C LANGUAGE

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

Mother language: 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.

System programming language: It can be used to do low-level programming. It is generally used to create hardware devices, OS, drivers, kernels, etc. For example, Linux kernel is written in C.

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.

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.

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

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

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

History of C Language

History of C language is interesting to know. Here we are going to discuss brief history of c language.

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

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

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

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

Features of C Language

C is the widely used language. It provides a lot of **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 **structured approach** (to break the problem into parts), **rich set of library functions**, **data types** etc.

2) Machine Independent or Portable

Unlike assembly language, c programs **can be executed in many machines** with little bit or no change. But it is not platform-independent.

3) Mid-level programming language

C is **also used to do low level programming**. It is used to develop system applications such as kernel, driver etc. It **also supports the feature of high level language**. That is why it is known as mid-level language.

4) Structured prorgramming 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.

5) Rich Library

C **provides a lot of inbuilt functions** that makes 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.

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.

10) Extensible

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

First C Program

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

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

#include <stdio.h> includes the **standard input output** library functions. The printf() function is defined in stdio.h .

int main() The **main() function is the entry point of every program** in c language.

printf() The printf() function is **used to print data** on the console.

return 0 The return 0 statement, returns execution status to the OS. The 0 value is used for successful execution and 1 for unsuccessful execution.

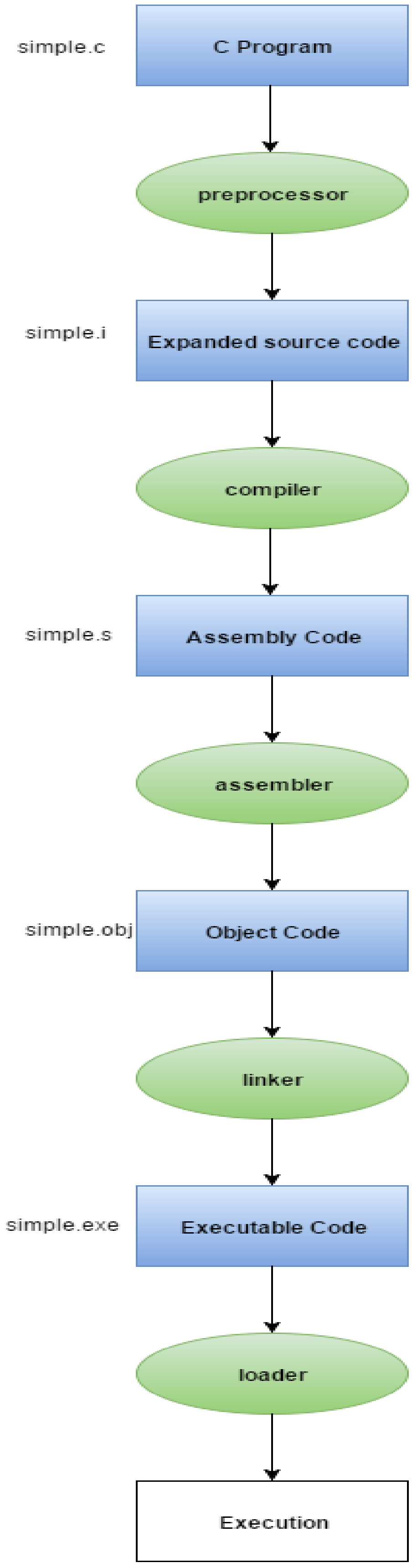
How to compile and run the c program

Compile for F9

Run for F10

Compile & run for F11

Execution Flow


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

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

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);
```

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 sum 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
```

Program to print cube of 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
```

Tokens in C

A C program consists of various tokens and a token is either a keyword, an identifier, a constant, a string literal, or a symbol.

C tokens are of six types. They are,

1. Keywords	(eg: int, while),
2. Identifiers	(eg: main, total),
3. Constants	(eg: 10, 20),
4. Strings	(eg: “total”, “hello”),
5. Special symbols	(eg: (), {}),
6. Operators	(eg: +, /,-,*)

C TOKENS EXAMPLE PROGRAM:

```
int main()
{
  int x, y, total;
  x = 10, y = 20;
  total = x + y;
```

```
   printf ("Total = %d \n", total);
}
```

- main – identifier
- {,}, (,) – delimiter
- int – keyword
- x, y, total – identifier
- main, {, }, (,), int, x, y, total – tokens

Identifiers

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

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

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

Keywords

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

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

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**
Integer 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", "srit" 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) C 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. The #define preprocessor directive is used to define constant or micro substitution. It can use any basic data type.

Syntax:

1. #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
```

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

Semicolons

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

Given below are two different statements −

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

return 0;
```

Whitespace in C

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

```
int age;
```

there must be at least one whitespace character (usually a space) between int and age for the compiler to be able to distinguish them.

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 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 single line comment in C.

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

Output:

```
Hello C
```

Even you can place comment after statement. For example:

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

Mult 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 multi line comment in C.

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

Output:

```
Hello C
```

C – Data Types

C data types are defined as the data storage format that a variable can store a data to perform a specific operation.

Data types are used to define a variable before to use in a program.

Size of variable, constant and array are determined by data types.

There are four data types in C language. They are,

Types	Data Types
Basic data types	int, char, float, double

Enumeration data type	enum
Derived data type	pointer, array, structure, union
Void data type	void

1.BASIC DATA TYPES IN C LANGUAGE:

1.1. CHARACTER DATA TYPE:

- Character data type allows a variable to store only one character.
- Storage size of character data type is 1. We can store only one character using character data type.
- “char” keyword is used to refer character data type.
- For example, ‘A’ can be stored using char datatype. You can’t store more than one character using char data type.
- Please refer C – Strings topic to know how to store more than one characters in a variable.

1.2.INTEGER DATA TYPE:

- Integer data type allows a variable to store numeric values.
- “int” keyword is used to refer integer data type.
- The storage size of int data type is 2 or 4 or 8 byte.
- It varies depend upon the processor in the CPU that we use. If we are using 16 bit processor, 2 byte (16 bit) of memory will be allocated for int data type.
- Like wise, 4 byte (32 bit) of memory for 32 bit processor and 8 byte (64 bit) of memory for 64 bit processor is allocated for int datatype.
- int (2 byte) can store values from -32,768 to +32,767
- int (4 byte) can store values from -2,147,483,648 to +2,147,483,647.
- If you want to use the integer value that crosses the above limit, you can go for “long int” and “long long int” for which the limits are very high.

Note:

- We can’t store decimal values using int data type.
- If we use int data type to store decimal values, decimal values will be truncated and we will get only whole number.
- In this case, float data type can be used to store decimal values in a variable.

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648

		to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

1.3. FLOATING POINT DATA TYPE:

Floating point data type consists of 3 types. They are,

1. float
2. double
3. long double

1. FLOAT:

- Float data type allows a variable to store decimal values.
- Storage size of float data type is 4bytes. This also varies depend upon the processor in the CPU as “int” data type.
- We can use up-to 6 digits after decimal using float data type.
- For example, 10.456789 can be stored in a variable using float data type.
- The range for float data type is from 1.2E-38 to 3.4E+38

2. DOUBLE:

- Double data type is also same as float data type which allows up-to 15 digits after decimal.
- Storage size of float data type is 8bytes

- The range for double data type is from 2.3E-308 to 1.7E+308

3. LONGE DOUBLE:

- Long double data type is also same as float data type which allows up-to 19 digits after decimal.
- Storage size of float data type is 10bytes
- The range for double data type is from 3.4E-4932 to 1.1E+4932

SIZEOF() FUNCTION IN C LANGUAGE:

sizeof() function is used to find the memory space allocated for each C data types.

```
#include <stdio.h>
#include <limits.h>
int main()
{
   int a;
   char b;
   float c;
   double d;
   printf("Storage size for int data type:%d
\n",sizeof(a));
   printf("Storage size for char data type:%d
\n",sizeof(b));
```

```
   printf("Storage size for float data type:%d
\n",sizeof(c));
   printf("Storage size for double data
type:%d\n",sizeof(d));
   return 0;
}
```

2. ENUMERATION DATA TYPE IN C LANGUAGE:

- Enumeration data type consists of named integer constants as a list.
- It start with 0 (zero) by default and value is incremented by 1 for the sequential identifiers in the list.

Enum syntax in C:

```
    enum identifier [optional{ enumerator-list }];
```

Enum example in C:

```
enum month { Jan, Feb, Mar }; or
/* Jan, Feb and Mar variables will be assigned to 0,
1 and 2 respectively by default */
enum month { Jan = 1, Feb, Mar };
/* Feb and Mar variables will be assigned to 2 and 3
respectively by default */
```

```
#include <stdio.h>
int main()
{
  enum MONTH { Jan = 0, Feb, Mar };
  enum MONTH month = Mar;
  if(month == 0)
  printf("Value of Jan");
  else if(month == 1)
  printf("Month is Feb");
  if(month == 2)
  printf("Month is Mar");
}
```

Output:

```
Month is March
```

3. DERIVED DATA TYPE IN C LANGUAGE:

Array, pointer, structure and union are called derived data type in C language.

To know more about derived data types, please visit "C – Array" , "C – Pointer" , "C – Structure" and "C – Union" topics .

4. VOID DATA TYPE IN C LANGUAGE:

Void is an empty data type that has no value.

This can be used in functions and pointers.

Please visit “C – Function” topic to know how to use void data type in function with simple call by value and call by reference example programs.

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:

```
datatype variable_list;
```

The example of declaring the variable is given below:

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

Initializing a variable means to provide it with a value. A variable can be initialized and defined in a single statement, like

```
    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 main(){
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 x=20;//global variable
void main(){
int y=10;//local variable
printf(“value of x=%d”,x);
printf(“value of y=%d”,y);
}
```

Static Variable

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

It retains its value between multiple function calls.

