not allow you to make any changes in the actual variables.	Allows you to make changes in the values of variables by using function calls.
Passing of variable	Values of variables are passed using a straightforward method.	Pointer variables are required to store the address of variables.
Value modification	Original value not modified.	The original value is modified.
Memory Location	Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in the same memory location
Safety	Actual arguments remain safe as they cannot be modified accidentally.	Actual arguments are not Safe. They can be accidentally modified, so you need to handle arguments operations carefully.

C pre – processor

As the name suggests Preprocessors are programs that process our source code before compilation. There are a number of steps involved between writing a program and executing a program.

The source code written by programmers is stored in the file program.c. This file is then processed by preprocessors and an expanded source code file is generated named program. This expanded file is compiled by the compiler and an object code file is generated named program .obj. Finally, the linker links this object code file to the object code of the library functions to generate the executable file program.exe.



Preprocessor programs provide preprocessors directives which tell the compiler to preprocess the source code before compiling. All of these preprocessor directives begin with a '#' (hash) symbol. The '#' symbol indicates that, whatever statement starts with #, is going to the preprocessor program, and preprocessor program will execute this statement. Let's see a list of preprocessor directives.

- #include
- #define
- #undef
- #ifdef
- #ifndef
- #if
- #else
- #elif
- #endif
- #error
- #pragma

A Preprocessor mainly performs three tasks

Removing comments: It removes all the comments. A comment is written only for the humans to understand the code. So, it is obvious that they are of no use to a machine. So, preprocessor removes all of them as they are not required in the execution and won't be executed as well.

This is how to see a file with removed comments in linux) :

Write a C code (let the file name be prog.c). Preprocess it using the command gcc -E prog.c

File inclusion: Including all the files from library that our program needs. In HLL we write **#include** which is a directive for the preprocessor that tells it to include the contents of the library file specified. For example, #include will tell the preprocessor to include all the contents in the library file stdio.h. This can also be written using double quotes – #include “stdio.h”

Note: If the filename is enclosed within angle brackets, the file is searched for in the standard compiler include paths. If the filename is enclosed within double quotes, the search path is expanded to include the current source directory.

Macro expansion: Macros can be called as small functions that are not as overhead to process. If we have to write a function (having a small definition) that needs to be called recursively (again and again), then we should prefer a macro over a function.

So, defining these macros is done by preprocessor.

#define SI 1000

Example.

```
#include <stdio.h>
// macro definition
#define LIMIT 5
int main()
{
    for (int i = 0; i < LIMIT; i++) {
        printf("%d ",i);
    }
    return 0;
}
```

Output:

0 1 2 3 4

In the above program, when the compiler executes the word LIMIT it replaces it with 5. The word 'LIMIT' in the macro definition is called a macro template and '5' is macro expansion.

Note: There is no semi-colon(';') at the end of macro definition. Macro definitions do not need a semi-colon to end.

Storage Class in C

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program.

They precede the type that they modify. A storage class in C is used to describe the following things:

- The variable scope.
- The location where the variable will be stored.
- The initialized value of a variable.
- A lifetime of a variable.
- Who can access a variable?

There are total four types of standard storage classes. The table below represents the storage classes in C.

Storage class	Purpose
auto	It is a default storage class.
extern	It is a global variable.
static	It is a local variable which is capable of returning a value even when control is transferred to the function call.
register	It is a variable which is stored inside a Register.

Storage classes in C

Storage Specifier	Storage	Initial value	Scope	Life
auto	stack	Garbage	Within block	End of block
extern	Data segment	Zero	global Multiple files	Till end of program
static	Data segment	Zero	Within block	Till end of program
register	CPU Register	Garbage	Within block	End of block



auto: This is the default storage class for all the variables declared inside a function or a block. Hence, the keyword `auto` is rarely used while writing programs in C language. Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope). Of course, these can be accessed within nested blocks within the parent block/function in which the auto variable was declared. However, they can be accessed outside their scope as well using the concept of pointers given here by pointing to the very exact memory location where the variables resides. They are assigned a garbage value by default whenever they are declared.

