

Lecture Notes

For

C Programming

By



LEARNERS TODAY, LEADERS TOMORROW

Introduction to C

C language Tutorial with programming approach for beginners and professionals, helps you to understand the C language tutorial easily. Our C tutorial explains each topic with programs.

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

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

It can be defined by the following ways:

- Mother language
- System programming language
- Procedure-oriented programming language
- Structured programming language
- Mid-level programming language

1) C as a mother language

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

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

2) C as a system programming language

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

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

3) C as a procedural language

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

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

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

4) C as a structured programming language

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

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

5) C as a mid-level programming language

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

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

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

First C Program

In this tutorial, all C programs are given with C compiler so that you can quickly change the C program code.

e.g.: main.c

```
#include <stdio.h>

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

A detailed description of above program is given in next chapters.

History of C Language

Dennis Ritchie - founder of C language

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

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

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

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
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

1) Simple –

C is a simple language in the sense that it provides a structured approach (to break the problem into parts), the rich set of library functions, data types, etc.

2) Machine Independent or Portable –

Unlike assembly language, c programs can be executed on different machines with some machine specific changes. Therefore, C is a machine independent language.

3) Mid-level programming language –

Although, C is intended to do low-level programming. 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.

4) Structured programming language –

C is a structured programming language in the sense that we can break the program into parts using functions. So, it is easy to understand and modify. Functions also provide code reusability.

5) Rich Library –

C provides a lot of inbuilt functions that make the development fast.

6) Memory Management –

It supports the feature of dynamic memory allocation. In C language, we can free the allocated memory at any time by calling the free() function.

7) Speed –

The compilation and execution time of C language is fast since there are lesser inbuilt functions and hence the lesser overhead.

8) Pointer –

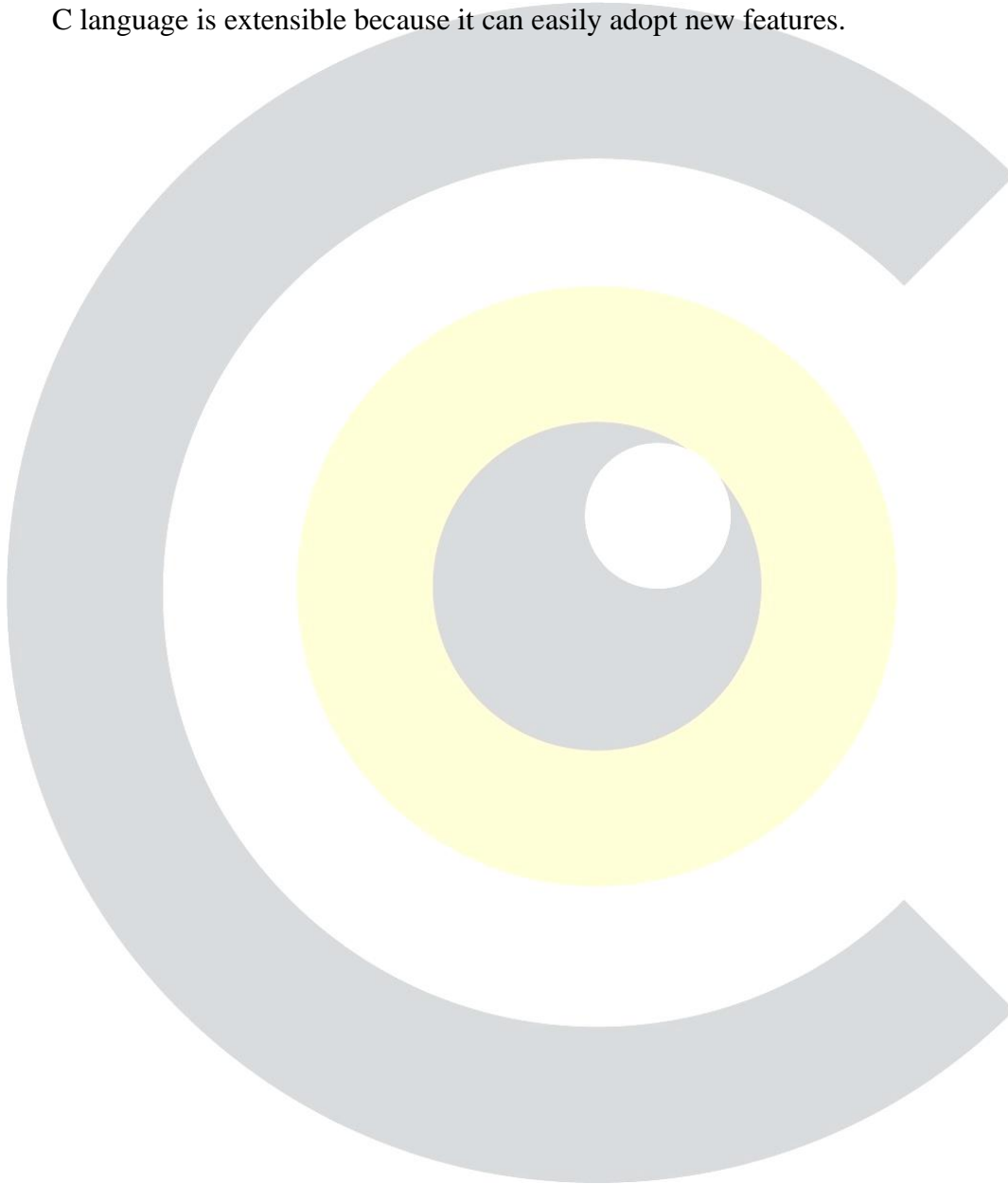
C provides the feature of pointers. We can directly interact with the memory by using the pointers. We can use pointers for memory, structures, functions, array, etc.

9) 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.

10) Extensible –

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



How to write C program

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

```
int main(){
```

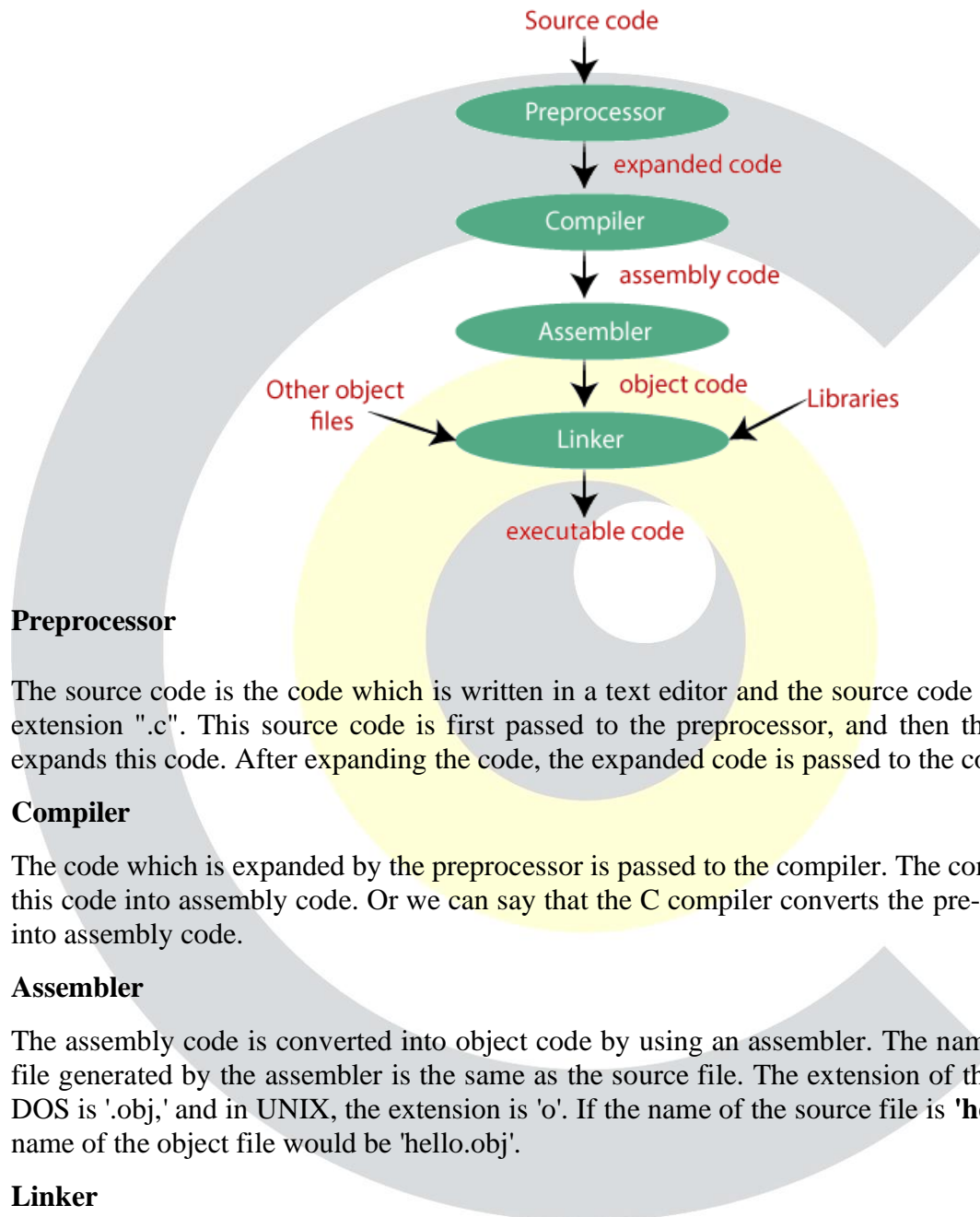
```
    printf("Hello C Language");
```

```
    return 0;
```

```
}
```

- `#include <stdio.h>` includes the standard input output library functions. The `printf()` function is defined in `stdio.h`.
- `int main()` The `main()` function is the entry point of every program in c language.
- `printf()` The `printf()` function is used to print data on the console.
- `return 0` The `return 0` statement, returns execution status to the OS. The 0 value is used for successful execution and 1 for unsuccessful execution.

Compilation process



Preprocessor

The source code is the code which is written in a text editor and the source code file is given an extension ".c". This source code is first passed to the preprocessor, and then the preprocessor expands this code. After expanding the code, the expanded code is passed to the compiler.

Compiler

The code which is expanded by the preprocessor is passed to the compiler. The compiler converts this code into assembly code. Or we can say that the C compiler converts the pre-processed code into assembly code.

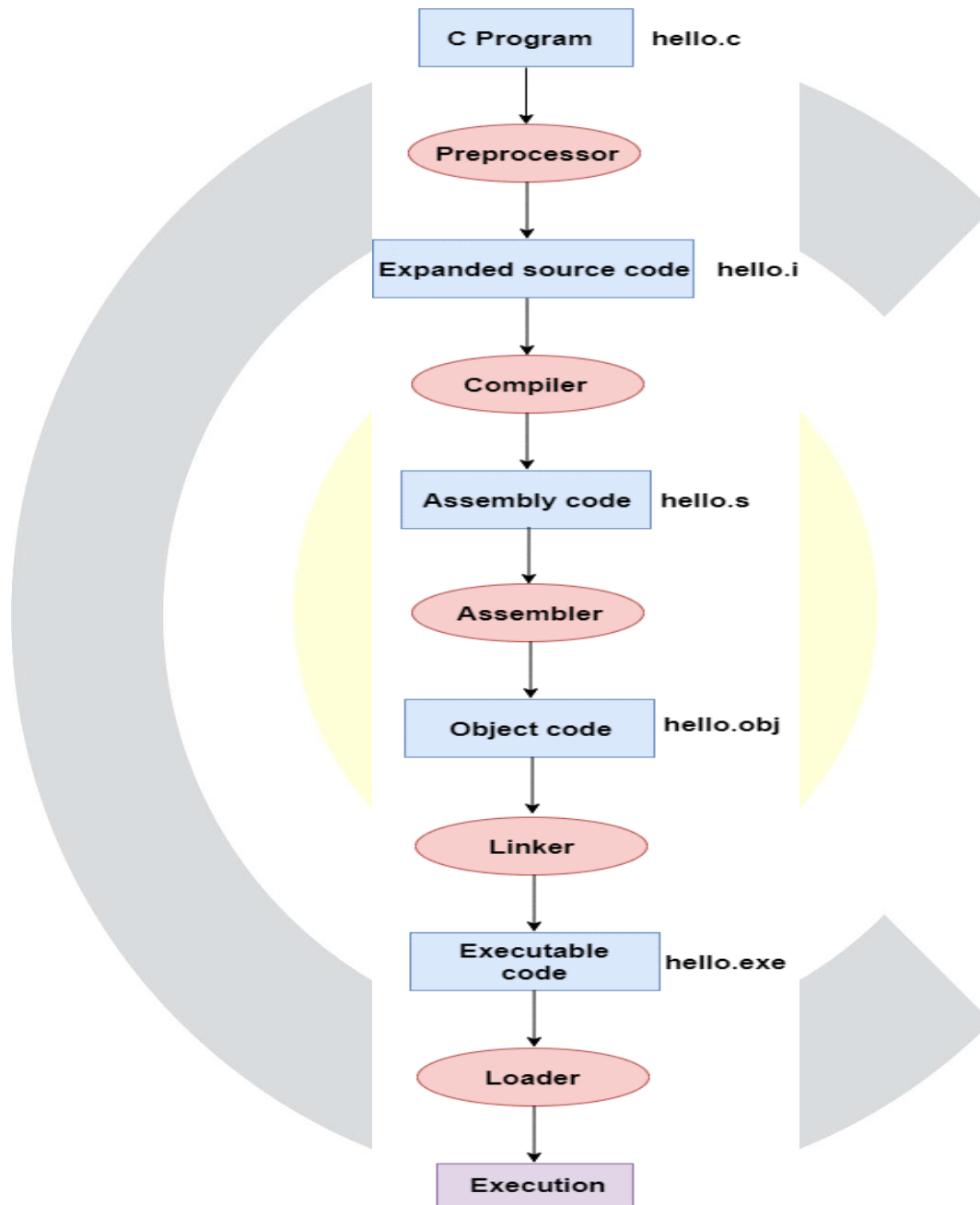
Assembler

The assembly code is converted into object code by using an assembler. The name of the object file generated by the assembler is the same as the source file. The extension of the object file in DOS is '.obj,' and in UNIX, the extension is '.o'. If the name of the source file is '**hello.c**', then the name of the object file would be 'hello.obj'.

Linker

Mainly, all the programs written in C use library functions. These library functions are pre-compiled, and the object code of these library files is stored with '.lib' (or '.a') extension. The main working of the linker is to combine the object code of library files with the object code of our program. Sometimes the situation arises when our program refers to the functions defined in other files; then linker plays a very important role in this. It links the object code of these files to our program. Therefore, we conclude that the job of the linker is to link the object code of our program

with the object code of the library files and other files. The output of the linker is the executable file. The name of the executable file is the same as the source file but differs only in their extensions. In DOS, the extension of the executable file is '.exe', and in UNIX, the executable file can be named as 'a.out'. For example, if we are using printf() function in a program, then the linker adds its associated code in an output file.



- Firstly, the input file, i.e., hello.c, is passed to the preprocessor, and the preprocessor converts the source code into expanded source code. The extension of the expanded source code would be hello.i.

- The expanded source code is passed to the compiler, and the compiler converts this expanded source code into assembly code. The extension of the assembly code would be hello.s.
- This assembly code is then sent to the assembler, which converts the assembly code into object code.
- After the creation of an object code, the linker creates the executable file. The loader will then load the executable file for the execution.

Important keywords & shortcuts:

#	Preprocessor
stdio.h	Standard input output header file
conio.h	Console input output header file
void	Return type
main()	Program main function
clrscr()	Clear screen
printf()	To print on screen
scanf()	To read / accept value from user
getch()	Get character
Alt + Enter	To minimize or maximize screen
F5	To minimize or maximize file
Alt + F9	To compile
Ctrl + F9	To run
F7	To check flow of program
F2	Save file
Shift + Del	Cut
Shift + Insert	Paste
Ctrl + Insert	Copy

printf() and scanf()

printf() and scanf() in C:

- inbuilt library functions, defined in stdio.h (header file).

printf() function

- The **printf() function** is used for output. It prints the given statement to the console.
- The syntax of printf() function is given below:

```
printf("format string",argument_list);
```

scanf() function

- The **scanf() function** is used for input. It reads the input data from the console.

```
scanf("format string",argument_list);
```

Format string	Data type
%d	Integer
%f	Float
%c	Character
%s	String

Variables in C

A **variable** is a name of the memory location. It is used to store data. Its value can be changed, and it can be reused many times.

It is a way to represent memory location through symbol so that it can be easily identified.

Let's see the syntax to declare a variable:

```
type variable_list;
```

```
int a=10, b=20;//declaring 2 variable of
```

```
41integer type
```

```
float f=20.08;
```

```
char c='A';
```

Rules for defining variables

- A variable can have alphabets, digits, and underscore.
- A variable name can start with the alphabet, and underscore only. It can't start with a digit.
- No whitespace is allowed within the variable name.
- A variable name must not be any reserved word or keyword, e.g. int, float, etc.

Types of Variables in C:

There are many types of variables in c:

1. local variable
2. global variable
3. static variable
4. automatic variable
5. external variable

Local Variable

A variable that is declared inside the function or block is called a local variable.

It must be declared at the start of the block.

```
void function1(){
```

```
int x=10;//local variable
```

```
}
```

You must have to initialize the local variable before it is used.

Global Variable

A variable that is declared outside the function or block is called a global variable. Any function can change the value of the global variable. It is available to all the functions. It must be declared at the start of the block.

```
int value=20;//global variable
void function1(){
int x=10;//local variable
}
```

Static Variable

A variable that is declared with the static keyword is called static variable.

It retains its value between multiple function calls.

```
void function1(){
int x=10;//local variable
static int y=10;//static variable
x=x+1;
y=y+1;
printf("%d,%d",x,y);
}
```

If you call this function many times, the **local variable will print the same value** for each function call, e.g, 11,11,11 and so on. But the **static variable will print the incremented value** in each function call, e.g. 11, 12, 13 and so on.

Automatic Variable

All variables in C that are declared inside the block, are automatic variables by default. We can explicitly declare an automatic variable using **auto keyword**.

```
void main(){
int x=10;//local variable (also automatic)
auto int y=20;//automatic variable
}
```

External Variable

We can share a variable in multiple C source files by using an external variable. To declare an external variable, you need to use **extern keyword**.

myfile.h

extern int x=10;//external variable (also global)

program1.c

#include "myfile.h"

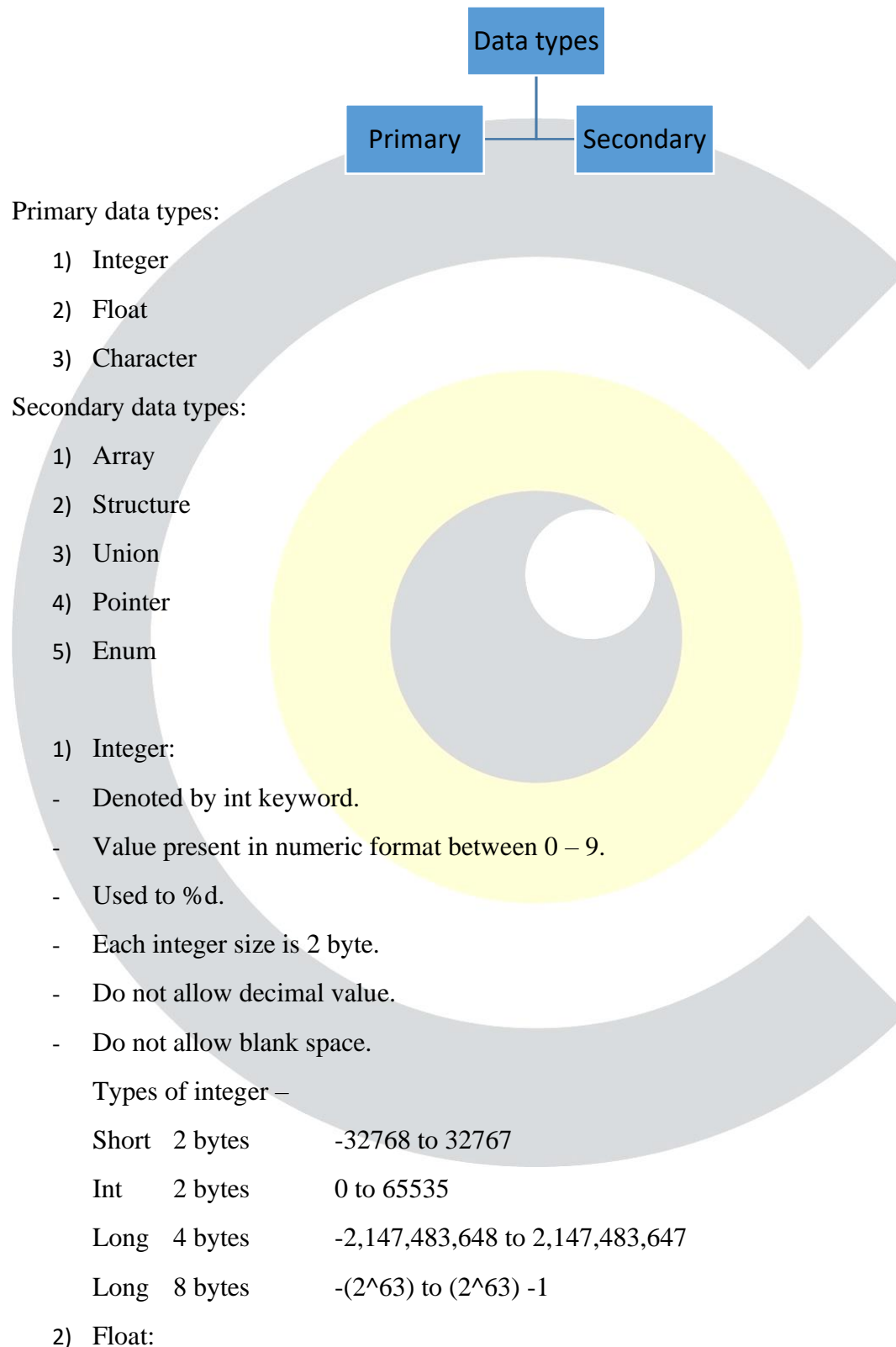
#include <stdio.h>

void printValue(){

 printf("Global variable: %d", global_variable);

}

Data types in C



- Denoted by float keyword.
- Value present in integer as well as decimal.
- Used to %f.
- Each float size is 4 byte.
- Do not allow blank space.

Types of float –

Float 4 byte

Double 8 byte

3) Character:

- Denoted by char keyword.
- Single character is present.
- Character written in single quotes.
- Used to %c.
- Each character size is 1 byte.
- Do not allow blank space.

4) String:

- Sequence of character is called string.
- More than one character is present.
- String always written in double quotes.
- Used to %s.
- Ex. char name[] = "Meenakshi".

Data Type	Memory (bytes)	Range	Format Specifier
short int	2	-32,768 to 32,767	%hd
unsigned short int	2	0 to 65,535	%hu
unsigned int	4	0 to 4,294,967,295	%u
int	4	-2,147,483,648 to 2,147,483,647	%d

long int	4	-2,147,483,648 to 2,147,483,647	%ld
unsigned long int	4	0 to 4,294,967,295	%lu
long long int	8	$-(2^{63})$ to $(2^{63})-1$	%lld
unsigned long int	8	0 to 18,446,744,073,709,551,615	%llu
signed char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
float	4		%f
double	8		%lf
long double	16		%Lf

Keywords in C

A keyword is a **reserved word**. You cannot use it as a variable name, constant name, etc. There are only 32 reserved words (keywords) in the C language.

A list of 32 keywords in the c language is given below:

Auto	break	case	char	const	continue	default	do
Double	else	enum	extern	float	for	goto	if
Int	long	register	return	short	signed	sizeof	static
Struct	switch	typedef	union	unsigned	void	volatile	while

We will learn about all the C language keywords later.

C identifiers

C identifiers represent the name in the C program, for example, variables, functions, arrays, structures, unions, labels, etc. An identifier can be composed of letters such as uppercase, lowercase letters, underscore, digits, but the starting letter should be either an alphabet or an underscore. If the identifier is not used in the external linkage, then it is called as an internal identifier. If the identifier is used in the external linkage, then it is called as an external identifier.

We can say that an identifier is a collection of alphanumeric characters that begins either with an alphabetical character or an underscore, which are used to represent various programming elements such as variables, functions, arrays, structures, unions, labels, etc. There are 52 alphabetical characters (uppercase and lowercase), underscore character, and ten numerical digits (0-9) that represent the identifiers. There is a total of 63 alphanumeric characters that represent the identifiers.

Rules for constructing C identifiers:

- 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.
- The length of the identifiers should not be more than 31 characters.
- Identifiers should be written in such a way that it is meaningful, short, and easy to read.

Types of identifiers

- Internal identifier
- External identifier

Internal Identifier

If the identifier is not used in the external linkage, then it is known as an internal identifier. The internal identifiers can be local variables.

External Identifier

If the identifier is used in the external linkage, then it is known as an external identifier. The external identifiers can be function names, global variables.

Difference between Keywords & Identifiers:

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.

Operators in C

An operator is simply a symbol that is used to perform operations. There can be many types of operations like arithmetic, logical, bitwise, etc.

There are following types of operators to perform different types of operations in C language.

○ Arithmetic Operators

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 Operators

Assume variable A holds 10 and variable B holds 20 then –

Operator	Description	Example
==	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 Operators

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.

○ Bitwise Operators

Assume variable 'A' holds 60 and variable 'B' holds 13, then –

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
	Binary OR Operator copies a bit if it exists in either operand.	(A B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = ~(60), i.e., -0111101
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

○ Assignment Operator

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

○ Conditional Operator

The conditional operator is also known as a ternary operator. The conditional statements are the decision-making statements which depends 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;

Comments in C

Comments in C language are used to provide information about lines of code. It is widely used for documenting code. There are 2 types of comments in the C language.

1. Single Line Comments
2. Multi-Line Comments

Single line comments are represented by double slash `//`. Let's see an example of a single line comment in C.

```
#include<stdio.h>
int main(){
    //printing information
    printf("Hello C");
    return 0;
}
```

Multi-Line comments are represented by slash asterisk `/* ... */`. It can occupy many lines of code, but it can't be nested. Syntax:

```
/*
code
to be commented
*/
```

Let's see an example of a multi-Line comment in C.

```
#include<stdio.h>
int main(){
    /*printing information
    Multi-Line Comment*/
    printf("Hello C");
    return 0;
}
```


Constants in C

A constant is a value or variable that can't be changed in the program, for example: 10, 20, 'a', 3.4, "c programming" etc.

The const keyword is used to define constant in C programming.

```
const float PI=3.14;
```

Now, the value of PI variable can't be changed.

```
#include<stdio.h>

int main(){

    const float PI=3.14;

    printf("The value of PI is: %f",PI);

    return 0;

}
```

Output:

The value of PI is: 3.140000

If you try to change the the value of PI, it will render compile time error.