```
#include<stdio.h>
void main()
{
        int i;
        for(i=0;i<5;i++)
        {
         int x=10;//local variable
         static int y=10;//static variable
         x=x+1;
         y=y+1;
         printf("\n x=%d,y=%d",x,y);
        }
}
```

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

Automatic Variable

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

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

External Variable

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

myfile.h

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

program1.c

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

C – Operators

- The symbols which are used to perform logical and mathematical operations in a C program are called C operators.
- These C operators join individual constants and variables to form expressions.
- Operators, functions, constants and variables are combined together to form expressions.
- Consider the expression A + B * 5. where, +, * are operators, A, B are variables, 5 is constant and A + B * 5 is an expression.

TYPES OF C OPERATORS:

1. Arithmetic operators
2. Assignment operators
3. Relational operators
4. Logical operators
5. Bit wise operators
6. Conditional operators (ternary operators)
7. Increment/decrement operators
8. Special operators

Arithmetic Operators

C Arithmetic operators are used to perform mathematical calculations like addition, subtraction, multiplication, division and modulus in C programs.

Arithmetic Operators/Operation	**Example**
+ (Addition)	A+B
– (Subtraction)	A-B
* (multiplication)	A*B
/ (Division)	A/B
% (Modulus)	A%B

EXAMPLE PROGRAM FOR C ARITHMETIC OPERATORS:

In this example program, two values “40” and “20” are used to perform arithmetic operations such as addition, subtraction, multiplication, division, modulus and output is displayed for each operation.

```
#include <stdio.h>

int main()
{
  int a=40,b=20, add,sub,mul,div,mod;
  add = a+b;
  sub = a-b;
  mul = a*b;
  div = a/b;
  mod = a%b;
  printf("Addition of a, b is : %d\n", add);
  printf("Subtraction of a, b is : %d\n", sub);
  printf("Multiplication of a, b is : %d\n", mul);
  printf("Division of a, b is : %d\n", div);
  printf("Modulus of a, b is : %d\n", mod);
}
```

OUTPUT:

```
Addition of a, b is : 60
Subtraction of a, b is : 20
Multiplication of a, b is : 800
```

```
Division of a, b is : 2
Modulus of a, b is : 0
```

RELATIONAL OPERATORS

Relational operators are used to find the relation between two variables. i.e. to compare the values of two variables in a C program.

Operators	Example/Description
>	x > y (x is greater than y)
<	x < y (x is less than y)
>=	x >= y (x is greater than or equal to y)
<=	x <= y (x is less than or equal to y)
==	x == y (x is equal to y)

!=	x != y (x is not equal to y)

EXAMPLE PROGRAM FOR RELATIONAL OPERATORS IN C:

- In this program, relational operator (==) is used to compare 2 values whether they are equal are not.
- If both values are equal, output is displayed as ” values are equal”. Else, output is displayed as “values are not equal”.
- Note : double equal sign (==) should be used to compare 2 values. We should not single equal sign (=).

```
#include <stdio.h>

main() {

   int a = 21;
   int b = 10;
   int c ;
   if( a == b ) {
      printf("Line 1 - a is equal
to b\n" );
   } else {
      printf("Line 1 - a is not
equal to b\n" );
   }

   if ( a < b ) {
      printf("Line 2 - a is less
than b\n" );
   } else {
      printf("Line 2 - a is not
less than b\n" );
   }

   if ( a > b ) {
      printf("Line 3 - a is
greater than b\n" );
   } else {
      printf("Line 3 - a is not
greater than b\n" );
   }

   /* Lets change value of a and b
*/
   a = 5;
   b = 20;

   if ( a <= b ) {
      printf("Line 4 - a is either
less than or equal to  b\n" );
   }

   if ( b >= a ) {
      printf("Line 5 - b is either
greater than  or equal to b\n" );
   }
}
```

result –

```
Line 1 - a is not equal to b
Line 2 - a is not less than b
Line 3 - a is greater than b
Line 4 - a is either less than or equal to  b
Line 5 - b is either greater than  or equal to b
```

LOGICAL OPERATORS

- These operators are used to perform logical operations on the given expressions.
- There are 3 logical operators in C language. They are, logical AND (&&), logical OR (||) and logical NOT (!).

Operators	Example/Description
&& (logical AND)	(x>5)&&(y<5) It returns true when both conditions are true
|| (logical OR)	(x>=10)||(y>=10)

	It returns true when at-least one of the condition is true
! (logical NOT)	!((x>5)&&(y<5)) It reverses the state of the operand “((x>5) && (y<5))” If “((x>5) && (y<5))” is true, logical NOT operator makes it false

Example

```
#include <stdio.h>

int main()
{
   int m=40,n=20;
   int o=20,p=30;
   if (m>n && m !=0)
   {
      printf("&& Operator : Both conditions are
true\n");
   }
   if (o>p || p!=20)
   {
      printf("|| Operator : Only one condition is
true\n");
   }
   if (!(m>n && m !=0))
   {
      printf("! Operator : Both conditions are true\n");
   }
   else
   {
      printf("! Operator : Both conditions are true. " \
      "But, status is inverted as false\n");
   }
}
```

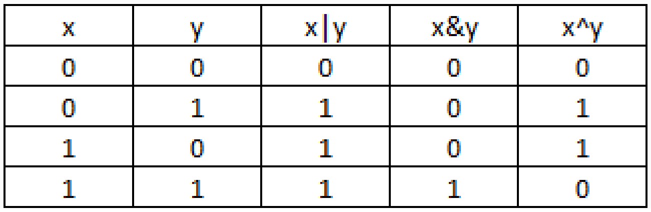
OUTPUT:

```
&& Operator : Both conditions are true
|| Operator : Only one condition is true
! Operator : Both conditions are true. But, status is
inverted as false
```

BIT WISE OPERATORS

- These operators are used to perform bit operations. Decimal values are converted into binary values which are the sequence of bits and bit wise operators work on these bits.
- Bit wise operators in C language are & (bitwise AND), | (bitwise OR), ~ (bitwise NOT), ^ (XOR), << (left shift) and >> (right shift).

TRUTH TABLE FOR BIT WISE OPERATION & BIT WISE OPERATORS:



x	y	x|y	x&y	x^y
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

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

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

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

Show Examples

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

		0011 0001
~	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = ~(60), i.e,. -0111101
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

```
#include <stdio.h>


main() {


   unsigned int a = 60; /* 60 =
0011 1100 */
   unsigned int b = 13; /* 13 =
0000 1101 */
   int c = 0;


   c = a & b;       /* 12 = 0000
1100 */
   printf("Line 1 - Value of c is
%d\n", c );


   c = a | b;       /* 61 = 0011
1101 */
   printf("Line 2 - Value of c is
%d\n", c );
```

```
   c = a ^ b;       /* 49 = 0011
0001 */
   printf("Line 3 - Value of c is
%d\n", c );

   c = ~a;          /*-61 = 1100
0011 */
   printf("Line 4 - Value of c is
%d\n", c );

   c = a << 2;     /* 240 = 1111
0000 */
   printf("Line 5 - Value of c is
%d\n", c );

   c = a >> 2;     /* 15 = 0000
1111 */
   printf("Line 6 - Value of c is
%d\n", c );
}
```

When you compile and execute the above program, it produces the following result −

```
Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
Line 6 - Value of c is 15
```

ASSIGNMENT OPERATORS

- In C programs, values for the variables are assigned using assignment operators.
- For example, if the value “10” is to be assigned for the variable “sum”, it can be assigned as “sum = 10;”
- There are 2 categories of assignment operators in C language. They are,
  1. Simple assignment operator (Example: =)
  2. Compound assignment operators (Example: +=, -=, *=, /=, %=, &=, ^=)

Operator	Description	Example
=	Simple assignment	C = A + B will assign

	operator. Assigns values from right side operands to left side operand	the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand	C -= A is equivalent to C = C - A

	from the left operand and assigns the result to the left operand.	
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right	C /= A is equivalent to C = C / A

	operand and assigns the result to the left operand.	
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2

&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Example

```
#include <stdio.h>

main() {

   int a = 21;
   int c ;

   c =  a;
   printf("Line 1 - =  Operator
Example, Value of c = %d\n", c );

   c +=  a;
   printf("Line 2 - += Operator
Example, Value of c = %d\n", c );

   c -=  a;
   printf("Line 3 - -= Operator
Example, Value of c = %d\n", c );

   c *=  a;
   printf("Line 4 - *= Operator
Example, Value of c = %d\n", c );

   c /=  a;
   printf("Line 5 - /= Operator
Example, Value of c = %d\n", c );


   c  = 200;

   c %=  a;

   printf("Line 6 - %= Operator
Example, Value of c = %d\n", c );


   c <<=  2;

   printf("Line 7 - <<= Operator
Example, Value of c = %d\n", c );


   c >>=  2;

   printf("Line 8 - >>= Operator
Example, Value of c = %d\n", c );


   c &=  2;

   printf("Line 9 - &= Operator
Example, Value of c = %d\n", c );
```

```
   c ^=  2;
   printf("Line 10 - ^= Operator
Example, Value of c = %d\n", c );


   c |=  2;
   printf("Line 11 - |= Operator
Example, Value of c = %d\n", c );
}
```

When you compile and execute the above program, it produces the following result −

