C Programming

Index:

1.Basics of C Programming

1.1 Introduction

1.2 Features

1.3 Setup

1.4 Character set in C

1.5 Keywords in C

2.Data types

2.1 Variable

2.2 Constant

2.3 literals

3.Input and output in C

3.1 printf()

3.2 scanf()

4.Operators

4.1 Arithmetic operator

4.2 Increment and decrement operator

4.3 Assignment operator

4.4 Relational operator

4.5 Logical operator

4.6 Bitwise operator

4.7 ternary operator

5.C preprocessors

5.1 #define

5.2 #include

5.3 #undef

5.4 #ifdef

5.5 #ifndef

5.6 #if

5.7 #else

5.8 #error

5.9 # pragma

6.Header files

7.Conditional statement

7.1 If statement

7.2 If else statement

7.3 Nested if else

7.4 If else ladder

7.5 switch statement

8.Loops

8.1 for loop

8.2 while loop

8.3 do while loop

8.4 break

8.5 continue

8.6 Goto statement

8.7 Difference

9.Functions

9.1Introduction to functions

9.2 Types of functions

9.3 Math.h functions

9,4 Defining Calling and passing argument to function

10.Recursion in C

11.Storage Classes in C

11.1 auto

11.2 extern

11.3 static

11.4 register

12.Arrays

12.1 Introduction to Arrays

12.2 Types of Array

12.3 Arrays and functions

13 Pointers

13.1Introduction to pointers

13.2Pointer and array

13.3Pointer and functions

14.Strings

14.1Introduction to String in C

14.2Library functions of Strings

15.Structure and Unions

15.1Introduction to Structure

15.2 structure and function

15.3 Introduction to Unions

15.4 Difference between Structure and Union

16.File Handing in C

16.1 Introduction to file handling

16.2 Types of files

16.3 File operations

17.Command line arguments

18.Error Handling

18.1 perror()

18.2 strerror()

19.Memory Management in C

19.1 malloc()

19.2 calloc()

19.3 realloc()

19.4 free()

20.Programs

21.C Interview Questions and Answer

22.Find Output of the following programs

23.Quick tip and Tricks.

1. Basics of C Programming:

1.1 Introduction:-

C programming is a general-purpose, procedural, imperative computer programming language developed in 1972 by Dennis M. Ritchie at the Bell Telephone Laboratories to develop the UNIX operating system. C is the most widely used computer language.

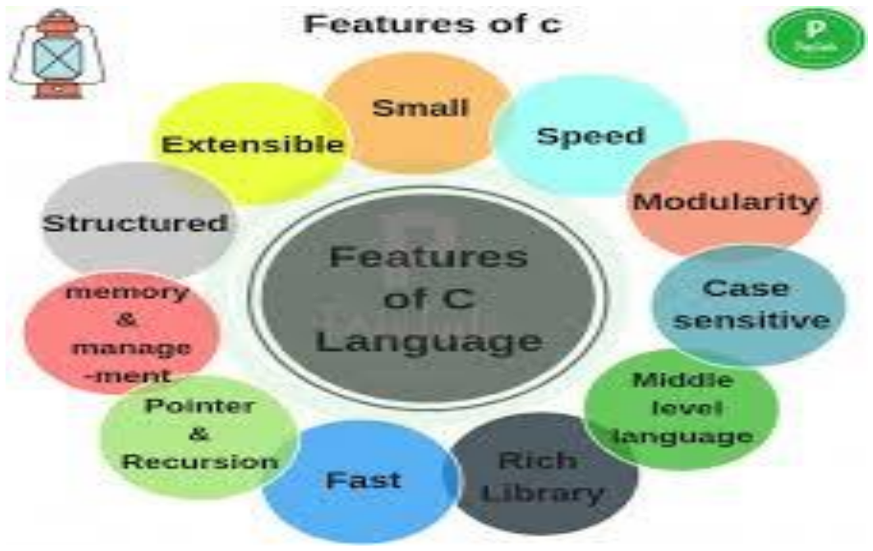
C programming language is a MUST for students and working professionals to become a great Software Engineer specially when they are working in Software Development Domain. I will list down some of the key advantages of learning C Programming**:**

- Easy to learn
- Structured language
- It produces efficient programs
- It can handle low-level activities
- It can be compiled on a variety of computer platforms

Worth to Know about C Language-

1) Oracle is written in C
2) Core libraries of android are written in C
3) MySql is written in C
4) Almost every device driver is written in C
5) Major part of web browser is written in C
6) Unix Operating system is developed in C
7) C is worlds most popular programming language

1.2 Features:-



1) Simple

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

---

2) Machine Independent or Portable

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

3) Mid-level programming language

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

---

4) Structured programming language

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

---

5) Rich Library

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

---

6) Memory Management

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

---

7) Speed

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

---

8) Pointer

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

---

9) Recursion

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

---

10) Extensible

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

1.3 Setup-

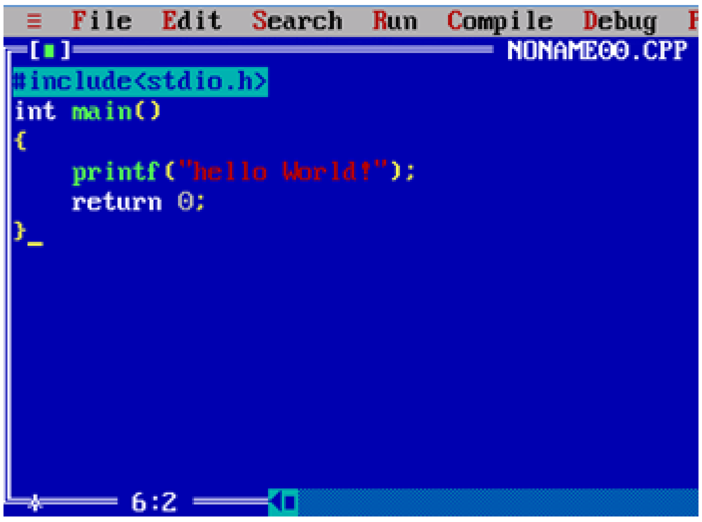
1) Download link of Turbo C- https://developerinsider.co/download-turbo-c-for-windows-7-8-8-1-and-windows-10-32-64-bit-full-screen/
2) Extract zip file
3) Run setup.exe file
4) Click next and accept terms and condition
5) Click Ok to open