```
#include<stdio.h>

int main(){

    const float PI=3.14;

    PI=4.5;

    printf("The value of PI is: %f",PI);

    return 0;

}
```

Output:

Compile Time Error: Cannot modify a const object

Literals:

Literals are the constant values assigned to the constant variables.

We can say that the literals represent the fixed values that cannot be modified.

It also contains memory but does not have references as variables.

For example, `const int =10;` is a constant integer expression in which 10 is an integer literal.

Tokens in C

Tokens in C is the most important element to be used in creating a program in C. We can define the token as the smallest individual element in C. For example, we cannot create a sentence without using words; similarly, we cannot create a program in C without using tokens in C. Therefore, we can say that tokens in C is the building block or the basic component for creating a program in C language.

Classification of tokens in C

Tokens in C language can be divided into the following categories:

- Keywords in C
- Identifiers in C
- Strings in C
- Operators in C
- Constant in C
- Special Characters in C

Special characters in C

Some special characters are used in C, and they have a special meaning which cannot be used for another purpose.

- Square brackets []: The opening and closing brackets represent the single and multidimensional subscripts.
- Simple brackets (): It is used in function declaration and function calling. For example, printf() is a pre-defined function.
- Curly braces { }: It is used in the opening and closing of the code. It is used in the opening and closing of the loops.
- Comma (,): It is used for separating for more than one statement and for example, separating function parameters in a function call, separating the variable when printing the value of more than one variable using a single printf statement.
- Hash/pre-processor (#): It is used for pre-processor directive. It basically denotes that we are using the header file.
- Asterisk (*): This symbol is used to represent pointers and also used as an operator for multiplication.
- Tilde (~): It is used as a destructor to free memory.
- Period (.): It is used to access a member of a structure or a union.

C Boolean

In C, Boolean is a data type that contains two types of values, i.e., 0 and 1. Basically, the bool type value represents two types of behavior, either true or false. Here, '0' represents false value, while '1' represents true value.

In C Boolean, '0' is stored as 0, and another integer is stored as 1. We do not require to use any header file to use the Boolean data type in C++, but in C, we have to use the header file, i.e., `stdbool.h`. If we do not use the header file, then the program will not compile.

Syntax:

bool variable_name;

In the above syntax, **bool** is the data type of the variable, and **variable_name** is the name of the variable.

Let's understand through an example.

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

int main()
{
    bool x=false; // variable initialization.
    if(x == true) // conditional statements
    {
        printf("The value of x is true");
    }
    else
        printf("The value of x is FALSE");
    return 0;
}
```

In the above code, we have used `<stdbool.h>` header file so that we can use the bool type variable in our program. After the declaration of the header file, we create the bool type variable 'x' and assigns a 'false' value to it. Then, we add the conditional statements, i.e., **if..else**, to determine whether the value of 'x' is true or not.

Output: The value of x is FALSE

Control Statements

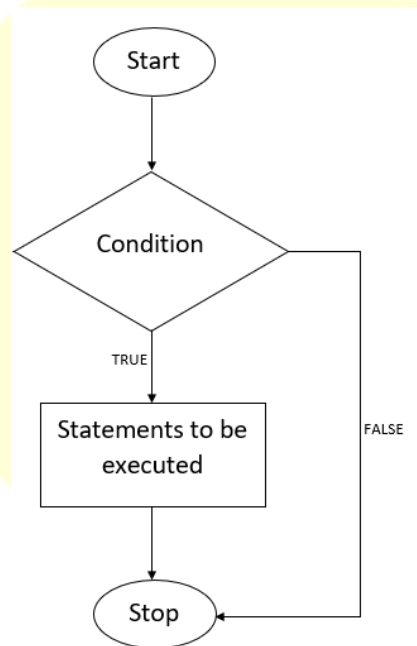
If Statement –

The if statement is used to check some given condition and perform some operations depending upon the correctness of that condition. It is mostly used in the scenario where we need to perform the different operations for the different conditions.

The syntax of the if statement is given below.

```
if(expression){  
    //code to be executed  
}
```

Flowchart of if statement in C



Let's see a simple example of C language if statement.

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

```
if(number%2==0){  
printf("%d is even number",number);  
}  
return 0;  
}
```

Output –

```
Enter a number:4  
4 is even number  
enter a number:5
```

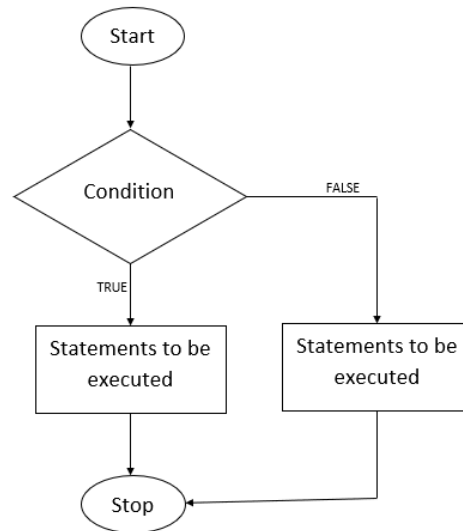
If-else Statement –

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. Here, we must notice that if and else block cannot be executed simultaneously. Using if-else statement is always preferable since it always invokes an otherwise case with every if condition.

The syntax of the if-else statement is given below.

```
if(expression){  
//code to be executed if condition is true  
}  
else{  
//code to be executed if condition is false  
}
```

Flowchart of the if-else statement in C



Let's see the simple example to check whether a number is even or odd using if-else statement in C language.

```
#include<stdio.h>
int main(){
int number=0;
printf("enter a number:");
scanf("%d",&number);
if(number%2==0){
printf("%d is even number",number);
}
else{
printf("%d is odd number",number);
}
return 0;
}
```

Output –

enter a number:4

4 is even number

enter a number:5

5 is odd number

If else-if ladder Statement –

The if-else-if ladder statement is an extension to the if-else statement. It is used in the scenario where there are multiple cases to be performed for different conditions. In if-else-if ladder statement, if a condition is true then the statements defined in the if block will be executed, otherwise if some other condition is true then the statements defined in the else-if block will be executed, at the last if none of the condition is true then the statements defined in the else block will be executed. There are multiple else-if blocks possible. It is similar to the switch case statement where the default is executed instead of else block if none of the cases is matched.

The syntax of the if else-if ladder statement is given below.

```
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  
}
```

The example of an if-else-if statement in C language is given below.

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



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

Output –

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

Nested If Statement –

Nested if else statements play an important role in C programming, it means you can use conditional statements inside another conditional statement.

The syntax for nested If statement is as given below.

```
if(test_expression one)
{
    if(test_expression two) {
        //Statement block Executes when the boolean test expression two is true.
    }
}
else
{
    //else statement block
}
```

The example of an nested if statement in C language is given below.

```
#include<stdio.h>
void main()
{
    int x=20,y=30;

    if(x==20)
    {
        if(y==30)
        {
            printf("value of x is 20, and value of y is 30.");
        }
    }
}
```

```
}
```

Output –

value of x is 20, and value of y is 30.

C Switch Statement –

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  
.....  
default:  
    code to be executed if all cases are not matched;  
}
```

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

C break statement –

The break is a keyword in C which is used to bring the program control out of the loop. The break statement is used inside loops or switch statement. The break statement breaks the loop one by one, i.e., in the case of nested loops, it breaks the inner loop first and then proceeds to outer loops. The break statement in C can be used in the following two scenarios:

- With switch case
- With loop

Syntax:

```
//loop or switch case
```

```
break;
```

Example –

```
#include<stdio.h>
#include<stdlib.h>
void main ()
{
    int i;
    for(i = 0; i<10; i++)
    {
        printf("%d ",i);
        if(i == 5)
            break;
    }
    printf("came outside of loop i = %d",i);
}
```

Output –

0 1 2 3 4 5 came outside of loop i = 5

C continue statement –

The continue statement in C language is used to bring the program control to the beginning of the loop. The continue statement skips some lines of code inside the loop and continues with the next iteration. It is mainly used for a condition so that we can skip some code for a particular condition.

Syntax:

```
//loop statements
```

```
continue;
```

```
//some lines of the code which is to be skipped
```

Example –

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

```
int main(){
```

```
int i=1;//initializing a local variable
```

```
//starting a loop from 1 to 10
```

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

```
if(i==5){//if value of i is equal to 5, it will continue the loop
```

```
continue;
```

```
}
```

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

```
}//end of for loop
```

```
return 0;
```

```
}
```

Output –

1

2

3

4

6

7

8

9

10

C goto statement –

The goto statement is known as jump statement in C. As the name suggests, goto is used to transfer the program control to a predefined label. The goto statement can be used to repeat some part of the code for a particular condition. It can also be used to break the multiple loops which can't be done by using a single break statement. However, using goto is avoided these days since it makes the program less readable and complicated.

Syntax:

label:

//some part of the code;

goto label;

goto example

Let's see a simple example to use goto statement in C language.

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

```
int main()
```

```
{
```

```
    int num,i=1;
```

```
    printf("Enter the number whose table you want to print?");
```

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

```
    table:
```

```
    printf("%d x %d = %d\n",num,i,num*i);
```

```
    i++;
```

```
    if(i<=10)
```

```
        goto table;
```

```
}
```

Output –

Enter the number whose table you want to print?10

$$10 \times 1 = 10$$

$$10 \times 2 = 20$$

$$10 \times 3 = 30$$

$$10 \times 4 = 40$$

$$10 \times 5 = 50$$

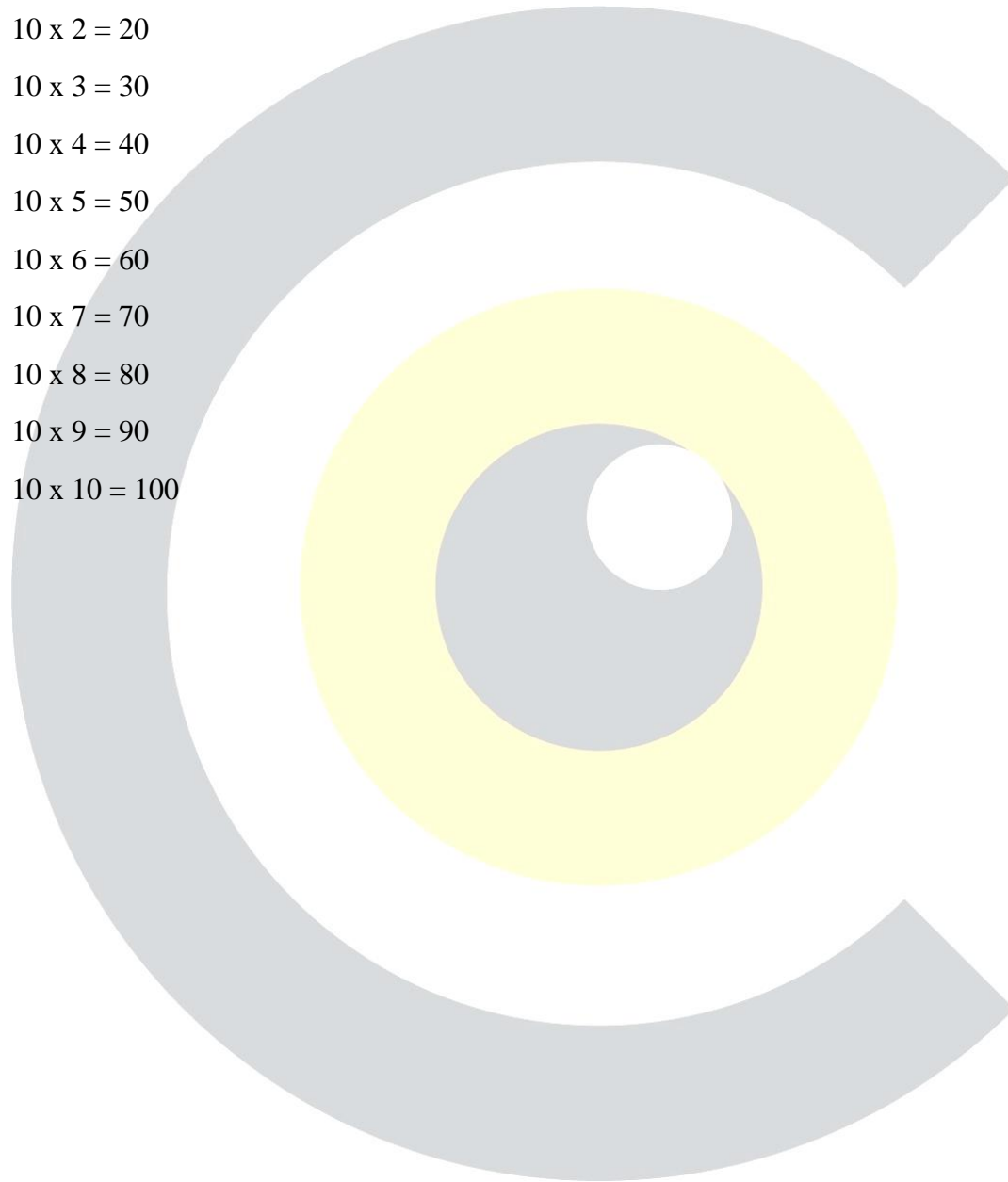
$$10 \times 6 = 60$$

$$10 \times 7 = 70$$

$$10 \times 8 = 80$$

$$10 \times 9 = 90$$

$$10 \times 10 = 100$$



Loops

The looping can be defined as repeating the same process multiple times until a specific condition satisfies. There are three types of loops used in the C language. In this part of the tutorial, we are going to learn all the aspects of C loops.

Types of C Loops –

There are three types of loops in C language that is given below:

- do while
- while
- for

do-while loop in C –

The do-while loop continues until a given condition satisfies. It is also called post tested loop. It is used when it is necessary to execute the loop at least once (mostly menu driven programs).

The syntax of do-while loop in c language is given below:

```
do{  
//code to be executed  
}while(condition);
```

There is given the simple program of c language do while loop where we are printing the values from 1 to 10.

```
#include<stdio.h>  
  
int main(){  
int i=1;  
do{  
printf("%d \n",i);  
i++;  
}while(i<=10);  
return 0;
```

```
}
```

Output –

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

while loop in C –

While loop is also known as a pre-tested loop. In general, a while loop allows a part of the code to be executed multiple times depending upon a given Boolean condition. It can be viewed as a repeating if statement. The while loop is mostly used in the case where the number of iterations is not known in advance.

The syntax of while loop in c language is given below:

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

Let's see the simple program of while loop that prints values from 1 to 10.

```
#include<stdio.h>

int main(){
int i=1;
```

```
while(i<=10){  
printf("%d \n",i);  
i++;  
}  
return 0;  
}
```

Output –

1
2
3
4
5
6
7
8
9
10

for loop in C –

The for loop in C language is used to iterate the statements or a part of the program several times. It is frequently used to traverse the data structures like the array and linked list.

The syntax of for loop in c language is given below:

```
for(Expression 1; Expression 2; Expression 3){  
//code to be executed  
}
```

Let's see the simple program of for loop that prints values from 1 to 10.

```
#include<stdio.h>

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

Output –

1
2
3
4
5
6
7
8
9
10

Nested Loops in C –

C supports nesting of loops in C. Nesting of loops is the feature in C that allows the looping of statements inside another loop. Let's observe an example of nesting loops in C.

