VG

BEGINNERS TO MASTERING C

LET'S C

Topics covered

Fundamentals of C

- What is C Language
- History of C
- Features of C
- How to install C
- First C Program
- Flow of C Program
- printf scanf
- Variables in C
- Data Types in C
- Keywords in C
- C Operators
- C Comments
- C Escape Sequence
- Constants in C

C Control Statements

- C if-else
- C switch
- C Loops
- C do-while loop
- C while loop
- C for loop
- C break
- C continue
- C goto
- Type Casting

C Functions

- What is function
- Call: Value & Reference
- Recursion in C
- Storage Classes

C Array

- 1-D Array
- 2-D Array
- Array to Function

C Pointers

- C Pointers
- C Pointer to Pointer
- C Pointer Arithmetic

C Dynamic Memory

- Dynamic memory

C Structure Union

- C Structure
- C Array of Structures
- C Nested Structure
- C Union

C File Handling

- C File Handling
- C fprintf() fscanf()
- C fputc() fgetc()
- C fputs() fgets()
- C fseek()
- C rewind()
- C ftell()

C Preprocessor

- C Preprocessor
- C Macros
- C #include
- C #define
- C #undef
- C #ifdef
- C #ifndef
- C #if
- C #else
- C #error
- C #pragma

C Command Line

- Command Line Arguments

C Strings

- String in C
- C gets() & puts()
- C String Functions
- C strlen()
- C strcpy()
- C strcat()
- C strcmp()
- C strrev()
- C strlwr()
- C strupr()
- C strstr()

C Math

- C Math Functions

Error Handling in C

- C –Error Handling

Searching and Sorting

- Linear Search
- Binary Search
- Selection Sort
- Bubble Sort
- Insertion Sort
- Merge Sort

Frequently Asked C programs

- Fibonacci Series
- Prime number
- Palindrome number
- Factorial
- Armstrong number
- Sum of Digits
- Reverse Number
- Swap two numbers without using third variable
- Print "hello" without using semicolon
- Assembly Program in C
- C Program without main() function
- Decimal to Binary
- Alphabet Triangle
- Number Triangle
- Fibonacci Triangle
- Number in Characters

Fundamentals of C

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**.

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

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

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

It can be defined by the following ways:

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

1) **C as a mother language**

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

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

2) C as a system programming language

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

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

3) C as a procedural language

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

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

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

4) C as a structured programming language

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

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

5) C as a mid-level programming language

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

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

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

Features of C Language

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

1. Simple
2. Machine Independent or Portable
3. Mid-level programming language
4. structured programming language
5. Rich Library
6. Memory Management
7. Fast Speed
8. Pointers
9. Recursion
10. Extensible

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 install C

There are many compilers available for c and c++. You need to download any one. Here, we are going to use **Turbo C++**. It will work for both C and C++. To install the Turbo C software, you need to follow following steps.

1. Download Turbo C++
2. Create turboc directory inside c drive and extract the tc3.zip inside c:\turboc
3. Double click on install.exe file
4. Click on the tc application file located inside c:\TC\BIN to write the c program

1) Download Turbo C++ software

You can download turbo c++ from the link below.

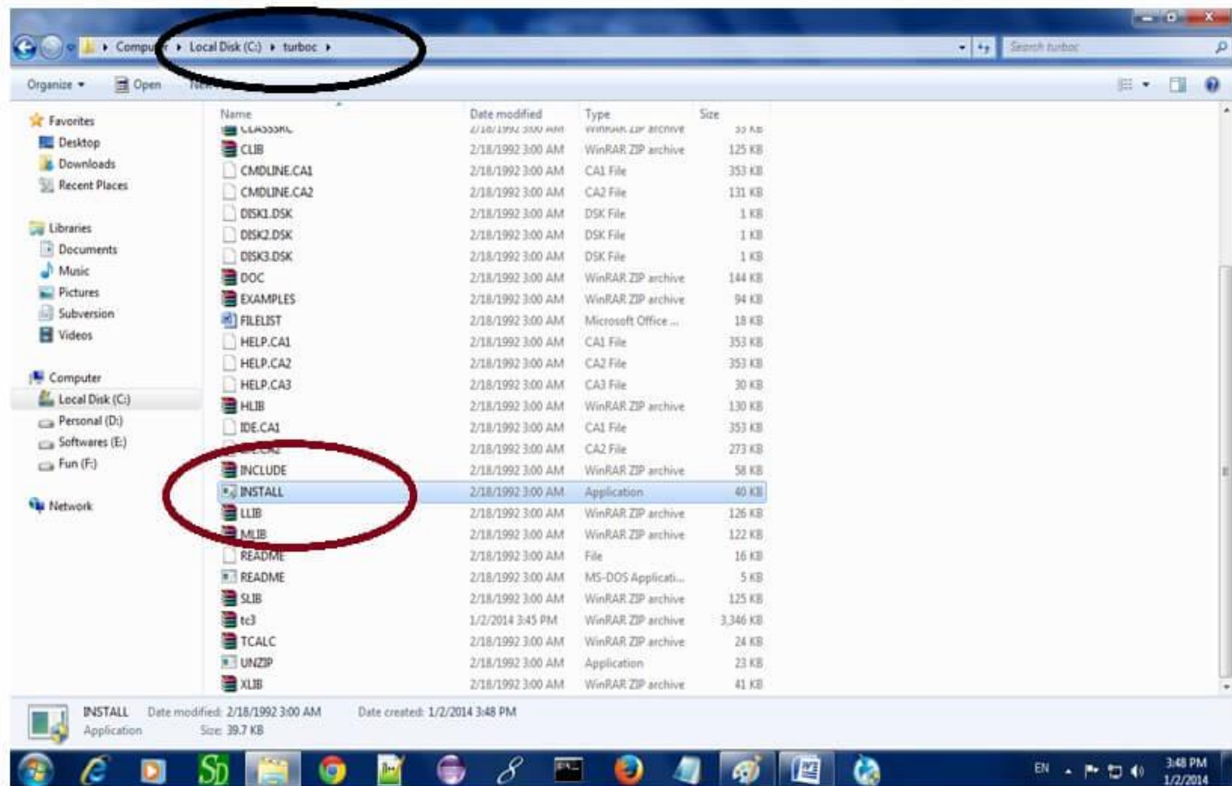
https://drive.google.com/file/d/1kPsM1wuhZ6xJKIxbX6tMuos49eIAtG6A/view?usp=sharing

2) Create turboc directory in c drive and extract the tc3.zip

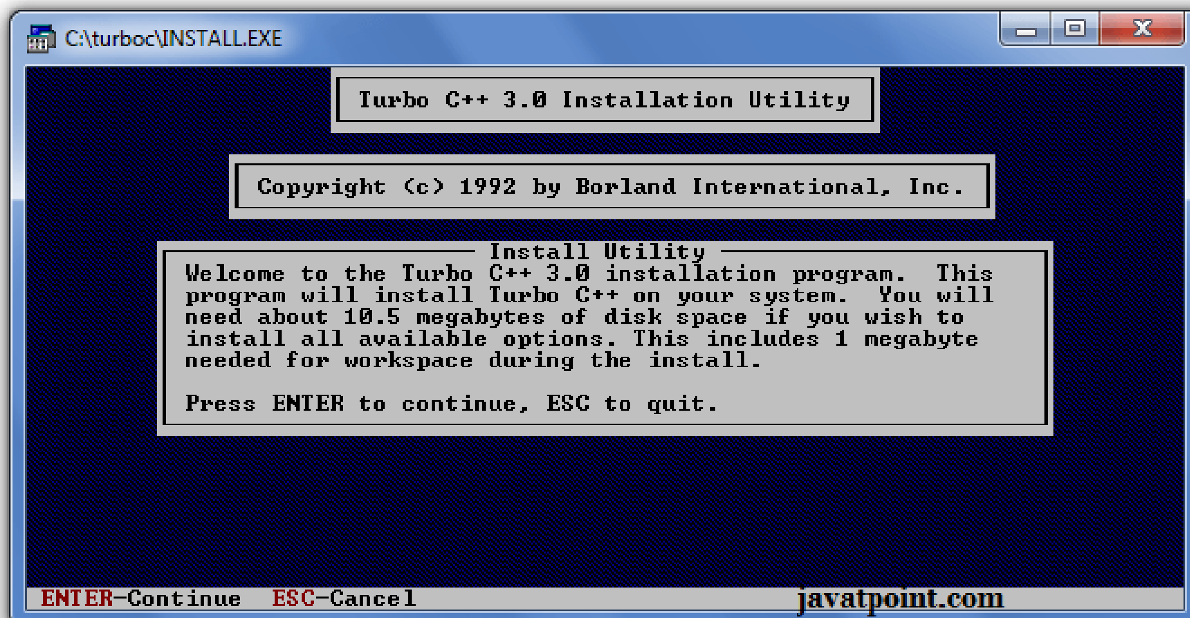
Now, you need to create a new directory turboc inside the c: drive. Now extract the tc3.zip file in c:\truboc directory.

3) Double click on the install.exe file and follow steps

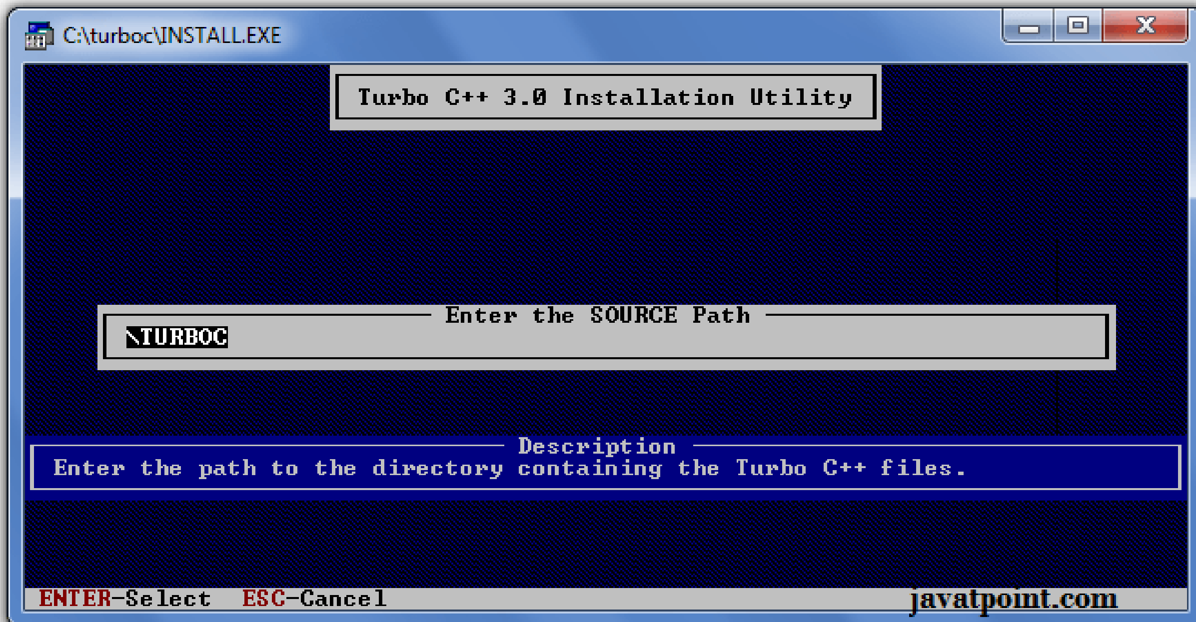
Now, click on the install icon located inside the c:\turboc

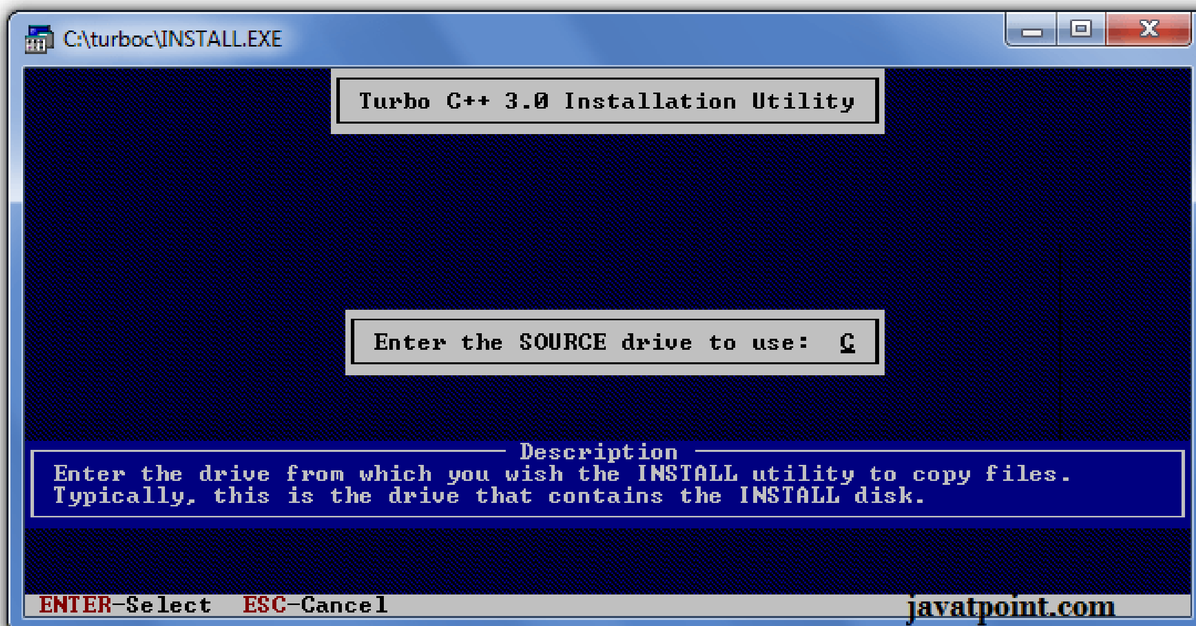


It will ask you to install c or not, press enter to install.



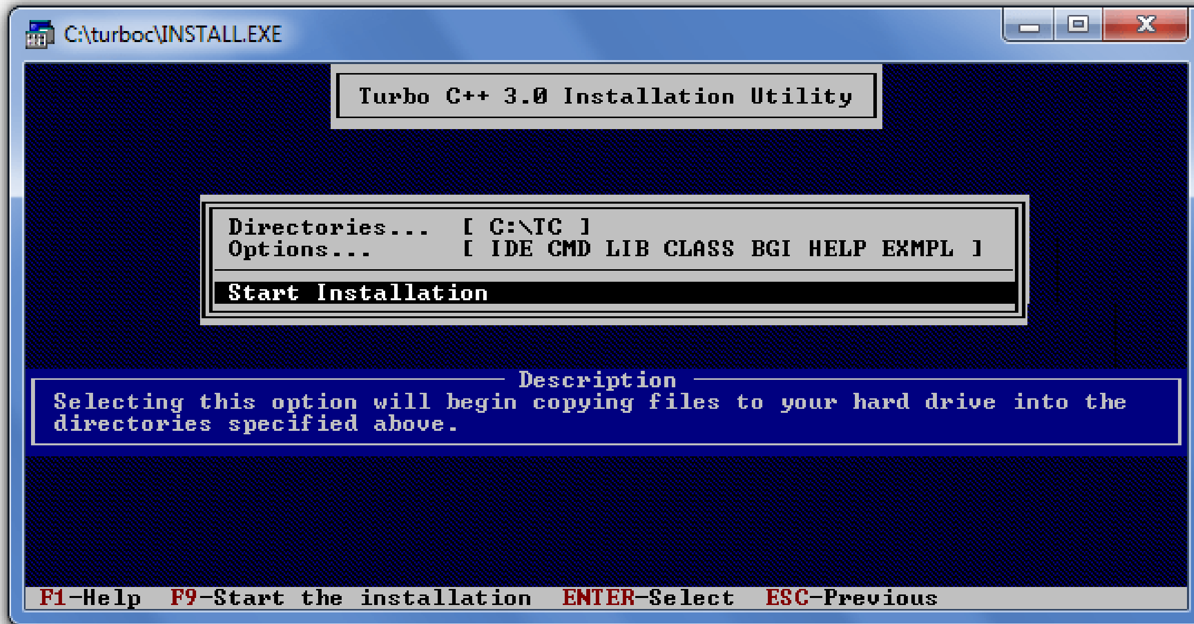
Change your drive to c, press c.



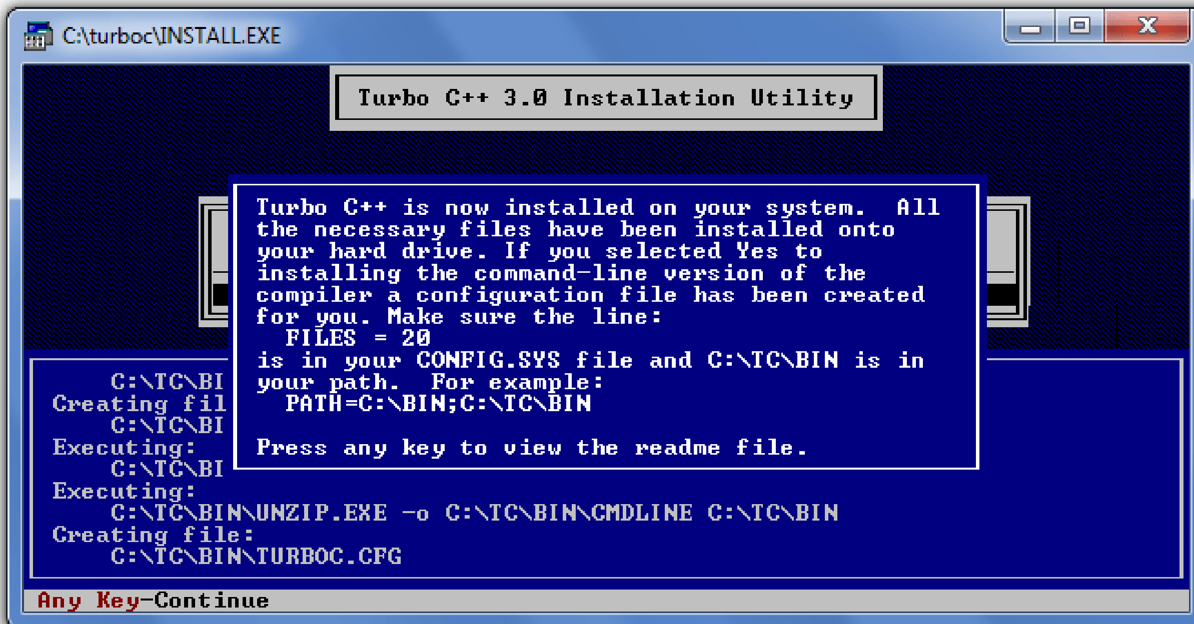


Press enter, it will look inside the c:\turboc directory for the required files.

Select Start installation by the down arrow key then press enter.

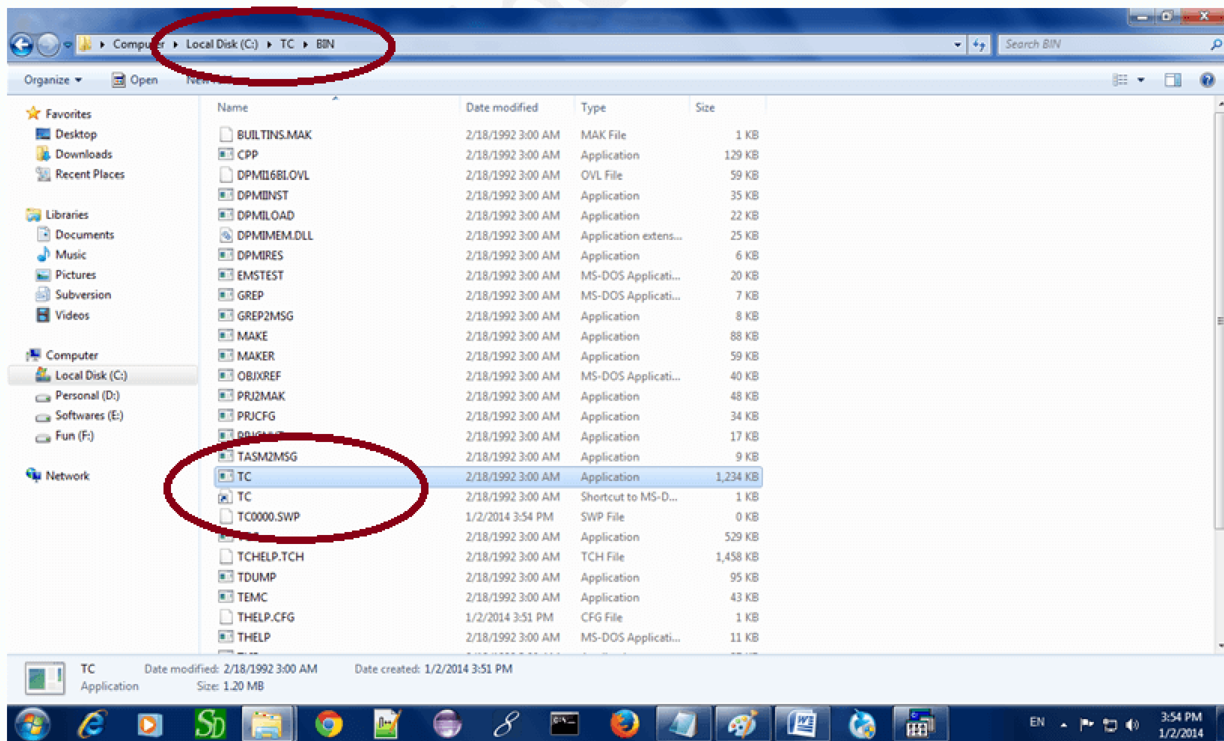


Now C is installed, press enter to read documentation or close the software.



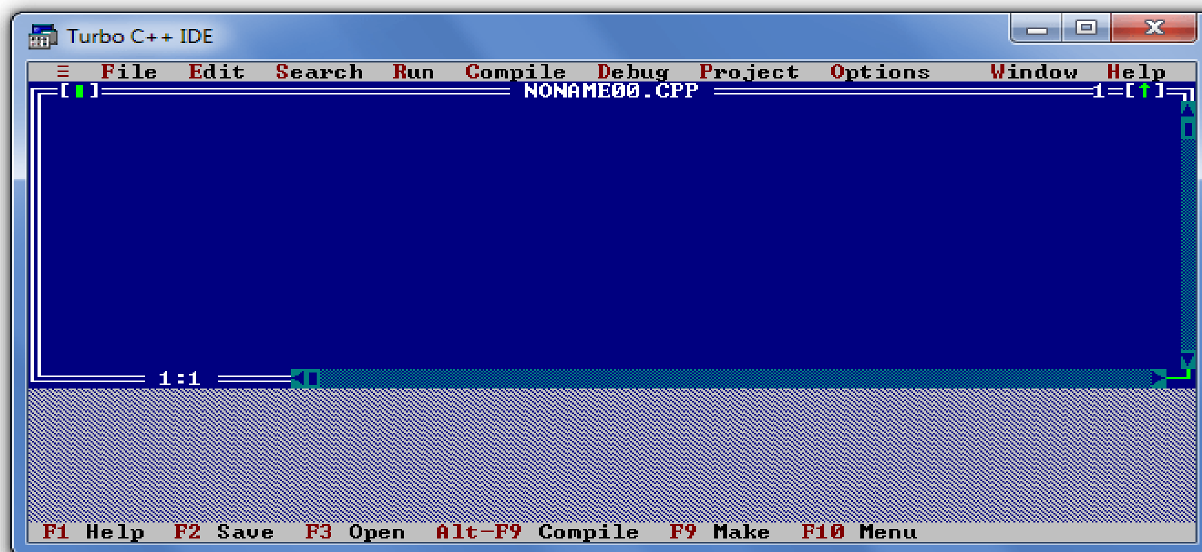
4) Click on the tc application located inside c:\TC\BIN

Now double click on the tc icon located in c:\TC\BIN directory to write the c program.



In windows 7 or window 8, it will show a dialog block to ignore and close the application because fullscreen mode is not supported. Click on Ignore button.

Now it will showing following console.



First C Program

Before starting the abcd of C language, you need to learn how to write, compile and run the first c program.

To write the first c program, open the C console and write the following code:

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

is called as preprocessor directive.

Include is a directory where all the header files are located. It's physical path is C:\TC

Stdio.h is the header file.

#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.

How to compile and run the c program

There are 2 ways to compile and run the c program, by menu and by shortcut.