extern: Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used. Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well. So an extern variable is nothing but a global variable initialized with a legal value where it is declared in order to be used elsewhere. It can be accessed within any function/block. Also, a normal global variable can be made extern as well by placing the ‘extern’ keyword before its declaration/definition in any function/block. This basically signifies that we are not initializing a new variable but instead we are using/accessing the global variable only. The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program. For more information on how extern variables work, have a look at [this link](#).

static: This storage class is used to declare static variables which are popularly used while writing programs in C language. Static variables have a property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope. So we can say that they are initialized only once and exist till the termination of the program. Thus, no new memory is allocated because they are not re-declared. Their scope is local to the function to which they were defined. Global static variables can be accessed anywhere in the program. By default, they are assigned the value 0 by the compiler.

register: This storage class declares register variables which have the same functionality as that of the auto variables. The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available. This makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program. If a free register is not available, these are then stored in the memory only. Usually few variables which are to be accessed very frequently in a program are declared with the register keyword which improves the running time of the program. An important and interesting point to be noted here is that we cannot obtain the address of a register variable using pointers.

To specify the storage class for a variable, the following syntax is to be followed:

Syntax:

```
storage_class var_data_type var_name;
```

```
auto int i; or extern int b; or static int c; or register int l;
```

Pointers in C

To access address of a variable, we use the unary operator **&** (ampersand) that returns the address of that variable. For example **&x** will give us address of variable **x**. We have used address numerous times while using the **scanf()** function. As

```
scanf("%d", &var);
```

Here, the value entered by the user is stored in the address of **var** variable. Let's take a example.

```
#include <stdio.h>

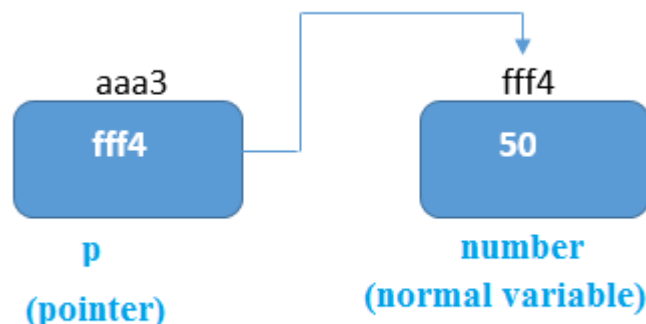
int main()
{
    int var = 5;
    printf("var: %d\n", var);
    printf("address of var: %p", &var);
    return 0;
}
```

Output

var: 5

address of var: 2686778

Pointers (pointer variables) are special variables that are used to store addresses rather than values. This variable can be of type **int**, **char**, **array**, **function**, or any other pointer. The asterisk (***** the same asterisk used for multiplication), declares a pointer.



Pointers

Pointer Syntax

Here is how we can declare pointers.

```
int* p;
```

```
int *p1;
```

```
int * p2;
```

Example

```
#include <stdio.h>

int main()
{
    int a=10; //variable declaration
    int *p;   //pointer variable declaration
    p=&a;     //store address of variable a in pointer p
    printf("Address stored in a variable p is:%x\n",p); //accessing the address
    printf("Value stored in a variable p is:%d\n",*p); //accessing the value
    return 0;
}
```

Output:

Address stored in a variable p is:60ff08

Value stored in a variable p is:10

Operator	Meaning
*	Serves 2 purpose <ol style="list-style-type: none">1. Declaration of a pointer2. Returns the value of the referenced variable
&	Serves only 1 purpose <ul style="list-style-type: none">• Returns the address of a variable

Write a 'C' program to compute the sum of all elements stored in an array Using pointers.

```
#include<stdio.h>
#include<conio.h>
main ()
{
int a [10], I, sum=0,*p;
printf ("enter 10 elements \n");
for (i=0; i<10; i++)
scanf ("%d", & a[i]);
p = a;
for (i = 0; i<10; i++)
{
sum = sum*p;
p++;
}
printf ("the sum is % d", sum);
getch ();
}
```

Pointer Arithmetic

We can perform addition and subtraction of integer constant from pointer variable.