Any number of loops can be defined inside another loop, i.e., there is no restriction for defining any number of loops. The nesting level can be defined at n times. You can define any type of loop inside another loop; for example, you can define 'while' loop inside a 'for' loop.

Syntax of Nested loop –

```
Outer_loop
{
    Inner_loop
    {
        // inner loop statements.
    }
    // outer loop statements.
}
```

Infinite Loop in C –

An infinite loop is a looping construct that does not terminate the loop and executes the loop forever. It is also called an indefinite loop or an endless loop. It either produces a continuous output or no output.

When to use an infinite loop

An infinite loop is useful for those applications that accept the user input and generate the output continuously until the user exits from the application manually. In the following situations, this type of loop can be used:

- All the operating systems run in an infinite loop as it does not exist after performing some tasks. It comes out of an infinite loop only when the user manually shuts down the system.
- All the servers run in an infinite loop as the server responds to all the client requests. It comes out of an indefinite loop only when the administrator shuts down the server manually.
- All the games also run in an infinite loop. The game will accept the user requests until the user exits from the game.

Type Casting in C

Typecasting allows us to convert one data type into other. In C language, we use cast operator for typecasting which is denoted by (type).

Syntax:

(type)value;

Without Type Casting:

```
int f= 9/4;  
printf("f : %d\n", f);//Output: 2
```

With Type Casting:

```
float f=(float) 9/4;  
printf("f : %f\n", f);//Output: 2.250000
```

Type Casting example

Let's see a simple example to cast int value into the float.

```
#include<stdio.h>  
int main(){  
float f= (float)9/4;  
printf("f : %f\n", f);  
return 0;  
}
```

Output:

f : 2.250000

C Functions

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. In other words, we can say that the collection of functions creates a program. The function is also known as procedure or subroutine in other programming languages.

Advantage of functions in C

There are the following advantages of C functions.

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

However, Function calling is always a overhead in a C program.

Function Aspects

There are three aspects of a C function.

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

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

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

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

Types of Functions

There are two types of functions in C programming:

Library Functions: are the functions which are declared in the C header files such as scanf(), printf(), gets(), puts(), ceil(), floor() etc.

User-defined functions: are the functions which are created by the C programmer, so that he/she can use it many times. It reduces the complexity of a big program and optimizes the code.

Return Value

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

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

Example without return value:

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

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

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

Example with return value:

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


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

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

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

Different aspects of function calling

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

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

Example for Function without argument and return value

Example 1

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

```
}
```

Output:

Hello Claritech

Example 2

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

Output:

Going to calculate the sum of two numbers:

Enter two numbers 10

24

The sum is 34

Example for Function without argument and with return value

Example 1

```
#include<stdio.h>

int sum();

void main()
{
    int result;
    printf("\nGoing to calculate the sum of two numbers:");
    result = sum();
    printf("%d",result);
}

int sum()
{
    int a,b;
    printf("\nEnter two numbers");
    scanf("%d %d",&a,&b);
    return a+b;
}
```

Output:

Going to calculate the sum of two numbers:

Enter two numbers 10

24

The sum is 34

Example 2: program to calculate the area of the square

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

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

Output:

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

Example for Function with argument and without return value

Example 1

```
#include<stdio.h>  
void sum(int, int);  
void main()  
{  
    int a,b,result;
```

```
printf("\nGoing to calculate the sum of two numbers:");  
printf("\nEnter two numbers:");  
scanf("%d %d",&a,&b);  
sum(a,b);  
}  
void sum(int a, int b)  
{  
    printf("\nThe sum is %d",a+b);  
}
```

Output:

Going to calculate the sum of two numbers:

Enter two numbers 10

24

The sum is 34

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

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

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

Output:

Going to calculate the average of five numbers:

Enter five numbers:10

20

30

40

50

The average of given five numbers : 30.000000

Example for Function with argument and with return value

Example 1

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

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

```
void main()
```

```
{
```

```
    int a,b,result;
```

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

```
    printf("\nEnter two numbers:");
```

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

```
    result = sum(a,b);
```

```
    printf("\nThe sum is : %d",result);
```

```
}
```

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

Output:

Going to calculate the sum of two numbers:

Enter two numbers:10

20

The sum is : 30

Example 2: Program to check whether a number is even or odd

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

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

Output:

Going to check whether a number is even or odd

Enter the number: 100

The number is even

Call by value in C & Call by reference in C

Call by value in C

In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.

In call by value method, we cannot modify the value of the actual parameter by the formal parameter.

In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.

The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

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

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

Output –

Before function call x=100

Before adding value inside function num=100

After adding value inside function num = 200

After function call x =100

Assignment –

1)Write a program swap the values of the two variables with call by value method.

Call by reference in C

In call by reference, the address of the variable is passed into the function call as the actual parameter.

The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.

In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

Consider the following example for the call by reference.

```
#include<stdio.h>

void change(int *num) {
    printf("Before adding value inside function num=%d \n",*num);
    (*num) += 100;
    printf("After adding value inside function num=%d \n", *num);
}

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

```
return 0;
}
```

Output –

Before function call x=100

Before adding value inside function num=100

After adding value inside function num=200

After function call x=200

Assignment –

1) Write a program swap the values of the two variables with call by reference method.

Difference between call by value and call by reference in c

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

Recursion in C

Recursion is the process which comes into existence when a function calls a copy of itself to work on a smaller problem. Any function which calls itself is called recursive function, and such function calls are called recursive calls. Recursion involves several numbers of recursive calls. However, it is important to impose a termination condition of recursion. Recursion code is shorter than iterative code however it is difficult to understand.

Recursion cannot be applied to all the problem, but it is more useful for the tasks that can be defined in terms of similar subtasks. For Example, recursion may be applied to sorting, searching, and traversal problems.

Generally, iterative solutions are more efficient than recursion since function call is always overhead. Any problem that can be solved recursively, can also be solved iteratively. However, some problems are best suited to be solved by the recursion, for example, tower of Hanoi, Fibonacci series, factorial finding, etc.

In the following example, recursion is used to calculate the factorial of a number.

```
#include <stdio.h>
int fact (int);
int main()
{
    int n,f;
    printf("Enter the number whose factorial you want to calculate?");
    scanf("%d",&n);
    f = fact(n);
    printf("factorial = %d",f);
}
int fact(int n)
{
    if (n==0)
    {
```

```
    return 0;
}
else if ( n == 1)
{
    return 1;
}
else
{
    return n*fact(n-1);
}
}
```

Output –

Enter the number whose factorial you want to calculate?5

factorial = 120

Assignment –

1) Write a program to print Fibonacci series using recursion.

C 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. It also has the capability to store the collection of derived data types, such as pointers, structure, etc. The array is the simplest data structure where each data element can be randomly accessed by using its index number.

C array is beneficial if you have to store similar elements. For example, if we want to store the marks of a student in 6 subjects, then we don't need to define different variables for the marks in the different subject. Instead of that, we can define an array which can store the marks in each subject at the contiguous memory locations.

By using the array, we can access the elements easily. Only a few lines of code are required to access the elements of the array.

Properties of Array

The array contains the following properties.

- Each element of an array is of same data type and carries the same size, i.e., int = 4 bytes.
- Elements of the array are stored at contiguous memory locations where the first element is stored at the smallest memory location.
- Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

Advantage of C Array

- Code Optimization: Less code to access the data.
- Ease of traversing: By using the for loop, we can retrieve the elements of an array easily.
- Ease of sorting: To sort the elements of the array, we need a few lines of code only.
- Random Access: We can access any element randomly using the array.

Disadvantage of C Array

- Fixed Size: Whatever size, we define at the time of declaration of the array, we can't exceed the limit. So, it doesn't grow the size dynamically like LinkedList which we will learn later.

Declaration of C Array

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

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

Now, let us see the example to declare the array.

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

Here, int is the data_type, marks are the array_name, and 5 is the array_size.

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;
```

C array example

```
#include<stdio.h>  
  
int main(){  
    int i=0;  
    int marks[5];//declaration of array  
    marks[0]=80;//initialization of array  
    marks[1]=60;  
    marks[2]=70;  
    marks[3]=85;  
    marks[4]=75;
```

```
//traversal of array
for(i=0;i<5;i++){
printf("%d \n",marks[i]);
} //end of for loop
return 0;
}
```

Output –

```
80
60
70
85
75
```

C Array: Declaration with Initialization

We can initialize the c array at the time of declaration. Let's see the code.

```
int marks[5]={20,30,40,50,60};
```

In such case, there is no requirement to define the size. So it may also be written as the following code.

```
int marks[]={20,30,40,50,60};
```

Let's see the C program to declare and initialize the array in C.

```
#include<stdio.h>

int main(){
int i=0;
int marks[5]={20,30,40,50,60}; //declaration and initialization of array

//traversal of array
for(i=0;i<5;i++){
```



```
printf("%d \n",marks[i]);  
}  
return 0;  
}
```

Output –

20
30
40
50
60

Assignment –

1)Write a program to sort an array.

Two-Dimensional Array in C

The two-dimensional array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns. However, 2D arrays are created to implement a relational database lookalike data structure. It provides ease of holding the bulk of data at once which can be passed to any number of functions wherever required.

Declaration of two-dimensional Array in C

The syntax to declare the 2D array is given below.

```
data_type array_name[rows][columns];
```

Consider the following example.

```
int twodimen[4][3];
```

Here, 4 is the number of rows, and 3 is the number of columns.

Initialization of 2D Array in C

In the 1D array, we don't need to specify the size of the array if the declaration and initialization are being done simultaneously. However, this will not work with 2D arrays. We will have to define at least the second dimension of the array. The two-dimensional array can be declared and defined in the following way.

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

Two-dimensional array example in C

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

```
int main(){
```

```
int i=0,j=0;
```

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

```
//traversing 2D array
```

```
for(i=0;i<4;i++){  
    for(j=0;j<3;j++){  
        printf("arr[%d] [%d] = %d \n",i,j,arr[i][j]);  
    }//end of j  
}//end of i  
return 0;  
}
```

Output

```
arr[0][0] = 1  
arr[0][1] = 2  
arr[0][2] = 3  
arr[1][0] = 2  
arr[1][1] = 3  
arr[1][2] = 4  
arr[2][0] = 3  
arr[2][1] = 4  
arr[2][2] = 5  
arr[3][0] = 4  
arr[3][1] = 5  
arr[3][2] = 6
```

Return an Array in C

An array is a type of data structure that stores a fixed-size of a homogeneous collection of data. In short, we can say that array is a collection of variables of the same type.

For example, if we want to declare 'n' number of variables, n1, n2...n., if we create all these variables individually, then it becomes a very tedious task. In such a case, we create an array of variables having the same type. Each element of an array can be accessed using an index of the element.

Let's first see how to pass a single-dimensional array to a function.

Passing array to a function

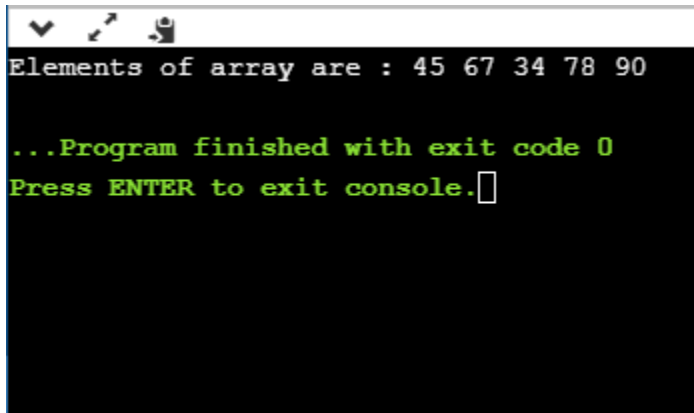
```
#include <stdio.h>

void getarray(int arr[])
{
    printf("Elements of array are : ");
    for(int i=0;i<5;i++)
    {
        printf("%d ", arr[i]);
    }
}

int main()
{
    int arr[5]={45,67,34,78,90};
    getarray(arr);
    return 0;
}
```

In the above program, we have first created the array **arr[]** and then we pass this array to the function **getarray()**. The **getarray()** function prints all the elements of the array **arr[]**.

Output –



Passing array to a function as a pointer

Now, we will see how to pass an array to a function as a pointer.

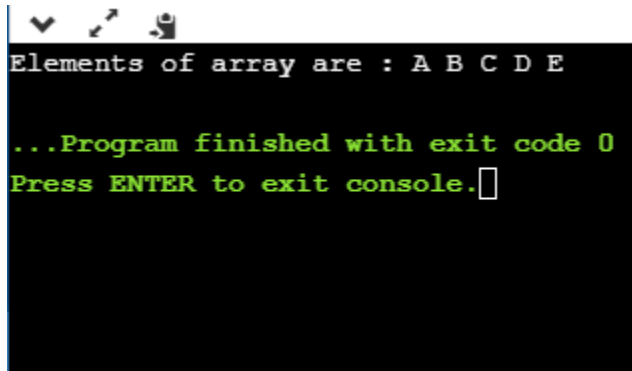
```
#include <stdio.h>

void printarray(char *arr)
{
    printf("Elements of array are : ");
    for(int i=0;i<5;i++)
    {
        printf("%c ", arr[i]);
    }
}

int main()
{
    char arr[5]={'A','B','C','D','E'};
    printarray(arr);
    return 0;
}
```

In the above code, we have passed the array to the function as a pointer. The function **printarray()** prints the elements of an array.

Output



```
Elements of array are : A B C D E

...Program finished with exit code 0
Press ENTER to exit console.
```

Note: From the above examples, we observe that array is passed to a function as a reference which means that array also persists outside the function.