By menu

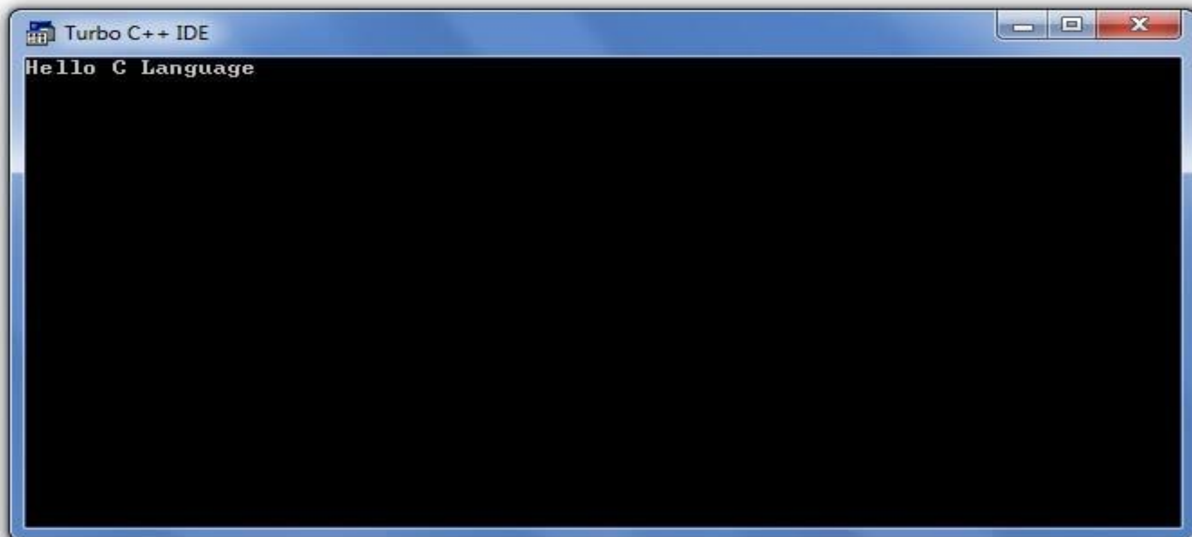
Now **click on the compile menu then compile sub menu** to compile the c program.

Then **click on the run menu then run sub menu** to run the c program.

By shortcut

Or, press ctrl+f9 keys compile and run the program directly.

You will see the following output on user screen.



You can view the user screen any time by pressing the **alt+f5** keys.

Now **press Esc** to return to the turbo c++ console.

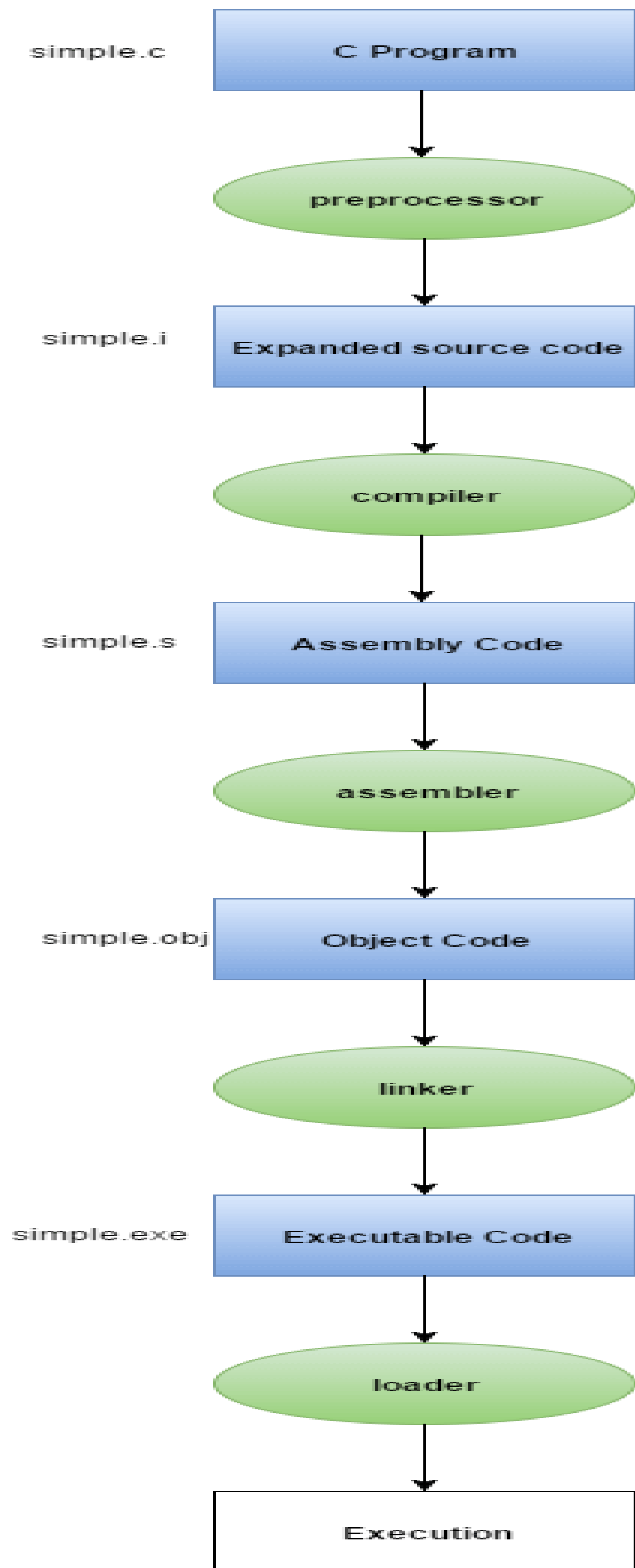
Flow of C Program

The C program follows many steps in execution. To understand the flow of C program well, let us see a simple program first.

```
#include <stdio.h>
#include <conio.h>
int main()
{
	printf("Hello C Language");
	getch();
	return 0;
}
```

Execution Flow

Let's try to understand the flow of above program by the figure given below.


simple.c
C Program
preprocessor
simple.i
Expanded source code
compiler
simple.s
Assembly Code
assembler
simple.obj
Object Code
linker
simple.exe
Executable Code
loader
Execution

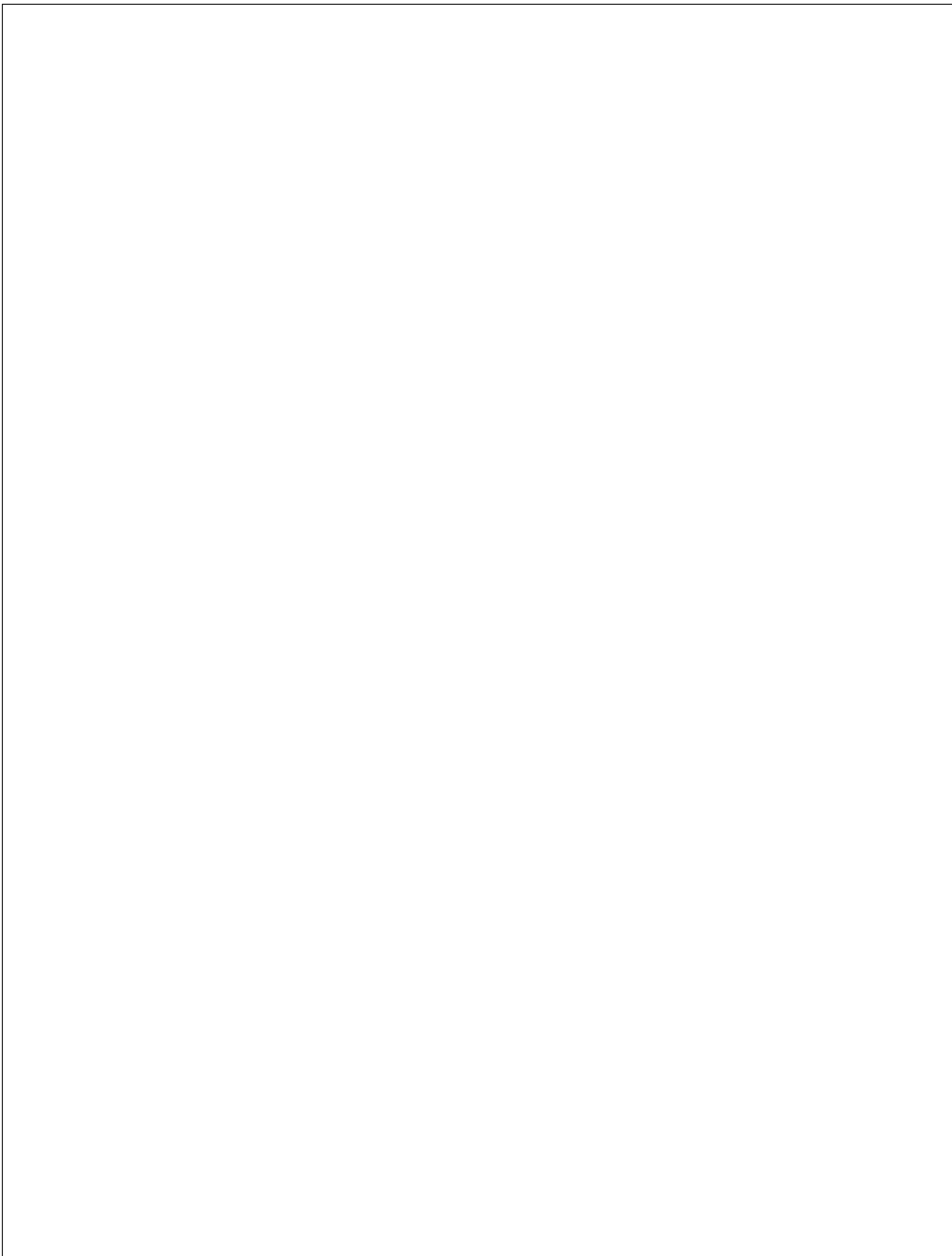
1) C program (source code) is sent to preprocessor first. The preprocessor is responsible to convert preprocessor directives into their respective values. The preprocessor generates an expanded source code.

2) Expanded source code is sent to compiler which compiles the code and converts it into assembly code.

3) The assembly code is sent to assembler which assembles the code and converts it into object code. Now a simple.obj file is generated.

4) The object code is sent to linker which links it to the library such as header files. Then it is converted into executable code. A simple.exe file is generated.

5) The executable code is sent to loader which loads it into memory and then it is executed. After execution, output is sent to console.



printf() and scanf() in C

The printf() and scanf() functions are used for input and output in C language. Both functions are 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:

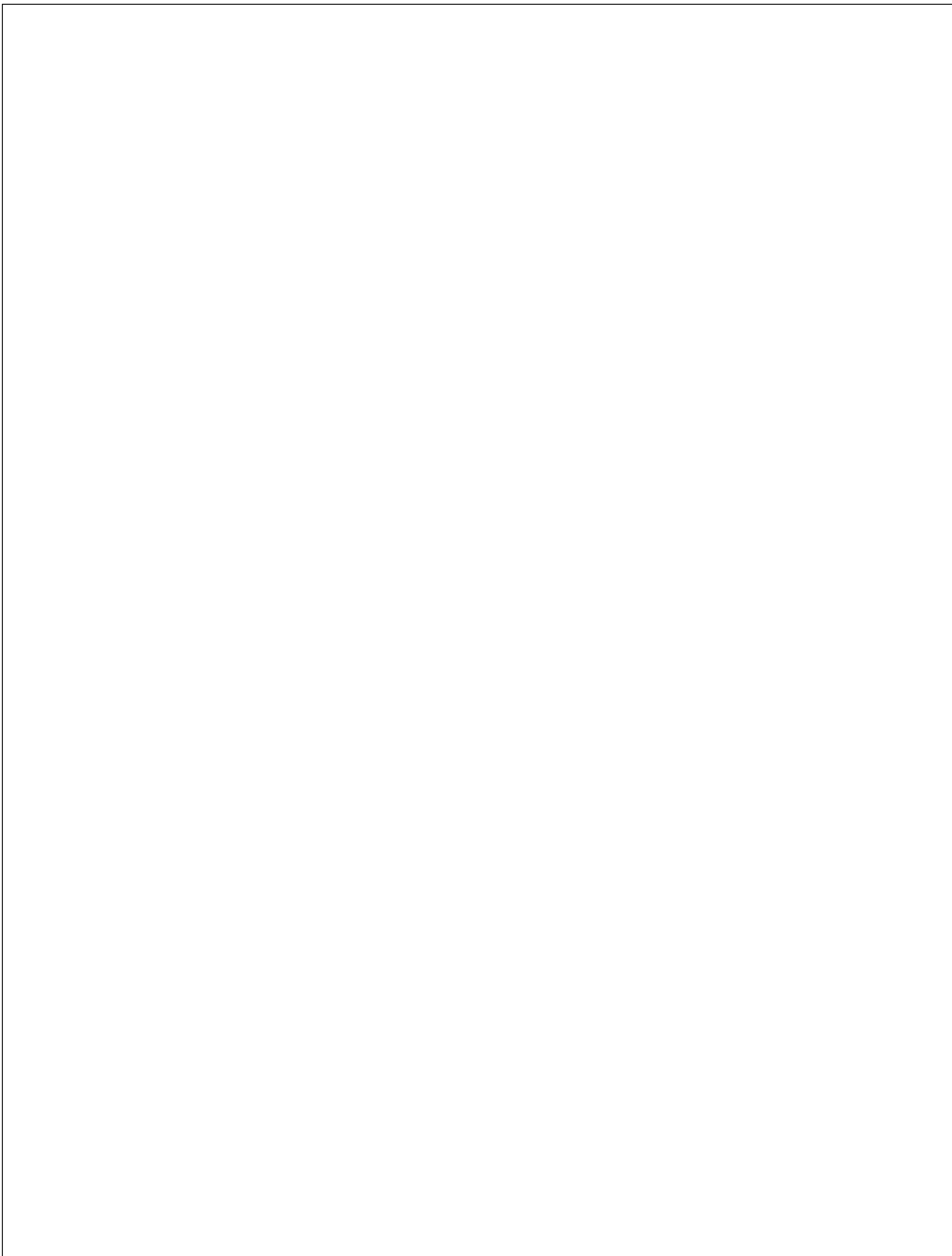
```
printf("format string",argument_list);
```

The **format string** can be %d (integer), %c (character), %s (string), %f (float) etc.

scanf() function

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

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



Program to print cube of given number

Let's see a simple example of c language that gets input from the user and prints the cube of the given number.

```
#include<stdio.h>
int main()
{
        int number;
        printf("enter a number:");
        scanf("%d",&number);
        printf("cube of number is:%d ",number*number*number);
        return 0;
}
```

Output

```
enter a number:5
cube of number is:125
```

The **scanf("%d",&number)** statement reads integer number from the console and stores the given value in number variable.

The **printf("cube of number is:%d ",number*number*number)** statement prints the cube of number on the console.

Program to print sum of 2 numbers

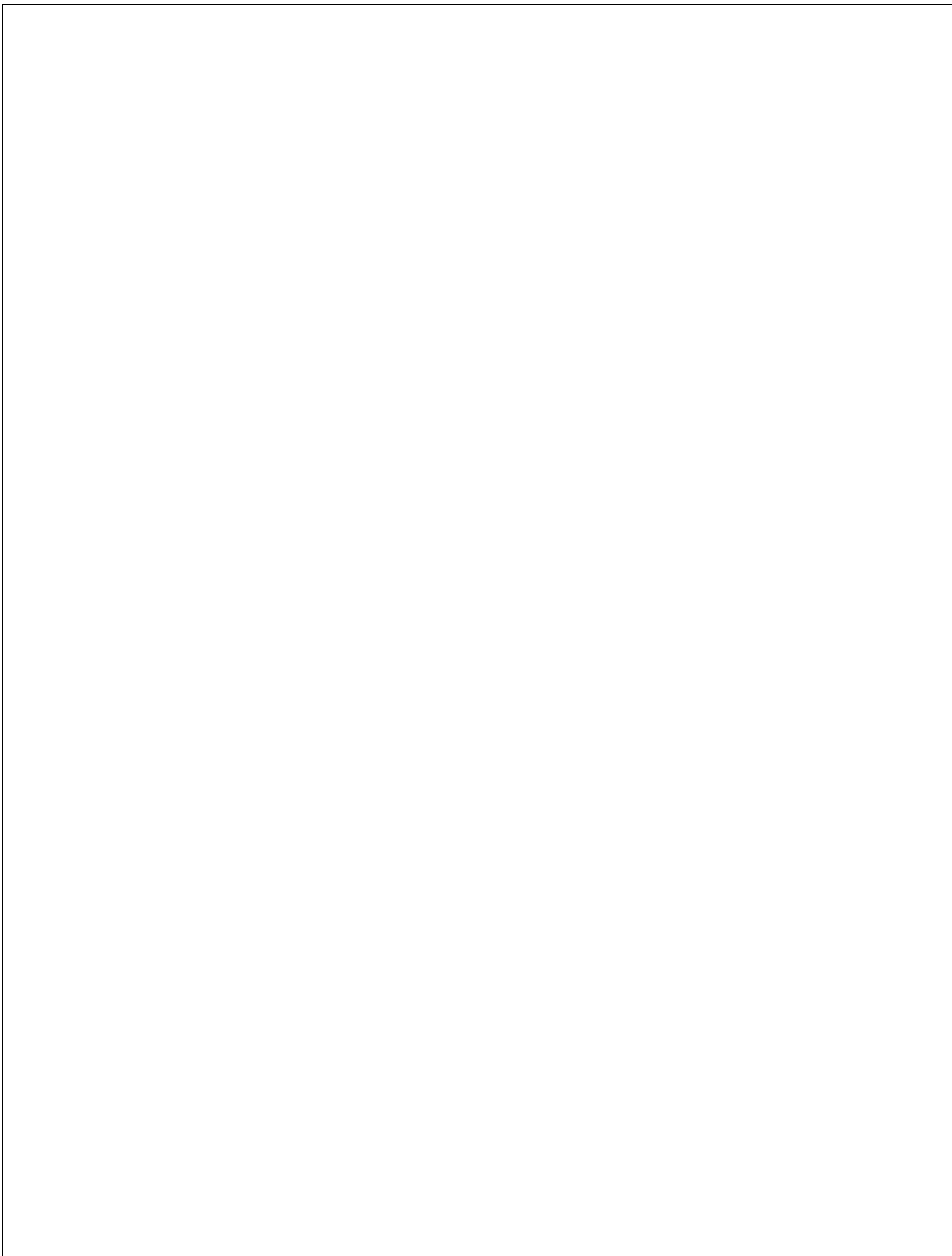
Let's see a simple example of input and output in C language that prints addition of 2 numbers.

```
#include<stdio.h>
int main()
{
        int x=0,y=0,result=0;
        printf("enter first number:");
        scanf("%d",&x);
        printf("enter second number:");
        scanf("%d",&y);
        result=x+y;
        printf("sum of 2 numbers:%d ",result);
        return 0;
}
```

Output

```
enter first number:9
```

```
enter second number:9
sum of 2 numbers:18
```



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;

The example of declaring the variable is given below:

```c
int a;
float b;
char c;
```

Here, a, b, c are variables. The int, float, char are the data types.

We can also provide values while declaring the variables as given below:

```c
int a=10,b=20;//declaring 2 variable of integer type
float f=20.8;
char c='A';
```

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;

The example of declaring the variable is given below:

```c
int a;
float b;
char c;
```

Here, a, b, c are variables. The int, float, char are the data types.

We can also provide values while declaring the variables as given below:

```
int a=10,b=20;//declaring 2 variable of integer type
float f=20.8;
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.

Valid variable names:

```
int a;
int _ab;
int a30;
```

Invalid variable names

```
int 2;
int a b;
int long;
```

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
}
```

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

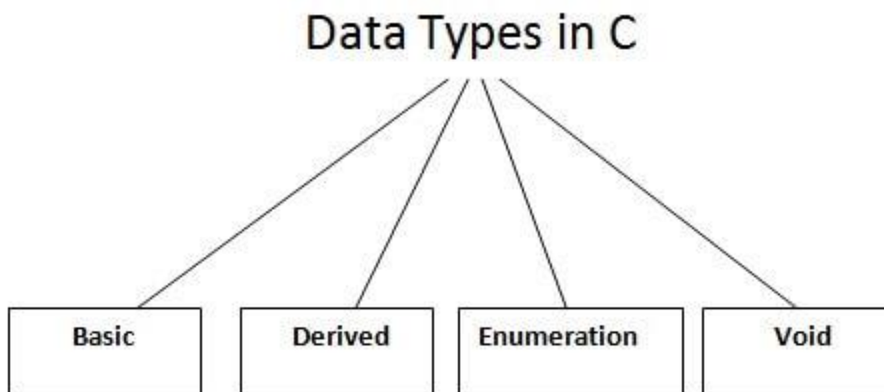
```
extern int x=10;//external variable (also global)
```

program1.c

```
#include "myfile.h"
#include <stdio.h>
void printValue()
{
   printf("Global variable: %d", global_variable);
}
```

Data Types in C

A data type specifies the type of data that a variable can store such as integer, floating, character, etc.



There are the following data types in C language.

Types	Data Types
Basic Data Type	int, char, float, double
Derived Data Type	array, pointer, structure, union
Enumeration Data Type	enum
Void Data Type	void

Basic Data Types

The basic data types are integer-based and floating-based. C language supports both signed and unsigned literals.

The memory size of the basic data types may change according to 32 or 64-bit operating system.

Let's see the basic data types. Its size is given **according to 32-bit architecture**.

Data Types	Memory Size	Range
char	1 byte	−128 to 127
signed char	1 byte	−128 to 127
unsigned char	1 byte	0 to 255
short	2 byte	−32,768 to 32,767
signed short	2 byte	−32,768 to 32,767
unsigned short	2 byte	0 to 65,535
int	2 byte	−32,768 to 32,767
signed int	2 byte	−32,768 to 32,767
unsigned int	2 byte	0 to 65,535
short int	2 byte	−32,768 to 32,767
signed short int	2 byte	−32,768 to 32,767
unsigned short int	2 byte	0 to 65,535
long int	4 byte	-2,147,483,648 to 2,147,483,647
signed long int	4 byte	-2,147,483,648 to 2,147,483,647
unsigned long int	4 byte	0 to 4,294,967,295
float	4 byte	
double	8 byte	
long double	10 byte	

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 Operators

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
- Relational Operators
- Shift Operators
- Logical Operators
- Bitwise Operators
- Ternary or Conditional Operators
- Assignment Operator
- Misc Operator

Precedence of Operators in C

The precedence of operator species that which operator will be evaluated first and next. The associativity specifies the operator direction to be evaluated; it may be left to right or right to left.

Let's understand the precedence by the example given below:

```
int value=10+20*10;
```

The value variable will contain **210** because * (multiplicative operator) is evaluated before + (additive operator).

The precedence and associativity of C operators is given below:

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

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

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;
}
```

Output: `Hello C`

Even you can place the comment after the statement. For example:

```
printf("Hello C");//printing information
```

Multi Line Comments

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;
}
```

Output: `Hello C`

Escape Sequence in C

An escape sequence in C language is a sequence of characters that doesn't represent itself when used inside string literal or character.

It is composed of two or more characters starting with backslash \. For example: \n represents new line.

List of Escape Sequences in C

Escape Sequence	Meaning
\a	Alarm or Beep
\b	Backspace
\f	Form Feed
\n	New Line
\r	Carriage Return
\t	Tab (Horizontal)
\v	Vertical Tab
\\	Backslash
\'	Single Quote
\"	Double Quote
\?	Question Mark
\nnn	octal number
\xhh	hexadecimal number
\0	Null

Escape Sequence Example

```
#include<stdio.h>
int main()
{
    int number=50;
    printf("You\nare\nlearning\n\'c\' language\n\"Do you know C language\"");
    return 0;
}
```

Output:

```
You
are
learning
'c' language
"Do you know C language"
```

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.

There are different types of constants in C programming.

List of Constants in C

Constant	Example
Decimal Constant	10, 20, 450 etc.
Real or Floating-point Constant	10.3, 20.2, 450.6 etc.
Octal Constant	021, 033, 046 etc.
Hexadecimal Constant	0x2a, 0x7b, 0xaa etc.
Character Constant	'a', 'b', 'x' etc.
String Constant	"c", "c program", "c in vivek" etc.

2 ways to define constant in C

There are two ways to define constant in C programming.

1. const keyword
2. #define preprocessor

1) Const keyword

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
```

2) C #define preprocessor

The #define preprocessor is also used to define constant. We will learn about #define preprocessor directive later.

C if else Statement

The if-else statement in C is used to perform the operations based on some specific condition. The operations specified in if block are executed if and only if the given condition is true.

There are the following variants of if statement in C language.

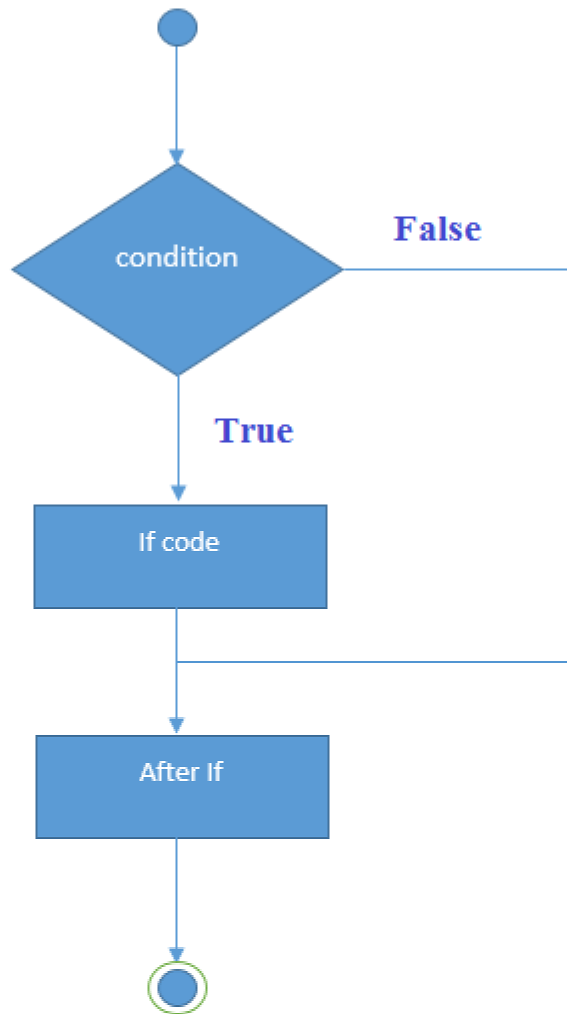
- If statement
- If-else statement
- If else-if ladder
- Nested if

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
```

Program to find the largest number of the three.