Addition

```
ptr1 = ptr1 + 2;
```

Subtraction

```
ptr1 = ptr1 - 2;
```

We can not perform addition, multiplication and division operations on two pointer variables.

For Example:

ptr1 + ptr2 is not valid

However we can subtract one pointer variable from another pointer variable. We can use increment and decrement operator along with pointer variable to increment or decrement the address contained in pointer variable.

For Example:

```
ptr1++;  
ptr2--;
```

Multiplication

Example:

```
int x = 10, y = 20, z;
```

```
int *ptr1 = &x;
```

```
int *ptr2 = &y;
```

```
z = *ptr1 * *ptr2 ;
```

Will assign 200 to variable z.

Unit-5

PPC

Structures in C

Arrays allow us to define type of variables that can hold several data items of the same kind. Similarly **structure** is another user defined data type available in C that allows us to combine data items of different kinds.

So, **Structure in c is a user-defined data type that enables us to store the collection of different data types.**

Each element of a structure is called a member.

Lets say we need to store the data of students like student name, age, address, id etc. One way of doing this would be creating a different variable for each attribute, however when we need to store the data of multiple students then in that case, we would need to create these several variables again for each student. This is such a big headache to store data in this way.

We can solve this problem easily by using structure. We can create a structure that has members for name, id, address and age and then we can create the variables of this structure for each student.

The **,struct** keyword is used to define the structure.

Syntax:

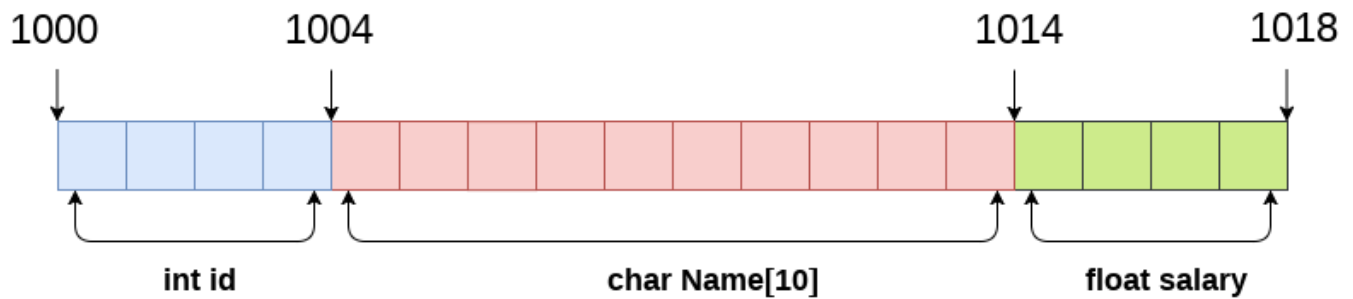
```
struct structure_name
{
    data_type member1;
    data_type member2;
    .
    .
    data_type memeberN;
};
```

Let's see the example to define a structure for an entity employee in c.

```
struct employee
{ int id;
  char name[20];
  float salary;
};
```

The following image shows the memory allocation of the structure employee that is defined in the above example.

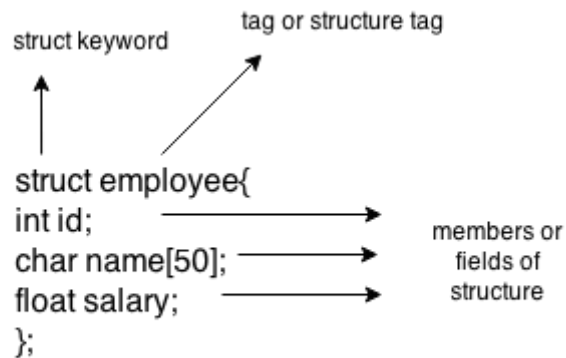
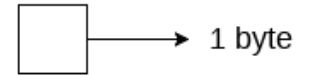
Here, **struct** is the keyword; **employee** is the name of the structure; **id**, **name**, and **salary** are the members or fields of the structure.



```
struct Employee
{
    int id;
    char Name[10];
    float salary;
} emp;
```

sizeof (emp) = 4 + 10 + 4 = 18 bytes

where;
sizeof (int) = 4 byte
sizeof (char) = 1 byte
sizeof (float) = 4 byte



Structure in C

Declaring structure variable

We can declare a variable for the structure so that we can access the member of the structure easily. There are two ways to declare structure variable:

1. By struct keyword within main() function
2. By declaring a variable at the time of defining the structure.

1st way:

Let's see the example to declare the structure variable by struct keyword. It should be declared within the main function.

```
struct employee
{
    int id;
    char name[50];
    float salary;
};
```

Now write given code inside the main() function.

```
struct employee e1, e2;
```

The variables e1 and e2 can be used to access the values stored in the structure. Here, e1 and e2 can be treated in the same way as the objects in C++ and Java.

2nd way:

Let's see another way to declare variable at the time of defining the structure.

```
struct employee
```

```
{ int id;
```

```
  char name[50];
```

```
  float salary;
```

```
}e1,e2;
```

If number of variables are not fixed, use the 1st approach. It provides you the flexibility to declare the structure variable many times.

If no. of variables are fixed, use 2nd approach. It saves your code to declare a variable in main() function.

Accessing members of the structure

There are two ways to access structure members:

1. By . (member or dot operator)
2. By -> (structure pointer operator)

Let's see the code to access the id member of p1 variable by. (member) operator.

```
p1.id
```

C Structure example

Let's see a simple example of structure in C language.

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

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

```
struct employee
```

```
{ int id;
```

```
  char name[50];
```

```
}e1; //declaring e1 variable for structure
```

```
int main( )
```

```
{
```

```
  //store first employee information
```

```
  e1.id=101;
```

```
  strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
```

```

//printing first employee information
printf( "employee 1 id : %d\n", e1.id);
printf( "employee 1 name : %s\n", e1.name);
return 0;
}

```

Output:

```

employee 1 id : 101
employee 1 name : Sonoo Jaiswal

```

Example-2 Let's see another example of the structure in C language to store many employees information.

```

#include<stdio.h>
#include <string.h>
struct employee
{
    int id;
    char name[50];
    float salary;
}e1,e2; //declaring e1 and e2 variables for structure
int main( )
{
    //store first employee information
    e1.id=101;
    strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
    e1.salary=56000;

    //store second employee information
    e2.id=102;
    strcpy(e2.name, "James Bond");
    e2.salary=126000;
    //printing first employee information
    printf( "employee 1 id : %d\n", e1.id);
    printf( "employee 1 name : %s\n", e1.name);
    printf( "employee 1 salary : %f\n", e1.salary);
}

```

```

//printing second employee information
printf( "employee 2 id : %d\n", e2.id);
printf( "employee 2 name : %s\n", e2.name);
printf( "employee 2 salary : %f\n", e2.salary);
return 0;
}

```

Output:

```

employee 1 id : 101
employee 1 name : Sonoo Jaiswal
employee 1 salary : 56000.000000
employee 2 id : 102
employee 2 name : James Bond
employee 2 salary : 126000.000000

```

Example-Write a program in C to Store Information of a Student Using Structure.