Steps:-

1) File > New

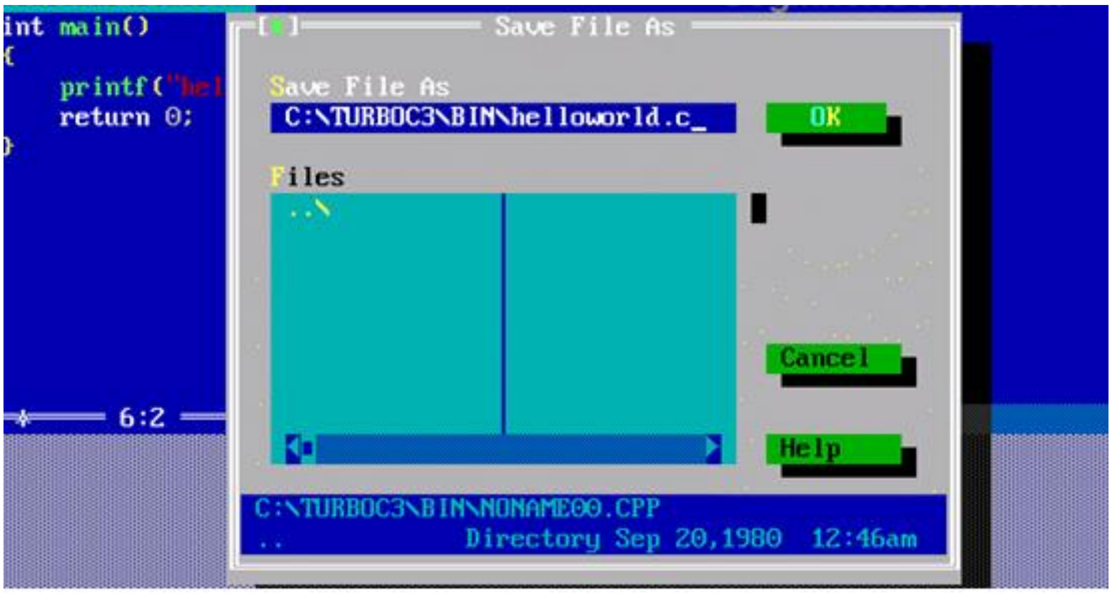
   Write the following program –

```
#include<stdio.h>
int main()
{
/*My first program*/
        printf("hello World!");

        return 0;
}
```



```
#include<stdio.h>
int main()
{
     printf("hello World!");
     return 0;
}_
```

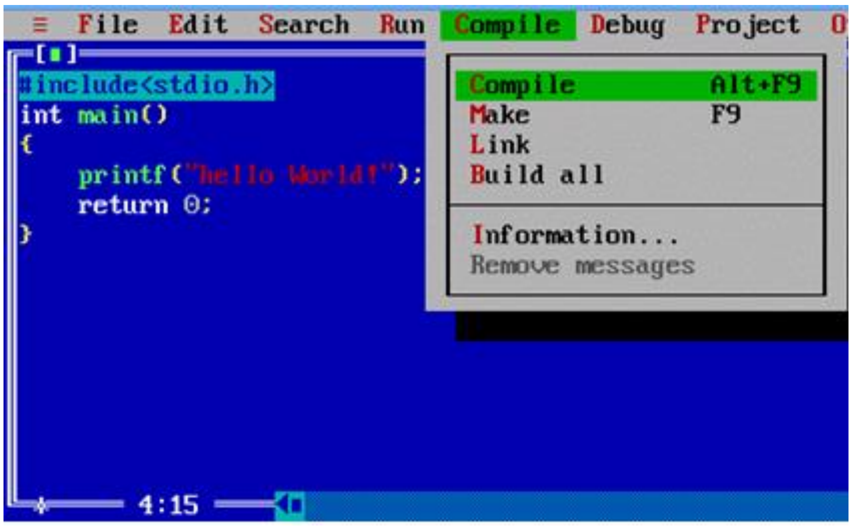
2) Save the program using F2 (OR file > Save), remember the extension should be ".c". In the below screenshot I have given the name as helloworld.c.



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

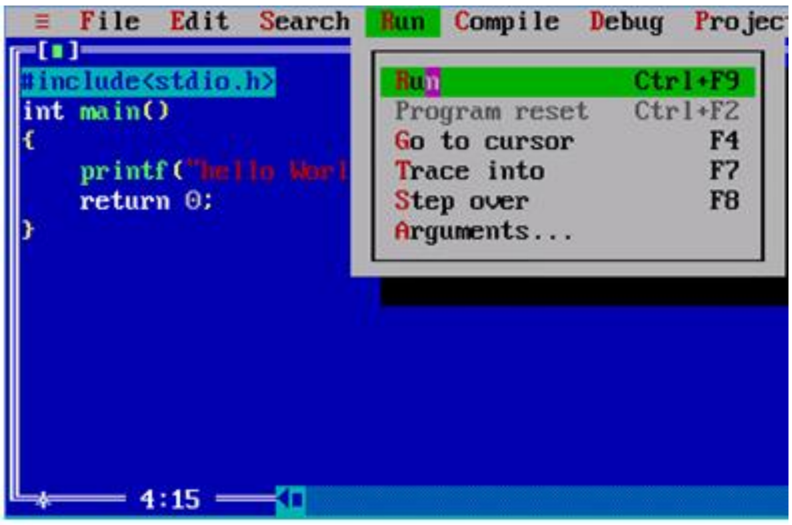
Save File As
Save File As
C:\TURBOC3\BIN\helloworld.c_
OK
Files
..\
Cancel
Help
C:\TURBOC3\BIN\NONAME00.CPP
..   Directory Sep 20,1980 12:46am
6:2

3) Compile the program using Alt + F9 OR Compile > Compile (as shown in the below screenshot).



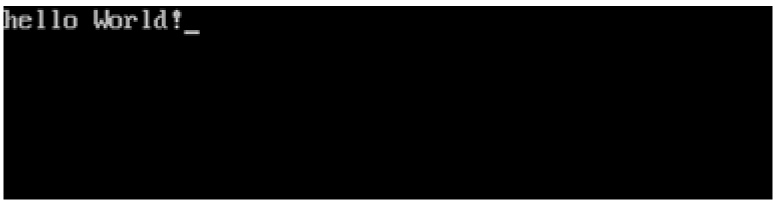
```
#include<stdio.h>
int main()
{
    printf("hello World!");
    return 0;
}
```

4) Press Ctrl + F9 to Run (or select Run > Run in menu bar) the C program.



```
≡ File  Edit  Search  Run  Compile  Debug  Projec
#include<stdio.h>
int main()
{
    printf("hello Worl
    return 0;
}
```

5) Alt+F5 to view the output of the program at the output screen.



```
hello World!_
```

Explanation of program-

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

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

1.4 Character set in C

The characters in C are grouped into the following two categories:

1. Source character set
    a. Alphabets
    b. Digits
    c. Special Characters
    d. White Spaces
2. Execution character set
    a. Escape Sequence

ALPHABETS		
	Uppercase letters	A-Z
	Lowercase letters	a-z

DIGITS	0, 1, 2, 3, 4, 5, 6, 7, 8, 9

SPECIAL CHARACTERS

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

WHITESPACE CHARACTERS

\b	blank space
\t	horizontal tab
\v	vertical tab
\r	carriage return
\f	form feed
\n	new line
\\	Back slash
\’	Single quote
\"	Double quote
\?	Question mark
\0	Null
\a	Alarm (bell)