```
Line 1 - =  Operator Example, Value of c = 21
Line 2 - += Operator Example, Value of c = 42
Line 3 - -= Operator Example, Value of c = 21
Line 4 - *= Operator Example, Value of c = 441
Line 5 - /= Operator Example, Value of c = 21
Line 6 - %= Operator Example, Value of c = 11
Line 7 - <<= Operator Example, Value of c = 44
Line 8 - >>= Operator Example, Value of c = 11
Line 9 - &= Operator Example, Value of c = 2
Line 10 - ^= Operator Example, Value of c = 0
Line 11 - |= Operator Example, Value of c = 2
```

CONDITIONAL OR TERNARY OPERATORS

- Conditional operators return one value if condition is true and returns another value is condition is false.
- This operator is also called as ternary operator.

Syntax : (Condition? true_value: false_value);

Example : (A > 100 ? 0 : 1);

- In above example, if A is greater than 100, 0 is returned else 1 is returned. This is equal to if else conditional statements.

EXAMPLE

```
#include <stdio.h>

int main()
{
  int x=1, y ;
  y = ( x ==1 ? 2 : 0 ) ;
  printf("x value is %d\n", x);
  printf("y value is %d", y);
}
```

OUTPUT:

```
x value is 1
y value is 2
```

Increment/decrement Operators

- Increment operators are used to increase the value of the variable by one and decrement operators are used to decrease the value of the variable by one in C programs.
- Syntax:
  Increment operator: ++var_name; (or) var_name++;
  Decrement operator: – -var_name; (or) var_name – -;
- Example:
  Increment operator : ++ i ; i ++ ;
  Decrement operator : – – i ; i – – ;

Increment Operator in C Programming

1. Increment operator is used to increment the current value of variable by adding integer 1.
2. Increment operator can be applied to only variables.

3. Increment operator is denoted by ++.

Different Types of Increment Operation

In C Programming we have two types of increment operator i.e Pre-Increment and Post-Increment Operator.

A. Pre Increment Operator

Pre-increment operator is used to increment the value of variable before using in the expression. In the Pre-Increment value is first incremented and then used inside the expression.

```
b = ++y;
```

In this example suppose the value of variable ‘y’ is 5 then value of variable ‘b’ will be 6 because the value of ‘y’ gets modified before using it in a expression.

B. Post Increment Operator

Post-increment operator is used to increment the value of variable as soon as after executing expression completely in which post increment is used. In the Post-Increment value is first used in a expression and then incremented.

```
b = x++;
```

In this example suppose the value of variable ‘x’ is 5 then value of variable ‘b’ will be 5 because old value of ‘x’ is used.

Sample program

```
#include<stdio.h>


void main()
{
int a,b,x=10,y=10;


a = x++;
b = ++y;


printf("Value of a : %d",a);
printf("Value of b : %d",b);
}
```

Output :

```
Value of a : 10
Value of b : 11
```

decrement operator

Different Types of Decrement Operation :

When decrement operator used in C Programming then it can be used as pre-decrement or post-decrement operator.

A. Pre Decrement Operator

Pre-decrement operator is used to decrement the value of variable before using in the expression. In the Pre-decrement value is first decremented and then used inside the expression.

```
b = --var;
```

Suppose the value of variable var is 10 then we can say that value of variable ‘var’ is firstly decremented then updated value will be used in the expression.

B. Post Decrement Operator

Post-decrement operator is used to decrement the value of variable immediatly after executing expression completely in which post decrement is used. In the Post-decrement old value is first used in a expression and then old value will be decrement by 1.

```
b = var--;
```

Value of variable ‘var’ is 5. Same value will be used in expression and after execution of expression new value will be 4.

C Program

```
#include<stdio.h>


void main()
{
int a,b,x=10,y=10;


a = x--;
b = --y;


printf("Value of a : %d",a);
printf("Value of b : %d",b);


}
```

Output :

```
Value of a : 10
Value of b : 9
```

SPECIAL OPERATORS

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

Operators	Description
&	This is used to get the address of the variable. Example : &a will give

	address of a.
*	This is used as pointer to a variable. Example : * a where, * is pointer to the variable a.
Sizeof ()	This gives the size of the variable. Example : size of

	(char) will give us 1.

EXAMPLE PROGRAM FOR & AND * OPERATORS IN C:

In this program, “&” symbol is used to get the address of the variable and “*” symbol is used to get the value of the variable that the pointer is pointing to. Please refer **C – pointer** topic to know more about pointers.

```
#include <stdio.h>
int main()
{
int *ptr, q;
q = 50;
/* address of q is assigned to ptr */
ptr = &q;
/* display q's value using ptr variable */
printf("%d", *ptr);
return 0;
}
```

OUTPUT:

```
50
```

EXAMPLE PROGRAM FOR SIZEOF() OPERATOR IN C:

sizeof() operator is used to find the memory space allocated for each C data types.

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

int main()
{
int a;
char b;
float c;
double d;
printf("Storage size for int data type:%d
\n",sizeof(a));
printf("Storage size for char data type:%d
\n",sizeof(b));
printf("Storage size for float data type:%d
\n",sizeof(c));
printf("Storage size for double data
type:%d\n",sizeof(d));
return 0;
}
```

OUTPUT:

```
Storage size for int data
type:4
Storage size for char data
type:1
Storage size for float data
type:4
Storage size for double data
type:8
```

Decision Making

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.

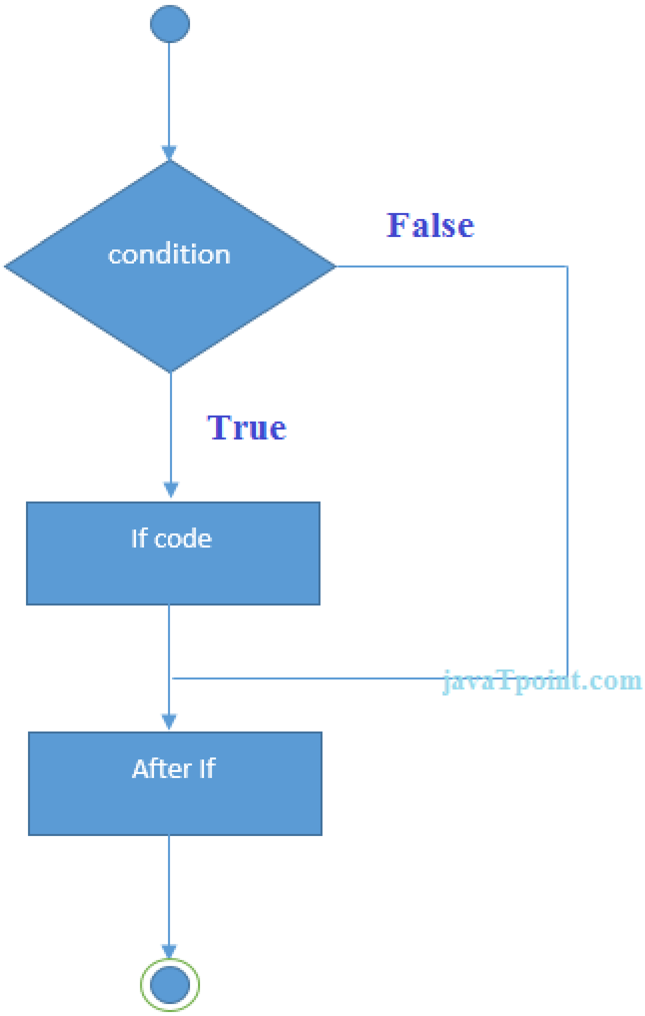
- If statement
- If-else statement
- If else-if ladder
- Nested if

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

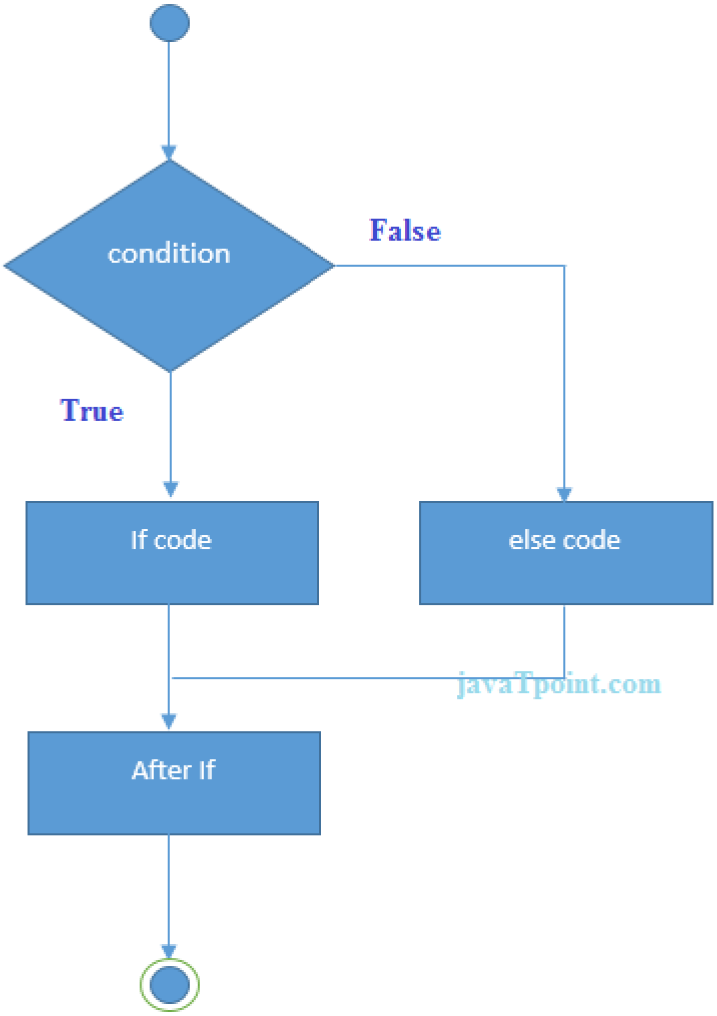
```
Enter three numbers?
12 23 34
34 is largest
```