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

Output

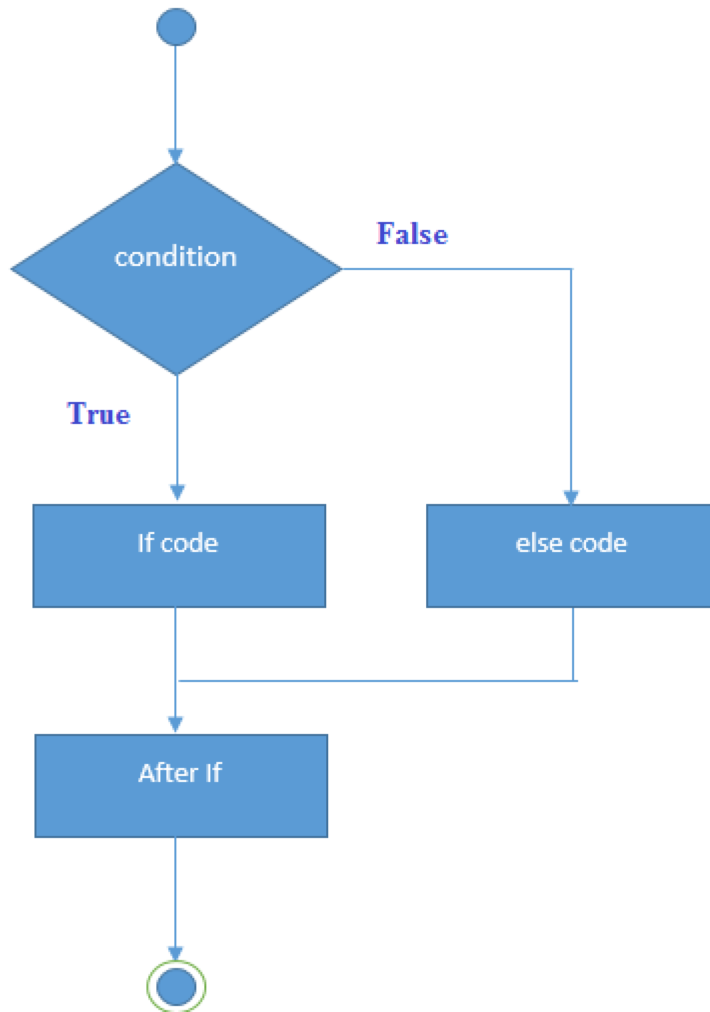
```
Enter three numbers?
12 23 34
34 is largest
```

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 simiulteneously. 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
```

Program to check whether a person is eligible to vote or not.

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

Output

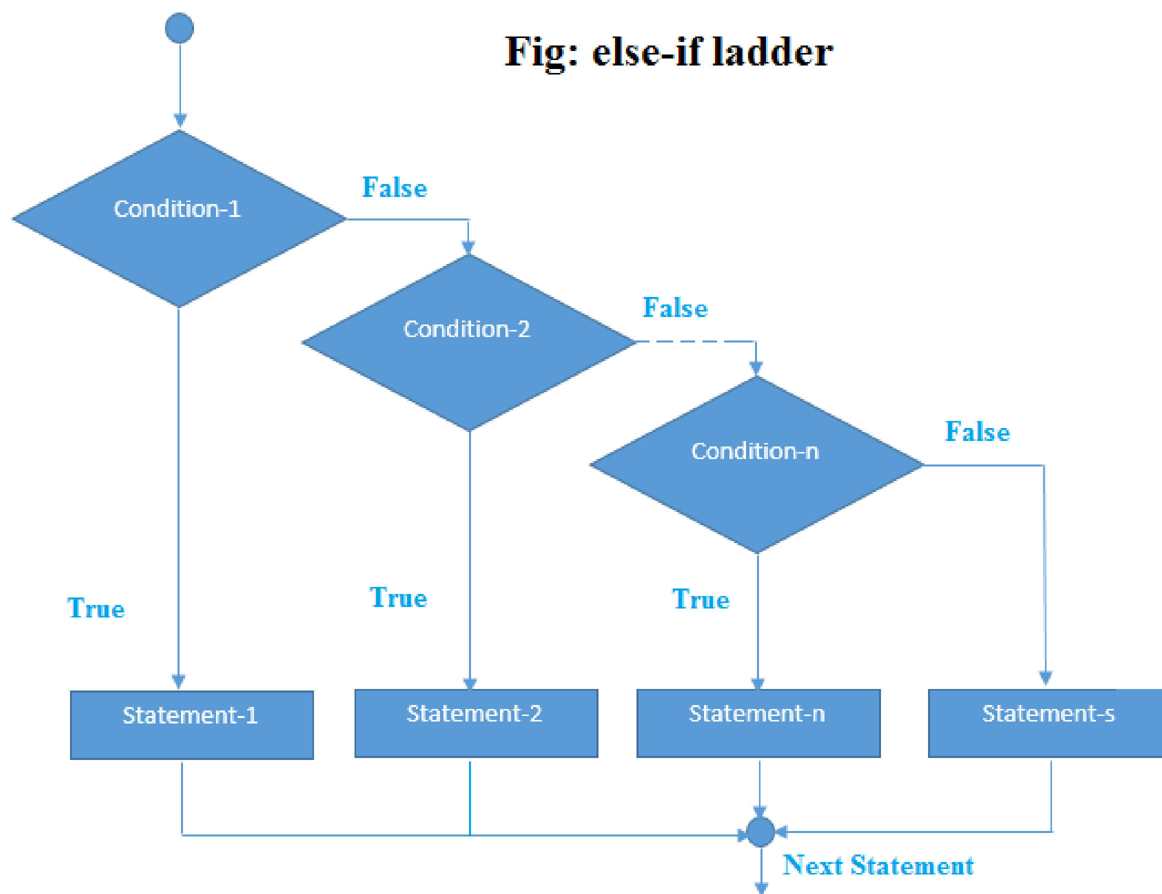
```
Enter your age?18
You are eligible to vote...
Enter your age?13
Sorry ... you can't vote
```

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.

```
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
}
```

Flowchart of else-if ladder statement in C



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
```

Program to calculate the grade of the student according to the specified marks.

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

```
  }
}
```

Output

```
Enter your marks?10
Sorry you are fail ...

Enter your marks?40
You scored grade C ...

Enter your marks?90
Congrats ! you scored grade A ...
```

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;
}
```

Rules for switch statement in C language

1) The *switch expression* must be of an integer or character type.

2) The *case value* must be an integer or character constant.

3) The *case value* can be used only inside the switch statement.

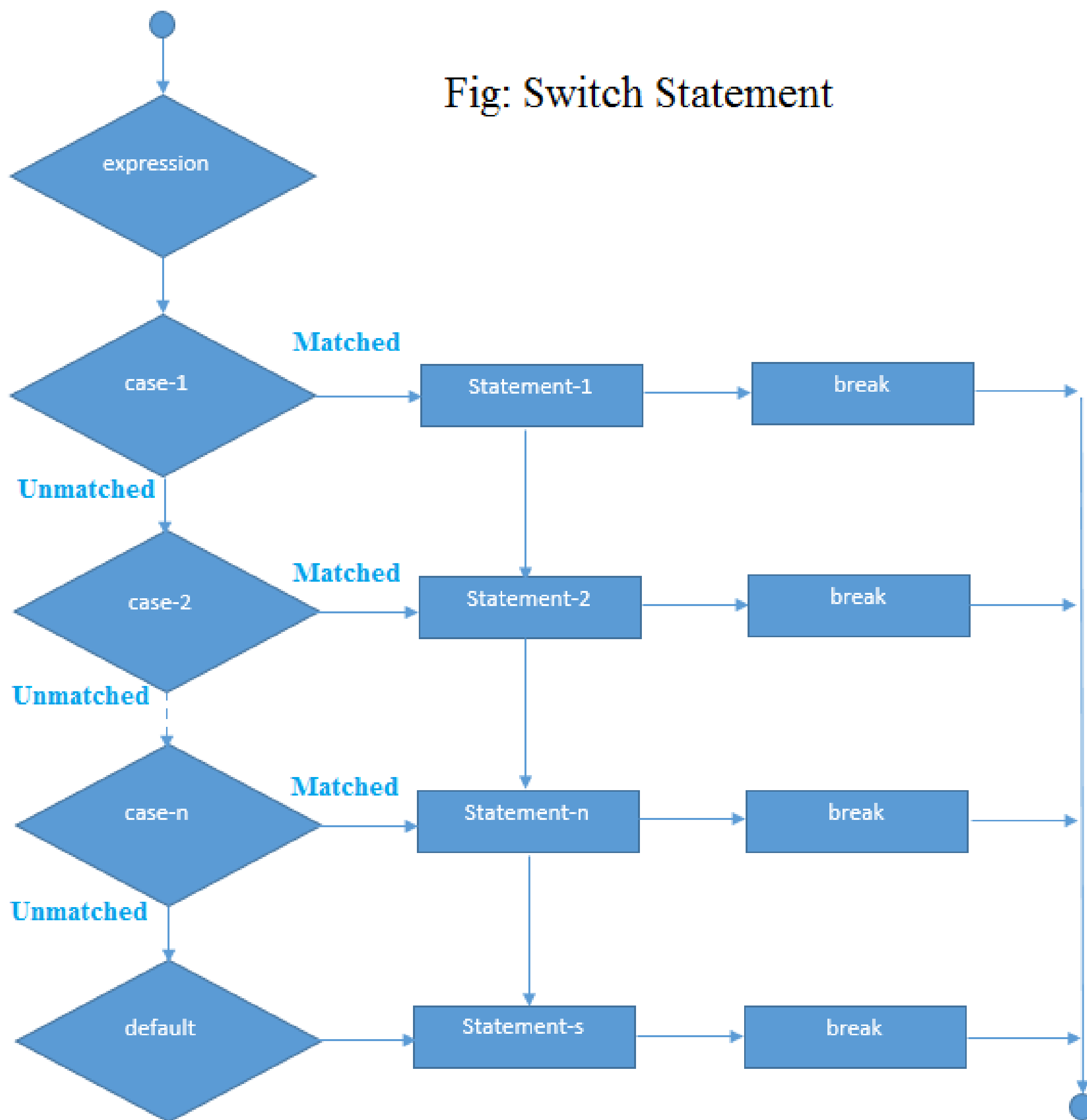
4) The *break statement* in switch case is not must. It is optional. If there is no break statement found in the case, all the cases will be executed present after the matched case. It is known as *fall through* the state of C switch statement.

Let's try to understand it by the examples. We are assuming that there are following variables.

```
int x,y,z;
char a,b;
float f;
```

Valid Switch	Invalid Switch	Valid Case	Invalid Case
switch(x)	switch(f)	case 3;	case 2.5;
switch(x>y)	switch(x+2.5)	case 'a';	case x;
switch(a+b-2)		case 1+2;	case x+2;
switch(func(x,y))		case 'x'>'y';	case 1,2,3;

Flowchart of switch statement in C



Functioning of switch case statement

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

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

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

Output

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

enter a number:50
number is equal to 50
```

Switch case example 2

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

```
        break;
        default:
        printf(" Hello bye ");
    }
}
```

Output

```
hi
```

C Switch statement is fall-through

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

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

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

Output

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

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

Nested switch case statement

We can use as many switch statement as we want inside a switch statement. Such type of statements is called nested switch case statements. Consider the following example.

```
#include <stdio.h>
int main ()
{
  int i = 10;
  int j = 20;
  switch(i)
  {
      case 10:
      printf("the value of i evaluated in outer switch: %d\n",i);
      case 20:
      switch(j)
     {
          case 20:
           printf("The value of j evaluated in nested switch: %d\n",j);
      }
   }
  printf("Exact value of i is : %d\n", i );
  printf("Exact value of j is : %d\n", j );
  return 0;
}
```

Output

```
the value of i evaluated in outer switch: 10
The value of j evaluated in nested switch: 20
Exact value of i is : 10
Exact value of j is : 20
```

C 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.

Why use loops in C language?

The looping simplifies the complex problems into the easy ones. It enables us to alter the flow of the program so that instead of writing the same code again and again, we can repeat the same code for a finite number of times. For example, if we need to print the first 10 natural numbers then, instead of using the printf statement 10 times, we can print inside a loop which runs up to 10 iterations.

Advantage of loops in C

1) It provides code reusability.

2) Using loops, we do not need to write the same code again and again.

2) Using loops, we can traverse over the elements of data structures (array or linked lists).

Types of C Loops

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

1. do while
2. while
3. 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).

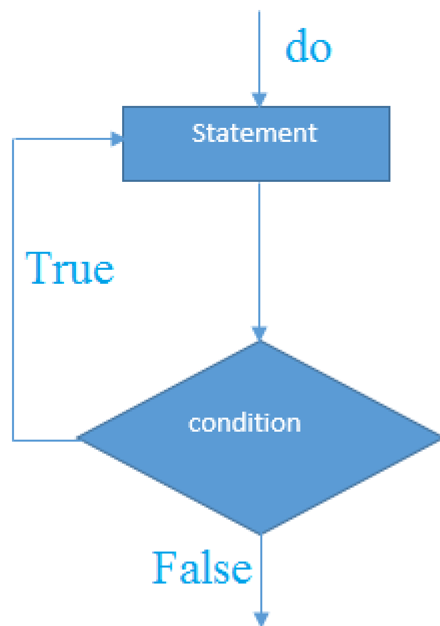
The syntax of do-while loop in c language is given below:

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

```c
#include<stdio.h>
#include<stdlib.h>
void main ()
{
   char c;
   int choice,dummy;
   do{
   printf("\n1. Print Hello\n2. Print Vivek\n3. Exit\n");
   scanf("%d",&choice);
   switch(choice)
   {
      case 1 :
      printf("Hello");
      break;
      case 2:
      printf("Vivek");
      break;
      case 3:
      exit(0);
      break;
      default:
      printf("please enter valid choice");
   }
   printf("do you want to enter more?");
   scanf("%d",&dummy);
   scanf("%c",&c);
   }while(c=='y');
}
```

Output

```
1. Print Hello
2. Print Vivek
3. Exit
1
Hello
do you want to enter more?
y

1. Print Hello
2. Print Vivek
3. Exit
2
Vivek
do you want to enter more?
n
```



do while example

There is given the simple program of c language do while loop where we are printing the table of 1.

```
#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
```

Program to print table for the given number using do while loop

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

Output

```
Enter a number: 5
5
10
15
20
25
30
35
40
45
50
```

Infinitive do while loop

The do-while loop will run infinite times if we pass any non-zero value as the conditional expression.

```
do{
//statement
}while(1);
```

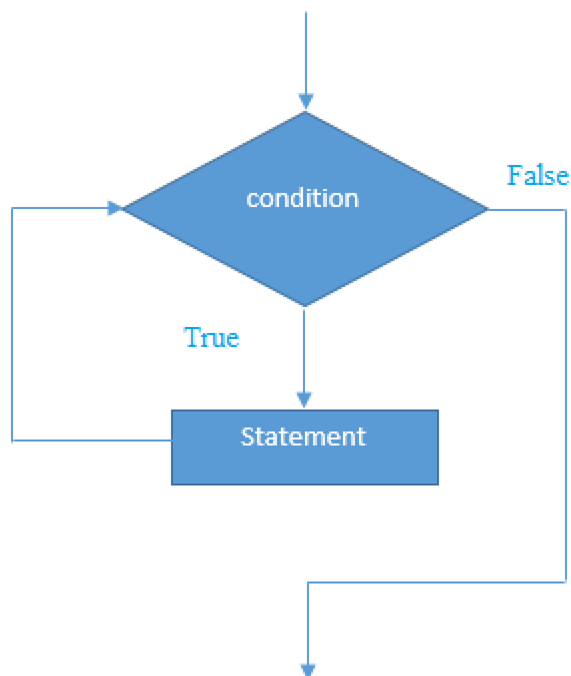
while loop in C

The while loop in c is to be used in the scenario where we don't know the number of iterations in advance. The block of statements is executed in the while loop until the condition specified in the while loop is satisfied. It is also called a pre-tested loop.

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

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

Flowchart of while loop in C



Example of the while loop in C language

Let's see the simple program of while loop that prints table of 1.

```
#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
```

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

```
#include<stdio.h>
int main()
{
	int i=1,number=0,b=9;
	printf("Enter a number: ");
	scanf("%d",&number);
	while(i<=10)
	{
		printf("%d \n",(number*i));
		i++;
	}
	return 0;
}
```

Output

```
Enter a number: 50
50
100
150
200
250
300
350
```

```
400
450
500
```

Properties of while loop

- A conditional expression is used to check the condition. The statements defined inside the while loop will repeatedly execute until the given condition fails.
- The condition will be true if it returns 0. The condition will be false if it returns any non-zero number.
- In while loop, the condition expression is compulsory.
- Running a while loop without a body is possible.
- We can have more than one conditional expression in while loop.
- If the loop body contains only one statement, then the braces are optional.

Example 1

```
#include<stdio.h>
void main ()
{
   int j = 1;
   while(j+=2,j<=10)
   {
      printf("%d ",j);
   }
   printf("%d",j);
}
```

Output

```
3 5 7 9 11
```

Example 2

```
#include<stdio.h>
void main ()
{
   while()
   {
      printf("hello Vivek");
   }
}
```

Output

```
compile time error: while loop can't be empty
```

Example 3

```
#include<stdio.h>
void main ()
{
   int x = 10, y = 2;
   while(x+y-1)
   {
      printf("%d %d",x--,y--);
   }
}
```

Output

```
infinite loop
```

Infinitive while loop in C

If the expression passed in while loop results in any non-zero value then the loop will run the infinite number of times.

```
while(1){
//statement
}
```

for loop in C

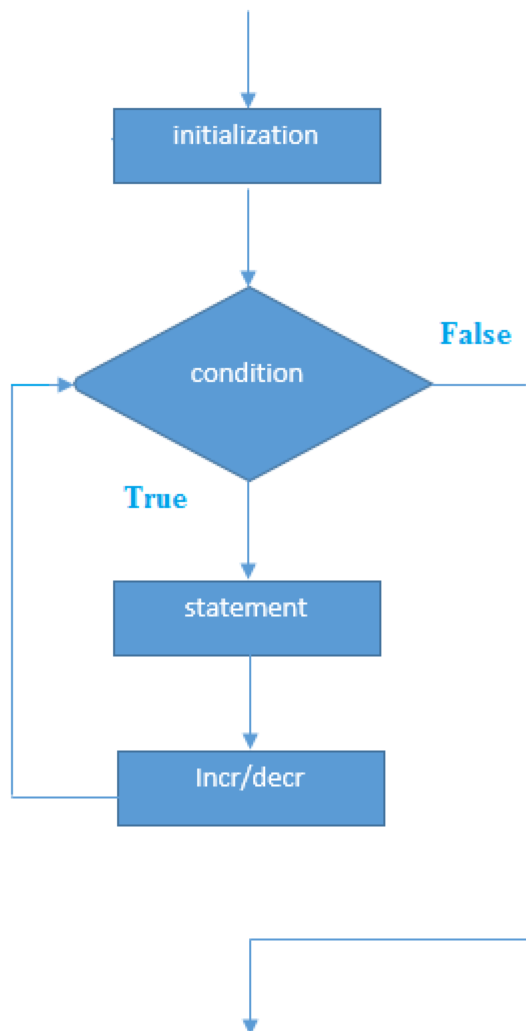
The **for loop in C language** is used to iterate the statements or a part of the program several times. It is frequently used to traverse the data structures like the array and linked list.

Syntax of for loop in C

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

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

Flowchart of for loop in C



C for loop Examples

Let's see the simple program of for loop that prints table of 1.

```
#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
```

C Program: Print table for the given number using C for loop

```
#include<stdio.h>
int main()
{
	int i=1,number=0;
	printf("Enter a number: ");
	scanf("%d",&number);
	for(i=1;i<=10;i++){
		printf("%d \n",(number*i));
	}
	return 0;
}
```

Output

```
Enter a number: 2
2
```

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

Properties of Expression 1

- The expression represents the initialization of the loop variable.
- We can initialize more than one variable in Expression 1.
- Expression 1 is optional.
- In C, we can not declare the variables in Expression 1. However, It can be an exception in some compilers.

Example 1

```
#include <stdio.h>
int main()
{
    int a,b,c;
    for(a=0,b=12,c=23;a<2;a++)
    {
        printf("%d ",a+b+c);
    }
}
```

Output

```
35 36
```

Example 2

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

Output

```
1 2 3 4
```

Properties of Expression 2

- Expression 2 is a conditional expression. It checks for a specific condition to be satisfied. If it is not, the loop is terminated.
- Expression 2 can have more than one condition. However, the loop will iterate until the last condition becomes false. Other conditions will be treated as statements.
- Expression 2 is optional.
- Expression 2 can perform the task of expression 1 and expression 3. That is, we can initialize the variable as well as update the loop variable in expression 2 itself.
- We can pass zero or non-zero value in expression 2. However, in C, any non-zero value is true, and zero is false by default.

Example 1

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

Output

```
0 1 2 3 4
```

Example 2

```
#include <stdio.h>
int main()
{
   int i,j,k;
   for(i=0,j=0,k=0;i<4,k<8,j<10;i++)
   {
      printf("%d %d %d\n",i,j,k);
      j+=2;
      k+=3;
   }
}
```

Output

```
0 0 0
1 2 3
2 4 6
```

```
3 6 9
4 8 12
```

Example 3

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

Output

```
infinite loop
```

Properties of Expression 3

- Expression 3 is used to update the loop variable.
- We can update more than one variable at the same time.
- Expression 3 is optional.

Example 1

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

Output

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

Loop body

The braces {} are used to define the scope of the loop. However, if the loop contains only one statement, then we don't need to use braces. A loop without a body is possible. The braces work as a block separator, i.e., the value variable declared inside for loop is valid only for that block and not outside. Consider the following example.

```
#include<stdio.h>
void main ()
{
   int i;
   for(i=0;i<10;i++)
   {
      int i = 20;
      printf("%d ",i);
   }
}
```

```
20 20 20 20 20 20 20 20 20 20
```

Infinitive for loop in C

To make a for loop infinite, we need not give any expression in the syntax. Instead of that, we need to provide two semicolons to validate the syntax of the for loop. This will work as an infinite for loop.

```
#include<stdio.h>
void main ()
{
   for(;;)
   {
      printf("welcome to vivek");
   }
}
```

If you run this program, you will see above statement infinite times.

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:

1. With switch case
2. With loop

Syntax:

```
//loop or switch case
break;
```

Flowchart of break in c

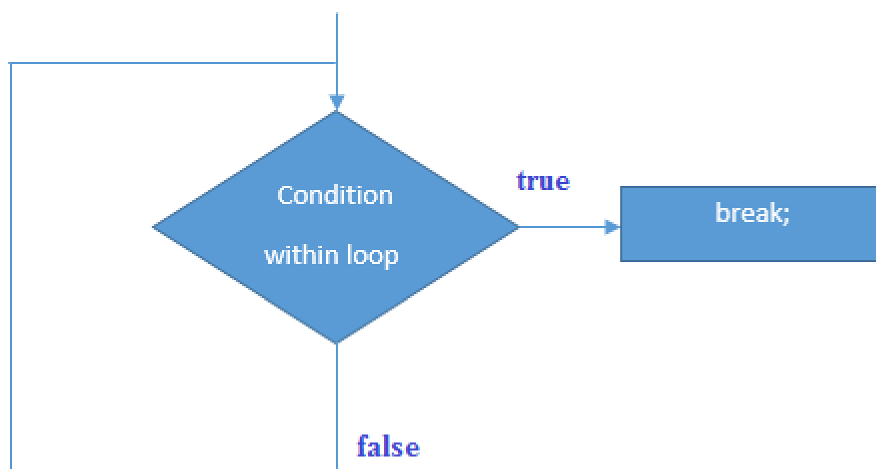


Figure: Flowchart of break statement

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 break statement with the nested loop

In such case, it breaks only the inner loop, but not outer loop.

```
#include<stdio.h>
int main()
{
        int i=1,j=1;//initializing a local variable
        for(i=1;i<=3;i++)
        {
                for(j=1;j<=3;j++)
                {
                        printf("%d &d\n",i,j);
                        if(i==2 && j==2){
                                break;//will break loop of j only
                        }
                }//end of for loop
         }
        return 0;
}
```

Output

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

As you can see the output on the console, 2 3 is not printed because there is a break statement after printing i==2 and j==2. But 3 1, 3 2 and 3 3 are printed because the break statement is used to break the inner loop only.

break statement with while loop

Consider the following example to use break statement inside while loop.

```
#include<stdio.h>
void main ()
{
   int i = 0;
   while(1)
   {
      printf("%d  ",i);
      i++;
      if(i == 10)
      break;
   }
   printf("came out of while loop");
}
```

Output

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

break statement with do-while loop

Consider the following example to use the break statement with a do-while loop.

```
#include<stdio.h>
void main ()
{
  int n=2,i,choice;
  do
  {
     i=1;
     while(i<=10)
     {
        printf("%d X %d = %d\n",n,i,n*i);
        i++;
     }
```

```
    printf("do you want to continue with the table of %d , enter any nonzero value to continu
e.",n+1);
       scanf("%d",&choice);
    if(choice == 0)
       {
          break;
       }
       n++;
   }while(1);
}
```

Output

```
2 X 1 = 2
2 X 2 = 4
2 X 3 = 6
2 X 4 = 8
2 X 5 = 10
2 X 6 = 12
2 X 7 = 14
2 X 8 = 16
2 X 9 = 18
2 X 10 = 20
do you want to continue with the table of 3 , enter any non-zero value to
continue.1
3 X 1 = 3
3 X 2 = 6
3 X 3 = 9
3 X 4 = 12
3 X 5 = 15
3 X 6 = 18
3 X 7 = 21
3 X 8 = 24
3 X 9 = 27
3 X 10 = 30
do you want to continue with the table of 4 , enter any non-zero value to
continue.0
```

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
```

Continue statement example 1

```
#include<stdio.h>
void main ()
{
   int i = 0;
   while(i!=10)
   {
      printf("%d", i);
      continue;
      i++;
   }
}
```

Output

```
infinite loop
```

Continue statement example 2

```
#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
```

As you can see, 5 is not printed on the console because loop is continued at i==5.

C continue statement with inner loop

In such case, C continue statement continues only inner loop, but not outer loop.