Character	**ASCII value**	**Escape Sequence**
Null	000	\0
Back space Moves previous position	008	\b
Horizontal tab Moves next horizontal tab	009	\t
New line	010	\n
Vertical tab Moves next vertical tab	011	\v
Form feed Moves initial position of next page	012	\f
Carriage return Moves beginning of the line	013	\r
Double quote Present Double quotes	034	\"
Single quote Present Apostrophe	039	\'
Question mark Present Question Mark	063	\?
Back slash Present back slash	092	\\
Octal number	\000	
Hexadecimal number	\x	

1.5 Keywords in c

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

2. Data types in C :

2.1 Variable-

In programming, a variable is a container (storage area) to hold data.
To indicate the storage area, each variable should be given a unique name (identifier). Variable names are just the symbolic representation of a memory location. For example:

```
int playerScore = 95;
```

Here, playerScore is a variable of int type. Here, the variable is assigned an integer value 95.
The value of a variable can be changed, hence the name variable.

```
char ch = 'a';
// some code
ch = 'l';
```

Rules for naming a variable

1) A variable name can only have letters (both uppercase and lowercase letters), digits and underscore.
2) The first letter of a variable should be either a letter or an underscore.
3) There is no rule on how long a variable name (identifier) can be. However, you may run into problems in some compilers if the variable name is longer than 31 characters.

2.2 Constant-

If you want to define a variable whose value cannot be changed, you can use the const keyword. This will create a constant. For example,

```
const double PI = 3.14;
```

Notice, we have added keyword const.

Here, PI is a symbolic constant; its value cannot be changed.

```
const double PI = 3.14;

PI = 2.9; //Error
```

2.3 Literals-

Literals are data used for representing fixed values. They can be used directly in the code. For example: 1, 2.5, 'c' etc.

Here, 1, 2.5 and 'c' are literals. Why? You cannot assign different values to these terms.

1. Integers

An integer is a numeric literal(associated with numbers) without any fractional or exponential part. There are three types of integer literals in C programming:

decimal (base 10)

octal (base 8)

hexadecimal (base 16)

For example:

Decimal: 0, -9, 22 etc

Octal: 021, 077, 033 etc

Hexadecimal: 0x7f, 0x2a, 0x521 etc

2. Floating-point Literals

A floating-point literal is a numeric literal that has either a fractional form or an exponent form. For example:

-2.0

0.0000234

-0.22E-5

3. Characters

A character literal is created by enclosing a single character inside single quotation marks. For example: 'a', 'm', 'F', '2', '}' etc.

4. String Literals

A string literal is a sequence of characters enclosed in double-quote marks. For example:

```
"good"                  //string constant
""                      //null string constant
"      "                //string constant of six white space
"x"                     //string constant having a single character.
"Earth is round\n"      //prints string with a newline
```

Data Types

Here's a table containing commonly used types in C programming for quick access.

Type	Size (bytes)	Format Specifier
int	at least 2, usually 4	%d
char	1	%c
float	4	%f
double	8	%lf
short int	2 usually	%hd
unsigned int	at least 2, usually 4	%u
long int	at least 4, usually 8	%li
long long int	at least 8	%lli
unsigned long int	at least 4	%lu
unsigned long long int	at least 8	%llu
signed char	1	%c
unsigned char	1	%c
long double	at least 10, usually 12 or 16	%Lf

Example-

```
int myVar=10;

float salary=10.4;

double price=10.44;

char test = 'h';
```

Program-

```
#include <stdio.h>

int main() {

  short a;

  long b;

  long long c;

  long double d;

  printf("size of short = %d bytes\n", sizeof(a));

  printf("size of long = %d bytes\n", sizeof(b));

  printf("size of long long = %d bytes\n", sizeof(c));

  printf("size of long double= %d bytes\n", sizeof(d));

  return 0;

}
```

3.Input and output in C

Output in C

Printf()-It is not a keyword,It is predefined function.

It is used to print any statement in C on monitor.

It can also print value of expression or value or variable.

Ex.1 Printing a Statement

```
#include <stdio.h>
int main()
{
Clrscr();
   printf("C Programming");
   printf("C Programming \n with Vineet");
getch();
   return 0;
}
```

Ex.2 Printing integer value

```
#include <stdio.h>
int main()
{
   int testInteger = 5;
   printf("Number = %d", testInteger);
   return 0;
}
```

Ex.3 Printing a float value

```
#include <stdio.h>
int main()
{
   float number1 = 13.5;
```

```
    double number2 = 12.4;

    printf("number1 = %f\n", number1);
    printf("number2 = %lf", number2);
    return 0;
}
```

Ex.4 Printing a character

```
#include <stdio.h>
int main()
{
    char chr = 'a';
    printf("character = %c.", chr);
    return 0;
}
```

Input in C-

scanf()-It is not a keyword,It is a predefined function

It is used to take input from user from keyboard

Ex.1 Integer

```
#include <stdio.h>
int main()
{
    int testInteger;
    printf("Enter an integer: ");
    scanf("%d", &testInteger);
    printf("Number = %d",testInteger);
    return 0;
}
```

Ex.2 Float

```
#include <stdio.h>
int main()
{
    float num1;
    double num2;

    printf("Enter a number: ");
    scanf("%f", &num1);
    printf("Enter another number: ");
    scanf("%lf", &num2);

    printf("num1 = %f\n", num1);
    printf("num2 = %lf", num2);

    return 0;
}
```

Ex.3 Character

```
#include <stdio.h>
int main()
{
    char chr;
    printf("Enter a character: ");
    scanf("%c",&chr);
    printf("You entered %c.", chr);
    return 0;
}
```

Ex.4 Multiple value

```
#include <stdio.h>

int main()

{

   int a;

   float b;


   printf("Enter integer and then a float: ");


   // Taking multiple inputs

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


   printf("You entered %d and %f", a, b);

   return 0;

}
```

4.Operators

4.1 Arithmetic operators-

An arithmetic operator performs mathematical operations such as addition, subtraction, multiplication, division etc on numerical values (constants and variables).

Operator	Meaning of Operator
+	addition or unary plus
-	subtraction or unary minus
*	multiplication
/	division
%	remainder after division (modulo division)

Example-

```
#include <stdio.h>

int main()

{

    int a = 9,b = 4, c;

    c = a+b;

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

    c = a-b;

    printf("a-b = %d \n",c);

    c = a*b;

    printf("a*b = %d \n",c);

    c = a/b;

    printf("a/b = %d \n",c);

    c = a%b;

    printf("Remainder when a divided by b = %d \n",c);
    return 0;

}
```

4.2 Increment and decrement operator

C programming has two operators increment ++ and decrement -- to change the value of an operand (constant or variable) by 1.