How to return an array from a function

Returning pointer pointing to the array

```
#include <stdio.h>

int *getarray()
{
    int arr[5];
    printf("Enter the elements in an array : ");
    for(int i=0;i<5;i++)
    {
        scanf("%d", &arr[i]);
    }
    return arr;
}

int main()
{
```

```
int *n;
n=getarray();
printf("\nElements of array are :");
for(int i=0;i<5;i++)
{
    printf("%d", n[i]);
}
return 0;
}
```

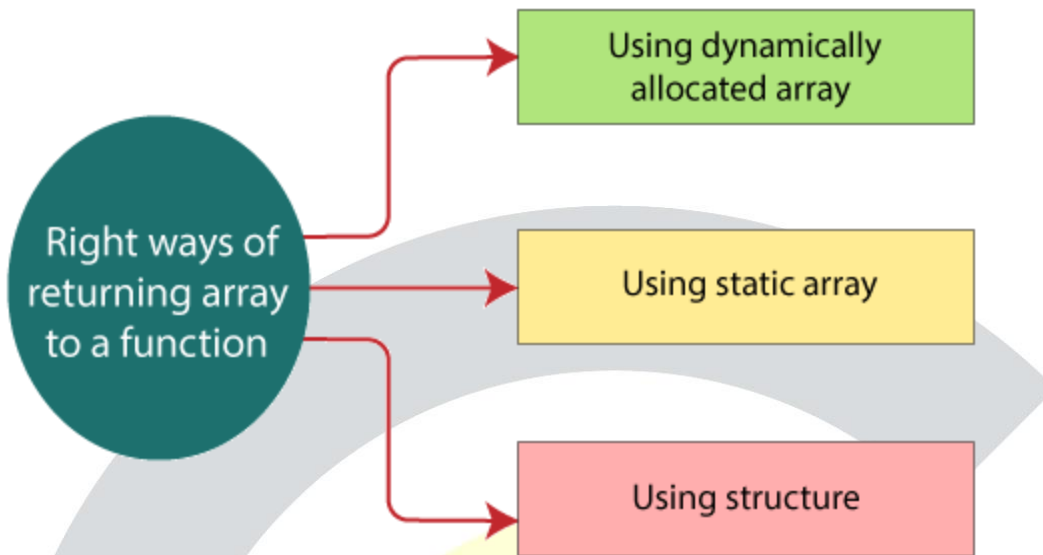
In the above program, **getarray()** function returns a variable 'arr'. It returns a local variable, but it is an illegal memory location to be returned, which is allocated within a function in the stack. Since the program control comes back to the **main()** function, and all the variables in a stack are freed. Therefore, we can say that this program is returning memory location, which is already de-allocated, so the output of the program is a **segmentation fault**.

Output

```
Array inside function: 1
2
3
4
5
Array outside function:
Segmentation fault (core dumped)
```

There are three right ways of returning an array to a function:

- Using dynamically allocated array
- Using static array
- Using structure



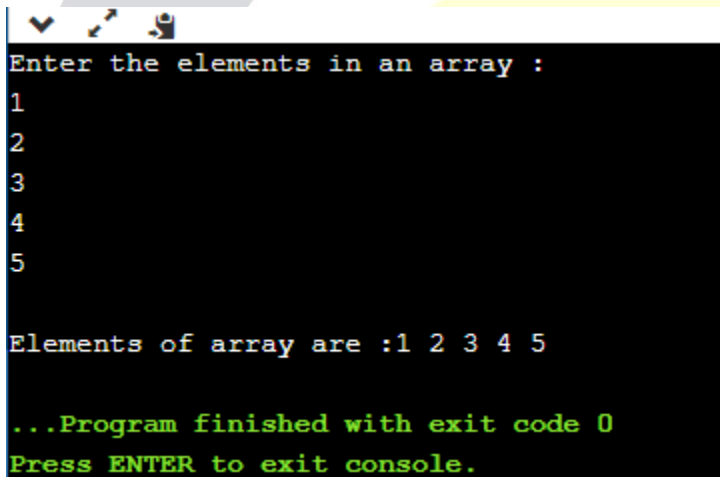
Returning array by passing an array which is to be returned as a parameter to the function.

```
#include <stdio.h>
int *getarray(int *a)
{
    printf("Enter the elements in an array : ");
    for(int i=0;i<5;i++)
    {
        scanf("%d", &a[i]);
    }
    return a;
}
int main()
{
    int *n;
```



```
int a[5];
n=getarray(a);
printf("\nElements of array are :");
for(int i=0;i<5;i++)
{
    printf("%d", n[i]);
}
return 0;
}
```

Output



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

Elements of array are :1 2 3 4 5

...Program finished with exit code 0
Press ENTER to exit console.
```

Using Static Variable

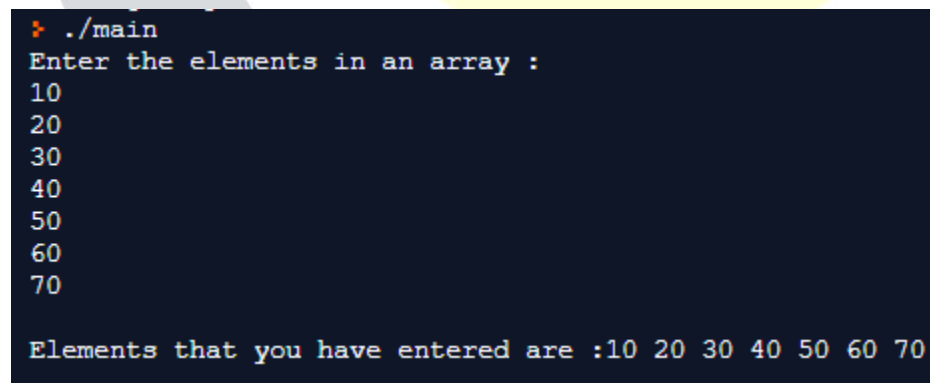
```
#include <stdio.h>

int *getarray()
{
    static int arr[7];
    printf("Enter the elements in an array : ");
    for(int i=0;i<7;i++)
    {
        scanf("%d",&arr[i]);
    }
}
```

```
}  
  
return arr;  
  
}  
  
int main()  
{  
    int *ptr;  
    ptr=getarray();  
    printf("\nElements that you have entered are :");  
    for(int i=0;i<7;i++)  
    {  
        printf("%d ", ptr[i]);  
    }  
}
```

In the above code, we have created the variable **arr[]** as static in **getarray()** function, which is available throughout the program. Therefore, the function **getarray()** returns the actual memory location of the variable '**arr**'.

Output



```
➤ ./main  
Enter the elements in an array :  
10  
20  
30  
40  
50  
60  
70  
  
Elements that you have entered are :10 20 30 40 50 60 70
```

Using Structure

The structure is a user-defined data type that can contain a collection of items of different types. Now, we will create a program that returns an array by using structure.

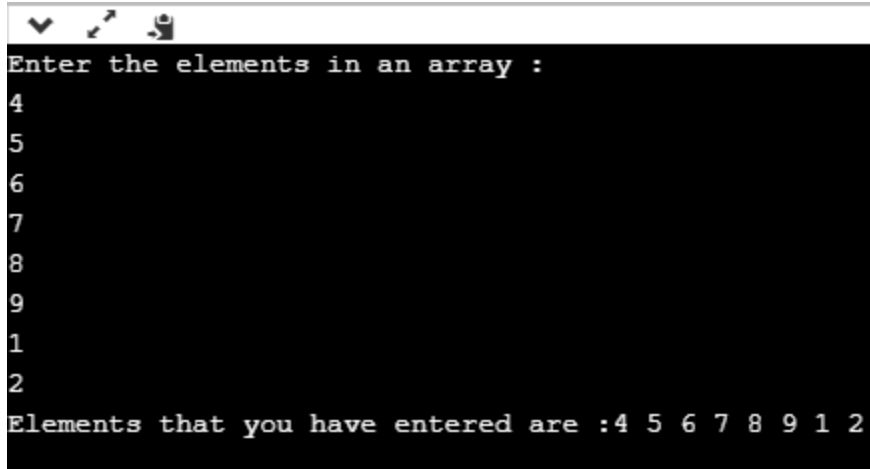
```
#include <stdio.h>
#include<malloc.h>

struct array
{
    int arr[8];
};

struct array getarray()
{
    struct array y;
    printf("Enter the elements in an array : ");
    for(int i=0;i<8;i++)
    {
        scanf("%d",&y.arr[i]);
    }
    return y;
}

int main()
{
    struct array x=getarray();
    printf("Elements that you have entered are :");
    for(int i=0;x.arr[i]!='\0';i++)
    {
        printf("%d ", x.arr[i]);
    }
    return 0;
}
```

Output



```
Enter the elements in an array :
4
5
6
7
8
9
1
2
Elements that you have entered are :4 5 6 7 8 9 1 2
```

C language passing an array to function example

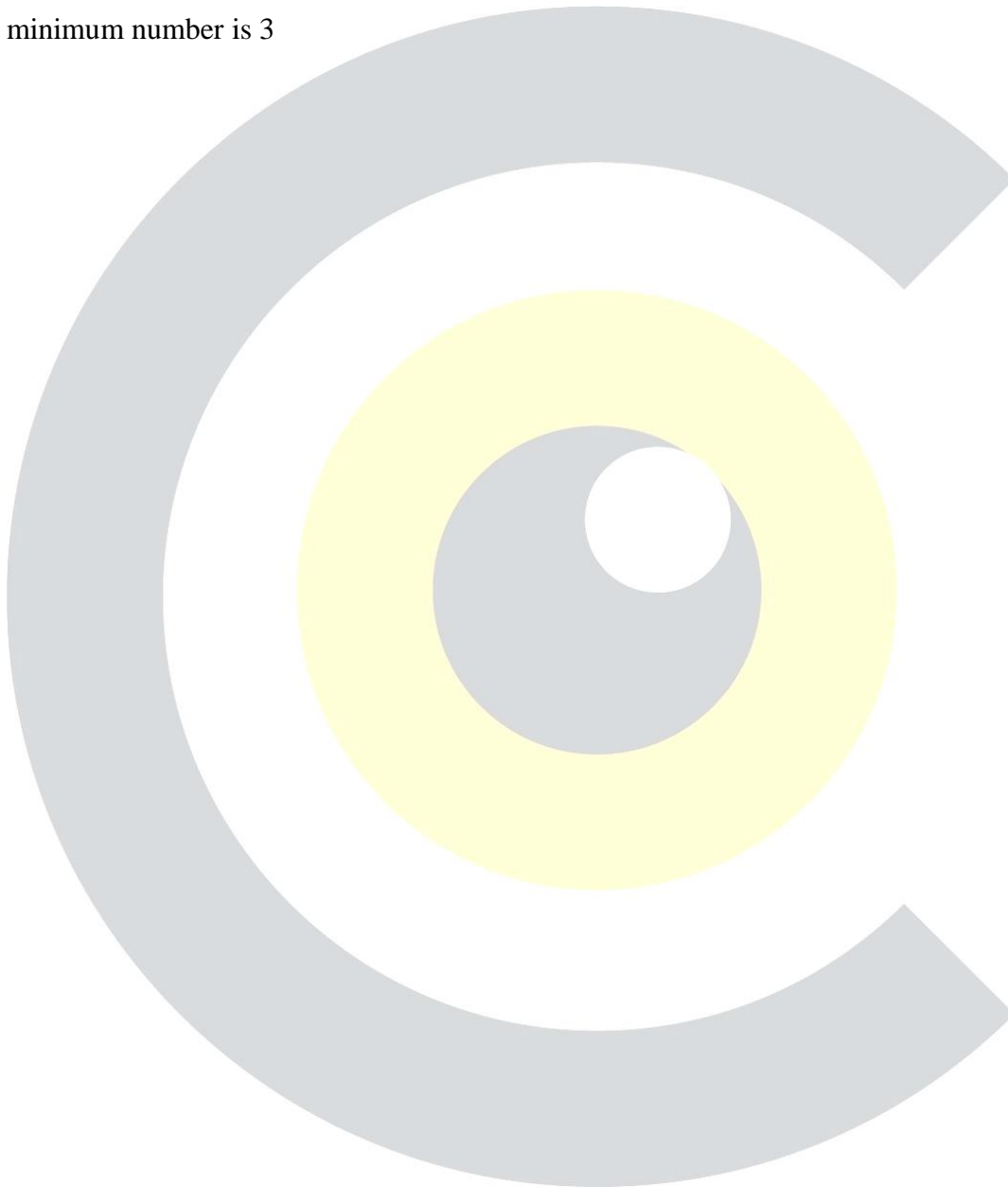
```
#include<stdio.h>
int minarray(int arr[],int size){
int min=arr[0];
int i=0;
for(i=1;i<size;i++){
if(min>arr[i]){
min=arr[i];
}
} //end of for
return min;
} //end of function

int main(){
int i=0,min=0;
int numbers[]={4,5,7,3,8,9}; //declaration of array

min=minarray(numbers,6); //passing array with size
printf("minimum number is %d \n",min);
```

```
return 0;  
}
```

Output –
minimum number is 3



C Pointers

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte.

Consider the following example to define a pointer which stores the address of an integer.

```
int n = 10;

int* p = &n; // Variable p of type pointer is pointing to the address of the variable n of type integer.
```

Declaring a pointer

The pointer in c language can be declared using * (asterisk symbol). It is also known as indirection pointer used to dereference a pointer.

```
int *a;//pointer to int
char *c;//pointer to char
```

Pointer Example

An example of using pointers to print the address and value is given below.

As you can see in the above figure, pointer variable stores the address of number variable, i.e., fff4. The value of number variable is 50. But the address of pointer variable p is aaa3.

By the help of * (indirection operator), we can print the value of pointer variable p.

Let's see the pointer example as explained for the above figure.

```
#include<stdio.h>

int main(){
int number=50;
int *p;
```

```
p=&number;//stores the address of number variable  
printf("Address of p variable is %x \n",p); // p contains the address of the number therefore  
printing p gives the address of number.  
printf("Value of p variable is %d \n",*p); // As we know that * is used to dereference a pointer  
therefore if we print *p, we will get the value stored at the address contained by p.  
return 0;  
}
```

Output –

Address of number variable is fff4

Address of p variable is fff4

Value of p variable is 50

Pointer to array

```
int arr[10];  
int (*p)[10]=&arr; // Variable p of type pointer is pointing to the address of an integer array arr.
```

Pointer to a function

```
void show (int);  
void(*p)(int) = &show; // Pointer p is pointing to the address of a function
```

Pointer to structure

```
struct st {  
    int i;  
    float f;  
}ref;  
struct st *p = &ref;
```

Advantage of pointer

- Pointer reduces the code and improves the performance, it is used to retrieving strings, trees, etc. and used with arrays, structures, and functions.
- We can return multiple values from a function using the pointer.
- It makes you able to access any memory location in the computer's memory.

Usage of pointer

There are many applications of pointers in c language.

1) Dynamic memory allocation

In c language, we can dynamically allocate memory using malloc() and calloc() functions where the pointer is used.

2) Arrays, Functions, and Structures

Pointers in c language are widely used in arrays, functions, and structures. It reduces the code and improves the performance.

Address Of (&) Operator

The address of operator '&' returns the address of a variable. But, we need to use %u to display the address of a variable.