```
#include<stdio.h>
int main()
{
    int i=1,j=1;//initializing a local variable
    for(i=1;i<=3;i++)
    {
        for(j=1;j<=3;j++)
        {
            if(i==2 && j==2){
                continue;//will continue loop of j only
            }
        printf("%d %d\n",i,j);
        }
    }//end of for loop
    return 0;
}
```

Output

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

```
3 2
3 3
```

As you can see, 2 2 is not printed on the console because inner loop is continued at i==2 and j==2.

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 statment 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 complecated.

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 x 1 = 10
10 x 2 = 20
10 x 3 = 30
10 x 4 = 40
10 x 5 = 50
10 x 6 = 60
10 x 7 = 70
10 x 8 = 80
10 x 9 = 90
10 x 10 = 100
```

When should we use goto?

The only condition in which using goto is preferable is when we need to break the multiple loops using a single statement at the same time. Consider the following example.

```
#include <stdio.h>
int main()
{
  int i, j, k;
  for(i=0;i<10;i++)
  {
    for(j=0;j<5;j++)
    {
      for(k=0;k<3;k++)
      {
        printf("%d %d %d\n",i,j,k);
        if(j == 3)
        {
          goto out;
        }
      }
    }
  }
  out:
  printf("came out of the loop");
}
```

```
0 0 0
0 0 1
0 0 2
0 1 0
0 1 1
0 1 2
0 2 0
0 2 1
0 2 2
0 3 0
came out of the loop
```

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;

Note: It is always recommended to convert the lower value to higher for avoiding data loss.

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

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

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

Types of Functions

There are two types of functions in C programming:

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

Return Value

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

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

Example without return value:

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

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

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

Example with return value:

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

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

```
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("Vivek");
}
```

Output `Hello Vivek`

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;
}
```

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
```

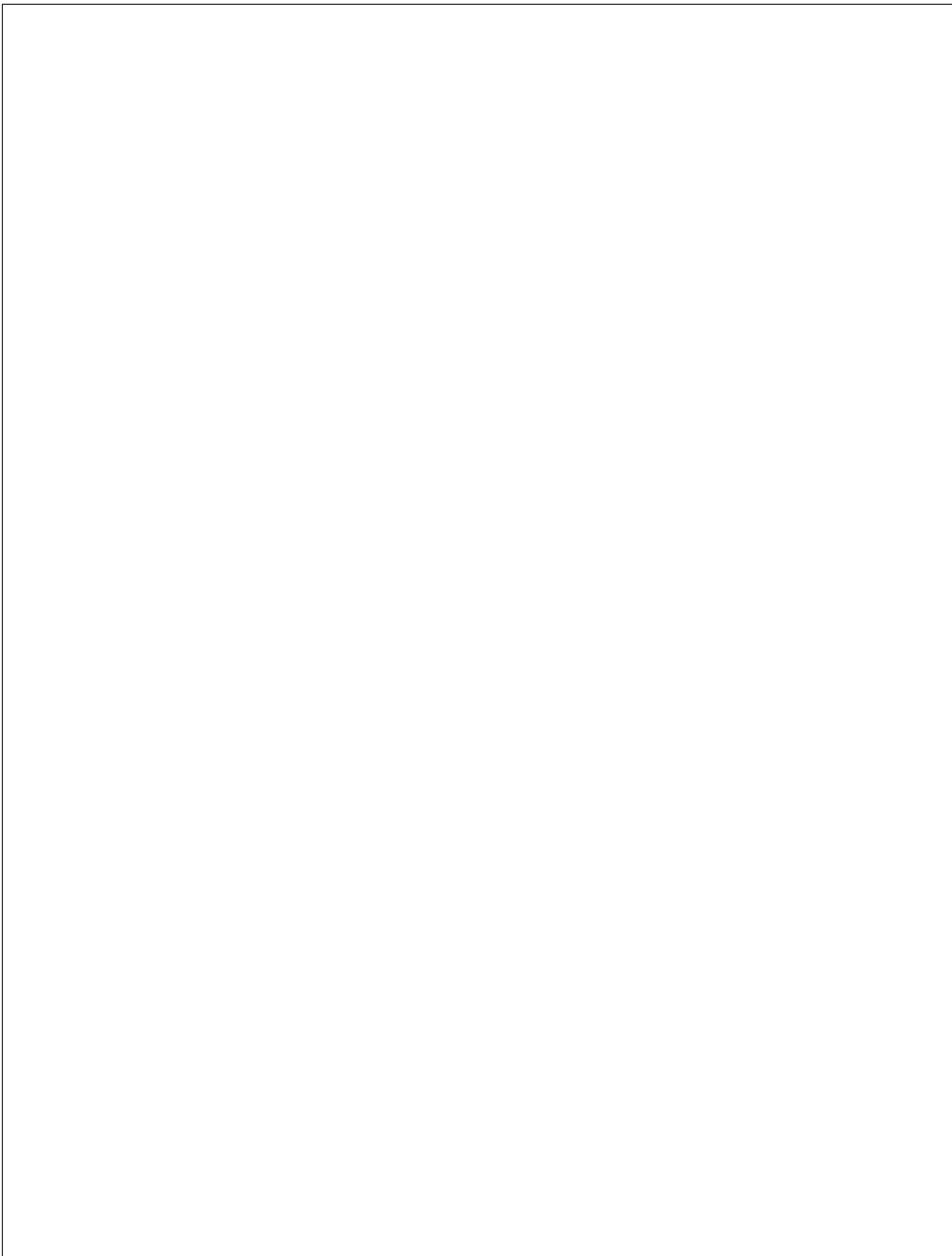
C Library Functions

Library functions are the inbuilt function in C that are grouped and placed at a common place called the library. Such functions are used to perform some specific operations. For example, printf is a library function used to print on the console. The library functions are created by the designers of compilers. All C standard library functions are defined inside the different header files saved with the extension **.h**. We need to include these header files in our program to make use of the library functions defined in such header files. For example, To use the library functions such as printf/scanf we need to include stdio.h in our program which is a header file that contains all the library functions regarding standard input/output.

The list of mostly used header files is given in the following table.

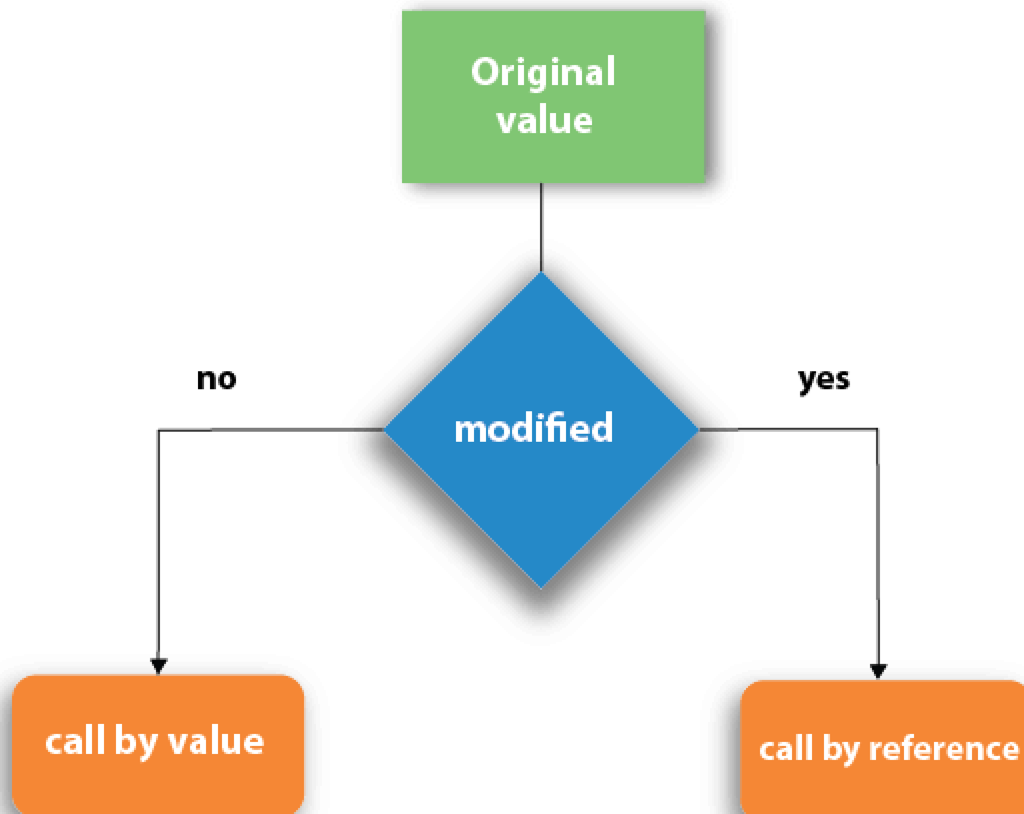
Header file	Description
stdio.h	This is a standard input/output header file. It contains all the library functions regarding input/output.
conio.h	This is a console input/output header file.
string.h	It contains all string related library functions like gets(), puts(),etc.
stdlib.h	This header file contains all the general library functions like malloc(), calloc(), exit(), et
math.h	This header file contains all the math operations related functions like sqrt(), pow(), etc
time.h	This header file contains all the time-related functions.
ctype.h	This header file contains all character handling functions.
stdarg.h	Variable argument functions are defined in this header file.
signal.h	All the signal handling functions are defined in this header file.
setjmp.h	This file contains all the jump functions.
locale.h	This file contains locale functions.
errno.h	This file contains error handling functions.

	assert.h	This file contains diagnostics functions.



Call by value and Call by reference in C

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



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

Call by value in C

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

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

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

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

Output

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

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

```
#include <stdio.h>
void swap(int , int); //prototype of the function
int main()
{
   int a = 10;
   int b = 20;
   printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value
   of a and b in main
   swap(a,b);
   printf("After swapping values in main a = %d, b = %d\n",a,b); // The value of actual paa
   meters do not change by changing the formal parameters in call by value, a = 10, b = 2
}

void swap (int a, int b)
{
   int temp;
```

```
    temp = a;
    a=b;
    b=temp;
    printf("After swapping values in function a = %d, b = %d\n",a,b); // Formal parameters,
a = 20, b = 10
}
```

Output

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

Call by reference in C

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

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

Output

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

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

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

Output

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

Difference between call by value and call by reference in C

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

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
```

We can understand the above program of the recursive method call by the method given below:

```
return 5 * factorial(4) = 120
  └── return 4 * factorial(3) = 24
        └── return 3 * factorial(2) = 6
              └── return 2 * factorial(1) = 2
                    └── return 1 * factorial(0) = 1

1 * 2 * 3 * 4 * 5 = 120
```

Recursive Function

A recursive function performs the tasks by dividing it into the subtasks. There is a termination condition defined in the function which is satisfied by some specific subtask. After this, the recursion stops and the final result is returned from the function.

The case at which the function doesn't recur is called the base case whereas the instances where the function keeps calling itself to perform a subtask, is called the recursive case. All the recursive functions can be written using this format.

Pseudocode for writing any recursive function is given below.

```
if (test_for_base)
{
    return some_value;
}
else if (test_for_another_base)
{
    return some_another_value;
```

```
}
else
{
   // Statements;
   recursive call;
}
```

Example of recursion in C

```
#include<stdio.h>
int fibonacci(int);
void main ()
{
   int n,f;
   printf("Enter the value of n?");
   scanf("%d",&n);
   f = fibonacci(n);
   printf("%d",f);
}
int fibonacci (int n)
{
   if (n==0)
   {
   return 0;
   }
   else if (n == 1)
   {
      return 1;
   }
   else
   {
      return fibonacci(n-1)+fibonacci(n-2);
   }
}
```

```
Enter the value of n?12
144
```

Memory allocation of Recursive method

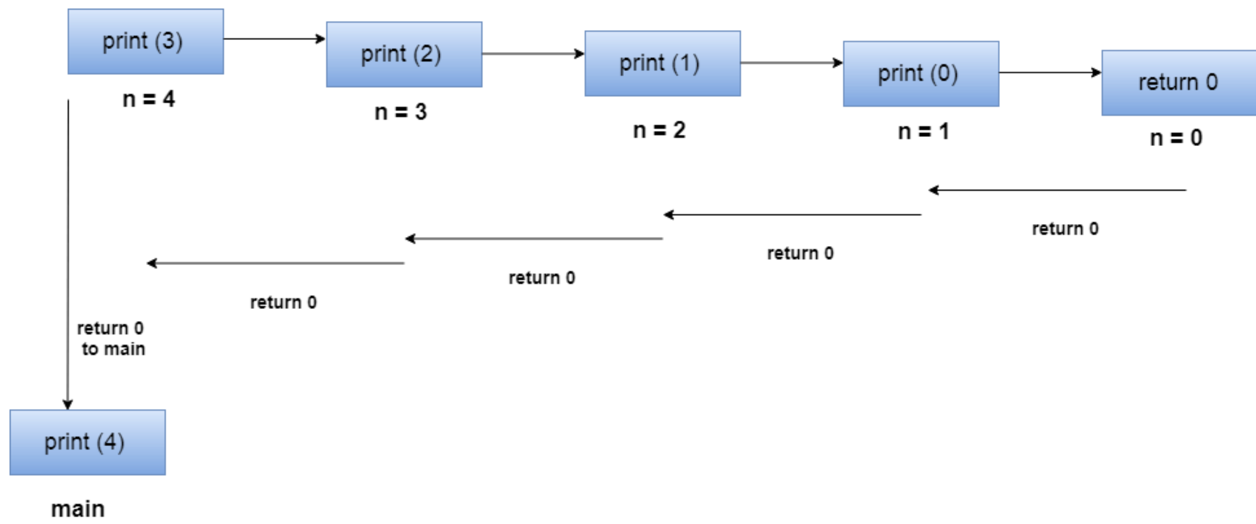
Each recursive call creates a new copy of that method in the memory. Once some data is returned by the method, the copy is removed from the memory. Since all the variables and other stuff declared inside function get stored in the stack, therefore a separate stack is maintained at each recursive call. Once the value is returned from the corresponding function, the stack gets destroyed. Recursion involves so much complexity in resolving and tracking the values at each recursive call. Therefore we need to maintain the stack and track the values of the variables defined in the stack.

Let us consider the following example to understand the memory allocation of the recursive functions.

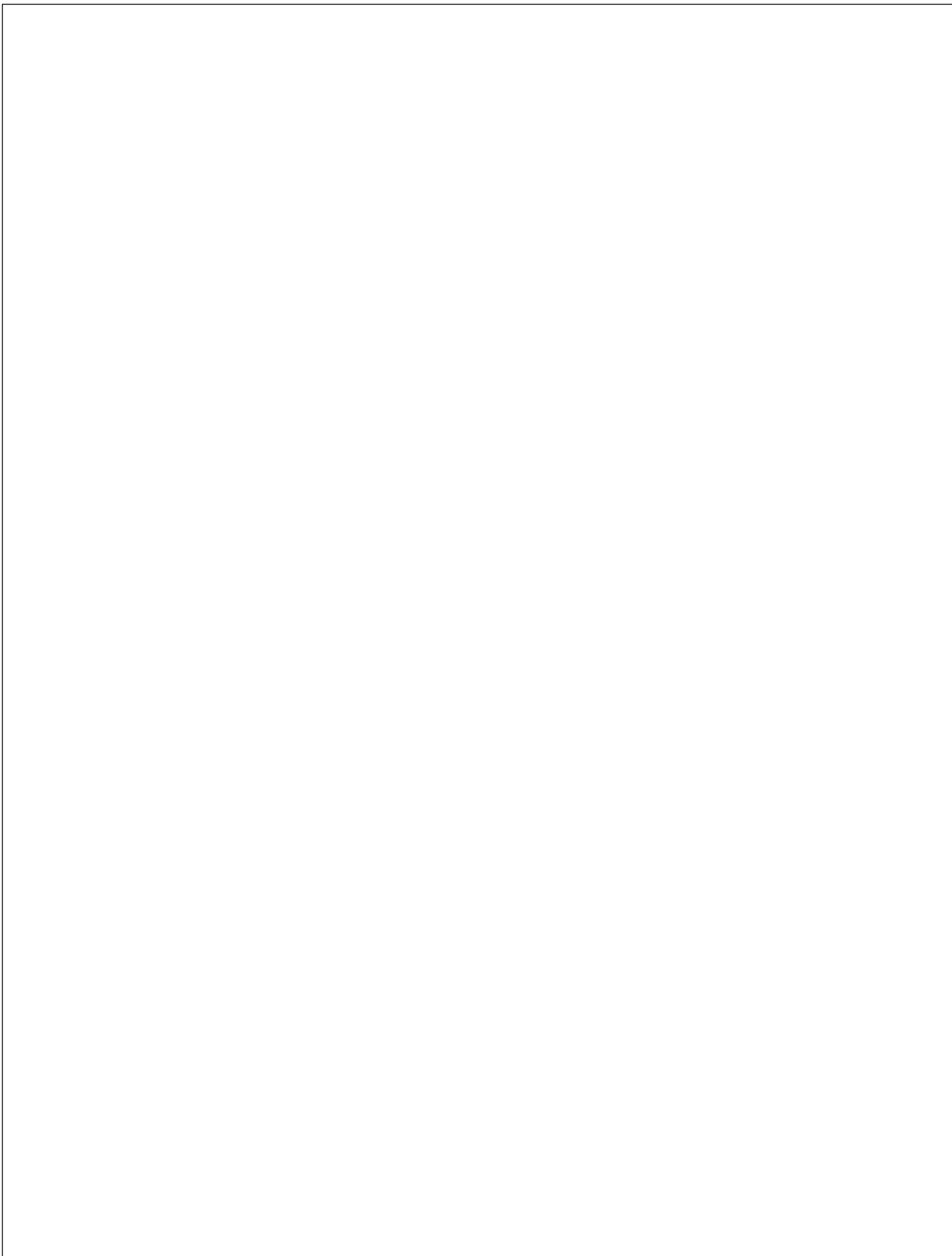
```
int display (int n)
{
   if(n == 0)
      return 0; // terminating condition
   else
   {
      printf("%d",n);
      return display(n-1); // recursive call
   }
}
```

Explanation

Let us examine this recursive function for n = 4. First, all the stacks are maintained which prints the corresponding value of n until n becomes 0, Once the termination condition is reached, the stacks get destroyed one by one by returning 0 to its calling stack. Consider the following image for more information regarding the stack trace for the recursive functions.



Stack tracing for recursive function call



Storage Classes in C

Storage classes in C are used to determine the lifetime, visibility, memory location, and initial value of a variable. There are four types of storage classes in C

- Automatic
- External
- Static
- Register

Storage Classes	**Storage Place**	**Default Value**	**Scope**	**Lifetime**
Auto	RAM	Garbage Value	Local	Within function
Extern	RAM	Zero	Global	Till the end of the main program Maybe declared anywhere in the program
Static	RAM	Zero	Local	Till the end of the main program, Retains value between multiple functions call
Register	Register	Garbage Value	Local	Within the function

Automatic

- Automatic variables are allocated memory automatically at runtime.
- The visibility of the automatic variables is limited to the block in which they are defined.

  The scope of the automatic variables is limited to the block in which they are defined.
- The automatic variables are initialized to garbage by default.
- The memory assigned to automatic variables gets freed upon exiting from the block.
- The keyword used for defining automatic variables is auto.
- Every local variable is automatic in C by default.

Example 1

```
#include <stdio.h>
int main()
{
int a; //auto
char b;
float c;
```

```
printf("%d %c %f",a,b,c); // printing initial default value of automatic variables a, b, and c.
return 0;
}
```

Output:

```
garbage garbage garbage
```

Example 2

```
#include <stdio.h>
int main()
{
int a = 10,i;
printf("%d ",++a);
{
int a = 20;
for (i=0;i<3;i++)
{
printf("%d ",a); // 20 will be printed 3 times since it is the local value of a
}
}
printf("%d ",a); // 11 will be printed since the scope of a = 20 is ended.
}
```

Output:

```
11 20 20 20 11
```

Static

- The variables defined as static specifier can hold their value between the multiple function calls.
- Static local variables are visible only to the function or the block in which they are defined.
- A same static variable can be declared many times but can be assigned at only one time.
- Default initial value of the static integral variable is 0 otherwise null.
- The visibility of the static global variable is limited to the file in which it has declared.
- The keyword used to define static variable is static.

Example 1

```
#include<stdio.h>
static char c;
static int i;
static float f;
static char s[100];
void main ()
{
printf("%d %d %f %s",c,i,f); // the initial default value of c, i, and f will be printed.
}
```

Output:

```
0 0 0.000000 (null)
```

Example 2

```
#include<stdio.h>
void sum()
{
static int a = 10;
static int b = 24;
printf("%d %d \n",a,b);
a++;
b++;
}
void main()
{
int i;
for(i = 0; i< 3; i++)
{
sum(); // The static variables holds their value between multiple function calls.
}
}
```

Output:

```
10 24
11 25
12 26
```

Register

- The variables defined as the register is allocated the memory into the CPU registers depending upon the size of the memory remaining in the CPU.
- We can not dereference the register variables, i.e., we can not use &operator for the register variable.

- The access time of the register variables is faster than the automatic variables.
- The initial default value of the register local variables is 0.
- The register keyword is used for the variable which should be stored in the CPU register. However, it is compiler?s choice whether or not; the variables can be stored in the register.
- We can store pointers into the register, i.e., a register can store the address of a variable.
- Static variables can not be stored into the register since we can not use more than one storage specifier for the same variable.

Example 1

```
#include <stdio.h>
int main()
{
register int a; // variable a is allocated memory in the CPU register. The initial default vale of a is 0.
printf("%d",a);
}
```

Output:

```
0
```

Example 2

```
#include <stdio.h>
int main()
{
register int a = 0;
printf("%u",&a); // This will give a compile time error since we can not access the address of a register variable.
}
```

Output:

```
main.c:5:5: error: address of register variable ?a? requested
printf("%u",&a);
^~~~~~
```

External

- The external storage class is used to tell the compiler that the variable defined as extern is declared with an external linkage elsewhere in the program.

- The variables declared as extern are not allocated any memory. It is only declaration and intended to specify that the variable is declared elsewhere in the program.
- The default initial value of external integral type is 0 otherwise null.
- We can only initialize the extern variable globally, i.e., we can not initialize the external variable within any block or method.
- An external variable can be declared many times but can be initialized at only once.
- If a variable is declared as external then the compiler searches for that variable to be initialized somewhere in the program which may be extern or static. If it is not, then the compiler will show an error.

Example 1

```
#include <stdio.h>
int main()
{
extern int a;
printf("%d",a);
}
```

Output

```
main.c:(.text+0x6): undefined reference to `a'
collect2: error: ld returned 1 exit status
```

Example 2

```
#include <stdio.h>
int a;
int main()
{
extern int a; // variable a is defined globally, the memory will not be allocated to a
printf("%d",a);
}
```

Output

```
0
```

Example 3

```
#include <stdio.h>
int a;
int main()
{
extern int a = 0; // this will show a compiler error since we can not use extern and initializ
er at same time
printf("%d",a);
}
```

Output

```
compile time error
main.c: In function ?main?:
main.c:5:16: error: ?a? has both ?extern? and initializer
extern int a = 0;
```

Example 4

```
#include <stdio.h>
int main()
{
extern int a; // Compiler will search here for a variable a defined and initialized somewhere
 in the pogram or not.
printf("%d",a);

int a = 20;
```

Output

```
20
```

Example 5

```
extern int a;
int a = 10;
#include <stdio.h>
int main()
{
printf("%d",a);
}
int a = 20; // compiler will show an error at this line
```

Output

```
compile time error
```

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

1) Code Optimization: Less code to the access the data.

2) Ease of traversing: By using the for loop, we can retrieve the elements of an array easily.

3) Ease of sorting: To sort the elements of the array, we need a few lines of code only.

4) Random Access: We can access any element randomly using the array.

Disadvantage of C Array

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

Declaration of C Array

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;
```

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

Initialization of Array

C array example

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

C Array Example: Sorting an array

In the following program, we are using bubble sort method to sort the array in ascending order.

```
#include<stdio.h>
void main ()
{
   int i, j,temp;
   int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
   for(i = 0; i<10; i++)
   {
      for(j = i+1; j<10; j++)
      {
         if(a[j] > a[i])
         {
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
         }
      }
   }
   printf("Printing Sorted Element List ...\n");
   for(i = 0; i<10; i++)
   {
      printf("%d\n",a[i]);
   }
}
```

Program to print the largest and second largest element of the array.

```
#include<stdio.h>
void main ()
{
   int arr[100],i,n,largest,sec_largest;
   printf("Enter the size of the array?");
   scanf("%d",&n);
   printf("Enter the elements of the array?");
   for(i = 0; i<n; i++)
```

```
    {
        scanf("%d",&arr[i]);
    }
    largest = arr[0];
    sec_largest = arr[1];
    for(i=0;i<n;i++)
    {
        if(arr[i]>largest)
        {
            sec_largest = largest;
            largest = arr[i];
        }
        else if (arr[i]>sec_largest && arr[i]!=largest)
        {
            sec_largest=arr[i];
        }
    }
    printf("largest = %d, second largest = %d",largest,sec_largest);

}
```

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
```

C 2D array example: Storing elements in a matrix and printing it.