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
```

f 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

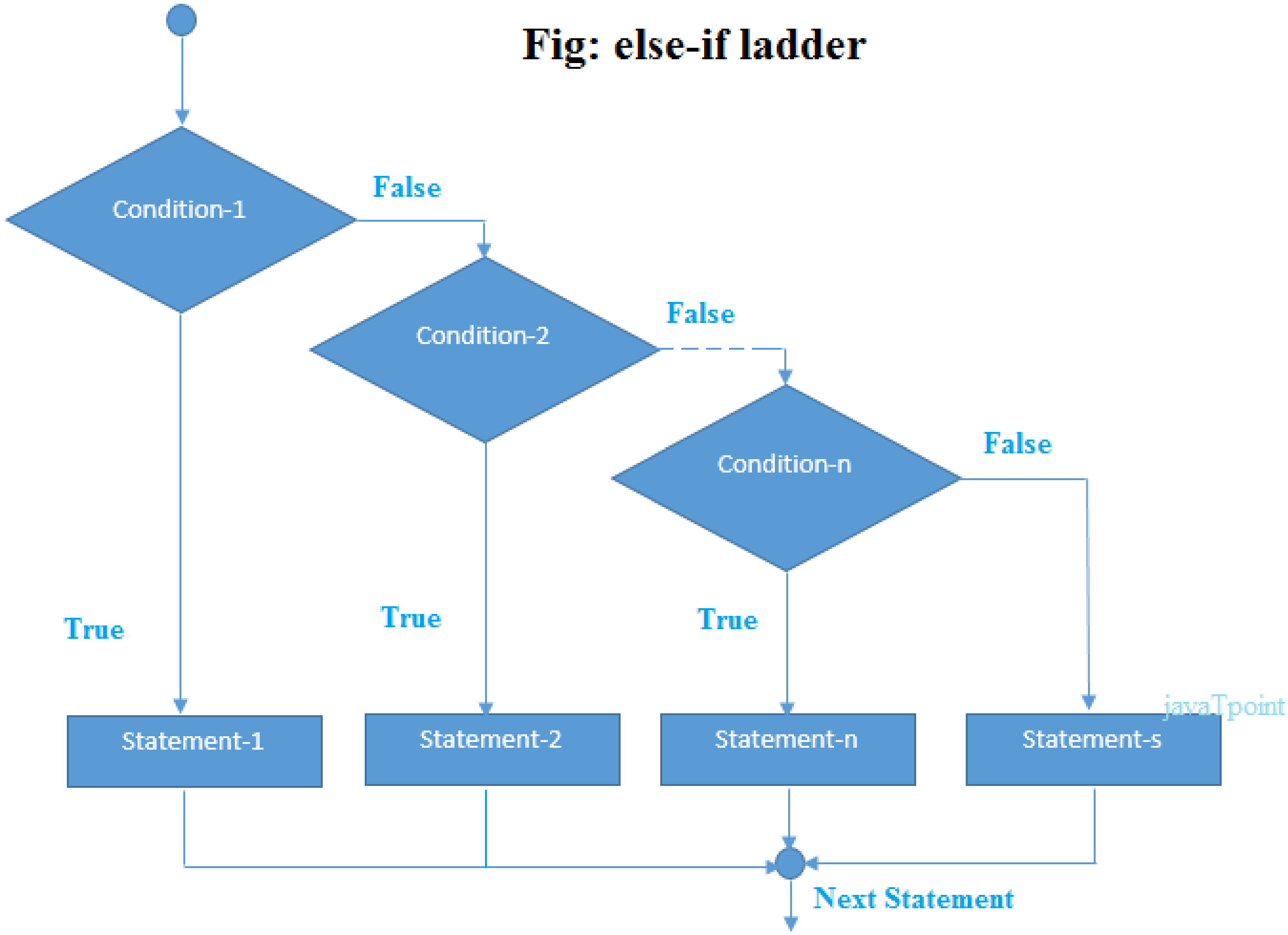


Fig: else-if ladder

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 ...
```

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 possibles 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.

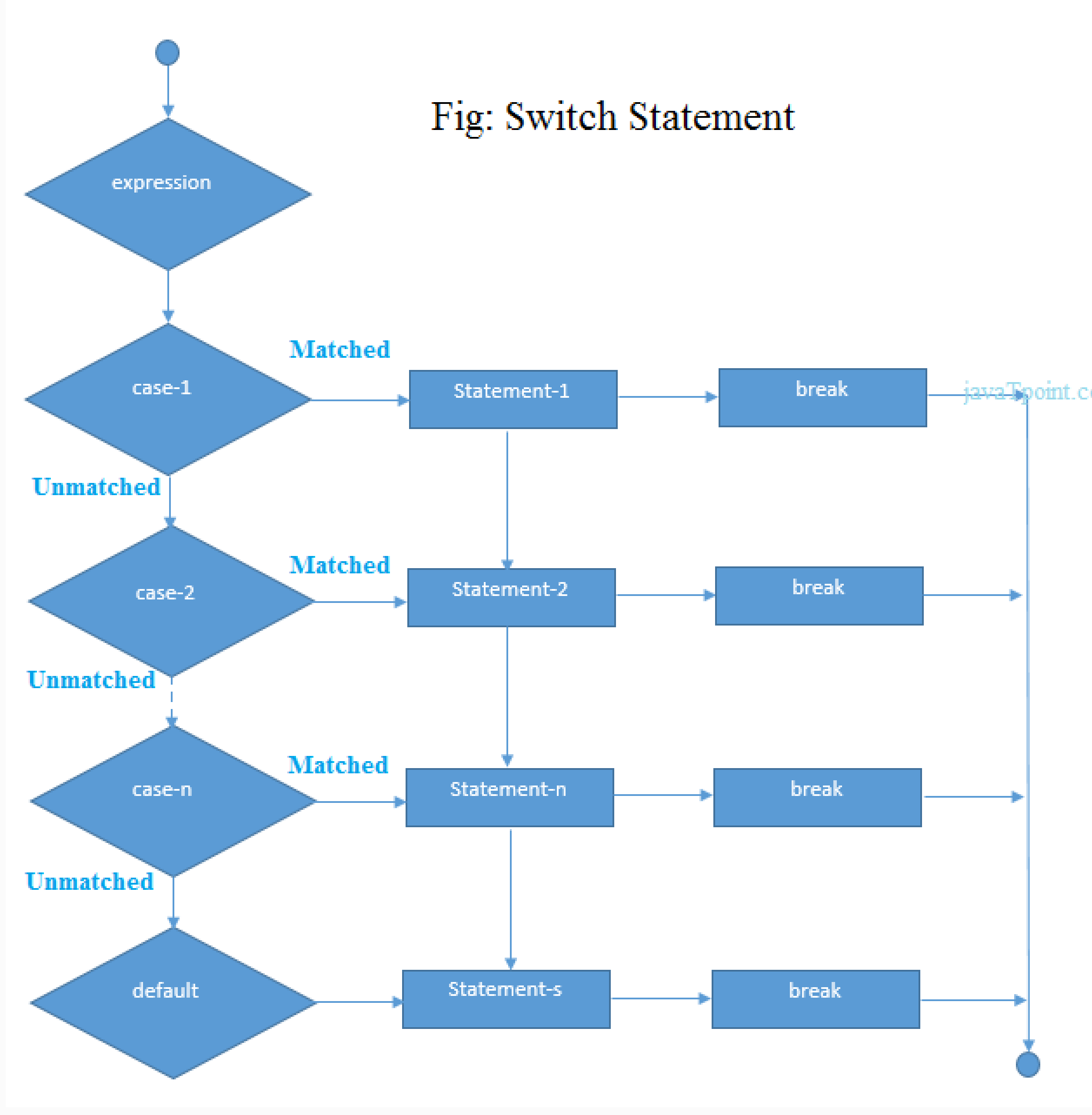
Let's try to understand it by the examples. We are assuming that there are following variables.

1. int x,y,z;
2. char a,b;
3. float f;

alid 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;

lowchart of switch statement in C



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

1. #include<stdio.h>
2. 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
```

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
```

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. for
2. while
3. do while

for loop in C

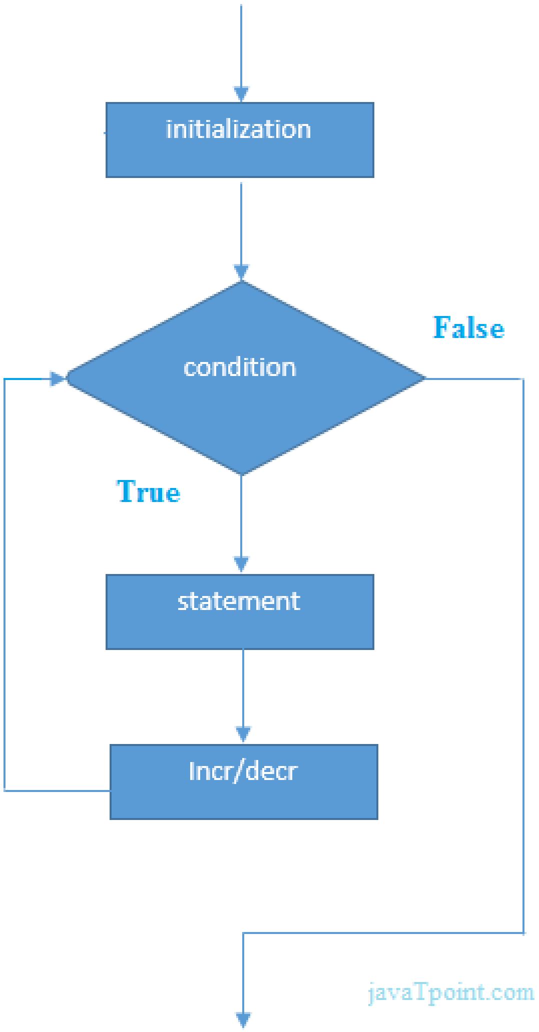
The for loop in C language is used to iterate the statements or a part of the program several times. It is frequently used to traverse the data structures like the array and linked list.