```
#include<stdio.h>

int main(){
int number=50;
printf("value of number is %d, address of number is %u",number,&number);
return 0;
}
```

Output –

value of number is 50, address of number is fff4

NULL Pointer

A pointer that is not assigned any value but NULL is known as the NULL pointer. If you don't have any address to be specified in the pointer at the time of declaration, you can assign NULL value. It will provide a better approach.

```
int *p=NULL;
```

In the most libraries, the value of the pointer is 0 (zero).

Pointer Program to swap two numbers without using the 3rd variable.

```
#include<stdio.h>

int main(){
int a=10,b=20,*p1=&a,*p2=&b;
printf("Before swap: *p1=%d *p2=%d",*p1,*p2);
*p1=*p1+*p2;
*p2=*p1-*p2;
*p1=*p1-*p2;
printf("\nAfter swap: *p1=%d *p2=%d",*p1,*p2);
return 0;
}
```

Output –

Before swap: *p1=10 *p2=20

After swap: *p1=20 *p2=10

C Double Pointer (Pointer to Pointer)

As we know that, a pointer is used to store the address of a variable in C. Pointer reduces the access time of a variable. However, In C, we can also define a pointer to store the address of another pointer. Such pointer is known as a double pointer (pointer to pointer). The first pointer

is used to store the address of a variable whereas the second pointer is used to store the address of the first pointer. Let's understand it by the diagram given below.

pointer to pointer in c

The syntax of declaring a double pointer is given below.

```
int **p; // pointer to a pointer which is pointing to an integer.
```

Consider the following example.

```
#include<stdio.h>
void main ()
{
    int a = 10;
    int *p;
    int **pp;
    p = &a; // pointer p is pointing to the address of a
    pp = &p; // pointer pp is a double pointer pointing to the address of pointer p
    printf("address of a: %x\n",p); // Address of a will be printed
    printf("address of p: %x\n",pp); // Address of p will be printed
    printf("value stored at p: %d\n",*p); // value stored at the address contained by p i.e. 10 will be printed
    printf("value stored at pp: %d\n",**pp); // value stored at the address contained by the pointer stored at pp
}
```

Output –

address of a: d26a8734

address of p: d26a8738

value stored at p: 10

value stored at pp: 10

C double pointer example

Let's see an example where one pointer points to the address of another pointer.

As you can see in the above figure, p2 contains the address of p (fff2), and p contains the address of number variable (fff4).

```
#include<stdio.h>
int main(){
int number=50;
int *p;//pointer to int
int **p2;//pointer to pointer
p=&number;//stores the address of number variable
p2=&p;
printf("Address of number variable is %x \n",&number);
printf("Address of p variable is %x \n",p);
printf("Value of *p variable is %d \n",*p);
printf("Address of p2 variable is %x \n",p2);
printf("Value of **p2 variable is %d \n",*p);
return 0;
}
```

Output –

Address of number variable is fff4

Address of p variable is fff4

Value of *p variable is 50

Address of p2 variable is fff2

Value of **p variable is 50

Pointer Arithmetic in C

We can perform arithmetic operations on the pointers like addition, subtraction, etc. However, as we know that pointer contains the address, the result of an arithmetic operation performed on the pointer will also be a pointer if the other operand is of type integer. In pointer-from-pointer subtraction, the result will be an integer value. Following arithmetic operations are possible on the pointer in C language:

- Increment
- Decrement
- Addition
- Subtraction

Incrementing Pointer in C

If we increment a pointer by 1, the pointer will start pointing to the immediate next location. This is somewhat different from the general arithmetic since the value of the pointer will get increased by the size of the data type to which the pointer is pointing.

We can traverse an array by using the increment operation on a pointer which will keep pointing to every element of the array, perform some operation on that, and update itself in a loop.

The Rule to increment the pointer is given below:

$$\text{new_address} = \text{current_address} + i * \text{size_of}(\text{data type})$$

Where i is the number by which the pointer get increased.

32-bit

For 32-bit int variable, it will be incremented by 4 bytes.

64-bit

For 64-bit int variable, it will be incremented by 8 bytes.

Let's see the example of incrementing pointer variable on 64-bit architecture.

```
#include<stdio.h>

int main(){
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p+1;
printf("After increment: Address of p variable is %u \n",p); // in our case, p will get incremented
by 4 bytes.
return 0;
}
```

Output –

Address of p variable is 3214864300

After increment: Address of p variable is 3214864304

Traversing an array by using pointer

```
#include<stdio.h>

void main ()
{
    int arr[5] = { 1, 2, 3, 4, 5 };
    int *p = arr;
    int i;
    printf("printing array elements...\n");
    for(i = 0; i < 5; i++)
    {
```

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

Output –

printing array elements...

1 2 3 4 5

Decrementing Pointer in C

Like increment, we can decrement a pointer variable. If we decrement a pointer, it will start pointing to the previous location. The formula of decrementing the pointer is given below:

$$\text{new_address} = \text{current_address} - i * \text{size_of}(\text{data type})$$

32-bit

For 32-bit int variable, it will be decremented by 2 bytes.

64-bit

For 64-bit int variable, it will be decremented by 4 bytes.

Let's see the example of decrementing pointer variable on 64-bit OS.

```
#include <stdio.h>  
void main(){  
    int number=50;  
    int *p;//pointer to int  
    p=&number;//stores the address of number variable  
    printf("Address of p variable is %u \n",p);  
    p=p-1;  
    printf("After decrement: Address of p variable is %u \n",p); // P will now point to the immediate  
    previous location.
```

```
}
```

Output –

Address of p variable is 3214864300

After decrement: Address of p variable is 3214864296

C Pointer Addition

We can add a value to the pointer variable. The formula of adding value to pointer is given below:

$$\text{new_address} = \text{current_address} + (\text{number} * \text{size_of}(\text{data type}))$$

32-bit

For 32-bit int variable, it will add $2 * \text{number}$.

64-bit

For 64-bit int variable, it will add $4 * \text{number}$.

Let's see the example of adding value to pointer variable on 64-bit architecture.

```
#include<stdio.h>
int main(){
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p+3; //adding 3 to pointer variable
printf("After adding 3: Address of p variable is %u \n",p);
return 0;
}
```

Output –

Address of p variable is 3214864300

After adding 3: Address of p variable is 3214864312

As you can see, the address of p is 3214864300. But after adding 3 with p variable, it is 3214864312, i.e., $4 \times 3 = 12$ increment. Since we are using 64-bit architecture, it increments 12. But if we were using 32-bit architecture, it was incrementing to 6 only, i.e., $2 \times 3 = 6$. As integer value occupies 2-byte memory in 32-bit OS.

C Pointer Subtraction

Like pointer addition, we can subtract a value from the pointer variable. Subtracting any number from a pointer will give an address. The formula of subtracting value from the pointer variable is given below:

$$\text{new_address} = \text{current_address} - (\text{number} * \text{size_of}(\text{data type}))$$

32-bit

For 32-bit int variable, it will subtract $2 * \text{number}$.

64-bit

For 64-bit int variable, it will subtract $4 * \text{number}$.

Let's see the example of subtracting value from the pointer variable on 64-bit architecture.

```
#include<stdio.h>

int main(){
    int number=50;
    int *p;//pointer to int
    p=&number;//stores the address of number variable
    printf("Address of p variable is %u \n",p);
    p=p-3; //subtracting 3 from pointer variable
    printf("After subtracting 3: Address of p variable is %u \n",p);
    return 0;
```



```
}
```

Output –

Address of p variable is 3214864300

After subtracting 3: Address of p variable is 3214864288

You can see after subtracting 3 from the pointer variable, it is 12 (4*3) less than the previous address value.

However, instead of subtracting a number, we can also subtract an address from another address (pointer). This will result in a number. It will not be a simple arithmetic operation, but it will follow the following rule.

If two pointers are of the same type,

$\text{Address2} - \text{Address1} = (\text{Subtraction of two addresses}) / \text{size of data type which pointer points}$

Consider the following example to subtract one pointer from another.

```
#include<stdio.h>
void main ()
{
    int i = 100;
    int *p = &i;
    int *temp;
    temp = p;
    p = p + 3;
    printf("Pointer Subtraction: %d - %d = %d",p, temp, p-temp);
}
```

Output –

Pointer Subtraction: 1030585080 - 1030585068 = 3

C Strings

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

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

- By char array
- By string literal

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

```
char ch[10]={'c', 'l', 'a', 'r', 'i', 't', 'e', 'c', 'h', '\0'};
```

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

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

```
char ch[]={'c', 'l', 'a', 'r', 'i', 't', 'e', 'c', 'h', '\0'};
```

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

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

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

Difference between char array and string literal

There are two main differences between char array and literal.

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

String Example in C

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

```
#include<stdio.h>
#include <string.h>
int main(){
    char ch[11]={'c', 'l', 'a', 'r', 'i', 't', 'e', 'c', 'h', '\0'};
    char ch2[11]="claritech";

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

Output

Char Array Value is: claritech

String Literal Value is: claritech

Traversing String

Traversing the string is one of the most important aspects in any of the programming languages. We may need to manipulate a very large text which can be done by traversing the text.

Traversing string is somewhat different from the traversing an integer array. We need to know the length of the array to traverse an integer array, whereas we may use the null character in the case of string to identify the end the string and terminate the loop.

Hence, there are two ways to traverse a string.

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

Let's discuss each one of them.

Using the length of string

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

```
#include<stdio.h>

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

Output –

The number of vowels 3

Using the null character

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

```
#include<stdio.h>

void main ()
```

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

Output –

The number of vowels 3

Accepting string as the input

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

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

```
}
```

Output –

Enter the string?claritech is the best

You entered claritech

It is clear from the output that, the above code will not work for space separated strings. To make this code working for the space separated strings, the minor changed required in the scanf function, i.e., instead of writing `scanf("%s",s)`, we must write: `scanf("%[^\n]s",s)` which instructs the compiler to store the string `s` while the new line (`\n`) is encountered. Let's consider the following example to store the space-separated strings.

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

Output –

Enter the string?claritech is the best

You entered claritech is the best

Here we must also notice that we do not need to use address of (`&`) operator in scanf to store a string since string `s` is an array of characters and the name of the array, i.e., `s` indicates the base address of the string (character array) therefore we need not use `&` with it.

Some important points

However, there are the following points which must be noticed while entering the strings by using scanf.

- The compiler doesn't perform bounds checking on the character array. Hence, there can be a case where the length of the string can exceed the dimension of the character array which may always overwrite some important data.
- Instead of using scanf, we may use gets() which is an inbuilt function defined in a header file string.h. The gets() is capable of receiving only one string at a time.

C gets() & puts() functions

The gets() and puts() are declared in the header file stdio.h. Both the functions are involved in the input/output operations of the strings.

C gets() function

The gets() function enables the user to enter some characters followed by the enter key. All the characters entered by the user get stored in a character array. The null character is added to the array to make it a string. The gets() allows the user to enter the space-separated strings. It returns the string entered by the user.

Declaration

```
char[] gets(char[]);
```

Reading string using gets()

```
#include<stdio.h>

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

Output:

Enter the string?

claritech is the best

You entered claritech is the best

The `gets()` function is risky to use since it doesn't perform any array bound checking and keep reading the characters until the new line (enter) is encountered. It suffers from buffer overflow, which can be avoided by using `fgets()`. The `fgets()` makes sure that not more than the maximum limit of characters are read. Consider the following example.

```
#include<stdio.h>

void main()
{
    char str[20];
    printf("Enter the string? ");
    fgets(str, 20, stdin);
    printf("%s", str);
}
```

Output:

```
Enter the string? claritech is the best website
claritech is the b
```

C puts() function

The `puts()` function is very much similar to `printf()` function. The `puts()` function is used to print the string on the console which is previously read by using `gets()` or `scanf()` function. The `puts()` function returns an integer value representing the number of characters being printed on the console. Since, it prints an additional newline character with the string, which moves the cursor to the new line on the console, the integer value returned by `puts()` will always be equal to the number of characters present in the string plus 1.

Declaration

```
int puts(char[])
```

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

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

Output:

Enter your name: Sonoo Jaiswal

Your name is: Sonoo Jaiswal

C String Functions

There are many important string functions defined in "string.h" library.

No.	Function	Description
1)	strlen(string_name)	returns the length of string name.
2)	strcpy(destination, source)	copies the contents of source string to destination string.
3)	strcat(first_string, second_string)	concatenates or joins first string with second string. The result of the string is stored in first string.
4)	strcmp(first_string, second_string)	compares the first string with second string. If both strings are same, it returns 0.
5)	strrev(string)	returns reverse string.
6)	strlwr(string)	returns string characters in lowercase.
7)	strupr(string)	returns string characters in uppercase.

strlen() function

The strlen() function returns the length of the given string. It doesn't count null character '\0'.

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

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

```
int main(){
```

```
char ch[20]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};
```

```
printf("Length of string is: %d",strlen(ch));
```

```
return 0;  
}
```

Output:

Length of string is: 10

strcpy() function

The strcpy(destination, source) function copies the source string in destination.

```
#include<stdio.h>  
#include <string.h>  
int main(){  
    char ch[20]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};  
    char ch2[20];  
    strcpy(ch2,ch);  
    printf("Value of second string is: %s",ch2);  
    return 0;  
}
```

Output:

Value of second string is: claritech

strcat() function

The strcat(first_string, second_string) function concatenates two strings and result is returned to first_string.

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

```
char ch[10]={'h', 'e', 'l', 'l', 'o', '\0'};
char ch2[10]={'c', '\0'};
strcat(ch,ch2);
printf("Value of first string is: %s",ch);
return 0;
}
```

Output:

Value of first string is: helloc

strcmp() function

The strcmp(first_string, second_string) function compares two string and returns 0 if both strings are equal.