```
#include <stdio.h>
void main ()
{
   int arr[3][3],i,j;
   for (i=0;i<3;i++)
   {
      for (j=0;j<3;j++)
      {
         printf("Enter a[%d][%d]: ",i,j);
         scanf("%d",&arr[i][j]);
      }
   }
   printf("\n printing the elements ....\n");
   for(i=0;i<3;i++)
   {
      printf("\n");
      for (j=0;j<3;j++)
      {
         printf("%d\t",arr[i][j]);
      }
   }
}
```

Output

```
Enter a[0][0]: 56
Enter a[0][1]: 10
Enter a[0][2]: 30
Enter a[1][0]: 34
```

```
Enter a[1][1]: 21
Enter a[1][2]: 34

Enter a[2][0]: 45
Enter a[2][1]: 56
Enter a[2][2]: 78

 printing the elements ....

56      10      30
34      21      34
45      56      78
```

Passing Array to Function in C

In C, there are various general problems which requires passing more than one variable of the same type to a function. For example, consider a function which sorts the 10 elements in ascending order. Such a function requires 10 numbers to be passed as the actual parameters from the main function. Here, instead of declaring 10 different numbers and then passing into the function, we can declare and initialize an array and pass that into the function. This will resolve all the complexity since the function will now work for any number of values.

As we know that the array_name contains the address of the first element. Here, we must notice that we need to pass only the name of the array in the function which is intended to accept an array. The array defined as the formal parameter will automatically refer to the array specified by the array name defined as an actual parameter.

Consider the following syntax to pass an array to the function.

```
functionname(arrayname);//passing array
```

Methods to declare a function that receives an array as an argument

There are 3 ways to declare the function which is intended to receive an array as an argument.

First way:

```
return_type function(type arrayname[])
```

Declaring blank subscript notation [] is the widely used technique.

Second way:

```
return_type function(type arrayname[SIZE])
```

Optionally, we can define size in subscript notation [].

Third way:

```
return_type function(type *arrayname)
```

You can also use the concept of a pointer. In pointer chapter, we will learn about it.

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 function to sort the array

```
#include<stdio.h>
void Bubble_Sort(int[]);
void main ()
{
   int arr[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
   Bubble_Sort(arr);
}
void Bubble_Sort(int a[]) //array a[] points to arr.
{
int i, j,temp;
   for(i = 0; i<10; i++)
   {
      for(j = i+1; j<10; j++)
      {
```

```
            if(a[j] < a[i])
            {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
    printf("Printing Sorted Element List ...\n");
    for(i = 0; i<10; i++)
    {
        printf("%d\n",a[i]);
    }
}
```

Output

```
Printing Sorted Element List ...
7
9
10
12
23
23
34
44
78
101
```

Returning array from the function

As we know that, a function can not return more than one value. However, if we try to write the return statement as return a, b, c; to return three values (a,b,c), the function will return the last mentioned value which is c in our case. In some problems, we may need to return multiple values from a function. In such cases, an array is returned from the function.

Returning an array is similar to passing the array into the function. The name of the array is returned from the function. To make a function returning an array, the following syntax is used.

```
int * Function_name() {
//some statements;
return array_type;
}
```

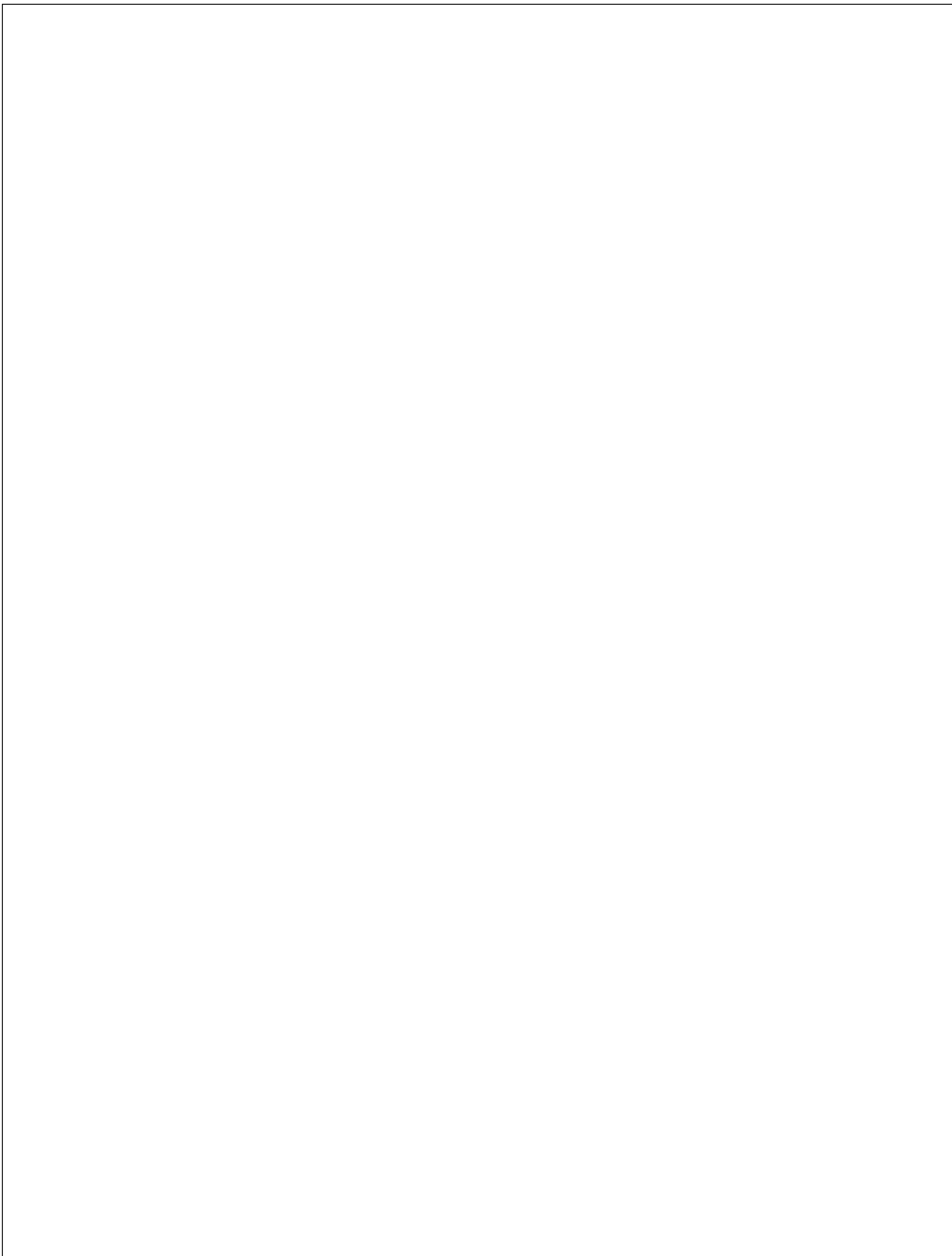
To store the array returned from the function, we can define a pointer which points to that array. We can traverse the array by increasing that pointer since pointer initially points to the base address of the array. Consider the following example that contains a function returning the sorted array.

```
#include<stdio.h>
int* Bubble_Sort(int[]);
void main ()
{
    int arr[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
    int *p = Bubble_Sort(arr), i;
    printf("printing sorted elements ...\n");
    for(i=0;i<10;i++)
    {
        printf("%d\n",*(p+i));
    }
}
int* Bubble_Sort(int a[]) //array a[] points to arr.
{
int i, j,temp;
    for(i = 0; i<10; i++)
    {
        for(j = i+1; j<10; j++)
        {
            if(a[j] < a[i])
            {
                temp = a[i];
```

```
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
    return a;
}
```

Output

```
Printing Sorted Element List ...
7
9
10
12
23
23
34
44
78
101
```



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.
```

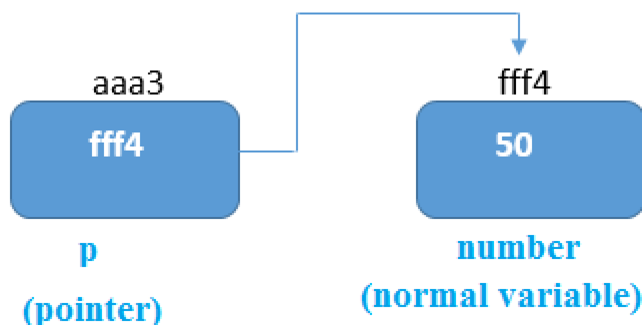
Declaring a pointer

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 poin
ter therefore if we print *p, we will get the value stored at the address contained by p.
return 0;
}
```

Output

```
Address of number variable is fff4
Address of p variable is fff4
Value of p variable is 50
```

Pointer to array

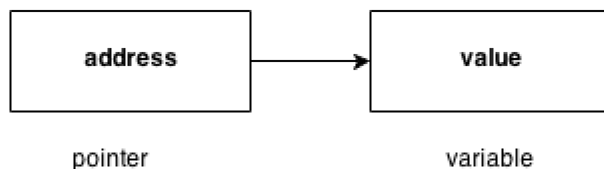
```
int arr[10];
int *p[10]=&arr; // Variable p of type pointer is pointing to the address of an integer array
arr.
```

Pointer to a function

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

Pointer to structure

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



Advantage of pointer

1) Pointer **reduces the code** and **improves the performance**, it is used to retrieving strings, trees, etc. and used with arrays, structures, and functions.

2) We can **return multiple values from a function** using the pointer.

3) It makes you able to **access any memory location** in the computer's memory.

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
```

Reading complex pointers

There are several things which must be taken into the consideration while reading the complex pointers in C. Lets see the precedence and associativity of the operators which are used regarding pointers.

Operator	Precedence	Associativity
(), []	1	Left to right
*, identifier	2	Right to left
Data type	3	-

Here,we must notice that,

- (): This operator is a bracket operator used to declare and define the function.
- []: This operator is an array subscript operator
- * : This operator is a pointer operator.
- Identifier: It is the name of the pointer. The priority will always be assigned to this.
- Data type: Data type is the type of the variable to which the pointer is intended to point. It also includes the modifier like signed int, long, etc).

How to read the pointer: int (*p)[10].

To read the pointer, we must see that () and [] have the equal precedence. Therefore, their associativity must be considered here. The associativity is left to right, so the priority goes to ().

Inside the bracket (), pointer operator * and pointer name (identifier) p have the same precedence. Therefore, their associativity must be considered here which is right to left, so the priority goes to p, and the second priority goes to *.

Assign the 3rd priority to [] since the data type has the last precedence. Therefore the pointer will look like following.

- char -> 4
- * -> 2
- p -> 1
- [10] -> 3

The pointer will be read as p is a pointer to an array of integers of size 10.

Example

How to read the following pointer?

```
int (*p)(int (*)[2], int (*)void))
```

Explanation

This pointer will be read as p is a pointer to such function which accepts the first parameter as the pointer to a one-dimensional array of integers of size two and the second parameter as the pointer to a function which parameter is void and return type is the integer.

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.

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.

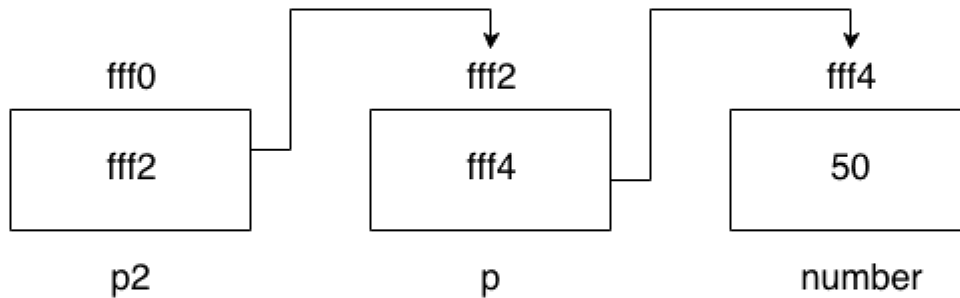
```
#include<stdio.h>
void main ()
{
    int a = 10;
    int *p;
    int **pp;
    p = &a; // pointer p is pointing to the address of a
    pp = &p; // pointer pp is a double pointer pointing to the address of pointer p
    printf("address of a: %x\n",p); // Address of a will be printed
    printf("address of p: %x\n",pp); // Address of p will be printed
    printf("value stored at p: %d\n",*p); // value stoted at the address contained by p i.e. 1
0 will be printed
    printf("value stored at pp: %d\n",**pp); // value stored at the address contained by the
pointer stoyred at pp
}
```

Output

```
address of a: d26a8734
address of p: d26a8738
value stored at p: 10
value stored at pp: 10
```

C double pointer example

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



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

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

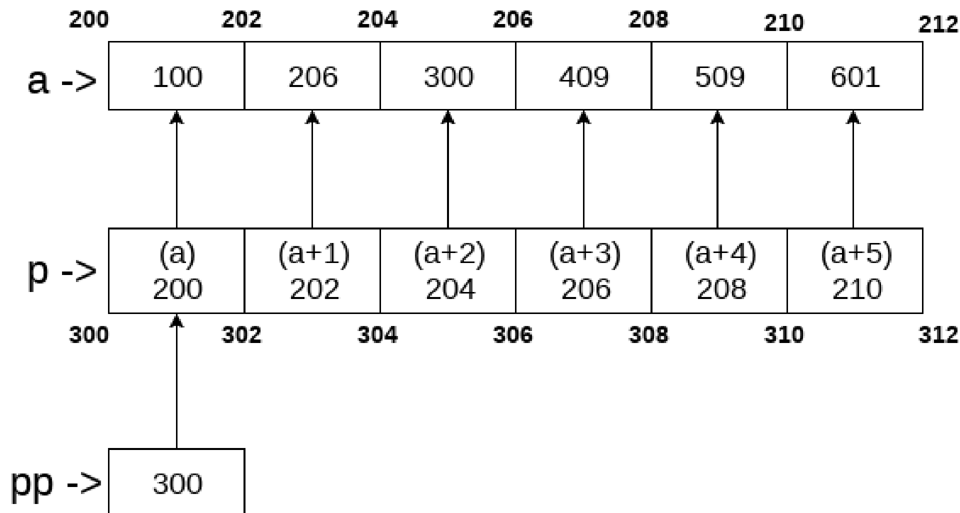
Output

```
Address of number variable is fff4
Address of p variable is fff4
Value of *p variable is 50
Address of p2 variable is fff2
Value of **p variable is 50
```

Q. What will be the output of the following program?

```
#include<stdio.h>
void main ()
{
    int a[10] = {100, 206, 300, 409, 509, 601}; //Line 1
    int *p[] = {a, a+1, a+2, a+3, a+4, a+5}; //Line 2
    int **pp = p; //Line 3
    pp++; // Line 4
    printf("%d %d %d\n",pp-p,*pp - a,**pp); // Line 5
    *pp++; // Line 6
    printf("%d %d %d\n",pp-p,*pp - a,**pp); // Line 7
    ++*pp; // Line 8
    printf("%d %d %d\n",pp-p,*pp - a,**pp); // Line 9
    ++**pp; // Line 10
    printf("%d %d %d\n",pp-p,*pp - a,**pp); // Line 11
}
```

Explanation



To access a[0] ⟶ a[0] = * (a) = *p[0] = **(p+0) = **(pp+0) = 100

In the above question, the pointer arithmetic is used with the double pointer. An array of 6 elements is defined which is pointed by an array of pointer p. The pointer array p is pointed by a double pointer pp. However, the above image gives you a brief idea about how the memory is being allocated to the array a and the pointer array p. The elements of p are the pointers that are pointing to every element of the array a. Since we know that the array

name contains the base address of the array hence, it will work as a pointer and can the value can be traversed by using *(a), *(a+1), etc. As shown in the image, a[0] can be accessed in the following ways.

- a[0]: it is the simplest way to access the first element of the array
- *(a): since a store the address of the first element of the array, we can access its value by using indirection pointer on it.
- *p[0]: if a[0] is to be accessed by using a pointer p to it, then we can use indirection operator (*) on the first element of the pointer array p, i.e., *p[0].
- **(pp): as pp stores the base address of the pointer array, *pp will give the value of the first element of the pointer array that is the address of the first element of the integer array. **p will give the actual value of the first element of the integer array.

Coming to the program, Line 1 and 2 declare the integer and pointer array relatively. Line 3 initializes the double pointer to the pointer array p. As shown in the image, if the address of the array starts from 200 and the size of the integer is 2, then the pointer array will contain the values as 200, 202, 204, 206, 208, 210. Let us consider that the base address of the pointer array is 300; the double pointer pp contains the address of pointer array, i.e., 300. Line number 4 increases the value of pp by 1, i.e., pp will now point to address 302.

Line number 5 contains an expression which prints three values, i.e., pp - p, *pp - a, **pp. Let's calculate them each one of them.

- pp = 302, p = 300 => pp-p = (302-300)/2 => pp-p = 1, i.e., 1 will be printed.
- pp = 302, *pp = 202, a = 200 => *pp - a = 202 - 200 = 2/2 = 1, i.e., 1 will be printed.
- pp = 302, *pp = 202, *(*pp) = 206, i.e., 206 will be printed.

Therefore as the result of line 5, The output 1, 1, 206 will be printed on the console. On line 6, *pp++ is written. Here, we must notice that two unary operators * and ++ will have the same precedence. Therefore, by the rule of associativity, it will be evaluated from right to left. Therefore the expression *pp++ can be rewritten as (*(pp++)). Since, pp = 302 which will now become, 304. *pp will give 204.

On line 7, again the expression is written which prints three values, i.e., pp-p, *pp-a, *pp. Let's calculate each one of them.

- pp = 304, p = 300 => pp - p = (304 - 300)/2 => pp-p = 2, i.e., 2 will be printed.
- pp = 304, *pp = 204, a = 200 => *pp-a = (204 - 200)/2 = 2, i.e., 2 will be printed.
- pp = 304, *pp = 204, *(*pp) = 300, i.e., 300 will be printed.

Therefore, as the result of line 7, The output 2, 2, 300 will be printed on the console. On line 8, ++*pp is written. According to the rule of associativity, this can be rewritten as, (++(*(pp))). Since, pp = 304, *pp = 204, the value of *pp = *(p[2]) = 206 which will now point to a[3].

On line 9, again the expression is written which prints three values, i.e., pp-p, *pp-a, *pp. Let's calculate each one of them.

- pp = 304, p = 300 => pp - p = (304 - 300)/2 => pp-p = 2, i.e., 2 will be printed.
- pp = 304, *pp = 206, a = 200 => *pp-a = (206 - 200)/2 = 3, i.e., 3 will be printed.
- pp = 304, *pp = 206, *(*pp) = 409, i.e., 409 will be printed.

Therefore, as the result of line 9, the output 2, 3, 409 will be printed on the console. On line 10, ++**pp is writen. according to the rule of associativity, this can be rewritten as, (++(*(*(pp)))). pp = 304, *pp = 206, **pp = 409, ++**pp => *pp = *pp + 1 = 410. In other words, a[3] = 410.

On line 11, again the expression is written which prints three values, i.e., pp-p, *pp-a, *pp. Let's calculate each one of them.

- pp = 304, p = 300 => pp - p = (304 - 300)/2 => pp-p = 2, i.e., 2 will be printed.
- pp = 304, *pp = 206, a = 200 => *pp-a = (206 - 200)/2 = 3, i.e., 3 will be printed.
- On line 8, **pp = 410.

Therefore as the result of line 9, the output 2, 3, 410 will be printed on the console.

At last, the output of the complete program will be given as:

Output

```
1 1 206
2 2 300
2 3 409
2 3 410
```

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
- Comparison

Incrementing Pointer in C

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

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

The Rule to increment the pointer is given below:

new_address= current_address + i * size_of(data type)

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

32-bit

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

64-bit

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

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

```
#include<stdio.h>
int main(){
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p+1;
printf("After increment: Address of p variable is %u \n",p); // in our case, p will get increme
nted 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:

new_address= current_address - i * size_of(data type)

32-bit

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

64-bit

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

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

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

new_address= current_address + (number * size_of(data type))

32-bit

For 32-bit int variable, it will add 2 * number.

64-bit

For 64-bit int variable, it will add 4 * number.

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

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

```
p=p+3;   //adding 3 to pointer variable
printf("After adding 3: Address of p variable is %u \n",p);
return 0;
}
```

Output

```
Address of p variable is 3214864300
After adding 3: Address of p variable is 3214864312
```

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

C Pointer Subtraction

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

new_address= current_address - (number * size_of(data type))

32-bit

For 32-bit int variable, it will subtract 2 * number.

64-bit

For 64-bit int variable, it will subtract 4 * 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,

Address2 -
Address1 = (Subtraction of two addresses)/size of data type which pointer points

Consider the following example to subtract one pointer from an another.

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

Output

```
Pointer Subtraction: 1030585080 - 1030585068 = 3
```

Illegal arithmetic with pointers

There are various operations which can not be performed on pointers. Since, pointer stores address hence we must ignore the operations which may lead to an illegal address, for example, addition, and multiplication. A list of such operations is given below.

- Address + Address = illegal
- Address * Address = illegal
- Address % Address = illegal
- Address / Address = illegal
- Address & Address = illegal
- Address ^ Address = illegal
- Address | Address = illegal

- ~Address = illegal

Pointer to function in C

As we discussed in the previous chapter, a pointer can point to a function in C. However, the declaration of the pointer variable must be the same as the function. Consider the following example to make a pointer pointing to the function.

```
#include<stdio.h>
int addition ();
int main ()
{
    int result;
    int (*ptr)();
    ptr = &addition;
    result = (*ptr)();
    printf("The sum is %d",result);
}
int addition()
{
    int a, b;
    printf("Enter two numbers?");
    scanf("%d %d",&a,&b);
    return a+b;
}
```

Output

```
Enter two numbers?10 15
The sum is 25
```

Pointer to Array of functions in C

To understand the concept of an array of functions, we must understand the array of function. Basically, an array of the function is an array which contains the addresses of functions. In other words, the pointer to an array of functions is a pointer pointing to an array which contains the pointers to the functions. Consider the following example

```
#include<stdio.h>
int show();
int showadd(int);
int (*arr[3])();
int (*(*ptr)[3])();

int main ()
{
   int result1;
   arr[0] = show;
   arr[1] = showadd;
   ptr = &arr;
   result1 = (**ptr)();
   printf("printing the value returned by show : %d",result1);
   (*(*ptr+1))(result1);
}
int show()
{
   int a = 65;
   return a++;
}
int showadd(int b)
{
   printf("\nAdding 90 to the value returned by show: %d",b+90);
}
```

Output

```
printing the value returned by show : 65
Adding 90 to the value returned by show: 155
```

Dynamic memory allocation in C

The concept of **dynamic memory allocation in c language** *enables the C programmer to allocate memory at runtime*. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

1. malloc()
2. calloc()
3. realloc()
4. free()

Before learning above functions, let's understand the difference between static memory allocation and dynamic memory allocation.

tatic memory allocation	dynamic memory allocation
memory is allocated at compile time.	memory is allocated at run time.
memory can't be increased while executing program.	memory can be increased while executing program.
used in array.	used in linked list.

Now let's have a quick look at the methods used for dynamic memory allocation.

malloc()	allocates single block of requested memory.
calloc()	allocates multiple block of requested memory.
realloc()	reallocates the memory occupied by malloc() or calloc() functions.
free()	frees the dynamically allocated memory.

malloc() function in C

The malloc() function allocates single block of requested memory.

It doesn't initialize memory at execution time, so it has garbage value initially.

It returns NULL if memory is not sufficient.

The syntax of malloc() function is given below:

ptr=(cast-type*)malloc(byte-size)

Let's see the example of malloc() function.

```
#include<stdio.h>
#include<stdlib.h>
int main(){
  int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc
    if(ptr==NULL)
    {
        printf("Sorry! unable to allocate memory");
        exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {
        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
return 0;
}
```

Output

```
Enter elements of array: 3
Enter elements of array: 10
10
10
Sum=30
```

calloc() function in C

The calloc() function allocates multiple block of requested memory.

It initially initialize all bytes to zero.

It returns NULL if memory is not sufficient.

The syntax of calloc() function is given below:

ptr=(cast-type*)calloc(number, byte-size)

Let's see the example of calloc() function.