Increment ++ increases the value by 1 whereas decrement -- decreases the value by 1. These two operators are unary operators, meaning they only operate on a single operand.

Increment/Decrement operators are of two types:

Prefix increment/decrement operator.

Postfix increment/decrement operator.

++x is same as x = x + 1 or x += 1

--x is same as x = x - 1 or x -= 1

The prefix increment/decrement operator immediately increases or decreases the current value of the variable. This value is then used in the expression. Let’s take an example:

y = ++x;

Here first, the current value of x is incremented by 1. The new value of x is then assigned to y.

The postfix increment/decrement operator causes the current value of the variable to be used in the expression, then the value is incremented or decremented. For example:

y = x++;

Here first, the current value of x is assigned to y then x is incremented.

Ex-

```
#include <stdio.h>

int main()

{

    int a = 10, b = 100;

    float c = 10.5, d = 100.5;


    printf("++a = %d \n", ++a);//11
    printf("--b = %d \n", --b);//99

    printf("++c = %f \n", ++c);//11.5

    printf("--d = %f \n", --d);//99.5

    return 0;

}
```

4.3 Assignment operator

An assignment operator is used for assigning a value to a variable. The most common assignment operator is =

Operator	Example	Same as
=	a = b	a = b
+=	a += b	a = a+b
-=	a -= b	a = a-b
*=	a *= b	a = a*b
/=	a /= b	a = a/b
%=	a %= b	a = a%b

```
#include <stdio.h>

int main()

{

    int a = 5, c;

    c = a;      // c is 5

    printf("c = %d\n", c);

    c += a;     // c is 10

    printf("c = %d\n", c);

    c -= a;     // c is 5

    printf("c = %d\n", c);

    c *= a;     // c is 25

    printf("c = %d\n", c);

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

    return 0;
}
```

4.4 Relational operator-

A relational operator checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0.

Relational operators are used in decision making and loops.

Operator	**Meaning of Operator**	**Example**
==	Equal to	5 == 3 is evaluated to 0
>	Greater than	5 > 3 is evaluated to 1
<	Less than	5 < 3 is evaluated to 0
!=	Not equal to	5 != 3 is evaluated to 1
>=	Greater than or equal to	5 >= 3 is evaluated to 1
<=	Less than or equal to	5 <= 3 is evaluated to 0

Example-

```
#include <stdio.h>
int main()
{
    int a = 5, b = 5, c = 10;
    printf("%d == %d is %d \n", a, b, a == b);
    printf("%d == %d is %d \n", a, c, a == c);
    printf("%d > %d is %d \n", a, b, a > b);
    printf("%d > %d is %d \n", a, c, a > c);

    printf("%d < %d is %d \n", a, b, a < b);

    printf("%d < %d is %d \n", a, c, a < c);

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

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

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

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

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

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

    return 0;

}
```

4.5 Logical operator

An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false. Logical operators are commonly used in decision making in C programming.

Operator

&&- Logical AND. True only if all operands are true

If c = 5 and d = 2 then, expression ((c==5) && (d>5)) equals to 0.

|| - Logical OR. True only if either one operand is true

If c = 5 and d = 2 then, expression ((c==5) || (d>5)) equals to 1.

! - Logical NOT. True only if the operand is 0

If c = 5 then, expression !(c==5) equals to 0.

Example: Logical Operators

```
#include <stdio.h>

int main()

{

    int a = 5, b = 5, c = 10, result;
    result = (a == b) && (c > b);

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

    result = (a == b) && (c < b);

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

    result = (a == b) || (c < b);

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

    result = (a != b) || (c < b);

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

    result = !(a != b);

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

    result = !(a == b);

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

    return 0;

}
```

4.6 C Bitwise Operators

Bitwise AND operator &

The output of bitwise AND is 1 if the corresponding bits of two operands is 1. If either bit of an operand is 0, the result of corresponding bit is evaluated to 0.

Let us suppose the bitwise AND operation of two integers 12 and 25.

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bit Operation of 12 and 25

```
  00001100

& 00011001

  ________

  00001000  = 8 (In decimal)
```

Example #1: Bitwise AND

```
#include <stdio.h>

int main()

{

   int a = 12, b = 25;

   printf("Output = %d", a&b);

   return 0;

}
```

Output = 8

Bitwise OR operator |

The output of bitwise OR is 1 if at least one corresponding bit of two operands is 1. In C Programming, bitwise OR operator is denoted by |.

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bitwise OR Operation of 12 and 25

```
  00001100
| 00011001
  ________
  00011101  = 29 (In decimal)
```

Example #2: Bitwise OR

```
#include <stdio.h>

int main()

{

   int a = 12, b = 25;

   printf("Output = %d", a|b);

   return 0;

}
```

Output = 29

Bitwise XOR (exclusive OR) operator ^

The result of bitwise XOR operator is 1 if the corresponding bits of two operands are opposite. It is denoted by ^.

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bitwise XOR Operation of 12 and 25

```
  00001100
^ 00011001
  ________
  00010101  = 21 (In decimal)
```

Example #3: Bitwise XOR

```
#include <stdio.h>

int main()

{

    int a = 12, b = 25;

    printf("Output = %d", a^b);

    return 0;

}
```

Output = 21

Bitwise complement operator ~

Bitwise compliment operator is an unary operator (works on only one operand). It changes 1 to 0 and 0 to 1. It is denoted by ~.

35 = 00100011 (In Binary)

Bitwise complement Operation of 35

```
~ 00100011
  ________
  11011100  = 220 (In decimal)
```

Twist in bitwise complement operator in C Programming

The bitwise complement of 35 (~35) is -36 instead of 220, but why?

For any integer n, bitwise complement of n will be -(n+1). To understand this, you should have the knowledge of 2's complement.

Shift Operators in C programming

There are two shift operators in C programming:

1) Right shift operator
2) Left shift operator.

Right Shift Operator

Right shift operator shifts all bits towards right by certain number of specified bits. It is denoted by >>.

212 = 11010100 (In binary)

212>>2 = 00110101 (In binary) [Right shift by two bits]

212>>7 = 00000001 (In binary)

212>>8 = 00000000

212>>0 = 11010100 (No Shift)

Left Shift Operator

Left shift operator shifts all bits towards left by certain number of specified bits. It is denoted by <<.

212 = 11010100 (In binary)

212<<1 = 110101000 (In binary) [Left shift by one bit]

212<<0 =11010100 (Shift by 0)

212<<4 = 110101000000 (In binary) =3392(In decimal)

Example #5: Shift Operators

```
#include <stdio.h>

int main()

{

   int num=212, i;

   for (i=0; i<=2; ++i)

       printf("Right shift by %d: %d\n", i, num>>i);
```

```
    printf("\n");
    for (i=0; i<=2; ++i)
       printf("Left shift by %d: %d\n", i, num<<i);
    return 0;
}
```

```
Right Shift by 0: 212
Right Shift by 1: 106
Right Shift by 2: 53

Left Shift by 0: 212
Left Shift by 1: 424
Left Shift by 2: 848
```

4.7 Ternary operator-

Programmers use the ternary operator for decision making in place of longer if and else conditional statements.