Here, we are using gets() function which reads string from the console.

```
#include<stdio.h>
#include <string.h>
int main(){
    char str1[20],str2[20];
    printf("Enter 1st string: ");
    gets(str1);//reads string from console
    printf("Enter 2nd string: ");
    gets(str2);
    if(strcmp(str1,str2)==0)
        printf("Strings are equal");
    else
        printf("Strings are not equal");
    return 0;
```

```
}
```

Output:

Enter 1st string: hello

Enter 2nd string: hello

Strings are equal

strrev() function

The strrev(string) function returns reverse of the given string. Let's see a simple example of strrev() function.

```
#include<stdio.h>
#include <string.h>
int main(){
    char str[20];
    printf("Enter string: ");
    gets(str);//reads string from console
    printf("String is: %s",str);
    printf("\nReverse String is: %s",strrev(str));
    return 0;
}
```

Output:

Enter string: claritech

String is: claritech

Reverse String is: hcetiralc

strlwr() function

The strlwr(string) function returns string characters in lowercase. Let's see a simple example of strlwr() function.

```
#include<stdio.h>
#include <string.h>
int main(){
    char str[20];
    printf("Enter string: ");
    gets(str);//reads string from console
    printf("String is: %s",str);
    printf("\nLower String is: %s",strlwr(str));
    return 0;
}
```

Output:

Enter string: Claritech

String is: Claritech

Lower String is: claritech

strupr() function

The strupr(string) function returns string characters in uppercase. Let's see a simple example of strupr() function.

```
#include<stdio.h>
#include <string.h>
int main(){
    char str[20];
    printf("Enter string: ");
```

```
gets(str);//reads string from console  
printf("String is: %s",str);  
printf("\nUpper String is: %s",strupr(str));  
return 0;  
}
```

Output:

Enter string: claritech

String is: claritech

Upper String is: CLARITECH

C Math

C Programming allows us to perform mathematical operations through the functions defined in `<math.h>` header file. The `<math.h>` header file contains various methods for performing mathematical operations such as `sqrt()`, `pow()`, `ceil()`, `floor()` etc.

C Math Functions

There are various methods in `math.h` header file. The commonly used functions of `math.h` header file are given below.

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

C Structure

In C, there are cases where we need to store multiple attributes of an entity. It is not necessary that an entity has all the information of one type only. It can have different attributes of different data types. For example, an entity **Student** may have its name (string), roll number (int), marks (float). To store such type of information regarding an entity student, we have the following approaches:

- Construct individual arrays for storing names, roll numbers, and marks.
- Use a special data structure to store the collection of different data types.

Let's look at the first approach in detail.

```
#include<stdio.h>

void main ()
{
    char names[2][10],dummy; // 2-
    dimensional character array names is used to store the names of the students
    int roll_numbers[2],i;
    float marks[2];
    for (i=0;i<3;i++)
    {

        printf("Enter the name, roll number, and marks of the student %d",i+1);
        scanf("%s %d %f",&names[i],&roll_numbers[i],&marks[i]);
        scanf("%c",&dummy); // enter will be stored into dummy character at each iteration
    }

    printf("Printing the Student details ...\n");
    for (i=0;i<3;i++)
    {
        printf("%s %d %f\n",names[i],roll_numbers[i],marks[i]);
    }
}
```

```
}
```

Output

Enter the name, roll number, and marks of the student 1Arun 90 91

Enter the name, roll number, and marks of the student 2Varun 91 56

Enter the name, roll number, and marks of the student 3Sham 89 69

Printing the Student details...

Arun 90 91.000000

Varun 91 56.000000

Sham 89 69.000000

The above program may fulfill our requirement of storing the information of an entity student. However, the program is very complex, and the complexity increase with the amount of the input. The elements of each of the array are stored contiguously, but all the arrays may not be stored contiguously in the memory. C provides you with an additional and simpler approach where you can use a special data structure, i.e., structure, in which, you can group all the information of different data type regarding an entity.

What is Structure

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. Structures can simulate the use of classes and templates as it can store various information

The **struct** keyword is used to define the structure. Let's see the syntax to define the structure in c.

```
struct structure_name
```

```
{
```

```
    data_type member1;
```

```
    data_type member2;
```

```
    .
```

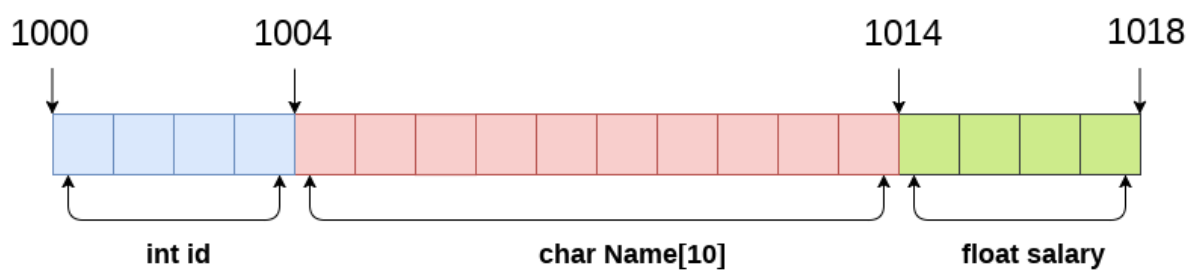
```
    .
```

```
data_type memberN;
};
```

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

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

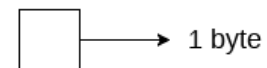
The following image shows the memory allocation of the structure employee that is defined in the above example.



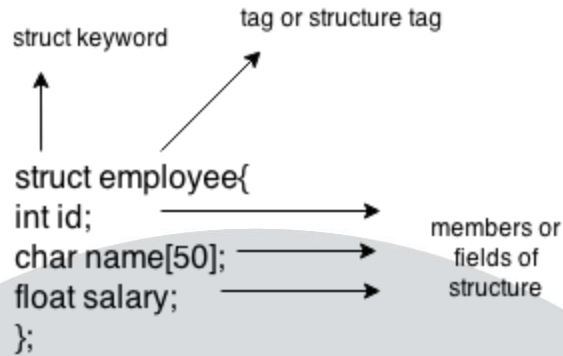
```
struct Employee
{
    int id;
    char Name[10];
    float salary;
} emp;
```

$\text{sizeof (emp)} = 4 + 10 + 4 = 18 \text{ bytes}$

where;
 $\text{sizeof (int)} = 4 \text{ byte}$
 $\text{sizeof (char)} = 1 \text{ byte}$
 $\text{sizeof (float)} = 4 \text{ byte}$



Here, **struct** is the keyword; **employee** is the name of the structure; **id**, **name**, and **salary** are the members or fields of the structure. Let's understand it by the diagram given below:



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

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;
```

Which approach is good

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 is fixed, use 2nd approach. It saves your code to declare a variable in main() function.

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

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
```

Nested Structure in C

C provides us the feature of nesting one structure within another structure by using which, complex data types are created. For example, we may need to store the address of an entity employee in a structure. The attribute address may also have the subparts as street number, city, state, and pin code. Hence, to store the address of the employee, we need to store the address of the employee into a separate structure and nest the structure address into the structure employee. Consider the following program.

Example of nested structure:

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

struct student {
    int roll;
    char name[20];
    int age;

    struct details {
        char email[30];
        long int phone;
    }d;
}s;

int main()
{
    printf("Enter roll no : ");
    scanf("%d",&s.roll);
    printf("\nEnter name : ");
    scanf("%s",&s.name);
```

```
printf("\nEnter age : ");
scanf("%d",&s.age);
printf("\nEnter email : ");
scanf("%s",&s.d.email);
printf("\nEnter phone no. : ");
scanf("%lld",&s.d.phone);

printf("\n\nYour roll no : %d",s.roll);
printf("\nYour name : %s",s.name);
printf("\nYour age : %d",s.age);
printf("\nYour email : %s",s.d.email);
printf("\nYour phone no. : %lld",s.d.phone);
return 0;
}
```

Output:

Enter roll no : 101

Enter name : Sanskruti

Enter age : 21

Enter email : Sanskruti@gmail.com

Enter phone no. : 9876543210

Your roll no : 101

Your name : Sanskruti

Your age : 21

Your email : Sanskruti@gmail.com

Your phone no. : 9876543210

Union in C

Union can be defined as a user-defined data type which is a collection of different variables of different data types in the same memory location. The union can also be defined as many members, but only one member can contain a value at a particular point in time.

Union is a user-defined data type, but unlike structures, they share the same memory location.

Let's understand this through an example.

```
struct abc
{
    int a;
    char b;
}
```

The above code is the user-defined structure that consists of two members, i.e., 'a' of type int and 'b' of type character. When we check the addresses of 'a' and 'b', we found that their addresses are different. Therefore, we conclude that the members in the structure do not share the same memory location.

When we define the union, then we found that union is defined in the same way as the structure is defined but the difference is that union keyword is used for defining the union data type, whereas the struct keyword is used for defining the structure. The union contains the data members, i.e., 'a' and 'b', when we check the addresses of both the variables then we found that both have the same addresses. It means that the union members share the same memory location.

Let's have a look at the pictorial representation of the memory allocation.

The below figure shows the pictorial representation of the structure. The structure has two members; i.e., one is of integer type, and the another one is of character type. Since 1 block is equal to 1 byte; therefore, 'a' variable will be allocated 4 blocks of memory while 'b' variable will be allocated 1 block of memory.

The below figure shows the pictorial representation of union members. Both the variables are sharing the same memory location and having the same initial address.

In union, members will share the memory location. If we try to make changes in any of the member then it will be reflected to the other member as well. Let's understand this concept through an example.

```
union abc
{
    int a;
    char b;
}var;
int main()
{
    var.a = 66;
    printf("\n a = %d", var.a);
    printf("\n b = %d", var.b);
}
```

In the above code, union has two members, i.e., 'a' and 'b'. The 'var' is a variable of union abc type. In the main() method, we assign the 66 to 'a' variable, so var.a will print 66 on the screen. Since both 'a' and 'b' share the memory location, var.b will print 'B' (ascii code of 66).

Deciding the size of the union

The size of the union is based on the size of the largest member of the union.

Let's understand through an example.

```
union abc{
    int a;
    char b;
    float c;
    double d;
};
int main()
```

```
{  
    printf("Size of union abc is %d", sizeof(union abc));  
    return 0;  
}
```

As we know, the size of int is 4 bytes, size of char is 1 byte, size of float is 4 bytes, and the size of double is 8 bytes. Since the double variable occupies the largest memory among all the four variables, so total 8 bytes will be allocated in the memory. Therefore, the output of the above program would be 8 bytes.

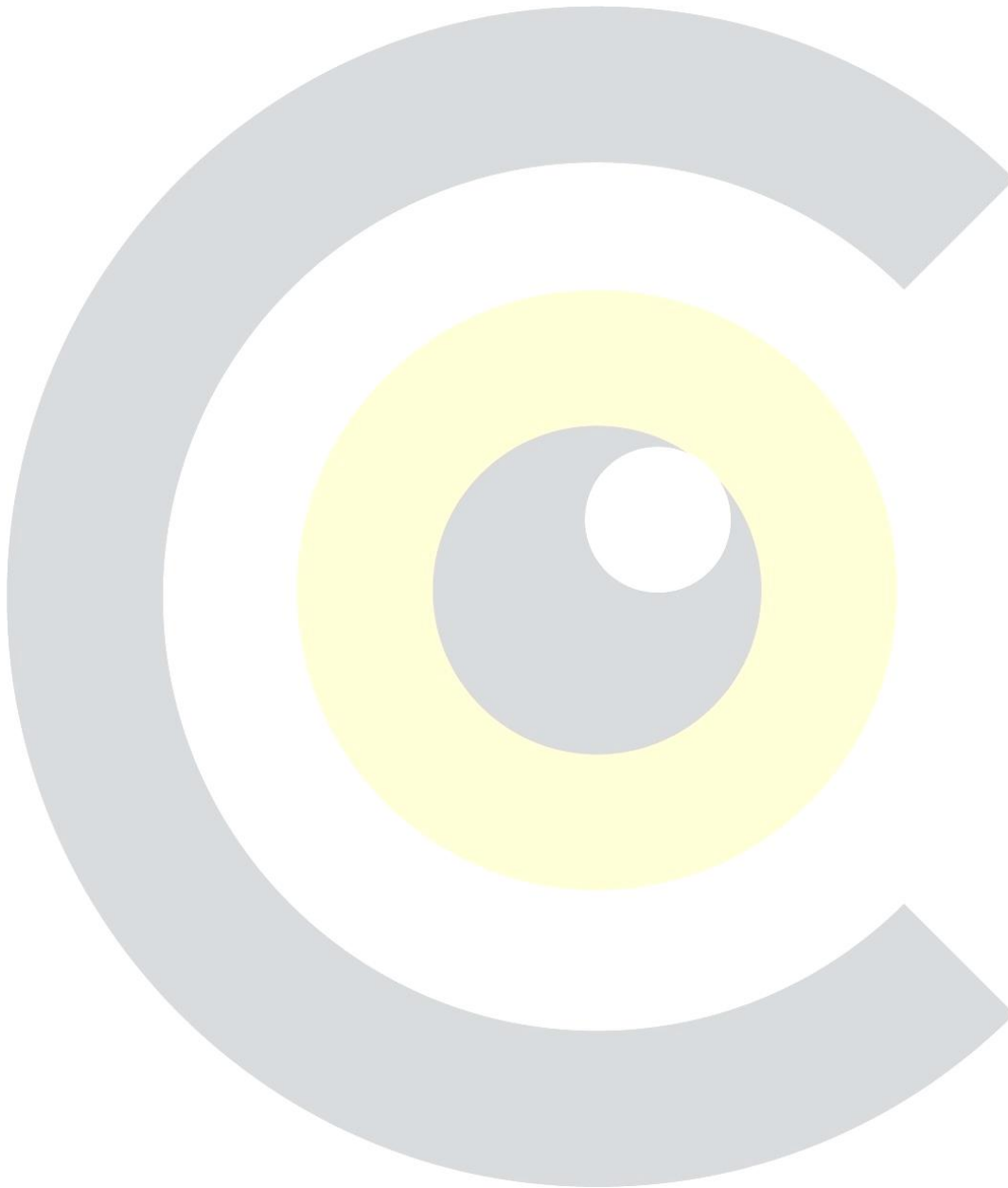
Accessing members of union using pointers

We can access the members of the union through pointers by using the (->) arrow operator.

Let's understand through an example.