```
#include<stdio.h>
#include<stdlib.h>
int main(){
 int n,i,*ptr,sum=0;
   printf("Enter number of elements: ");
   scanf("%d",&n);
   ptr=(int*)calloc(n,sizeof(int));  //memory allocated using calloc
   if(ptr==NULL)
   {
      printf("Sorry! unable to allocate memory");
      exit(0);
   }
   printf("Enter elements of array: ");
   for(i=0;i<n;++i)
   {
      scanf("%d",ptr+i);
      sum+=*(ptr+i);
   }
   printf("Sum=%d",sum);
   free(ptr);
return 0;
}
```

Output:

```
Enter elements of array: 3
Enter elements of array: 10
10
10
Sum=30
```

realloc() function in C

If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.

Let's see the syntax of realloc() function.

```
ptr=realloc(ptr, new-size)
```

free() function in C

The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.

Let's see the syntax of free() function.

```
free(ptr)
```

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-
dimensioanal 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 ca; simulate the use of classes and templates as it can store various information

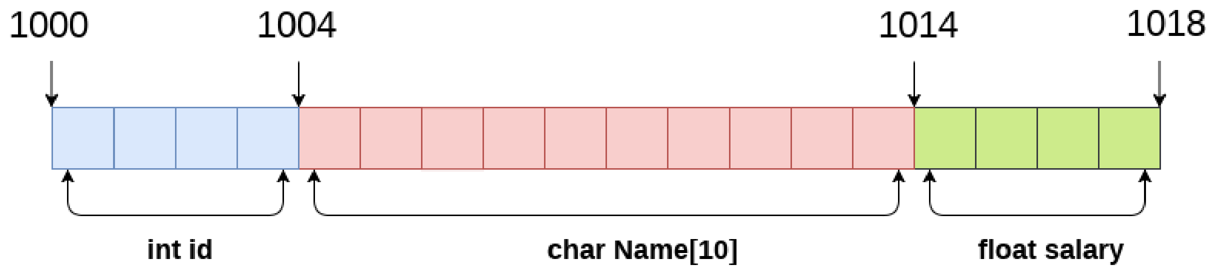
The **,struct** keyword is used to define the structure. Let's see the syntax to define the structure in c.

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

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

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

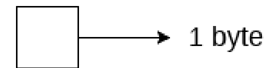
The following image shows the memory allocation of the structure employee that is defined in the above example.



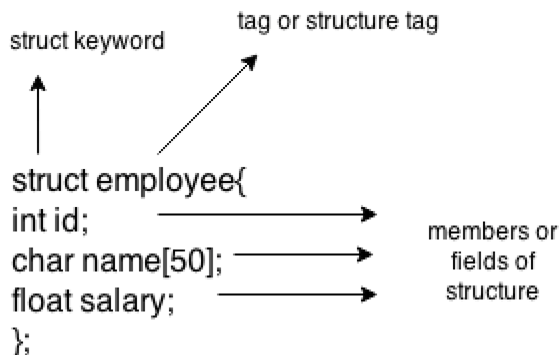

```
struct Employee
{
    int id;
    char Name[10];
    float salary;
} emp;
```

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

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



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:



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

Accessing members of the structure

There are two ways to access structure members:

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

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

p1.id

C Structure example

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

```
#include<stdio.h>
#include <string.h>
struct employee
{   int id;
    char name[50];
}e1;  //declaring e1 variable for structure
int main( )
{
   //store first employee information
   e1.id=101;
   strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
   //printing first employee information
   printf( "employee 1 id : %d\n", e1.id);
   printf( "employee 1 name : %s\n", e1.name);
return 0;
}
```

Output:

```
employee 1 id : 101
employee 1 name : Sonoo Jaiswal
```

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
```

C Array of Structures

Why use an array of structures?

Consider a case, where we need to store the data of 5 students. We can store it by using the structure as given below.

```
#include<stdio.h>
struct student
{
   char name[20];
   int id;
   float marks;
};
void main()
{
   struct student s1,s2,s3;
   int dummy;
   printf("Enter the name, id, and marks of student 1 ");
   scanf("%s %d %f",s1.name,&s1.id,&s1.marks);
   scanf("%c",&dummy);
   printf("Enter the name, id, and marks of student 2 ");
   scanf("%s %d %f",s2.name,&s2.id,&s2.marks);
   scanf("%c",&dummy);
   printf("Enter the name, id, and marks of student 3 ");
   scanf("%s %d %f",s3.name,&s3.id,&s3.marks);
   scanf("%c",&dummy);
   printf("Printing the details....\n");
   printf("%s %d %f\n",s1.name,s1.id,s1.marks);
   printf("%s %d %f\n",s2.name,s2.id,s2.marks);
   printf("%s %d %f\n",s3.name,s3.id,s3.marks);
}
```

Output

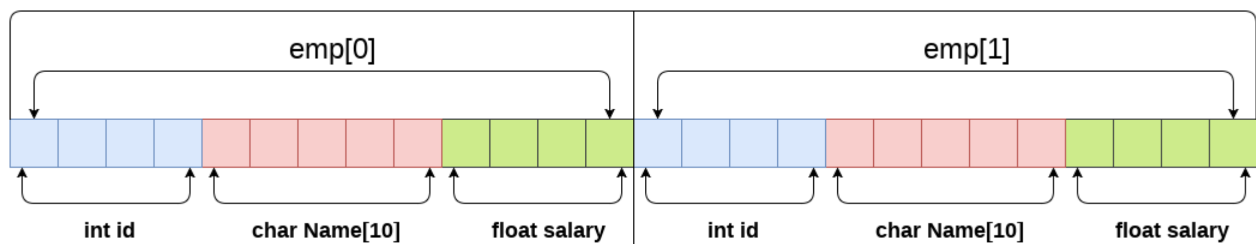
```
Enter the name, id, and marks of student 1 James 90 90
Enter the name, id, and marks of student 2 Adoms 90 90
Enter the name, id, and marks of student 3 Nick 90 90
Printing the details....
James 90 90.000000
Adoms 90 90.000000
Nick 90 90.000000
```

In the above program, we have stored data of 3 students in the structure. However, the complexity of the program will be increased if there are 20 students. In that case, we will have to declare 20 different structure variables and store them one by one. This will always be tough since we will have to declare a variable every time we add a student. Remembering the name of all the variables is also a very tricky task. However, c enables us to declare an array of structures by using which, we can avoid declaring the different structure variables; instead we can make a collection containing all the structures that store the information of different entities.

Array of Structures in C

An array of structres in C can be defined as the collection of multiple structures variables where each variable contains information about different entities. The array of structures in C are used to store information about multiple entities of different data types. The array of structures is also known as the collection of structures.

Array of structures




```
struct employee
{
    int id;
    char name[5];
    float salary;
};
struct employee emp[2];
```

sizeof (emp) = 4 + 5 + 4 = 13 bytes

sizeof (emp[2]) = 26 bytes

Let's see an example of an array of structures that stores information of 5 students and prints it.

```
#include<stdio.h>
#include <string.h>
struct student{
int rollno;
char name[10];
};
int main(){
int i;
```

```c
struct student st[5];
printf("Enter Records of 5 students");
for(i=0;i<5;i++){
printf("\nEnter Rollno:");
scanf("%d",&st[i].rollno);
printf("\nEnter Name:");
scanf("%s",&st[i].name);
}
printf("\nStudent Information List:");
for(i=0;i<5;i++){
printf("\nRollno:%d, Name:%s",st[i].rollno,st[i].name);
}
  return 0;
}
```

Output:

```
Enter Records of 5 students
Enter Rollno:1
Enter Name:Sonoo
Enter Rollno:2
Enter Name:Ratan
Enter Rollno:3
Enter Name:Vimal
Enter Rollno:4
Enter Name:James
Enter Rollno:5
Enter Name:Sarfraz

Student Information List:
Rollno:1, Name:Sonoo
Rollno:2, Name:Ratan
Rollno:3, Name:Vimal
Rollno:4, Name:James
Rollno:5, Name:Sarfraz
```

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.

```
#include<stdio.h>
struct address
{
   char city[20];
   int pin;
   char phone[14];
};
struct employee
{
   char name[20];
   struct address add;
};
void main ()
{
   struct employee emp;
   printf("Enter employee information?\n");
   scanf("%s %s %d %s",emp.name,emp.add.city, &emp.add.pin, emp.add.phone);
   printf("Printing the employee information....\n");
   printf("name: %s\nCity: %s\nPincode: %d\nPhone: %s",emp.name,emp.add.city,emp.a
dd.pin,emp.add.phone);
}
```

Output

```
Enter employee information?

Arun

Delhi

110001

1234567890

Printing the employee information....

name: Arun
```

```
City: Delhi

Pincode: 110001

Phone: 1234567890
```

The structure can be nested in the following ways.

1. By separate structure
2. By Embedded structure

1) Separate structure

Here, we create two structures, but the dependent structure should be used inside the main structure as a member. Consider the following example.

```
struct Date
{
  int dd;
  int mm;
  int yyyy;
};
struct Employee
{
  int id;
  char name[20];
  struct Date doj;
}emp1;
```

As you can see, doj (date of joining) is the variable of type Date. Here doj is used as a member in Employee structure. In this way, we can use Date structure in many structures.

2) Embedded structure

The embedded structure enables us to declare the structure inside the structure. Hence, it requires less line of codes but it can not be used in multiple data structures. Consider the following example.

```
struct Employee
{
  int id;
  char name[20];
  struct Date
   {
```

```
    int dd;
    int mm;
    int yyyy;
   }doj;
}emp1;
```

Accessing Nested Structure

We can access the member of the nested structure by Outer_Structure.Nested_Structure.member as given below:

```
e1.doj.dd
e1.doj.mm
e1.doj.yyyy
```

C Nested Structure example

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

```
#include <stdio.h>
#include <string.h>
struct Employee
{
  int id;
  char name[20];
  struct Date
   {
    int dd;
    int mm;
    int yyyy;
   }doj;
}e1;
int main( )
{
  //storing employee information
  e1.id=101;
  strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
  e1.doj.dd=10;
  e1.doj.mm=11;
  e1.doj.yyyy=2014;
```

```
  //printing first employee information
  printf( "employee id : %d\n", e1.id);
  printf( "employee name : %s\n", e1.name);
  printf( "employee date of joining (dd/mm/yyyy) : %d/%d/%d\n", e1.doj.dd,e1.doj.mm,e1.doj.yyyy);
  return 0;
}
```

Output:

```
employee id : 101
employee name : Sonoo Jaiswal
employee date of joining (dd/mm/yyyy) : 10/11/2014
```

Passing structure to function

Just like other variables, a structure can also be passed to a function. We may pass the structure members into the function or pass the structure variable at once. Consider the following example to pass the structure variable employee to a function display() which is used to display the details of an employee.

```
#include<stdio.h>
struct address
{
   char city[20];
   int pin;
   char phone[14];
};
struct employee
{
   char name[20];
   struct address add;
};
void display(struct employee);
void main ()
{
   struct employee emp;
   printf("Enter employee information?\n");
   scanf("%s %s %d %s",emp.name,emp.add.city, &emp.add.pin, emp.add.phone);
   display(emp);
}
void display(struct employee emp)
{
```

```
  printf("Printing the details....\n");
  printf("%s %s %d %s",emp.name,emp.add.city,emp.add.pin,emp.add.phone);
}
```

C Union

Like structure, **Union in c language** is *a user-defined data type* that is used to store the different type of elements.

At once, only one member of the union can occupy the memory. In other words, we can say that the size of the union in any instance is equal to the size of its largest element.

Structure	Union

struct Employee{
char x; // size 1 byte
int y; //size 2 byte
float z; //size 4 byte
}e1; //size of e1 = 7 byte
	union Employee{
	char x; // size 1 byte
	int y; //size 2 byte
	float z; //size 4 byte
	}e1; //size of e1 = 4 byte

size of e1= 1 + 2 + 4 = 7	**size of e1= 4 (maximum size of 1 element)**

Advantage of union over structure

It **occupies less memory** because it occupies the size of the largest member only.

Disadvantage of union over structure

Only the last entered data can be stored in the union. It overwrites the data previously stored in the union.

Defining union

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

```
union union_name
{
   data_type member1;
   data_type member2;
   .
   .
   data_type memeberN;
};
```

Let's see the example to define union for an employee in c.

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

C Union example

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

```
#include <stdio.h>
#include <string.h>
union employee
{   int id;
    char name[50];
}e1;  //declaring e1 variable for union
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 : 1869508435
employee 1 name : Sonoo Jaiswal
```

As you can see, id gets garbage value because name has large memory size. So only name will have actual value.

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.

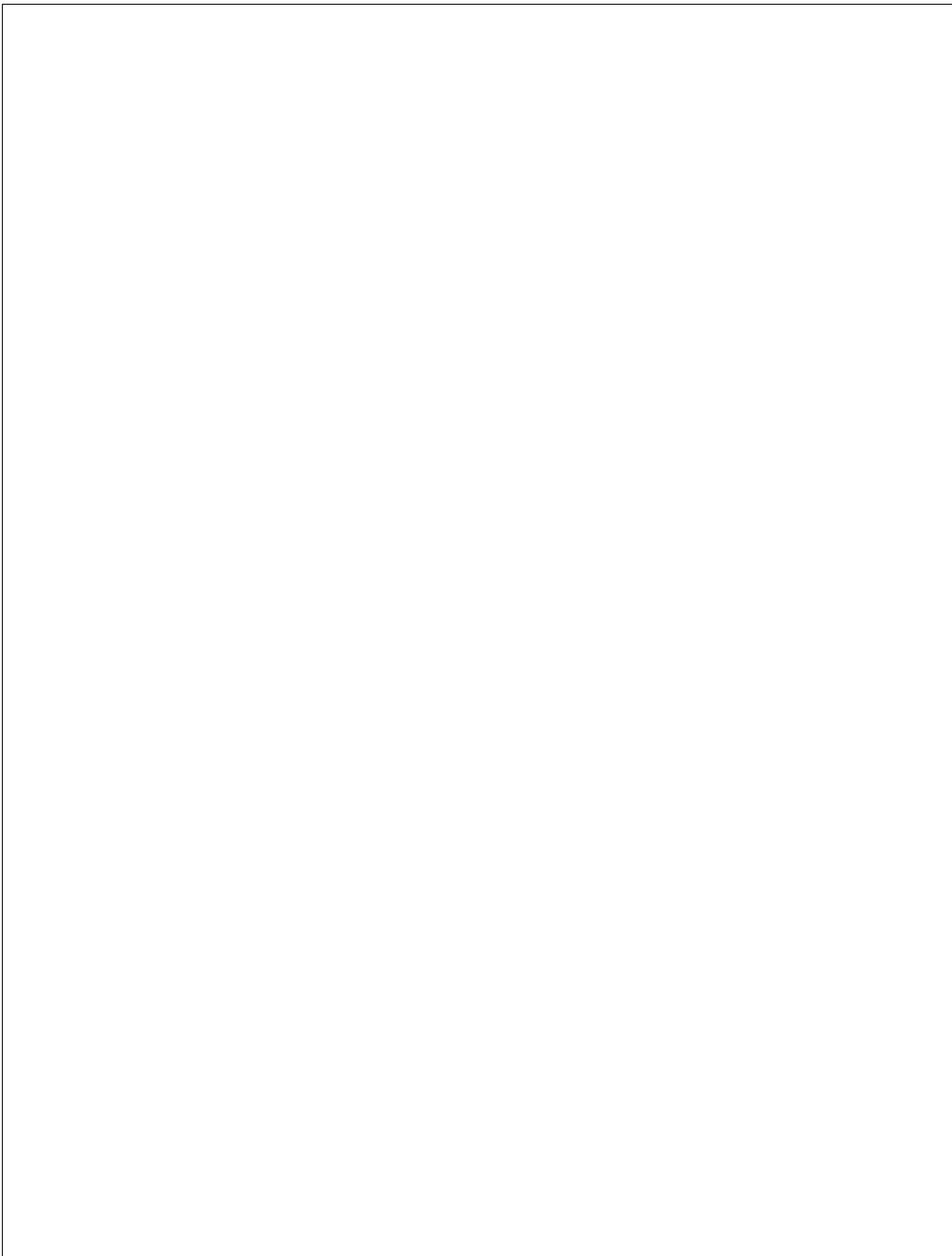
```
#include;
void main( )
{
```

```
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);



C fprintf() and fscanf()

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...
```

C File Example: Storing employee information

Let's see a file handling example to store employee information as entered by user from console. We are going to store id, name and salary of the employee.

```
#include <stdio.h>
void main()
{
   FILE *fptr;
   int id;
   char name[30];
   float salary;
   fptr = fopen("emp.txt", "w+");/*  open for writing */
   if (fptr == NULL)
   {
      printf("File does not exists \n");
      return;
   }
   printf("Enter the id\n");
   scanf("%d", &id);
   fprintf(fptr, "Id= %d\n", id);
   printf("Enter the name \n");
   scanf("%s", name);
   fprintf(fptr, "Name= %s\n", name);
   printf("Enter the salary\n");
   scanf("%f", &salary);
   fprintf(fptr, "Salary= %.2f\n", salary);
   fclose(fptr);
}
```

Output:

```
 Enter the id
1
Enter the name
sonoo
Enter the salary
120000
```

Now open file from current directory. For windows operating system, go to TC\bin directory, you will see emp.txt file. It will have following information.

emp.txt

```
Id= 1
Name= sonoo
Salary= 120000
```

C fputc() and fgetc()

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
```

C fputs() and fgets()

The fputs() and fgets() in C programming are used to write and read string from stream. Let's see examples of writing and reading file using fgets() and fgets() functions.

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 javatpoint", 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
```

As you can see, rewind() function moves the file pointer at beginning of the file that is why "this is simple text" is printed 2 times. If you don't call rewind() function, "this is simple text" will be printed only once.

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:

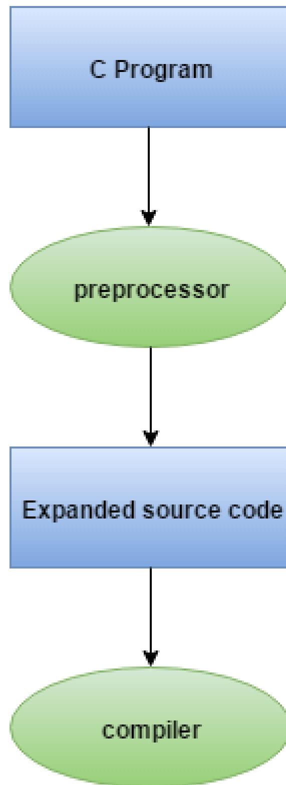
```
Size of file: 21 bytes
```

C Preprocessor Directives

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

All preprocessor directives starts with hash # symbol.

Let's see a list of preprocessor directives.



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

C Macros

A macro is a segment of code which is replaced by the value of macro. Macro is defined by #define directive. There are two types of macros:

1. Object-like Macros
2. Function-like Macros

Object-like Macros

The object-like macro is an identifier that is replaced by value. It is widely used to represent numeric constants. For example:

```
#define PI 3.14
```

Here, PI is the macro name which will be replaced by the value 3.14.

Function-like Macros

The function-like macro looks like function call. For example:

```
#define MIN(a,b) ((a)<(b)?(a):(b))
```

Here, MIN is the macro name.

C Predefined Macros

ANSI C defines many predefined macros that can be used in c program.

No.	Macro	Description
1	_DATE_	represents current date in "MMM DD YYYY" format.
2	_TIME_	represents current time in "HH:MM:SS" format.
3	_FILE_	represents current file name.
4	_LINE_	represents current line number.
5	_STDC_	It is defined as 1 when compiler complies with the ANSI standard.

C predefined macros example

File: simple.c

```
#include<stdio.h>
 int main(){
   printf("File :%s\n", __FILE__ );
   printf("Date :%s\n", __DATE__ );
   printf("Time :%s\n", __TIME__ );
   printf("Line :%d\n", __LINE__ );
   printf("STDC :%d\n", __STDC__ );
   return 0;
 }
```

Output:

```
File :simple.c
Date :Dec 6 2015
Time :12:28:46
Line :6
STDC :1
```

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.

1. #include <filename>
2. #include "filename"

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

#include notes:

Note 1: In #include directive, comments are not recognized. So in case of #include <a//b>, a//b is treated as filename.

Note 2: In #include directive, backslash is considered as normal text not escape sequence. So in case of #include <a\nb>, a\nb is treated as filename.

Note 3: You can use only comment after filename otherwise it will give error.

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
```

et's see an example of #define to create a macro.

```
#include <stdio.h>
#define MIN(a,b) ((a)<(b)?(a):(b))
void main() {
   printf("Minimum between 10 and 20 is: %d\n", MIN(10,20));
}
```

Output:

```
Minimum between 10 and 20 is: 10
```

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
```

But, if you don't define NOINPUT, it will ask user to enter a number.

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

```
Enter a:5
Value of a: 5
```

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
```

But, if you don't define INPUT, it will execute the code of #ifndef.

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

```
Value of a: 2
```

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
```

Let's see another example to understand the #if directive clearly.

```
#include <stdio.h>
#include <conio.h>
#define NUMBER 1
void main() {
clrscr();
#if (NUMBER==0)
printf("1 Value of Number is: %d",NUMBER);
#endif

#if (NUMBER==1)
printf("2 Value of Number is: %d",NUMBER);
#endif
getch();
}
```

Output:

```
2 Value of Number is: 1
```

C #else

The #else preprocessor directive evaluates the expression or condition if condition of #if is false. It can be used with #if, #elif, #ifdef and #ifndef directives.

Syntax:

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

Syntax with #elif:

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

C #else example

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

```
#include <stdio.h>
#include <conio.h>
#define NUMBER 1
void main() {
#if NUMBER==0
printf("Value of Number is: %d",NUMBER);
#else
print("Value of Number is non-zero");
#endif
getch();
}
```

Output:

```
Value of Number is non-zero
```

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
```

But, if you include math.h, it does not gives error.

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

Output:

```
2.645751
```

C #pragma

The #pragma preprocessor directive is used to provide additional information to the compiler. The #pragma directive is used by the compiler to offer machine or operating-system feature.

Syntax:

```
#pragma token
```

Different compilers can provide different usage of #pragma directive.

The turbo C++ compiler supports following #pragma directives.

```
#pragma argsused
#pragma exit
#pragma hdrfile
#pragma hdrstop
#pragma inline
#pragma option
#pragma saveregs
#pragma startup
#pragma warn
```

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

```
#include<stdio.h>
#include<conio.h>
void func() ;
#pragma startup func
#pragma exit func
void main(){
printf("\nI am in main");
getch();
}
void func(){
printf("\nI am in func");
getch();
}
```

Output:

```
I am in func
I am in main
I am in func
```

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.

C Strings

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

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

1. By char array
2. By string literal

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

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

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

0	1	2	3	4	5	6	7	8	9	10
j	a	v	a	t	p	o	i	n	t	\0

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

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

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

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

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

Difference between char array and string literal

There are two main differences between char array and literal.

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

String Example in C

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

```
#include<stdio.h>
#include <string.h>
int main(){
  char ch[11]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};
   char ch2[11]="javatpoint";

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

Output:

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

Traversing String

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

Hence, there are two ways to traverse a string.

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

Let's discuss each one of them.

Using the length of string

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

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

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

Output

```
The number of vowels 4
```

Using the null character

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

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

Output

```
The number of vowels 4
```

Accepting string as the input

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

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

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

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);
}
```

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

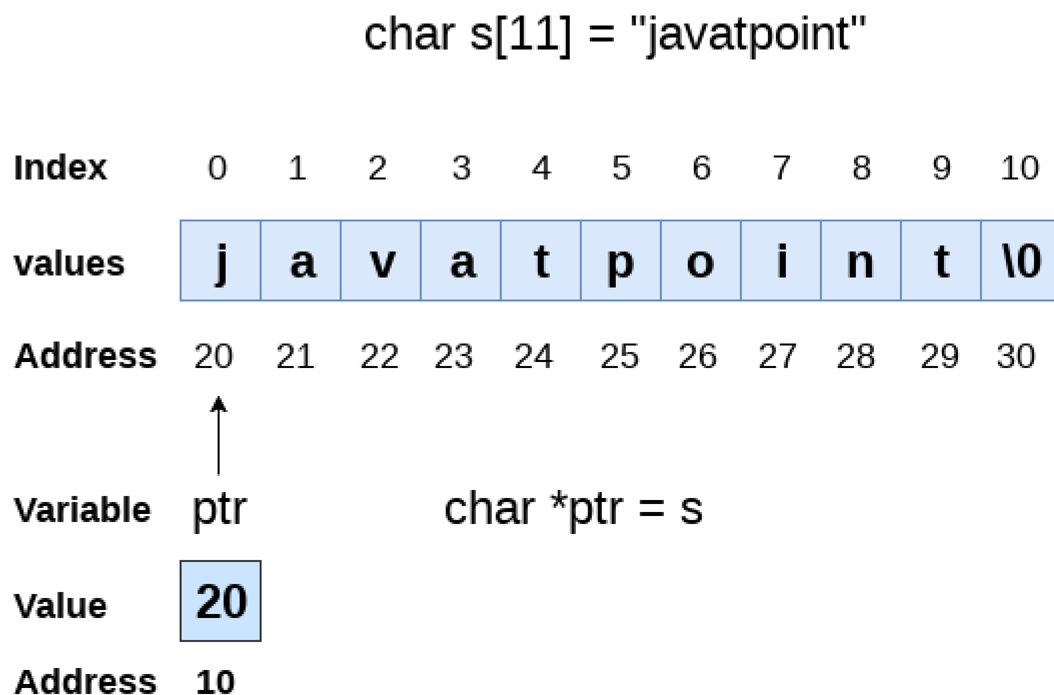
Pointers with strings

We have used pointers with the array, functions, and primitive data types so far. However, pointers can be used to point to the strings. There are various advantages of using pointers to point strings. Let us consider the following example to access the string via the pointer.