The ternary operator take three arguments:

1)The first is a comparison argument

2) The second is the result upon a true comparison

3) The third is the result upon a false comparison

Ex 1.Comparison using if else

```
int a = 10, b = 20, c;
if (a < b) {
   c = a;
}
else {
   c = b;
}
```

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

This example takes more than 10 lines, but that isn’t necessary. You can write the above program in just 3 lines of code using a ternary operator.

Syntax

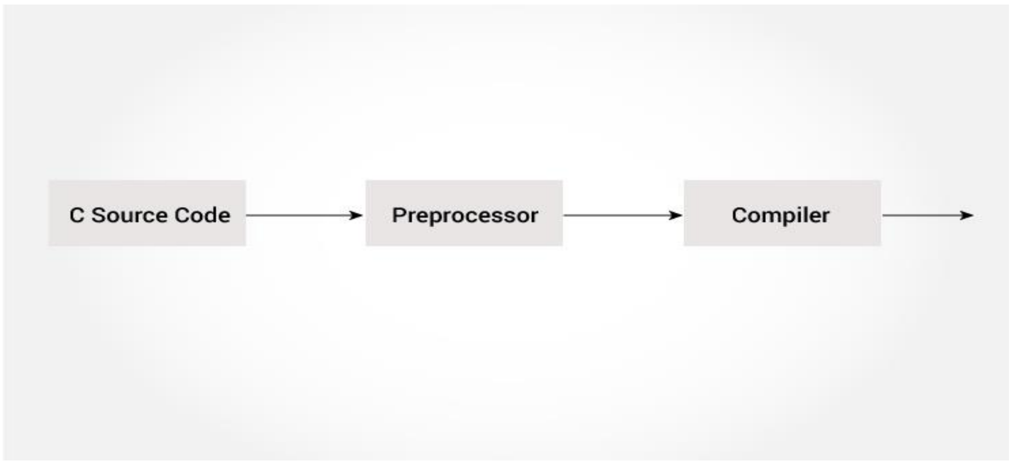
```
condition ? value_if_true : value_if_false
```

The statement evaluates to value_if_true if condition is met, and value_if_false otherwise.

```
int a = 10, b = 20, c;

c = (a < b) ? a : b;

printf("%d", c);//10
```

c is set equal to a, because the condition a < b was true.

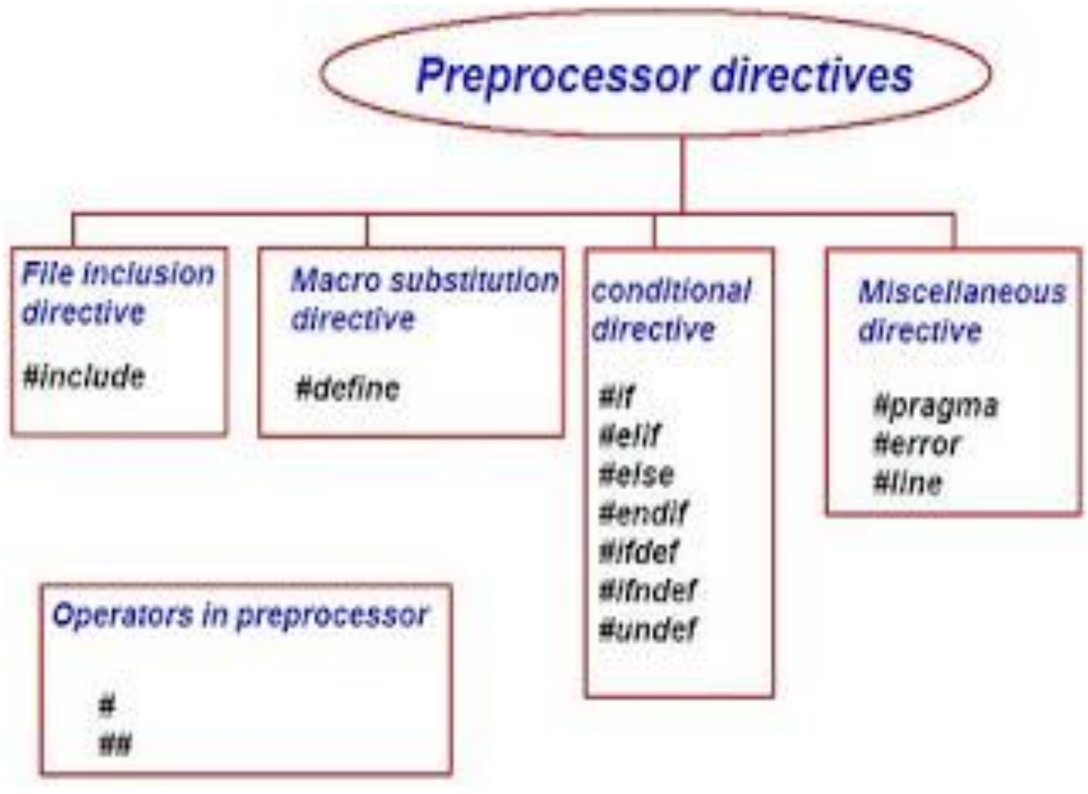
5.C Preprocessor



The C preprocessor is a macro preprocessor (allows you to define macros) that transforms your program before it is compiled. These transformations can be the inclusion of header file, macro expansions etc.

All preprocessing directives begin with a # symbol. For example,

#define PI 3.14



1) **#define:**

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

```
#define token value
```

Example-

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

2) **#include-**

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

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

```
#include <filename>
#include "filename"
```

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

Example-

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

3) **#undef**

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

Syntax:

```
#undef token
```

Example-

```
#include <stdio.h>
#define PI 3.14
#undef PI
```

```
main() {
  printf("%f",PI);
}
```
Output:

```
Compile Time Error: 'PI' undeclared
```

4) #ifdef

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

Syntax:
```
#ifdef MACRO
//code
#endif
```

Example-

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

Output-Value of a: 2

5) #ifndef

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

Syntax:
```
#ifndef MACRO
//code
#endif
```

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

```
void main() {
int a=0;
#ifndef INPUT
a=2;
#else
printf("Enter a:");
scanf("%d", &a);
#endif
printf("Value of a: %d\n", a);
getch();
}
```

Output:

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

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

6) #if

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

Syntax:

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

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

Output:

```
Value of Number is: 0
```

7) #else

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

Syntax:

```
#if expression
```

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

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

Output:

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

8) #error

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

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

Output:Compile Time Error: First include then compile

6.Header files

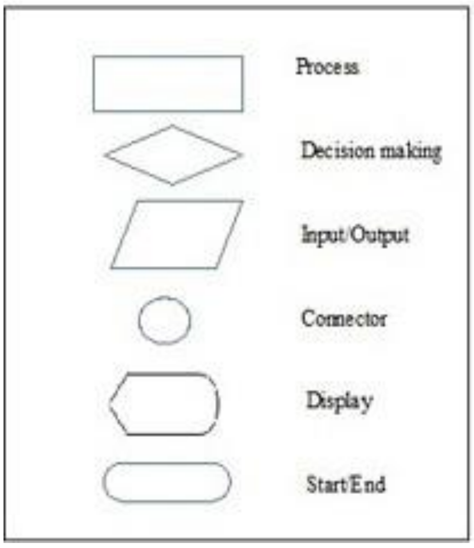
No.	Name	Description
1	stdio.h	Input/Output Functions
2	conio.h	console input/output
3	assert.h	Diagnostics Functions
4	ctype.h	Character Handling Functions
5	cocale.h	Localization Functions
6	math.h	Mathematics Functions
7	setjmp.h	Nonlocal Jump Functions
8	signal.h	Signal Handling Functions
9	stdarg.h	Variable Argument List Functions
10	stdlib.h	General Utility Functions
11	string.h	String Functions
12	time.h	Date and Time Functions
13	complex.h	A set of function for manipulating complex numbers
14	stdalign.h	For querying and specifying the alignment of objects
15	errno.h	For testing error codes
16	locale.h	Defines localization functions
17	stdatomic.h	For atomic operations on data shared between threads
18	stdnoreturn.h	For specifying non-returning functions
19	uchar.h	Types and functions for manipulating Unicode characters
20	fenv.h	A set of functions for controlling the floating-point

		environment
21	wchar.h	Defines wide string handling functions
22	tgmath.h	Type-generic mathematical functions
23	stdarg.h	Accessing a varying number of arguments passed to functions
24	stdbool.h	Defines a boolean data type

7.Conditional Statement:

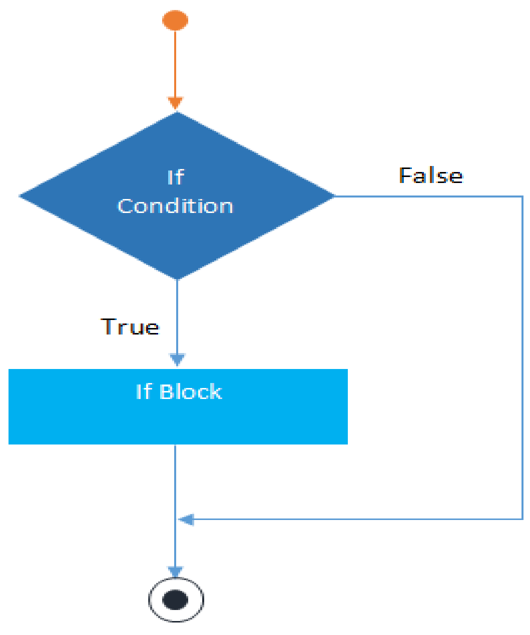
Conditional statements help you to make a decision based on certain conditions. These conditions are specified by a set of conditional statements having boolean expressions which are evaluated to a boolean value true or false

Flow chart shapes:



1) **If Statement**
   The single if statement in C language is used to execute the code if a condition is true. It is also called one-way selection statement.

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



If statement

Example-

```
#include<stdio.h>
#include<conio.h>
void main()
{
int num=0;
printf("enter the number");
scanf("%d",&num);
if(n%2==0)
{
printf("%d number in even",num);
}
getch();
}
```

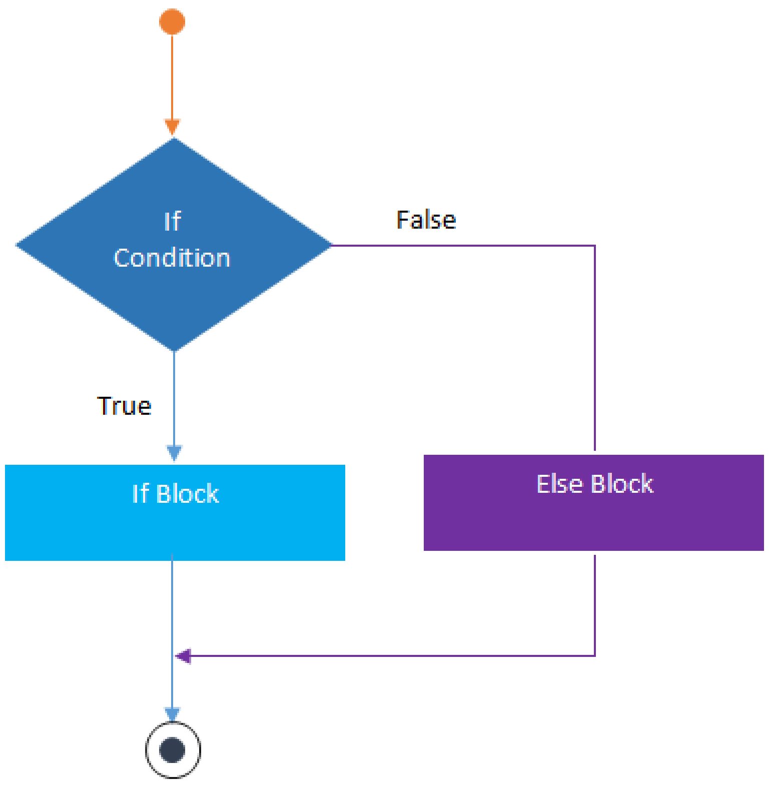
2) If else statement

The if-else statement in C language is used to execute the code if condition is true or false. It is also called two-way selection statement.

Syntax

```
if(expression)
{
 //Statements
}
else
```

```
{
 //Statements
}
```



If-Else statement

Example-

```
#include<stdio.h>
#include<conio.h>
void main()
{
int num=0;
printf("enter the number");
scanf("%d",&num);
if(n%2==0)
{
printf("%d number in even", num);
}
else
{
printf("%d number in odd",num);
}
getch();
}
```

3) Nested if else

The nested if...else statement is used when a program requires more than one test expression. It is also called a multi-way selection statement. When a series of the decision are involved in a statement, we use if else statement in nested form.