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.

```
1.#include<stdio.h>
2.int main(){
3.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);
    }
}
```

Output

```
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.

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

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

while loop in C

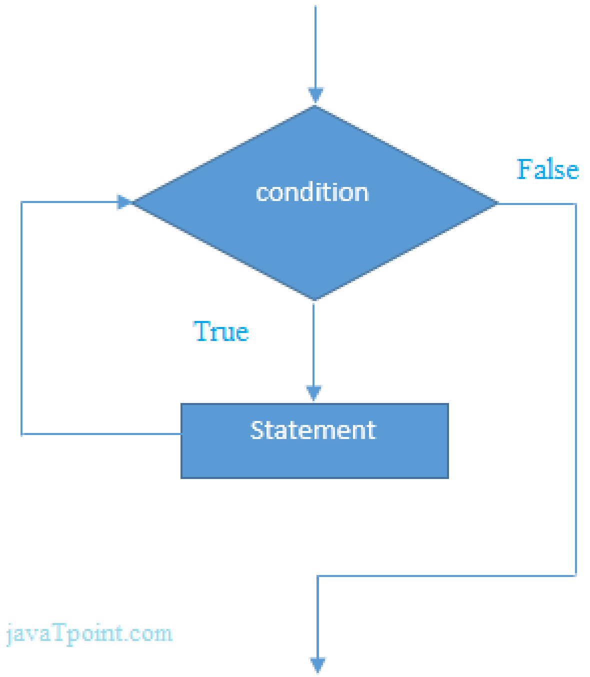
While loop is also known as a pre-tested loop. In general, a while loop allows a part of the code to be executed multiple times depending upon a given boolean condition. It can be viewed as a repeating if statement. The while loop is mostly used in the case where the number of iterations is not known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

Syntax of while loop in C language

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
```

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
}
```

do while loop in C

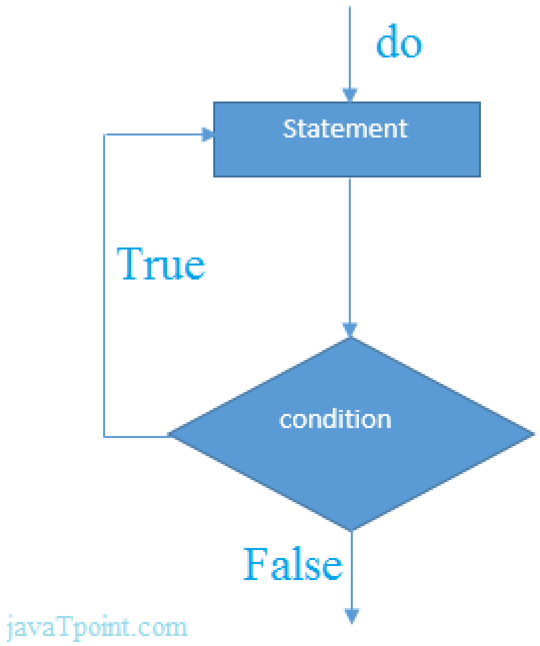
The do while loop is a post tested loop. Using the do-while loop, we can repeat the execution of several parts of the statements. The do-while loop is mainly used in the case where we need to execute the loop at least once. The do-while loop is mostly used in menu-driven programs where the termination condition depends upon the end user.

do while loop syntax

The syntax of the C language do-while loop is given below:

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

Flowchart of do while loop



Example 1

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

```
{
    char c;
    int choice,dummy;
    do{
    printf("\n1. Print Hello\n2. Print Javatpoint\n3. Exit\n");
    scanf("%d",&choice);
        switch(choice)
        {
           case 1 :
           printf("Hello");
           break;
           case 2:
           printf("Javatpoint");
           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 Javatpoint
3. Exit
1
Hello
do you want to enter more?
y

1. Print Hello
2. Print Javatpoint
3. Exit
2
Javatpoint
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);
```

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
```

2. break;

Flowchart of break in c

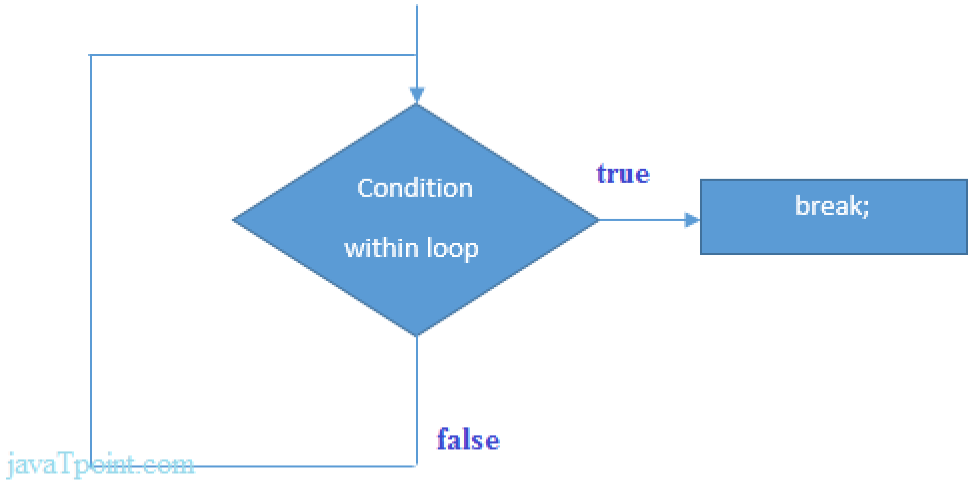


Figure: Flowchart of break statement

```
#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
```

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>
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.

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
```

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
```

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.

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;}

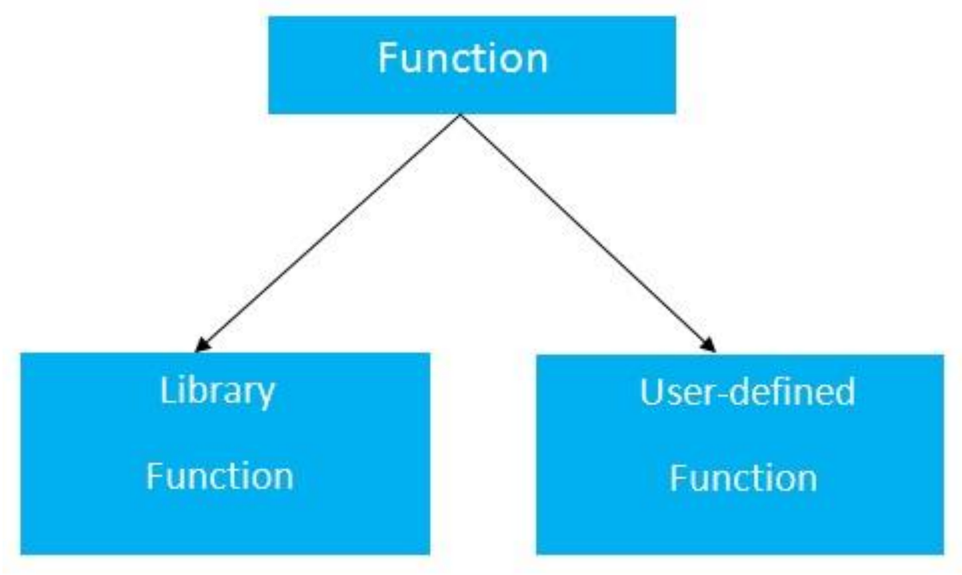
The syntax of creating function in c language is given below:

```
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:

```
1.     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("Sri Renuka");
}
```

Output

```
Hello Sri Renuka
```

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

Output

```
Going to calculate the sum of two
numbers:

Enter two numbers 10
24

The sum is 34
```

Example 2: program to calculate the area of the square

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

Output

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

Example for Function with argument and without return value

Example 1

```c
#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 x, int y)
{
   printf("\nThe sum is %d",x+y);
}
```

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 numbe
rs : %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 t
wo 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.

SN	Header file	Description
1	stdio.h	This is a standard input/output header file. It contains all the library functions regarding standard input/output.
2	conio.h	This is a console input/output header file.
3	string.h	It contains all string related library functions like gets(), puts(),etc.

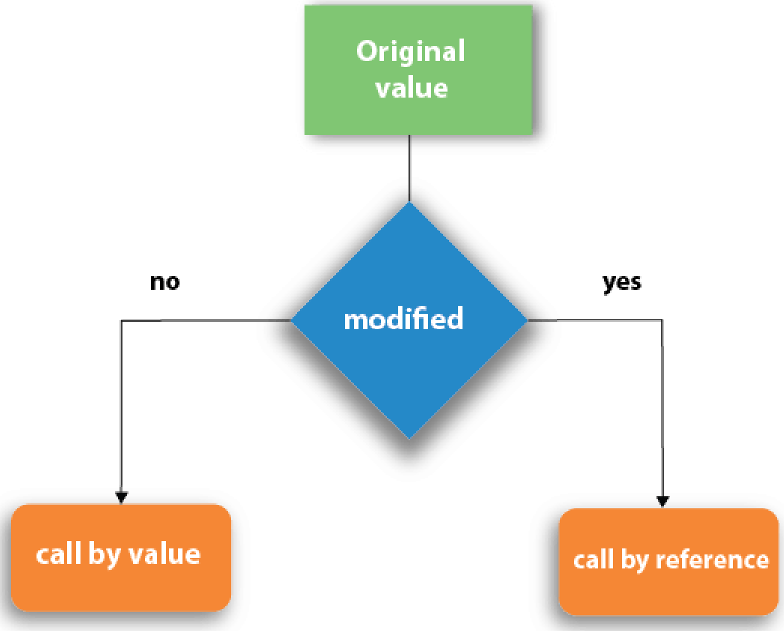
4	stdlib.h	This header file contains all the general library functions like malloc(), calloc(), exit(), etc.
5	math.h	This header file contains all the math operations related functions like sqrt(), pow(), etc.
6	time.h	This header file contains all the time-related functions.

7	ctype.h	This header file contains all character handling functions.
8	stdarg.h	Variable argument functions are defined in this header file.
9	signal.h	All the signal handling functions are defined in this header file.
10	setjmp.h	This file contains all the jump functions.
11	locale.h	This file contains locale functions.

12	errno.h	This file contains error handling functions.
13	assert.h	This file contains diagnostics functions.

Types of function

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



Call by value

In call by value method, the value of the actual parameters is copied into the formal parameters.

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.