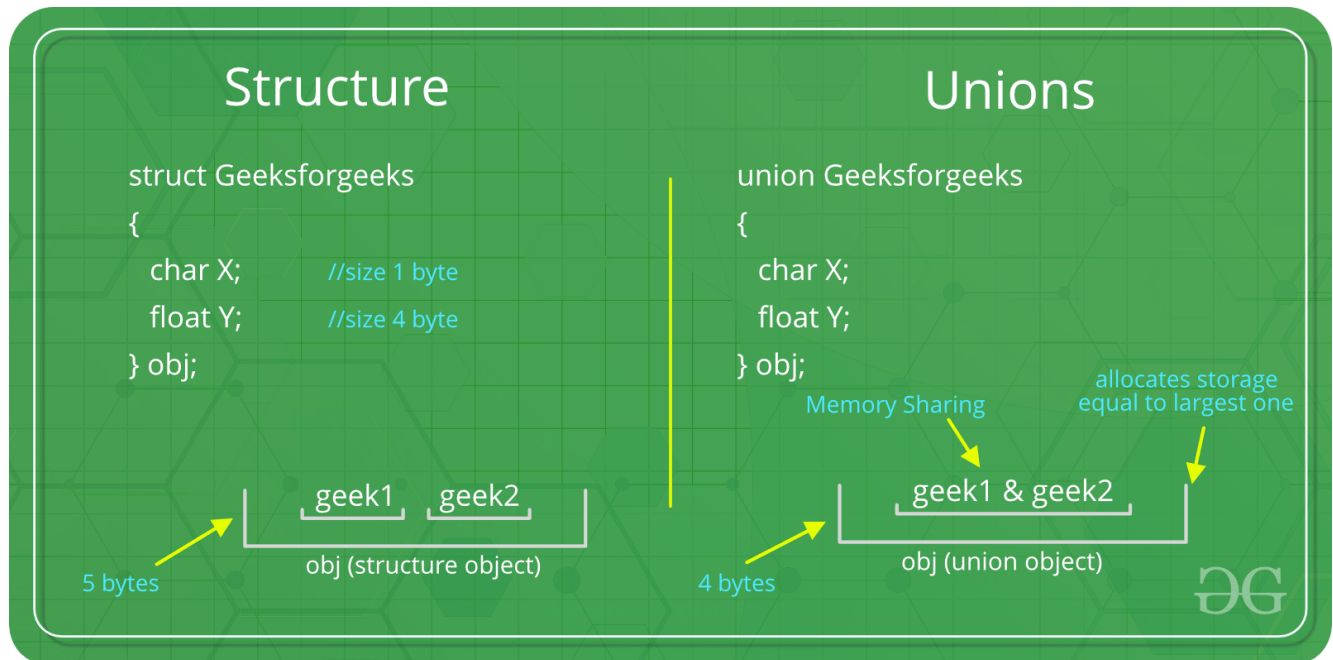
```

#include <stdio.h>
struct student
{
    char name[50];
    int roll;
    float marks;
} s;
int main()
{
    printf("Enter information:\n");
    printf("Enter name: ");
    fgets(s.name, sizeof(s.name), stdin);
    printf("Enter roll number: ");
    scanf("%d", &s.roll);
    printf("Enter marks: ");
    scanf("%f", &s.marks);
    printf("Displaying Information:\n");
    printf("Name: ");
    printf("%s", s.name);
    printf("Roll number: %d\n", s.roll);
    printf("Marks: %.1f\n", s.marks);
    return 0;
}

```

Union in C

A union is a special data type available in C that allows storing different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple purposes.



Defining a Union: To define a union, you must use the **union** statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program. The format of the union statement is as follows:

```
union [union name]
{
    member definition;
    member definition;
    ...
    member definition;
};
```

Example

```
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. To access pointer variables, we use also 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 understand 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

Why this difference in the size of union and structure variables?

Here, the size of sJob is 40 bytes because

the size of name[32] is 32 bytes

the size of salary is 4 bytes

the size of workerNo is 4 bytes

However, the size of uJob is 32 bytes. It's because the size of a union variable will always be the size of its largest element. In the above example, the size of its largest element, (name[32]), is 32 bytes.

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

Similarities between Structure and Union

1. Both are user-defined data types used to store data of different types as a single unit.
2. Their members can be objects of any type, including other structures and unions or arrays. A member can also consist of a bit field.
3. Both structures and unions support only assignment = and sizeof operators. The two structures or unions in the assignment must have the same members and member types.
4. A structure or a union can be passed by value to functions and returned by value by functions. The argument must have the same type as the function parameter. A structure or union is passed by value just like a scalar variable as a corresponding parameter.
5. ‘.’ operator is used for accessing members.

	STRUCTURE	UNION
Keyword	The keyword struct is used to define a structure	The keyword union is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is greater than or equal to the sum of sizes of its members.	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of union is equal to the size of largest member.
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.