Syntax

```
if( expression )
{
 if( expression1 )
 {
 statement-block1;
 }
 else
 {
 statement-block 2;
 }
}
else
{
 statement-block 3;
}
```

Example

```
#include<stdio.h>
#include<conio.h>
void main( )
{
 int a,b,c;
 clrscr();
 printf("Please Enter 3 number");
 scanf("%d%d%d",&a,&b,&c);
 if(a>b)
 {
 if(a>c)
 {
 printf("a is greatest");
 }
 else
 {
 printf("c is greatest");
 }
 }
 else
 {
 if(b>c)
 {
```

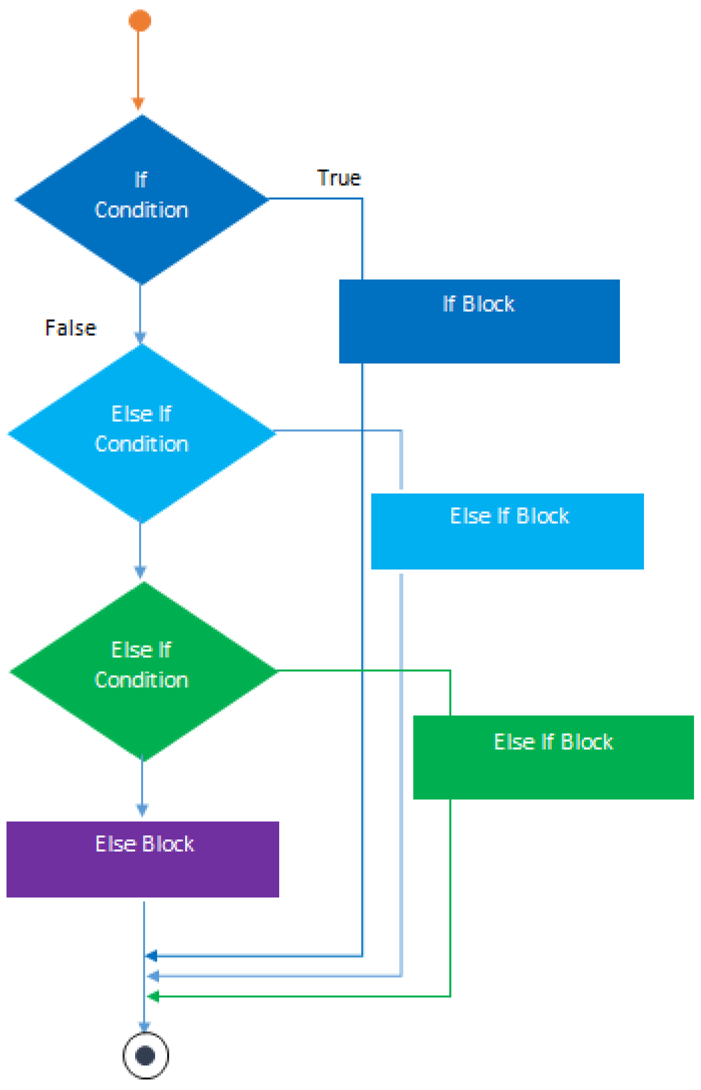
```
 printf("b is greatest");
 }
 else
 {
 printf("c is greatest");
 }
 }
getch();
}
```

4) If else ladder-

The if-else-if statement is used to execute one code from multiple conditions. It is also called multipath decision statement. It is a chain of if..else statements in which each if statement is associated with else if statement and last would be an else statement.

Syntax

```
if(condition1)
{
 //statements
}
else if(condition2)
{
 //statements
}
else if(condition3)
{
 //statements
}
else
{
 //statements
}
```



If-Else-If Ladder

Example-

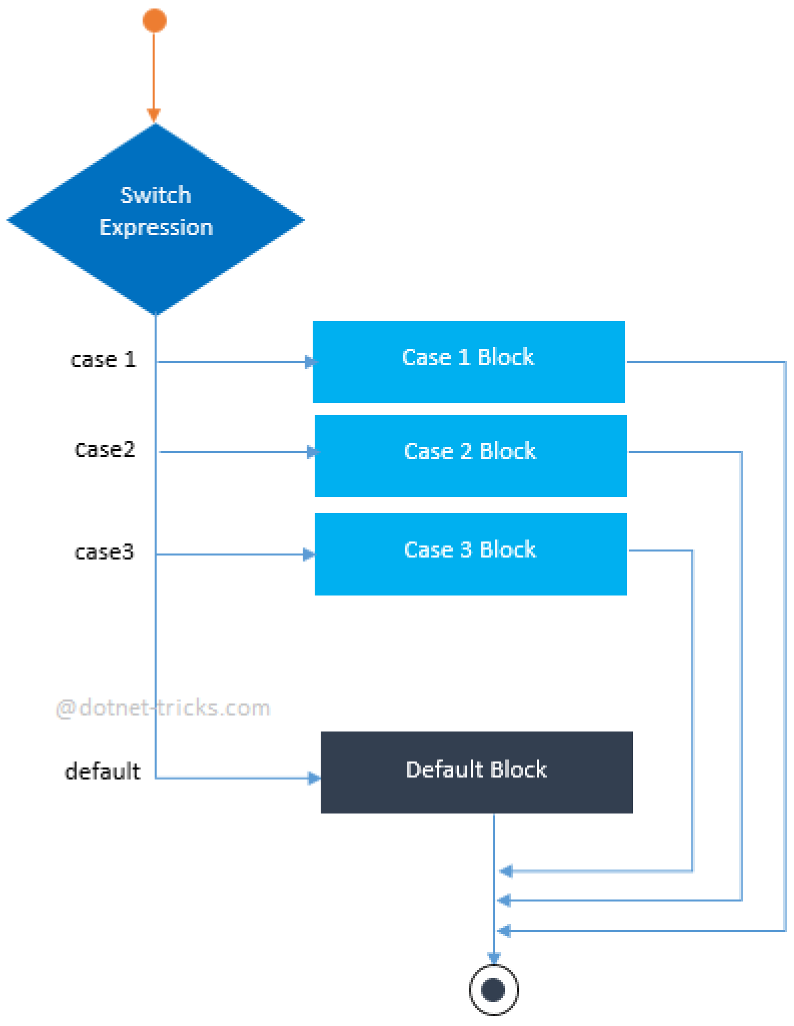
```
#include<stdio.h>
#include<conio.h>
void main( )
{
 int a;
 printf("enter a number");
 scanf("%d",&a);
 if( a%5==0 && a%8==0)
 {
 printf("divisible by both 5 and 8");
 }
 else if( a%8==0 )
 {
 printf("divisible by 8");
 }
 else if(a%5==0)
 {
 printf("divisible by 5");
 }
 else
 {
 printf("divisible by none");
 }
getch();
}
```

5) **Switch statement-**

switch statement acts as a substitute for a long if-else-if ladder that is used to test a list of cases. A switch statement contains one or more case labels which are tested against the switch expression. When the expression match to a case then the associated statements with that case would be executed.