```
#include<stdio.h>
void change(int num) {
    printf("Before adding value inside func
tion num=%d \n",num);
    num=num+100;
    printf("After adding value inside functi
on 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;
}
```

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

```
#include <stdio.h>
void swap(int , int); //prototype of the fu
nction
```

```
int main()
{
    int a = 10;
    int b = 20;
    printf("Before swapping the values in
main a = %d, b = %d\n",a,b); // printing t
he 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 ac
tual parameters do not change by changi
ng the formal parameters in call by value
, a = 10, b = 20
}
void swap (int a, int b)
{
    int temp;
    temp = a;
    a=b;
    b=temp;
    printf("After swapping values in functi
on a = %d, b = %d\n",a,b); // Formal para
meters, a = 20, b = 10
}
```

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.

```
#include<stdio.h>
void change(int *num) {
   printf("Before adding value inside func
tion num=%d \n",*num);
   (*num) += 100;
   printf("After adding value inside functi
on num=%d \n", *num);
}
```

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

return 0;
}
```

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

```
#include <stdio.h>
void swap(int *, int *); //prototype of th
e function
int main()
```

```
{
    int a = 10;
    int b = 20;
    printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and b in main
    swap(&a,&b);
    printf("After swapping values in main a = %d, b = %d\n",a,b); // The values of actual parameters do change in call by reference, a = 10, b = 20
}
void swap (int *a, int *b)
{
    int temp;
    temp = *a;
    *a=*b;
    *b=temp;
    printf("After swapping values in functi
on a = %d, b = %d\n",*a,*b); // Formal p
arameters, a = 20, b = 10
}
```

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.

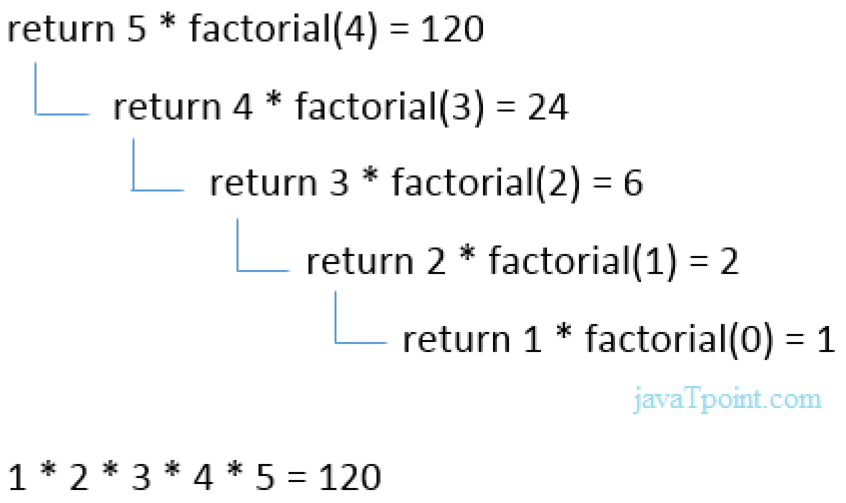
```
#include <stdio.h>
int fact (int);
int main()
{
   int n,f;
   printf("Enter the number whose factor
ial you want to calculate?");
   scanf("%d",&n);
   f = fact(n);
   printf("factorial = %d",f);
}
int fact(int n)
```

```
{
    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 figure 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
```

Fig: Recursion

Example of recursion in C

Let's see an example to find the nth term of the Fibonacci series.

```
#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);
    }
}
```

Output

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

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

---

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];
```

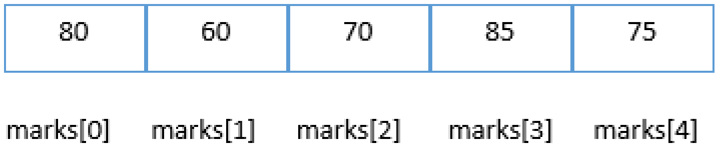
Here, int is the data_type, marks are the array_name, and 5 is the array_size.

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;
```



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

Initialization of Array

```
#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;
}
```

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;
}
```

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, 7
8, 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 element of the array.

```
#include<stdio.h>
void main ()
{
    int n;
    int arr[n],i,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];

    for(i=0;i<n;i++)
    {
        if(arr[i]>largest)
        {

            largest = arr[i];
        }

    }
    printf("largest = %d ",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.

Function_name(array_name);//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 arra
y 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[] point
s 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
```

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

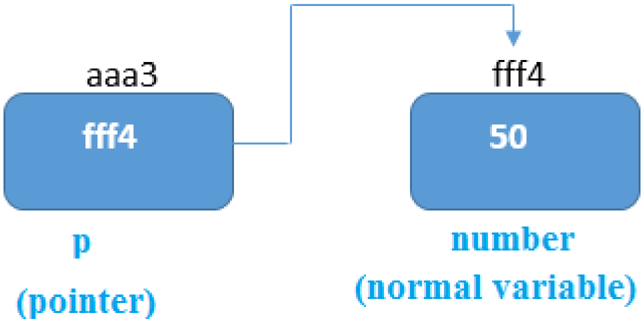
The pointer in c language can be declared using * (asterisk symbol). It is also known as indirection pointer used to dereference a pointer.

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.



javatpoint.com

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 derefere
nce a pointer therefore if we print *p, w
e 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.

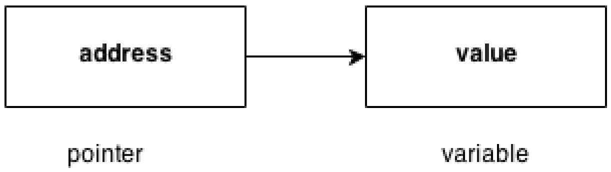
```
#include <stdio.h>


int main()
{
   int i;
   int a[5] = {1, 2, 3, 4, 5};
   int *p = a;     // same as int*p = &a[0]
   for (i = 0; i < 5; i++)
   {
      printf("%d", *p);
      p++;
   }

   return 0;
}
```

Pointer to a function

```
void show (int);
void(*p)(int) = &display; // Pointer p is p
ointing 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).

WRITE 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	Associativ
(), []	1	Left to rig
*, identifier	2	Right to le
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?

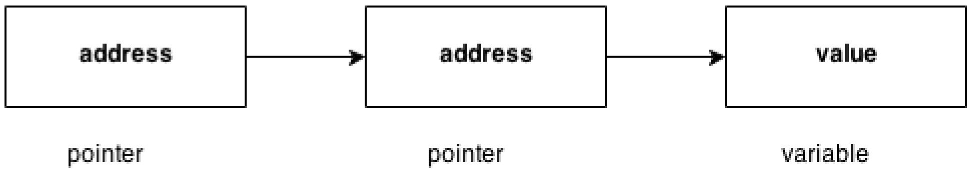
int (*p)(int (*)[2], int (*)void))

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.

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. 10 will be printed
    printf("value stored at pp: %d\n",**pp); // value stored at the address contained by the pointer stoyred at pp
}
```

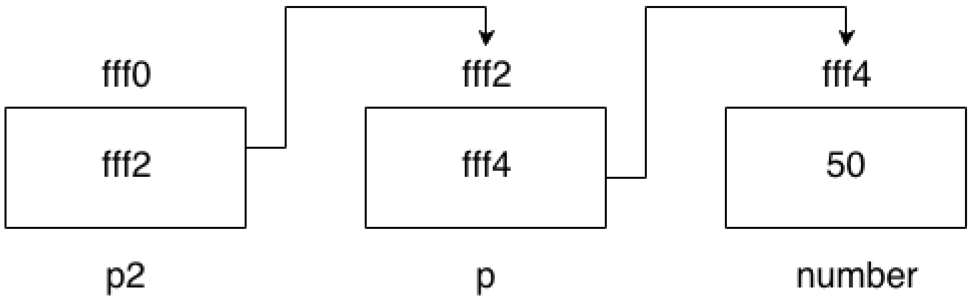
Output

```
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 num
ber 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",p
2);
printf("Value of **p2 variable is %d \n",*
p);
return 0;
```

```
}
```

Output

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

Pointer Arithmetic in C

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

Increment

Decrement

Addition

Subtraction

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 incremented by 4 bytes.
return 0;
}
```

Output

```
Address of p variable is 3214864300
After increment: Address of p variable is 3214864304
```

Traversing an array by using pointer

```
#include<stdio.h>
void main ()
{
   int arr[5] = {1, 2, 3, 4, 5};
```

```
    int *p = arr;
    int i;
    printf("printing array elements...\n");
    for(i = 0; i< 5; i++)
    {
        printf("%d  ",*(p+i));
    }
}
```

Output

```
printing array elements...
1  2  3  4  5
```

Decrementing Pointer in C

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

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 immidiate previous location.
}
```

Output

```
Address of p variable is 3214864300
After decrement: Address of p variable is 3214864296
```

C Pointer Addition

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

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
```

Strings

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

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

By char array

By string literal

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

```
char ch[10]={'s', 'r', 'i', 'r', 'e', 'n', 'u', 'k', '
a',  '\0'};
```

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

s	r	i	r	e	n	u	k	a	\0
0	1	2	3	4	5	6	7	8	9

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

```
char ch[]={'s', 'r', 'i', 'r', 'e', 'n', 'u', 'k', 'a',
'\0'};
```

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

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

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", ch
2);
 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] = "sri renuka";
    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",cou
nt);
}
```

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",cou
nt);
}
```

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.