```
#include<stdio.h>
void main ()
{
    char s[11] = "javatpoint";
    char *p = s; // pointer p is pointing to string s.
    printf("%s",p); // the string javatpoint is printed if we print p.
}
```

Output

```
javatpoint
```



As we know that string is an array of characters, the pointers can be used in the same way they were used with arrays. In the above example, p is declared as a pointer to the array of

characters s. P affects similar to s since s is the base address of the string and treated as a pointer internally. However, we can not change the content of s or copy the content of s into another string directly. For this purpose, we need to use the pointers to store the strings. In the following example, we have shown the use of pointers to copy the content of a string into another.

```
#include<stdio.h>
void main ()
{
   char *p = "hello javatpoint";
   printf("String p: %s\n",p);
   char *q;
   printf("copying the content of p into q...\n");
   q = p;
   printf("String q: %s\n",q);
}
```

Output

```
String p: hello javatpoint
copying the content of p into q...
String q: hello javatpoint
```

Once a string is defined, it cannot be reassigned to another set of characters. However, using pointers, we can assign the set of characters to the string. Consider the following example.

```
#include<stdio.h>
void main ()
{
   char *p = "hello javatpoint";
   printf("Before assigning: %s\n",p);
   p = "hello";
   printf("After assigning: %s\n",p);
}
```

Output

```
Before assigning: hello javatpoint
After assigning: hello
```

C gets() and 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?
javatpoint is the best
You entered javatpoint 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? javatpoint is the best website
javatpoint 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)	concats 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.

C String Length: 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
```

C Copy String: strcpy()

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: javatpoint
```

C String Concatenation: strcat()

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
```

C Compare String: strcmp()

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
```

C Reverse String: strrev()

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: javatpoint
String is: javatpoint
Reverse String is: tnioptavaj
```

C String Lowercase: strlwr()

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: JAVATpoint
String is: JAVATpoint
Lower String is: javatpoint
```

C String Uppercase: strupr()

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: javatpoint
String is: javatpoint
Upper String is: JAVATPOINT
```

C String strstr()

The strstr() function returns pointer to the first occurrence of the matched string in the given string. It is used to return substring from first match till the last character.

Syntax:

```
char *strstr(const char *string, const char *match)
```

String strstr() parameters

string: It represents the full string from where substring will be searched.

match: It represents the substring to be searched in the full string.

String strstr() example

```
#include<stdio.h>
#include <string.h>
int main(){
  char str[100]="this is javatpoint with c and java";
  char *sub;
  sub=strstr(str,"java");
  printf("\nSubstring is: %s",sub);
 return 0;
}
```

Output:

```
javatpoint with c and java
```

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	ceil(number)	rounds up the given number. It returns the integer value which is greater than or equal to given number.
2	floor(number)	rounds down the given number. It returns the integer value which is less than or equal to given number.
3	sqrt(number)	returns the square root of given number.
4	pow(base, exponent)	returns the power of given number.
5	abs(number)	returns the absolute value of given number.

C Math Example

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

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

Output:

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

C - Error Handling

As such, C programming does not provide direct support for error handling but being a system programming language, it provides you access at lower level in the form of return values. Most of the C or even Unix function calls return -1 or NULL in case of any error and set an error code errno. It is set as a global variable and indicates an error occurred during any function call. You can find various error codes defined in <error.h> header file.

So a C programmer can check the returned values and can take appropriate action depending on the return value. It is a good practice, to set errno to 0 at the time of initializing a program. A value of 0 indicates that there is no error in the program.

errno, perror(). and strerror()

The C programming language provides perror() and strerror() functions which can be used to display the text message associated with errno.

The perror() function displays the string you pass to it, followed by a colon, a space, and then the textual representation of the current errno value.

The strerror() function, which returns a pointer to the textual representation of the current errno value.

Let's try to simulate an error condition and try to open a file which does not exist. Here I'm using both the functions to show the usage, but you can use one or more ways of printing your errors. Second important point to note is that you should use stderr file stream to output all the errors.

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

extern int errno ;

int main () {

   FILE * pf;
   int errnum;
   pf = fopen ("unexist.txt", "rb");

   if (pf == NULL) {

      errnum = errno;
      fprintf(stderr, "Value of errno: %d\n", errno);
      perror("Error printed by perror");
      fprintf(stderr, "Error opening file: %s\n", strerror( errnum
));
   } else {
```

```
      fclose (pf);
   }

   return 0;
}
```

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

```
Value of errno: 2
Error printed by perror: No such file or directory
Error opening file: No such file or directory
```

Divide by Zero Errors

It is a common problem that at the time of dividing any number, programmers do not check if a divisor is zero and finally it creates a runtime error.

The code below fixes this by checking if the divisor is zero before dividing –

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

main() {

   int dividend = 20;
   int divisor = 0;
   int quotient;

   if( divisor == 0){
      fprintf(stderr, "Division by zero! Exiting...\n");
      exit(-1);
   }

   quotient = dividend / divisor;
   fprintf(stderr, "Value of quotient : %d\n", quotient );

   exit(0);
}
```

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

```
Division by zero! Exiting...
```

Program Exit Status

It is a common practice to exit with a value of EXIT_SUCCESS in case of program coming out after a successful operation. Here, EXIT_SUCCESS is a macro and it is defined as 0.

If you have an error condition in your program and you are coming out then you should exit with a status EXIT_FAILURE which is defined as -1. So let's write above program as follows −

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

main() {

   int dividend = 20;
   int divisor = 5;
   int quotient;

   if( divisor == 0) {
      fprintf(stderr, "Division by zero! Exiting...\n");
      exit(EXIT_FAILURE);
   }

   quotient = dividend / divisor;
   fprintf(stderr, "Value of quotient : %d\n", quotient );

   exit(EXIT_SUCCESS);
}
```

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

```
Value of quotient : 4
```

Searching and Sorting Techniques

Searching Technique

1. Linear Search

2. Binary Search

Linear Search

This is the simplest searching technique where each element is compared with the element to be searched.

- Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
- If x matches with an element, return the index.
- If x doesn’t match with any of elements, return -1

```
// C++ code to linearly search x in arr[]. If x

// is present then return its location, otherwise

// return -1

#include <stdio.h>

int search(int arr[], int n, int x)

{

	int i;

	for (i = 0; i < n; i++)

		if (arr[i] == x)

			return i;

	return -1;

}
```

```
int main(void)

{

        int arr[] = { 2, 3, 4, 10, 40 };

        int x = 10;

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

        int result = search(arr, n, x);

        (result == -1) ? printf("Element is not present in array")

                                        : printf("Element is present at index %d",

                                                                        result);

        return 0;

}
```

Binary Search

In computer science, a binary search or half-interval search algorithm finds the position of a target value within a sorted array. The binary search algorithm can be classified as a dichotomies divide-and-conquer search algorithm and executes in logarithmic time.

Pictorial presentation - Binary search algorithm

Number

0	1	2	3	4	5	6	7	8
6	12	17	23	38	45	77	84	90

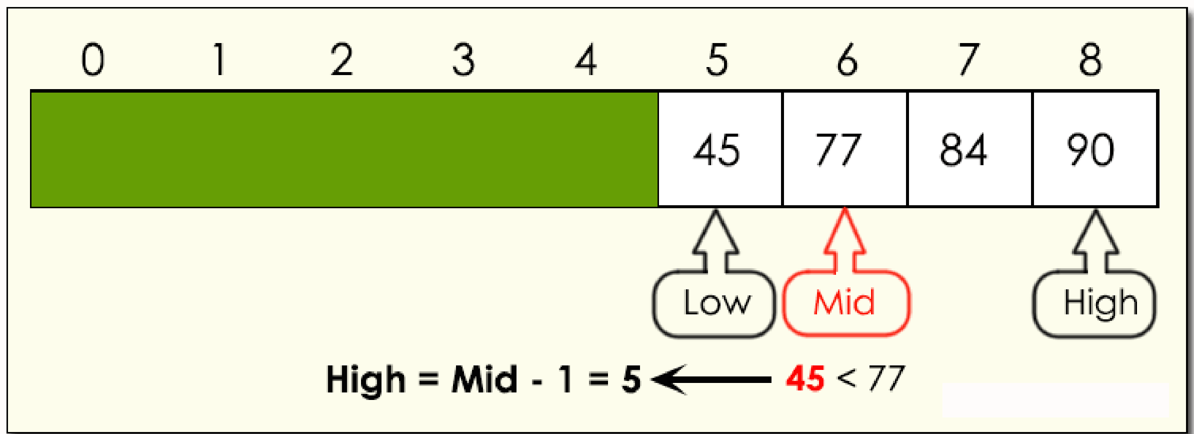
	Low	High	Mid
#1	0	8	4

Search (45)

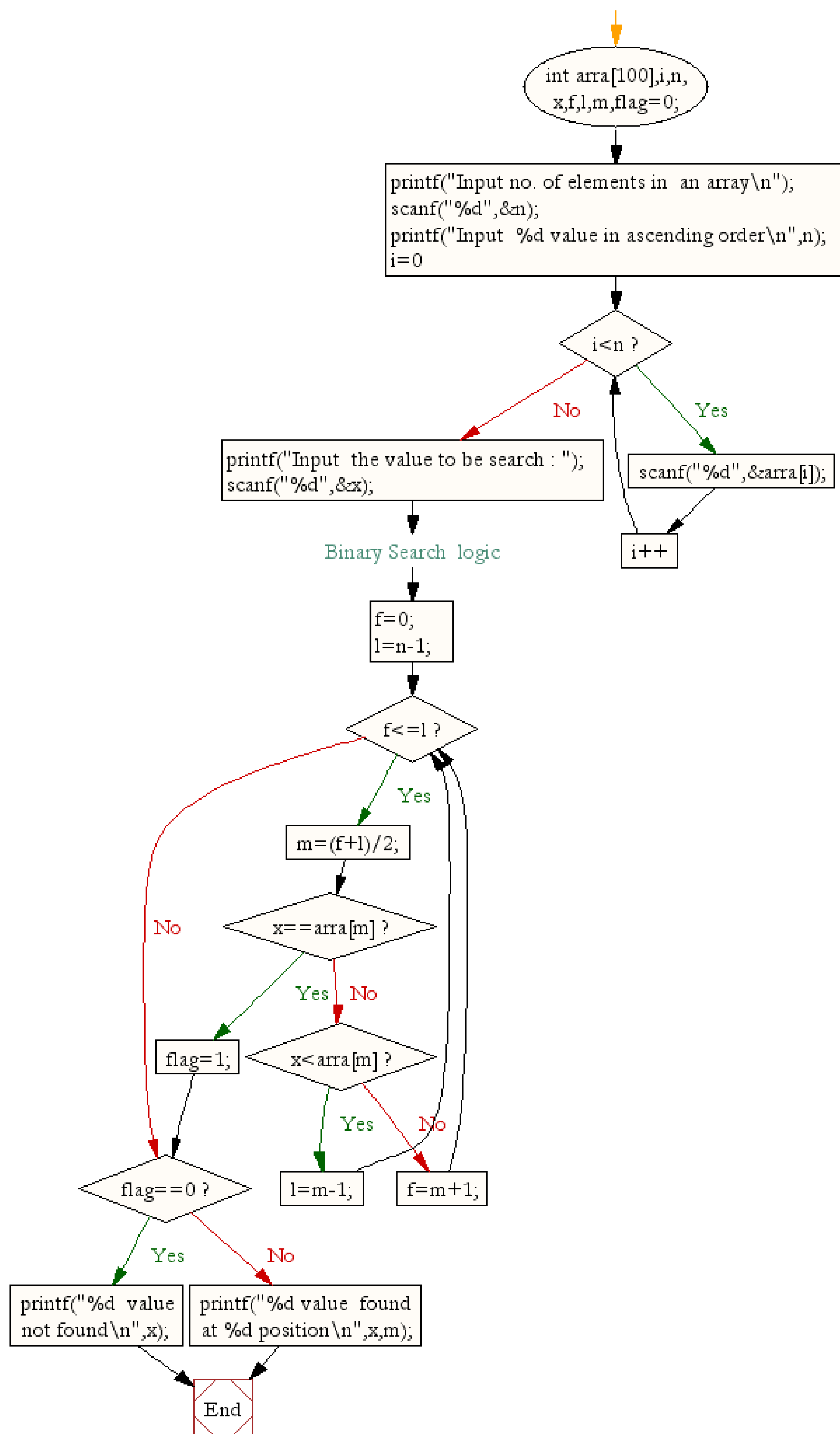
$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$

0	1	2	3	4	5	6	7	8
6	12	17	23	38	45	77	84	90
Low				Mid				High

38 < **45** → **Low = Mid + 1 = 5**



Low
High
Mid
#1
0
8
4
#2
5
8
6
Search (45)
$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$
0
1
2
3
4
5
6
7
8
45
77
84
90
Low
Mid
High
High = Mid - 1 = 5
45 < 77


int arra[100],i,n,
x,f,l,m,flag=0;
printf("Input no. of elements in an array\n");
scanf("%d",&n);
printf("Input %d value in ascending order\n",n);
i=0
i<n ?
No
Yes
printf("Input the value to be search : ");
scanf("%d",&x);
scanf("%d",&arra[i]);
i++
Binary Search logic
f=0;
l=n-1;
f<=l ?
Yes
No
m=(f+l)/2;
x==arra[m] ?
Yes
No
flag=1;
x<arra[m] ?
Yes
No
flag==0 ?
l=m-1;
f=m+1;
Yes
No
printf("%d value
not found\n",x);
printf("%d value found
at %d position\n",x,m);
End

Code for Binary Search

```
#include<stdio.h>
  void main()
  {
        int  arra[100],i,n,x,f,l,m,flag=0;
        printf("Input no. of elements in  an array\n");
        scanf("%d",&n);
        printf("Input  %d value in ascending order\n",n);
        for(i=0;i<n;i++)
        scanf("%d",&arra[i]);
        printf("Input  the value to be search : ");
        scanf("%d",&x);
        /* Binary Search  logic */
        f=0;l=n-1;
        while(f<=l)
        {
            m=(f+l)/2;
            if(x==arra[m])
            {
                flag=1;
                break;
            }
            else  if(x<arra[m])
                l=m-1;
            else
```

```
                f=m+1;
        }
        if(flag==0)
            printf("%d  value not found\n",x);
        else
            printf("%d value  found at %d position\n",x,m);
}
```

Sorting Techniques

Arranging the elements in ascending or descending order is called as Sorting

Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1) The subarray which is already sorted.
2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

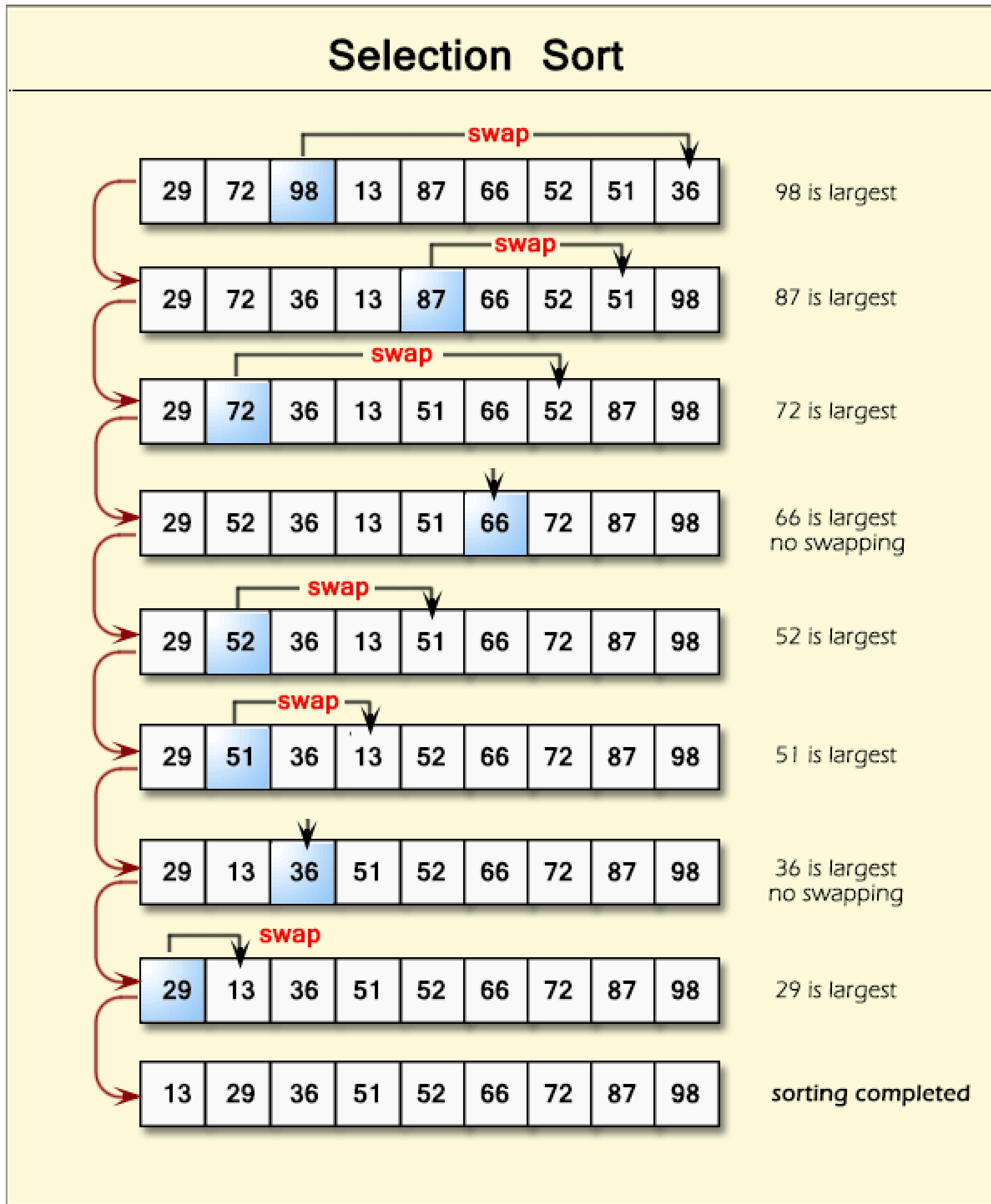
Note:

a) To find maximum of elements

b) To swap two elements

```
arr[] = 64 25 12 22 11


// Find the minimum element in arr[0...4]

// and place it at beginning

11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4]
11 12 25 22 64

// Find the minimum element in arr[2...4]
// and place it at beginning of arr[2...4]
11 12 22 25 64

// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 25 64
```

Pictorial presentation - Selection search algorithm :



Sample C Code:

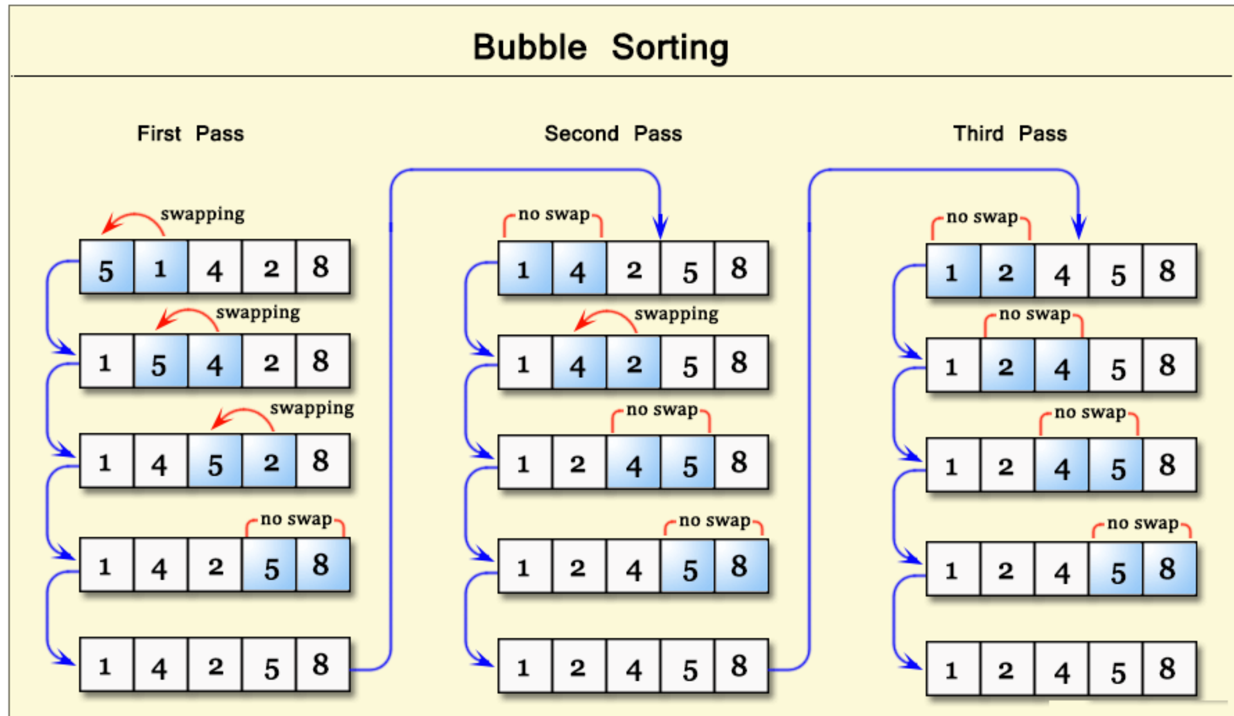
```
#include <stdio.h>
  int main()
  {
  int arr[10];
   int i, j, N, temp;
 /* function  declaration */
  int find_max(int  b[10], int k);
  void exchang(int  b[10], int k);
  printf("\nInput no. of values in the array : N");
  scanf("%d",&N);
  printf("\nInput  the elements one by one: ");
  for(i=0; i<N ;  i++)
  {
  scanf("%d",&arr[i]);
  }
  /* Selection sorting  begins */
  exchang(arr,N);
  printf("Sorted  array :\n");
  for(i=0; i< N ;  i++)
  {
  printf("%d\n",arr[i]);
  }
}
/* function to find the maximum value */
```

```
int find_max(int b[10], int k)
{
int max=0,j;
for(j = 1; j <=  k; j++)
{
if ( b[j] >  b[max])
{
max =  j;
}
}
return(max);
}
void exchang(int  b[10],int k)
{
int  temp, big, j;
for ( j=k-1;  j>=1; j--)
{
big =  find_max(b,j);
temp = b[big];
b[big] = b[j];
b[j] = temp;
}
return ;
}
```

Bubble Sort

Bubble Sort works by repeatedly swapping the adjacent elements if they are in wrong order.

Pictorial presentation - Bubble sort algorithm:



Example:
First Pass:
(**5** **1** 4 2 8) –> (**1** **5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5** **4** 2 8) –> (1 **4** **5** 2 8), Swap since 5 > 4
(1 4 **5** **2** 8) –> (1 4 **2** **5** 8), Swap since 5 > 2
(1 4 2 **5** **8**) –> (1 4 2 **5** **8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.
Second Pass:
(**1** **4** 2 5 8) –> (**1** **4** 2 5 8)
(1 **4** **2** 5 8) –> (1 **2** **4** 5 8), Swap since 4 > 2
(1 2 **4** **5** 8) –> (1 2 **4** **5** 8)
(1 2 4 **5** **8**) –> (1 2 4 **5** **8**)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:
(**1** **2** 4 5 8) –> (**1** **2** 4 5 8)
(1 **2** **4** 5 8) –> (1 **2** **4** 5 8)
(1 2 **4** **5** 8) –> (1 2 **4** **5** 8)
(1 2 4 **5** **8**) –> (1 2 4 **5** **8**)

Sample C Code:

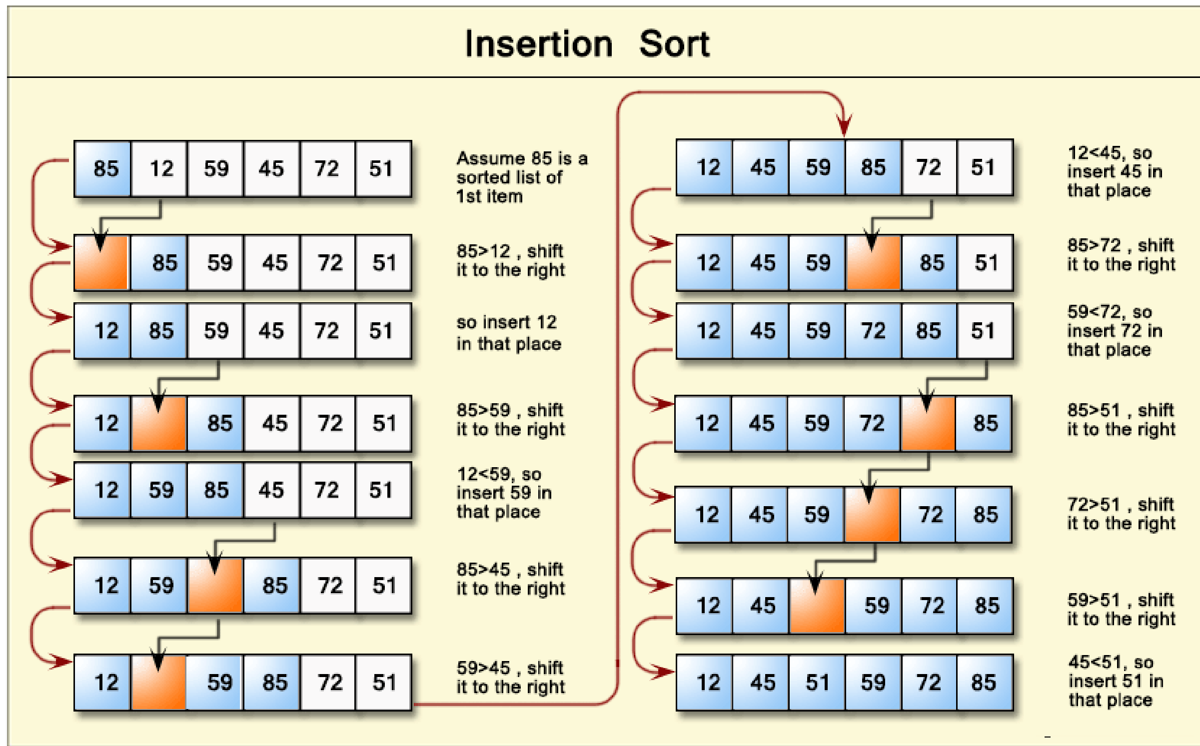
```
#include <stdio.h>

 void bubble_sort (int *x, int n) {

    int i, t, j = n, s = 1;

    while (s) {

        s = 0;

        for (i = 1; i < j; i++) {

            if (x[i] < x[i - 1]) {

                t = x[i];

                x[i] = x[i - 1];

                x[i - 1] = t;

                s = 1;