```
#include <stdio.h>  
  
union abc  
{  
    int a;  
    char b;  
};  
  
int main()  
{  
    union abc *ptr; // pointer variable declaration  
    union abc var;  
    var.a= 90;  
    ptr = &var;  
    printf("The value of a is : %d", ptr->a);  
    return 0;  
}
```

In the above code, we have created a pointer variable, i.e., `*ptr`, that stores the address of `var` variable. Now, `ptr` can access the variable 'a' by using the `(->)` operator. Hence the output of the above code would be 90.



File Handling in C

In programming, we may require some specific input data to be generated several numbers of times. Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console, and since the memory is volatile, it is impossible to recover the programmatically generated data again and again. However, if we need to do so, we may store it onto the local file system which is volatile and can be accessed every time. Here, comes the need of file handling in C.

File handling in C enables us to create, update, read, and delete the files stored on the local file system through our C program. The following operations can be performed on a file.

- Creation of the new file
- Opening an existing file
- Reading from the file
- Writing to the file
- Deleting the file

Functions for file handling

There are many functions in the C library to open, read, write, search and close the file. A list of file functions are given below:

No.	Function	Description
1	fopen()	opens new or existing file
2	fprintf()	write data into the file
3	fscanf()	reads data from the file
4	fputc()	writes a character into the file
5	fgetc()	reads a character from file
6	fclose()	closes the file
7	fseek()	sets the file pointer to given position
8	fputw()	writes an integer to file
9	fgetw()	reads an integer from file
10	ftell()	returns current position
11	rewind()	sets the file pointer to the beginning of the file

Opening File: fopen()

We must open a file before it can be read, write, or update. The fopen() function is used to open a file. The syntax of the fopen() is given below.

```
FILE *fopen( const char * filename, const char * mode );
```

The fopen() function accepts two parameters:

The file name (string). If the file is stored at some specific location, then we must mention the path at which the file is stored. For example, a file name can be like "c://some_folder/some_file.ext".

The mode in which the file is to be opened. It is a string.

We can use one of the following modes in the fopen() function.

Mode	Description
r	opens a text file in read mode
w	opens a text file in write mode
a	opens a text file in append mode
r+	opens a text file in read and write mode
w+	opens a text file in read and write mode
a+	opens a text file in read and write mode
rb	opens a binary file in read mode
wb	opens a binary file in write mode
ab	opens a binary file in append mode
rb+	opens a binary file in read and write mode
wb+	opens a binary file in read and write mode
ab+	opens a binary file in read and write mode

The fopen function works in the following way.

Firstly, It searches the file to be opened.

Then, it loads the file from the disk and place it into the buffer. The buffer is used to provide efficiency for the read operations.

It sets up a character pointer which points to the first character of the file.

Consider the following example which opens a file in write mode.

```
#include<stdio.h>

void main( )
{
FILE *fp ;
char ch ;
fp = fopen("file_handle.c","r") ;
while ( 1 )
{
ch = fgetc ( fp ) ;
if ( ch == EOF )
break ;
printf("%c",ch) ;
}
fclose (fp ) ;
}
```

Output –

The content of the file will be printed.

```
#include;

void main( )
{
```

```
FILE *fp; // file pointer
char ch;
fp = fopen("file_handle.c","r");
while ( 1 )
{
ch = fgetc ( fp ); //Each character of the file is read and stored in the character file.
if ( ch == EOF )
break;
printf("%c",ch);
}
fclose (fp );
}
```

Closing File: fclose()

The fclose() function is used to close a file. The file must be closed after performing all the operations on it. The syntax of fclose() function is given below:

```
int fclose( FILE *fp );
```

Writing File : fprintf() function

The fprintf() function is used to write set of characters into file. It sends formatted output to a stream.

Syntax:

```
int fprintf(FILE *stream, const char *format [, argument, ...])
```

Example:

```
#include <stdio.h>
main(){
    FILE *fp;
    fp = fopen("file.txt", "w");//opening file
```

```
fprintf(fp, "Hello file by fprintf...\n");//writing data into file  
fclose(fp);//closing file  
}
```

Reading File : fscanf() function

The fscanf() function is used to read set of characters from file. It reads a word from the file and returns EOF at the end of file.

Syntax:

```
int fscanf(FILE *stream, const char *format [, argument, ...])
```

Example:

```
#include <stdio.h>  
main(){  
    FILE *fp;  
    char buff[255];//creating char array to store data of file  
    fp = fopen("file.txt", "r");  
    while(fscanf(fp, "%s", buff)!=EOF){  
        printf("%s ", buff );  
    }  
    fclose(fp);  
}
```

Output:

Hello file by fprintf...

Writing File : fputc() function

The fputc() function is used to write a single character into file. It outputs a character to a stream.

Syntax:

```
int fputc(int c, FILE *stream)
```

Example:

```
#include <stdio.h>

main(){
    FILE *fp;

    fp = fopen("file1.txt", "w");//opening file
    fputc('a',fp);//writing single character into file
    fclose(fp);//closing file
}

file1.txt
a
```

Reading File : fgetc() function

The fgetc() function returns a single character from the file. It gets a character from the stream. It returns EOF at the end of file.

Syntax:

```
int fgetc(FILE *stream)
```

Example:

```
#include<stdio.h>
#include<conio.h>
void main(){
    FILE *fp;
    char c;
    clrscr();
    fp=fopen("myfile.txt","r");
    while((c=fgetc(fp))!=EOF){
        printf("%c",c);
    }
    fclose(fp);
}
```

```
getch();  
}
```

```
myfile.txt  
this is simple text message
```

Writing File : fputs() function

The fputs() function writes a line of characters into file. It outputs string to a stream.

Syntax:

```
int fputs(const char *s, FILE *stream)
```

Example:

```
#include<stdio.h>  
#include<conio.h>  
void main(){  
FILE *fp;  
clrscr();  
fp=fopen("myfile2.txt","w");  
fputs("hello c programming",fp);  
fclose(fp);  
getch();  
}
```

```
myfile2.txt  
hello c programming
```

Reading File : fgets() function

The fgets() function reads a line of characters from file. It gets string from a stream.

Syntax:

```
char* fgets(char *s, int n, FILE *stream)
```

Example:

```
#include<stdio.h>
#include<conio.h>
void main(){
FILE *fp;
char text[300];
clrscr();
fp=fopen("myfile2.txt","r");
printf("%s",fgets(text,200,fp));
fclose(fp);
getch();
}
```

Output:

hello c programming

C fseek() function

The fseek() function is used to set the file pointer to the specified offset. It is used to write data into file at desired location.

Syntax:

```
int fseek(FILE *stream, long int offset, int whence)
```

There are 3 constants used in the fseek() function for whence: SEEK_SET, SEEK_CUR and SEEK_END.

Example:

```
#include <stdio.h>

void main(){

    FILE *fp;

    fp = fopen("myfile.txt","w+");

    fputs("This is claritech", fp);

    fseek( fp, 7, SEEK_SET );

    fputs("sonoo jaiswal", fp);

    fclose(fp);

}
```

myfile.txt
This is sonoo jaiswal

C rewind() function

The rewind() function sets the file pointer at the beginning of the stream. It is useful if you have to use stream many times.

Syntax:

```
void rewind(FILE *stream)
```

Example:

File: file.txt
this is a simple text

File: rewind.c

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

void main(){

    FILE *fp;

    char c;
```



```
clrscr();  
fp=fopen("file.txt","r");  
while((c=fgetc(fp))!=EOF){  
printf("%c",c);  
}  
rewind(fp);//moves the file pointer at beginning of the file  
while((c=fgetc(fp))!=EOF){  
printf("%c",c);  
}  
fclose(fp);  
getch();  
}
```

Output:

this is a simple textthis is a simple text

C ftell() function

The ftell() function returns the current file position of the specified stream. We can use ftell() function to get the total size of a file after moving file pointer at the end of file. We can use SEEK_END constant to move the file pointer at the end of file.

Syntax:

```
long int ftell(FILE *stream)
```

Example:

File: ftell.c

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

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

```
void main (){
```

```
    FILE *fp;
```

```
int length;  
clrscr();  
fp = fopen("file.txt", "r");  
fseek(fp, 0, SEEK_END);  
length = ftell(fp);  
fclose(fp);  
printf("Size of file: %d bytes", length);  
getch();  
}
```

Output:

Size of file: 21 bytes

C Preprocessor Directives

The C preprocessor is a microprocessor that is used by compiler to transform your code before compilation. It is called micro preprocessor because it allows us to add macros.

C preprocessor

All preprocessor directives start with hash # symbol.

Let's see a list of preprocessor directives.

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

C #include

The #include preprocessor directive is used to paste code of given file into current file. It is used include system-defined and user-defined header files. If included file is not found, compiler renders error.

By the use of #include directive, we provide information to the preprocessor where to look for the header files. There are two variants to use #include directive.

- #include <filename>
- #include "filename"

The #include <filename> tells the compiler to look for the directory where system header files are held. In UNIX, it is \user\include directory.

The #include "filename" tells the compiler to look in the current directory from where program is running.

#include directive example

Let's see a simple example of #include directive. In this program, we are including stdio.h file because printf() function is defined in this file.

```
#include<stdio.h>

int main(){
    printf("Hello C");
    return 0;
}
```

Output:

Hello C

C #define

The #define preprocessor directive is used to define constant or micro substitution. It can use any basic data type.

Syntax:

```
#define token value
```

Let's see an example of #define to define a constant.

```
#include <stdio.h>
#define PI 3.14
main() {
    printf("%f",PI);
}
```

Output:

3.140000

C #undef

The #undef preprocessor directive is used to undefine the constant or macro defined by #define.

Syntax:

#undef token

Let's see a simple example to define and undefine a constant.

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

```
#define PI 3.14
```

```
#undef PI
```

```
main() {
```

```
    printf("%f",PI);
```

```
}
```

Output:

Compile Time Error: 'PI' undeclared

The #undef directive is used to define the preprocessor constant to a limited scope so that you can declare constant again.

Let's see an example where we are defining and undefining number variable. But before being undefined, it was used by square variable.

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

```
#define number 15
```

```
int square=number*number;
```

```
#undef number
```

```
main() {
```

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

```
}
```

Output:

225

C #ifdef

The #ifdef preprocessor directive checks if macro is defined by #define. If yes, it executes the code otherwise #else code is executed, if present.

Syntax:

```
#ifdef MACRO  
  
//code  
  
#endif
```

Syntax with #else:

```
#ifdef MACRO  
  
//successful code  
  
#else  
  
//else code  
  
#endif
```

C #ifdef example

Let's see a simple example to use #ifdef preprocessor directive.

```
#include <stdio.h>  
#include <conio.h>  
#define NOINPUT  
void main() {  
    int a=0;  
    #ifdef NOINPUT  
        a=2;  
    #else  
        printf("Enter a:");  
        scanf("%d", &a);  
    #endif  
    printf("Value of a: %d\n", a);
```

```
getch();  
}
```

Output:

Value of a: 2

C #ifndef

The #ifndef preprocessor directive checks if macro is not defined by #define. If yes, it executes the code otherwise #else code is executed, if present.

Syntax:

```
#ifndef MACRO  
//code  
#endif
```

Syntax with #else:

```
#ifndef MACRO  
//successful code  
#else  
//else code  
#endif
```

C #ifndef example

Let's see a simple example to use #ifndef preprocessor directive.

```
#include <stdio.h>  
#include <conio.h>  
#define INPUT  
void main() {  
    int a=0;  
    #ifndef INPUT
```

```
a=2;
#else
printf("Enter a:");
scanf("%d", &a);
#endif
printf("Value of a: %d\n", a);
getch();
}
```

Output:

Enter a: 5

Value of a: 5

C #if

The #if preprocessor directive evaluates the expression or condition. If condition is true, it executes the code otherwise #elseif or #else or #endif code is executed.

Syntax:

```
#if expression
//code
#endif
```

Syntax with #else:

```
#if expression
//if code
#else
//else code
#endif
```


Syntax with #elif and #else:

```
#if expression
//if code
#elif expression
//elif code
#else
//else code
#endif
```

C #if example

Let's see a simple example to use #if preprocessor directive.

```
#include <stdio.h>
#include <conio.h>
#define NUMBER 0
void main() {
    #if (NUMBER==0)
    printf("Value of Number is: %d",NUMBER);
    #endif
    getch();
}
```

Output:

Value of Number is: 0

C #error

The #error preprocessor directive indicates error. The compiler gives fatal error if #error directive is found and skips further compilation process.

C #error example

Let's see a simple example to use #error preprocessor directive.

```
#include<stdio.h>

#ifndef __MATH_H
#error First include then compile
#else
void main(){
    float a;
    a=sqrt(7);
    printf("%f",a);
}
#endif
```

Output:

Compile Time Error: First include then compile

Command Line Arguments in C

The arguments passed from command line are called command line arguments. These arguments are handled by main() function.

To support command line argument, you need to change the structure of main() function as given below.

```
int main(int argc, char *argv[] )
```

Here, argc counts the number of arguments. It counts the file name as the first argument.

The argv[] contains the total number of arguments. The first argument is the file name always.

Example

Let's see the example of command line arguments where we are passing one argument with file name.

```
#include <stdio.h>

void main(int argc, char *argv[] ) {
    printf("Program name is: %s\n", argv[0]);
    if(argc < 2){
        printf("No argument passed through command line.\n");
    }
    else{
        printf("First argument is: %s\n", argv[1]);
    }
}
```

Run this program as follows in Linux:

```
./program hello
```

Run this program as follows in Windows from command line:

program.exe hello

Output:

Program name is: program

First argument is: hello

If you pass many arguments, it will print only one.

./program hello c how r u

Output:

Program name is: program

First argument is: hello

But if you pass many arguments within double quote, all arguments will be treated as a single argument only.

./program "hello c how r u"

Output:

Program name is: program

First argument is: hello c how r u

You can write your program to print all the arguments. In this program, we are printing only argv[1], that is why it is printing only one argument.