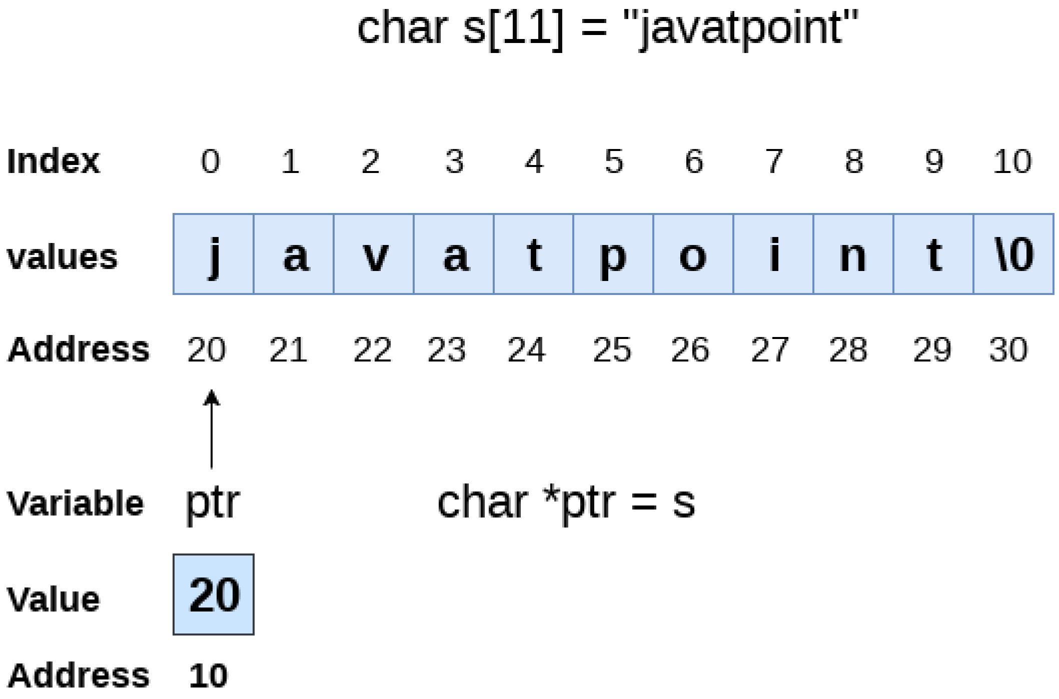
Pointers with strings

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 s
tring s.
    printf("%s",p); // the string javatpoint i
s printed if we print p.
}
```

Output

```
javatpoint
```



char s[11] = "javatpoint"

Index	0	1	2	3	4	5	6	7	8	9	10
values	j	a	v	a	t	p	o	i	n	t	\0
Address	20	21	22	23	24	25	26	27	28	29	30

Variable	ptr
Value	20
Address	10

char *ptr = s

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
```

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.

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
```

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
```

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
```

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
```

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(st
r));
 return 0;
}
```

Output:

```
Enter string: javatpoint
String is: javatpoint
Reverse String is: tnioptavaj
```

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
```

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
```

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
```

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.

malloc()

calloc()

realloc()

free()

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

static 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));  //memo
ry allocated using calloc
    if(ptr==NULL)
    {
        printf("Sorry! unable to allocate me
mory");
        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)
```

example:

```
#include <stdio.h>

#include <stdlib.h>

int main()

{

  int *ptr = (int *)malloc(sizeof(int)*2);

  int i;
```

```
   int *ptr_new;


   *ptr = 10;
   *(ptr + 1) = 20;


   ptr_new = (int *)realloc(ptr,
sizeof(int)*3);
   *(ptr_new + 2) = 30;
   for(i = 0; i < 3; i++)
       printf("%d ", *(ptr_new + i));


   return 0;
}
```

Output:

```
10 20 30
```

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)
```

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[3][10],dummy; // 2-dimensioanal character array names is used to store the names of the students
  int roll_numbers[3],i;
  float marks[3];
  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

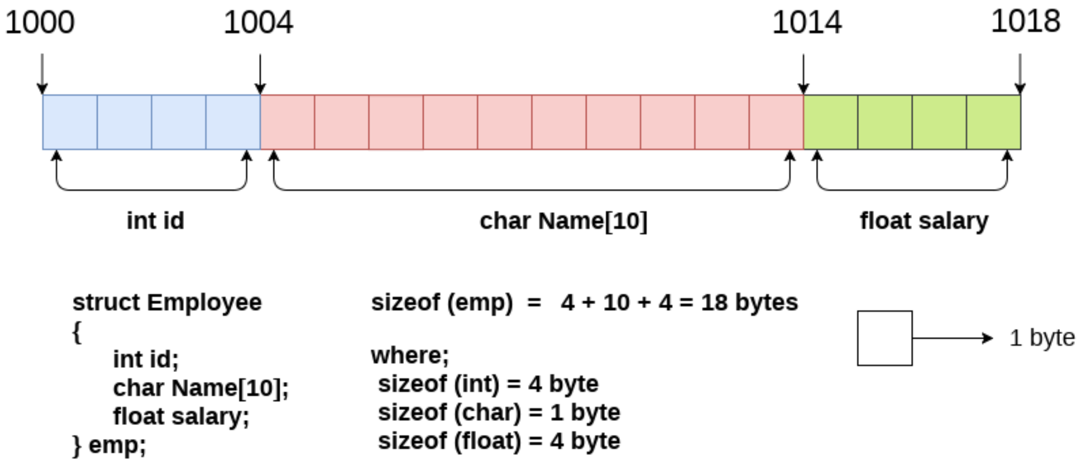
The **,struct** keyword is used to define the structure. Let's see the syntax to define the structure in c.

```
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[10];
    float salary;
};
```

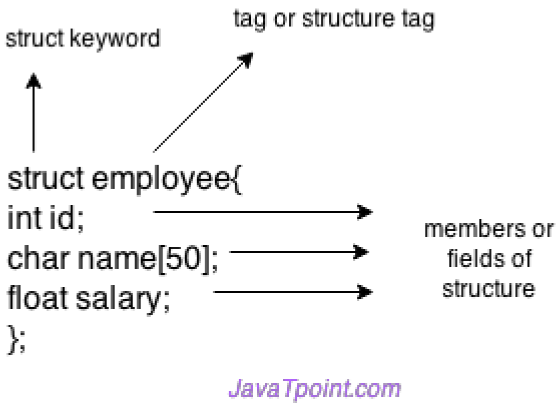
The following image shows the memory allocation of the structure employee that is defined in the above example.



```
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:



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

Declaring structure variable

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

By struct keyword within main() function

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:

By . (member or dot operator)

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 str
ucture
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
```

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 stu
dent 1 ");
    scanf("%s %d %f",s1.name,&s1.id,&s1.mark
s);
    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

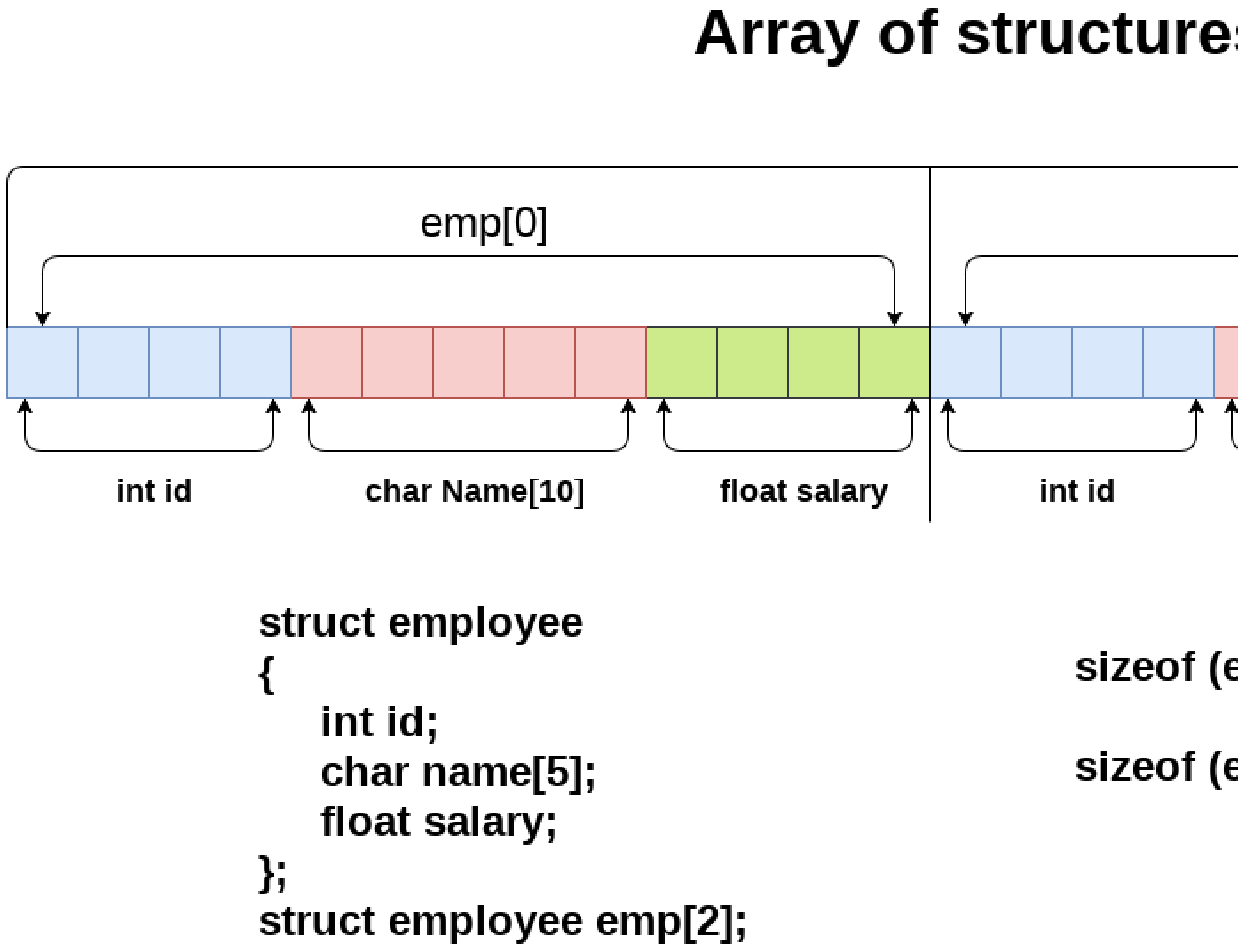
```
Enter the name, id, and marks
of student 1 James 90 90
Enter the name, id, and marks
of student 2 Adoms 90 90
Enter the name, id, and marks
of student 3 Nick 90 90
Printing the details....
```

```
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 structure

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

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;
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

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.