            }

        }

        j--;

    }

}


int main () {

    int x[] = {15, 56, 12, -21, 1, 659, 3, 83, 51, 3, 135, 0};

    int n = sizeof x / sizeof x[0];
```

```
    int i;
    for (i = 0; i < n; i++)
        printf("%d%s", x[i], i == n - 1 ? "\n" : " ");
    bubble_sort(x, n);
    for (i = 0; i < n; i++)
        printf("%d%s", x[i], i == n - 1 ? "\n" : " ");
    return 0;
}
```

Insertion sort

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than other algorithms such as quicksort, heapsort, or merge sort.

Pictorial presentation - Insertion search algorithm:



Another Example:
12, 11, 13, 5, 6
Let us loop for i = 1 (second element of the array) to 4 (last element of the array)

i = 1. Since 11 is smaller than 12, move 12 and insert 11 before 12
11, 12, 13, 5, 6
i = 2. 13 will remain at its position as all elements in **A[0..I-1]** are smaller than 13
11, 12, 13, 5, 6
i = 3. 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.
5, 11, 12, 13, 6
i = 4. 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.
5, 6, 11, 12, 13

Sample C Code:

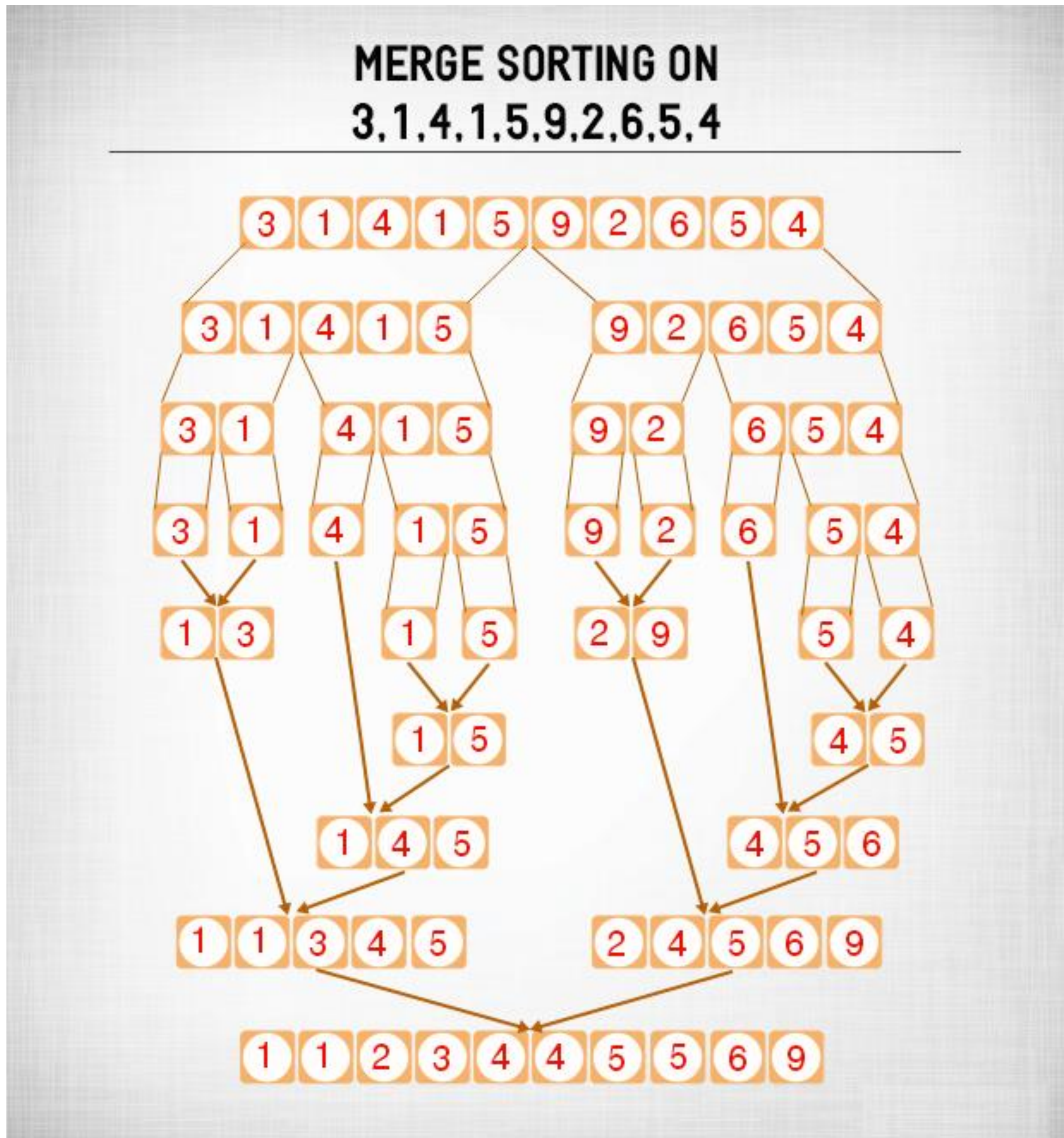
```
#include <bits/stdc++.h>

using namespace std;

/* Function to sort an array using insertion sort*/

void insertionSort(int arr[], int n)

{

    int i, key, j;

    for (i = 1; i < n; i++)

    {

        key = arr[i];

        j = i - 1;

        /* Move elements of arr[0..i-1], that are

        greater than key, to one position ahead

        of their current position */

        while (j >= 0 && arr[j] > key)

        {

            arr[j + 1] = arr[j];

            j = j - 1;

        }

        arr[j + 1] = key;

    }

}

// A utility function to print an array of size n

void printArray(int arr[], int n)
```

```
{
        int i;
        for (i = 0; i < n; i++)
                cout << arr[i] << " ";
        cout << endl;
}
/* Driver code */
int main()
{
        int arr[] = { 12, 11, 13, 5, 6 };
        int n = sizeof(arr) / sizeof(arr[0]);
        insertionSort(arr, n);
        printArray(arr, n);
        return 0;
}
```

Merge sort

Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.

Pictorial presentation - Merge search algorithm :



```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into two halves:
             middle m = (l+r)/2
     2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
             Call merge(arr, l, m, r)
```

```
/* C program for Merge Sort */

#include<stdlib.h>

#include<stdio.h>

// Merges two subarrays of arr[].

// First subarray is arr[l..m]

// Second subarray is arr[m+1..r]

void merge(int arr[], int l, int m, int r)

{

    int i, j, k;

    int n1 = m - l + 1;

    int n2 = r - m;

    /* create temp arrays */

    int L[n1], R[n2];

    /* Copy data to temp arrays L[] and R[] */

    for (i = 0; i < n1; i++)

        L[i] = arr[l + i];

    for (j = 0; j < n2; j++)

        R[j] = arr[m + 1+ j];

    /* Merge the temp arrays back into arr[l..r]*/
```

```
i = 0; // Initial index of first subarray
j = 0; // Initial index of second subarray
k = l; // Initial index of merged subarray
while (i < n1 && j < n2)
{
        if (L[i] <= R[j])
        {
                arr[k] = L[i];
                i++;
        }
        else
        {
                arr[k] = R[j];
                j++;
        }
        k++;
}
/* Copy the remaining elements of L[], if there
are any */
while (i < n1)
{
        arr[k] = L[i];
        i++;
        k++;
}
```

```
        /* Copy the remaining elements of R[], if there
        are any */
        while (j < n2)
        {
                arr[k] = R[j];
                j++;
                k++;
        }
}
/* l is for left index and r is right index of the
sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
        if (l < r)
        {
                // Same as (l+r)/2, but avoids overflow for
                // large l and h
                int m = l+(r-l)/2;
                // Sort first and second halves
                mergeSort(arr, l, m);
                mergeSort(arr, m+1, r);
                merge(arr, l, m, r);
        }
}
```

```
/* UTILITY FUNCTIONS */

/* Function to print an array */

void printArray(int A[], int size)

{

        int i;

        for (i=0; i < size; i++)

                printf("%d ", A[i]);

        printf("\n");

}

/* Driver program to test above functions */

int main()

{

        int arr[] = {12, 11, 13, 5, 6, 7};

        int arr_size = sizeof(arr)/sizeof(arr[0]);

        printf("Given array is \n");

        printArray(arr, arr_size);

        mergeSort(arr, 0, arr_size - 1);

        printf("\nSorted array is \n");

        printArray(arr, arr_size);

        return 0;

}
```

C Programs

C programs are frequently asked in the interview. These programs can be asked from basics, array, string, pointer, linked list, file handling etc. Let's see the list of c programs.

1) Fibonacci Series

Write a c program to print fibonacci series without using recursion and using recursion.

Fibonacci Series in C: In case of fibonacci series, *next number is the sum of previous two numbers* for example 0, 1, 1, 2, 3, 5, 8, 13, 21 etc. The first two numbers of fibonacci series are 0 and 1.

There are two ways to write the fibonacci series program:

- Fibonacci Series without recursion
- Fibonacci Series using recursion

Fibonacci Series in C without recursion

Let's see the fibonacci series program in c without recursion.

```
#include<stdio.h>
int main()
{
 int n1=0,n2=1,n3,i,number;
 printf("Enter the number of elements:");
 scanf("%d",&number);
 printf("\n%d %d",n1,n2);//printing 0 and 1
 for(i=2;i<number;++i)//loop starts from 2 because 0 and 1 are already printed
 {
  n3=n1+n2;
  printf(" %d",n3);
  n1=n2;
  n2=n3;
 }
  return 0;
 }
```

Output:

```
Enter the number of elements:15
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Fibonacci Series using recursion in C

Let's see the fibonacci series program in c using recursion.

```
#include<stdio.h>
   void printFibonacci(int n){
   static int n1=0,n2=1,n3;
   if(n>0){
       n3 = n1 + n2;
       n1 = n2;
       n2 = n3;
       printf("%d ",n3);
       printFibonacci(n-1);
   }
}
   int main(){
   int n;
   printf("Enter the number of elements: ");
   scanf("%d",&n);
   printf("Fibonacci Series: ");
   printf("%d %d ",0,1);
   printFibonacci(n-2);//n-2 because 2 numbers are already printed
  return 0;
 }
```

Output:

```
Enter the number of elements:15
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

2) Prime number

Write a c program to check prime number.

Prime number in C: **Prime number** is a number that is greater than 1 and divided by 1 or itself. In other words, prime numbers can't be divided by other numbers than itself or 1. For example 2, 3, 5, 7, 11, 13, 17, 19, 23.... are the prime numbers.

> *Note: Zero (0) and 1 are not considered as prime numbers. Two (2) is the only one even prime number because all the numbers can be divided by 2.*

Let's see the prime number program in C. In this c program, we will take an input from the user and check whether the number is prime or not.

```
#include<stdio.h>
int main(){
int n,i,m=0,flag=0;
printf("Enter the number to check prime:");
scanf("%d",&n);
m=n/2;
for(i=2;i<=m;i++)
{
if(n%i==0)
{
printf("Number is not prime");
flag=1;
break;
}
}
if(flag==0)
printf("Number is prime");
return 0;
 }
```

Output:

```
Enter the number to check prime:56
Number is not prime
Enter the number to check prime:23
Number is prime
```

3) Palindrome number

Write a c program to check palindrome number.

Palindrome number in c: A **palindrome number** is *a number that is same after reverse*. For example 121, 34543, 343, 131, 48984 are the palindrome numbers.

Palindrome number algorithm

- Get the number from user
- Hold the number in temporary variable
- Reverse the number
- Compare the temporary number with reversed number
- If both numbers are same, print palindrome number
- Else print not palindrome number

Let's see the palindrome program in C. In this c program, we will get an input from the user and check whether number is palindrome or not.

```
#include<stdio.h>
int main()
{
int n,r,sum=0,temp;
printf("enter the number=");
scanf("%d",&n);
temp=n;
while(n>0)
{
r=n%10;
sum=(sum*10)+r;
n=n/10;
}
if(temp==sum)
printf("palindrome number ");
else
printf("not palindrome");
return 0;
}
```

Output:

```
enter the number=151
palindrome  number

enter the number=5621
not palindrome  number
```

4) Factorial

Write a c program to print factorial of a number.

Factorial Program in C: Factorial of n is the *product of all positive descending integers*. Factorial of *n* is denoted by n!. For example:

5! = 5*4*3*2*1 = 120
3! = 3*2*1 = 6

Here, 5! is pronounced as "5 factorial", it is also called "5 bang" or "5 shriek".

The factorial is normally used in Combinations and Permutations (mathematics).

There are many ways to write the factorial program in c language. Let's see the 2 ways to write the factorial program.

- Factorial Program using loop
- Factorial Program using recursion

Factorial Program using loop

Let's see the factorial Program using loop.

```
#include<stdio.h>
int main()
{
 int i,fact=1,number;
 printf("Enter a number: ");
 scanf("%d",&number);
   for(i=1;i<=number;i++){
      fact=fact*i;
  }
  printf("Factorial of %d is: %d",number,fact);
 return 0;
 }
```

Output:

```
Enter a number: 5
Factorial of 5 is: 120
```

Factorial Program using recursion in C

Let's see the factorial program in c using recursion.

```
#include<stdio.h>
long factorial(int n)
{
  if (n == 0)
    return 1;
  else
    return(n * factorial(n-1));
}
void main()
{
  int number;
  long fact;
 printf("Enter a number: ");
 scanf("%d", &number);
  fact = factorial(number);
  printf("Factorial of %d is %ld\n", number, fact);
  return 0;
}
```

Output:

```
Enter a number: 6
Factorial of 5 is: 720
```

5) Armstrong number

Write a c program to check armstrong number.

Before going to write the c program to check whether the number is Armstrong or not, let's understand what is Armstrong number.

Armstrong number is *a number that is equal to the sum of cubes of its digits*. For example 0, 1, 153, 370, 371 and 407 are the Armstrong numbers.

Let's try to understand why **153** is an Armstrong number.

153 = (1*1*1)+(5*5*5)+(3*3*3)
where:
(1*1*1)=1
(5*5*5)=125
(3*3*3)=27
So:
1+125+27=153

Let's try to understand why **371** is an Armstrong number.

371 = (3*3*3)+(7*7*7)+(1*1*1)
where:
(3*3*3)=27
(7*7*7)=343
(1*1*1)=1
So:
27+343+1=371

Let's see the c program to check Armstrong Number in C.

```
#include<stdio.h>
 int main()
{
int n,r,sum=0,temp;
printf("enter the number=");
scanf("%d",&n);
temp=n;
while(n>0)
{
r=n%10;
sum=sum+(r*r*r);
```

```
n=n/10;
}
if(temp==sum)
printf("armstrong  number ");
else
printf("not armstrong number");
return 0;
}
```

Output:

```
enter the number=153
armstrong number

enter the number=5
not armstrong number
```

6) Sum of Digits

Write a c program to print sum of digits.

C program to sum each digit: We can write the sum of digits program in c language by the help of loop and mathematical operation only.

Sum of digits algorithm

To get sum of each digits by c program, use the following algorithm:

- Step 1: Get number by user
- Step 2: Get the modulus/remainder of the number
- Step 3: sum the remainder of the number
- Step 4: Divide the number by 10
- Step 5: Repeat the step 2 while number is greater than 0.

Let's see the sum of digits program in C.

```
#include<stdio.h>
 int main()
{
int n,sum=0,m;
printf("Enter a number:");
scanf("%d",&n);
while(n>0)
{
m=n%10;
sum=sum+m;
n=n/10;
}
printf("Sum is=%d",sum);
return 0;
}
```

Output:

```
Enter a number:654
Sum is=15
Enter a number:123
Sum is=6
```

7) Reverse Number

Write a c program to reverse given number.

We can reverse a number in c using loop and arithmetic operators. In this program, we are getting number as input from the user and reversing that number. Let's see a simple c example to reverse a given number.

```
#include<stdio.h>
 int main()
{
int n, reverse=0, rem;
printf("Enter a number: ");
  scanf("%d", &n);
  while(n!=0)
  {
     rem=n%10;
     reverse=reverse*10+rem;
     n/=10;
  }
  printf("Reversed Number: %d",reverse);
return 0;
}
```

Output:

```
Enter a number: 123
Reversed Number: 321
```

8) Swap two numbers without using third variable

Write a c program to swap two numbers without using third variable.

We can swap two numbers without using third variable. There are two common ways to swap two numbers without using third variable:

1. By + and -
2. By * and /

Program 1: Using + and -

Let's see a simple c example to swap two numbers without using third variable.

```
#include<stdio.h>
 int main()
{
int a=10, b=20;
printf("Before swap a=%d b=%d",a,b);
a=a+b;//a=30 (10+20)
b=a-b;//b=10 (30-20)
a=a-b;//a=20 (30-10)
printf("\nAfter swap a=%d b=%d",a,b);
return 0;
}
```

Output:

```
Before swap a=10 b=20
After swap a=20 b=10
```

Program 2: Using * and /

Let's see another example to swap two numbers using * and /.

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

```
int a=10, b=20;
printf("Before swap a=%d b=%d",a,b);
a=a*b;//a=200 (10*20)
b=a/b;//b=10 (200/20)
a=a/b;//a=20 (200/10)
 system("cls");
printf("\nAfter swap a=%d b=%d",a,b);
return 0;
}
```

Output:

```
Before swap a=10 b=20
After swap a=20 b=10
```

9) Print "hello" without using semicolon

Write a c program to print "hello" without using semicolon

We can print "hello" or "hello world" or anything else in C without using semicolon. There are various ways to do so:

1. Using if
2. Using switch
3. Using loop etc.

Program 1: Using if statement

Let's see a simple c example to print "hello world" using if statement and without using semicolon.

```
#include<stdio.h>
 void main()
{
 if(printf("hello world")){}
}
```

Program 2: Using switch statement

Let's see a simple c example to print "hello world" using switch statement and without using semicolon.

```
#include<stdio.h>
 Void main()
{
switch(printf("hello world")){}
}
```

Program 3: Using while loop

Let's see a simple c example to print "hello world" using while loop and without using semicolon.

```
#include<stdio.h>
 void main()
{
   while(!printf("hello world")){}
}
```

10) Assembly Program in C

Write a c program to add two numbers using assembly code.

We can write assembly program code inside c language program. In such case, all the assembly code must be placed inside asm{} block.

Let's see a simple assembly program code to add two numbers in c program.

```
#include<stdio.h>
void main() {
  int a = 10, b = 20, c;

  asm {
    mov ax,a
    mov bx,b
    add ax,bx
    mov c,ax
  }

  printf("c= %d",c);
}
```

Output:

```
c= 30
```

11) C Program without main() function

Write a c program to print "Hello" without using main() function.

We can write c program without using main() function. To do so, we need to use #define preprocessor directive.

Let's see a simple program to print "hello" without main() function.

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

Output:

```
Hello
```

12) Decimal to Binary

Write a c program to convert decimal number to binary.

Decimal to binary in C: We can convert any decimal number (base-10 (0 to 9)) into binary number(base-2 (0 or 1)) by c program.

Decimal Number

Decimal number is a base 10 number because it ranges from 0 to 9, there are total 10 digits between 0 to 9. Any combination of digits is decimal number such as 23, 445, 132, 0, 2 etc.

Binary Number

Binary number is a base 2 number because it is either 0 or 1. Any combination of 0 and 1 is binary number such as 1001, 101, 11111, 101010 etc.

Let's see the some binary numbers for the decimal number.

Decimal	Binary
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010

Decimal to Binary Conversion Algorithm

- Step 1: Divide the number by 2 through % (modulus operator) and store the remainder in array
- Step 2: Divide the number by 2 through / (division operator)
- Step 3: Repeat the step 2 until number is greater than 0

Let's see the c example to convert decimal to binary.

```
#include<stdio.h>
#include<stdlib.h>
int main(){
int a[10],n,i;
system ("cls");
printf("Enter the number to convert: ");
scanf("%d",&n);
for(i=0;n>0;i++)
{
a[i]=n%2;
n=n/2;
}
printf("\nBinary of Given Number is=");
for(i=i-1;i>=0;i--)
{
printf("%d",a[i]);
}
return 0;
}
```

Output:

```
Enter the number to convert: 5
Binary of Given Number is=101
```

13) Alphabet Triangle

Write a c program to print alphabet triangle.

There are different triangles that can be printed. Triangles can be generated by alphabets or numbers. In this c program, we are going to print alphabet triangles.

Let's see the c example to print alphabet triangle.

```
#include<stdio.h>
#include<stdlib.h>
int main(){
  int ch=65;
    int i,j,k,m;
  system("cls");
    for(i=1;i<=5;i++)
    {
        for(j=5;j>=i;j--)
            printf(" ");
        for(k=1;k<=i;k++)
            printf("%c",ch++);
            ch--;
        for(m=1;m<i;m++)
            printf("%c",--ch);
        printf("\n");
        ch=65;
    }
return 0;
}
```

Output:

```
     A
    ABA
   ABCBA
  ABCDCBA
 ABCDEDCBA
```

14) Number Triangle

Write a c program to print number triangle.

Like alphabet triangle, we can write the c program to print the number triangle. The number triangle can be printed in different ways.

Let's see the c example to print number triangle.

```
#include<stdio.h>
#include<stdlib.h>
int main(){
  int i,j,k,l,n;
system("cls");
printf("enter the range=");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
for(j=1;j<=n-i;j++)
{
printf(" ");
}
for(k=1;k<=i;k++)
{
printf("%d",k);
}
for(l=i-1;l>=1;l--)
{
printf("%d",l);
}
printf("\n");
}
return 0;
}
```

Output:

```
enter the range= 4
   1
  121
 12321
1234321
```

15) Fibonacci Triangle

Write a c program to generate fibonacci triangle.

In this program, we are getting input from the user for the limit for fibonacci triangle, and printing the fibonacci series for the given number of times (limit).

Let's see the c example to generate fibonacci triangle.

```
#include<stdio.h>
#include<stdlib.h>
int main(){
  int a=0,b=1,i,c,n,j;
system("cls");
   printf("Enter the limit:");
   scanf("%d",&n);
   for(i=1;i<=n;i++)
   {
      a=0;
      b=1;
      printf("%d\t",b);
      for(j=1;j<i;j++)
      {
         c=a+b;
         printf("%d\t",c);
         a=b;
         b=c;

      }
      printf("\n");
   }
return 0;
}
```

Output:

```
Enter the limit:9
1
1       1
1       1       2
1       1       2       3
1       1       2       3       5
1       1       2       3       5       8
1       1       2       3       5       8       13
1       1       2       3       5       8       13      21
1       1       2       3       5       8       13      21      34



Enter the limit:5
1
1       1
1       1       2
1       1       2       3
1       1       2       3       5
```

16) Number in Characters

Write a c program to convert number in characters.

Number in characters conversion: In c language, we can easily convert number in characters by the help of loop and switch case. In this program, we are taking input from the user and iterating this number until it is 0. While iteration, we are dividing it by 10 and the remainder is passed in switch case to get the word for the number.

Let's see the c program to convert number in characters.

```
#include<stdio.h>
#include<stdlib.h>
int main(){
long int n,sum=0,r;
system("cls");
printf("enter the number=");
scanf("%ld",&n);
while(n>0)
{
r=n%10;
sum=sum*10+r;
n=n/10;
}
n=sum;
while(n>0)
{
r=n%10;
switch(r)
{
case 1:
printf("one ");
break;
case 2:
printf("two ");
break;
case 3:
printf("three ");
break;
case 4:
printf("four ");
break;
```

```
case 5:
printf("five ");
break;
case 6:
printf("six ");
break;
case 7:
printf("seven ");
break;
case 8:
printf("eight ");
break;
case 9:
printf("nine ");
break;
case 0:
printf("zero ");
break;
default:
printf("tttt");
break;
}
n=n/10;
}
return 0;
}
```

Output:

```
enter the number=4321
four three two one
```