```c
#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.c

ity, &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.add.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.

By separate structure

By Embedded structure

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.

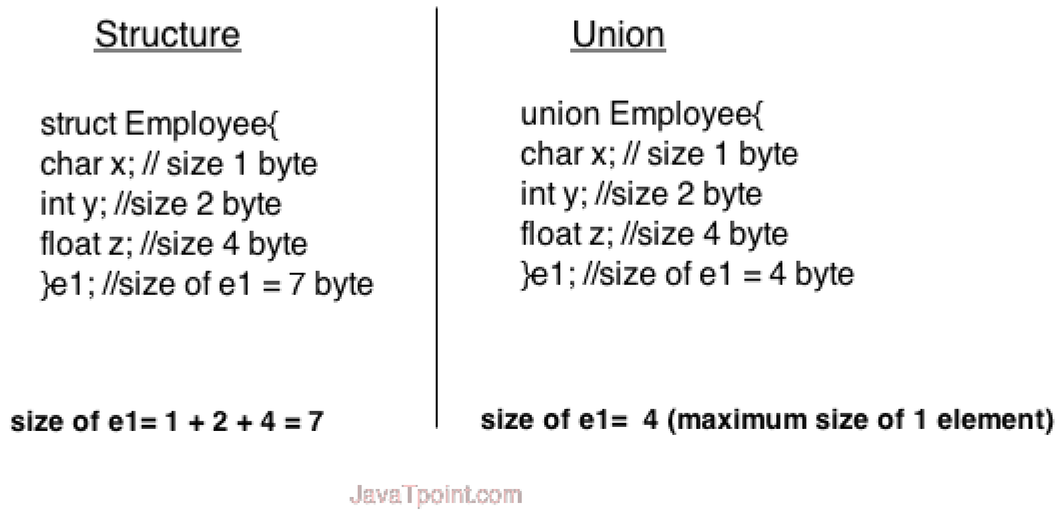
```
#include<stdio.h>
struct address
{
    char city[20];
    int pin;
    char phone[14];
};
struct employee
{
    char name[20];
    struct address add;
};
```

```
}
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

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

size of e1= 1 + 2 + 4 = 7

Union

```
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= 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
```

Programs

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.

```
#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;
 }
```

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.

```
#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;
}
```

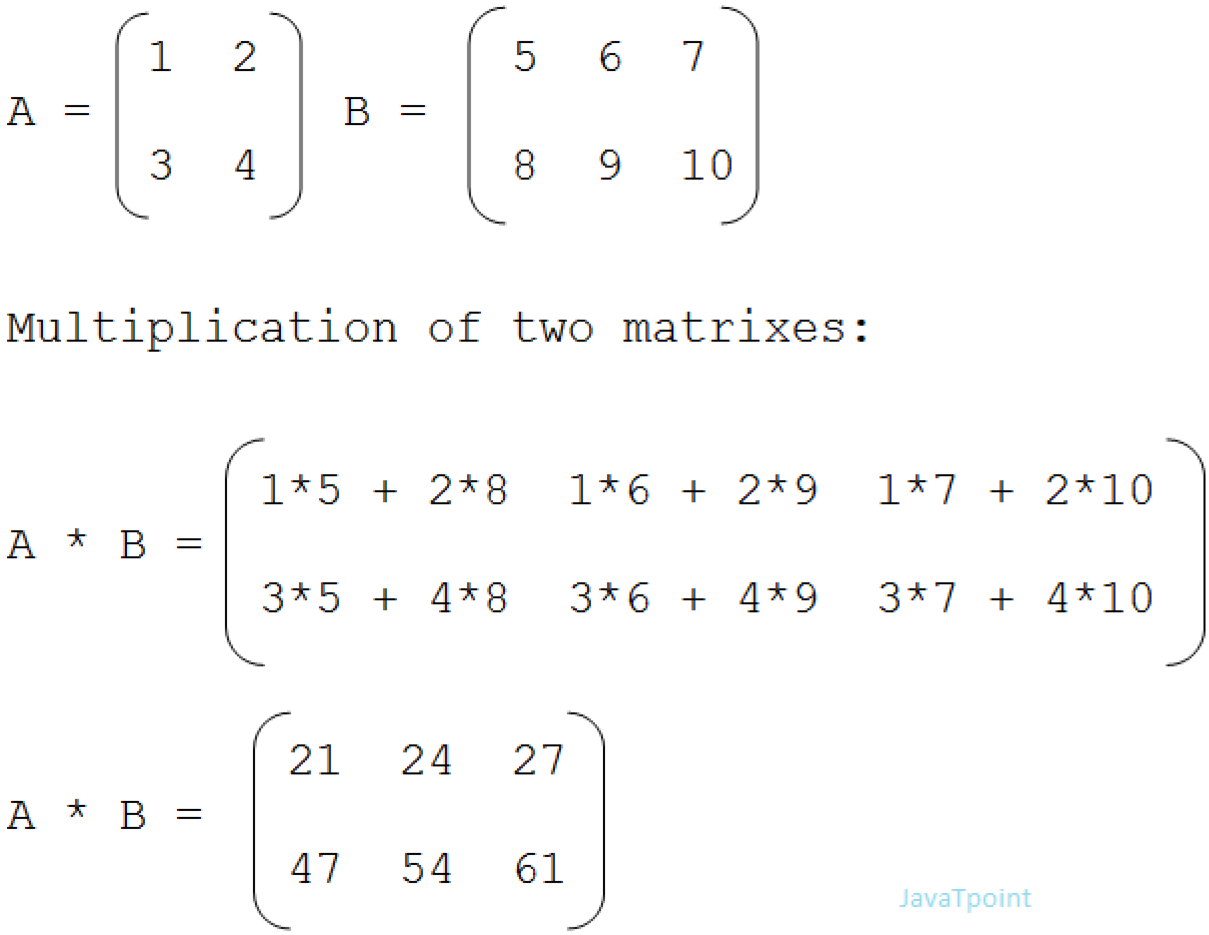
Output:

Matrix multiplication in C

Matrix multiplication in C: We can add, subtract, multiply and divide 2 matrices. To do so, we are taking input from the user for row number, column number, first matrix elements and second matrix elements. Then we are performing multiplication on the matrices entered by the user.

In matrix multiplication *first matrix one row element is multiplied by second matrix all column elements*.

Let's try to understand the matrix multiplication of **2*2 and 3*3** matrices by the figure given below:



$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad B = \begin{pmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \end{pmatrix}$$

Multiplication of two matrixes:

$$A * B = \begin{pmatrix} 1*5 + 2*8 & 1*6 + 2*9 & 1*7 + 2*10 \\ 3*5 + 4*8 & 3*6 + 4*9 & 3*7 + 4*10 \end{pmatrix}$$

$$A * B = \begin{pmatrix} 21 & 24 & 27 \\ 47 & 54 & 61 \end{pmatrix}$$

Let's see the program of matrix multiplication in C.

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

```
int a[10][10],b[10][10],mul[10][10],r,c,i,j,k;
system("cls");
printf("enter the number of row=");
scanf("%d",&r);
printf("enter the number of column=");
scanf("%d",&c);
printf("enter the first matrix element=\n");
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
scanf("%d",&a[i][j]);
}
}
printf("enter the second matrix element=\n");
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
scanf("%d",&b[i][j]);
}
}

printf("multiply of the matrix=\n");
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
mul[i][j]=0;
for(k=0;k<c;k++)
{
mul[i][j]+=a[i][k]*b[k][j];
}
}
}
//for printing result
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
printf("%d\t",mul[i][j]);
}
printf("\n");
}
return 0;
}
```

Output:

```
enter the number of row=3
enter the number of column=3
enter the first matrix element=
1 1 1
2 2 2
3 3 3
enter the second matrix element=
1 1 1
2 2 2
3 3 3
multiply of the matrix=
6 6 6
12 12 12
18 18 18
```

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
```

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
```

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
```

without 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
```

Assembly program in C

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
```