Syntax

```
Switch (expression)
{
 case value1:
 //Statements
 break;
 case value 2:
 //Statements
 break;
 case value 3:
 //Statements
 case value n:
 //Statements
 break;
 Default:
 //Statements
}
```



Switch Statement

Example-

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

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

    printf("Enter an operator (+, -, *, /): ");
    scanf("%c", &operator);
    printf("Enter two operands: ");
    scanf("%lf %lf",&n1, &n2);
    switch(operator)
    {
        case '+':
            printf("%.1lf + %.1lf = %.1lf",n1, n2, n1+n2);
            break;

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

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

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

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

    return 0;
}
```

8.Loops:

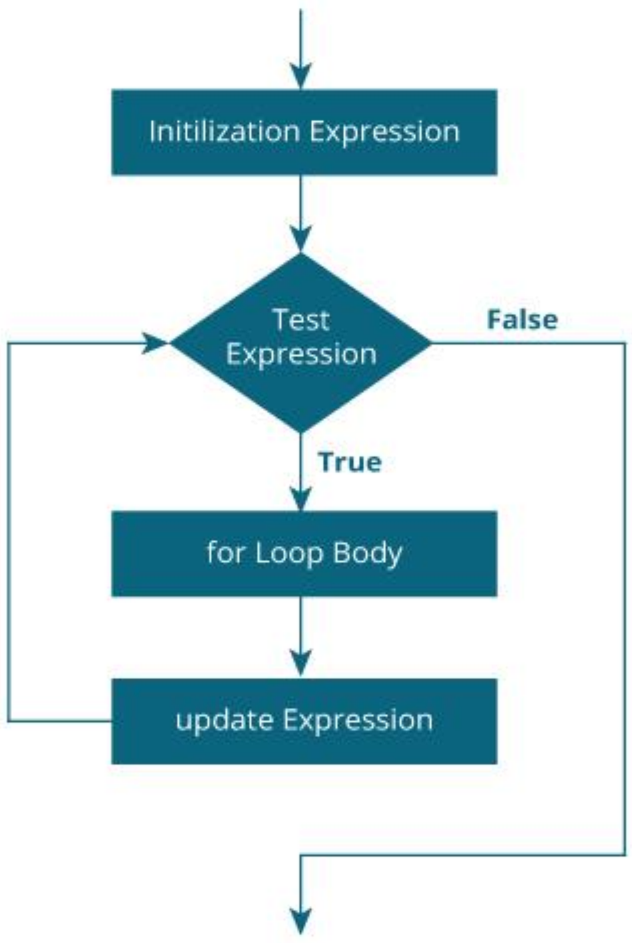
1) For loop-

The syntax of the for loop is:

```
for (initializationStatement; testExpression; updateStatement)
{
    // statements inside the body of loop
}
```

How for loop works?

1) The initialization statement is executed only once.
2) Then, the test expression is evaluated. If the test expression is evaluated to false, the for loop is terminated.
3) However, if the test expression is evaluated to true, statements inside the body of for loop are executed, and the update expression is updated.
4) Again the test expression is evaluated.
5) This process goes on until the test expression is false. When the test expression is false, the loop terminates.



Example 1: for loop

```
// Print numbers from 1 to 10

#include <stdio.h>

int main() {

  int i;

  for (i = 1; i < 11; ++i)

  {

    printf("%d ", i);

  }

return 0;

}
```

Output

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

Example 2: for loop

```
// Program to calculate the sum of first n natural numbers

// Positive integers 1,2,3...n are known as natural numbers

#include <stdio.h>

int main()

{

    int num, count, sum = 0;

    printf("Enter a positive integer: ");

    scanf("%d", &num);

    // for loop terminates when num is less than count

    for(count = 1; count <= num; ++count)

    {

        sum += count;

    }
    printf("Sum = %d", sum);

    return 0;

}
```

Output

```
Enter a positive integer: 10

Sum = 55
```

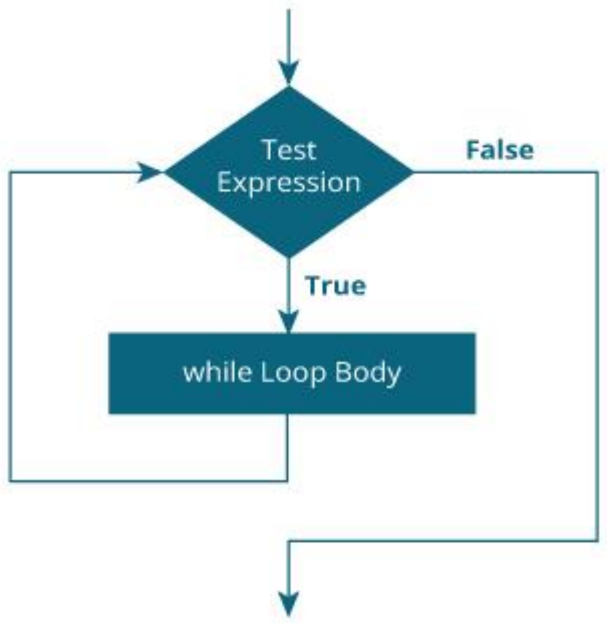
2) While loop

The syntax of the while loop is:

```
while (testExpression)

{

    // statements inside the body of the loop

}
```

How while loop works?

1) The while loop evaluates the test expression inside the parenthesis ().
2) If the test expression is true, statements inside the body of while loop are executed. Then, the test expression is evaluated again.
3) The process goes on until the test expression is evaluated to false.
4) If the test expression is false, the loop terminates (ends).



Example : while loop

```
// Print numbers from 1 to 5

#include <stdio.h>

int main()

{

    int i = 1;

    while (i <= 5)

    {

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

        ++i;

    }

    return 0;

}
```

Output

```
1

2

3

4

5
```

3) Do While loop

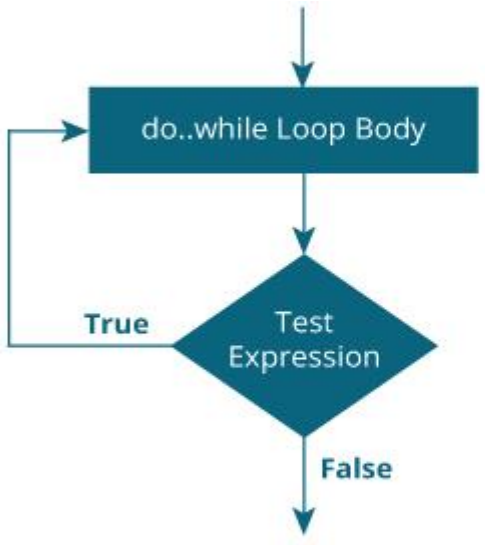
The do..while loop is similar to the while loop with one important difference. The body of do...while loop is executed at least once. Only then, the test expression is evaluated.

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

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

How do...while loop works?

1) The body of do...while loop is executed once. Only then, the test expression is evaluated.
2) If the test expression is true, the body of the loop is executed again and the test expression is evaluated.
3) This process goes on until the test expression becomes false.
4) If the test expression is false, the loop ends.



Example : do...while loop

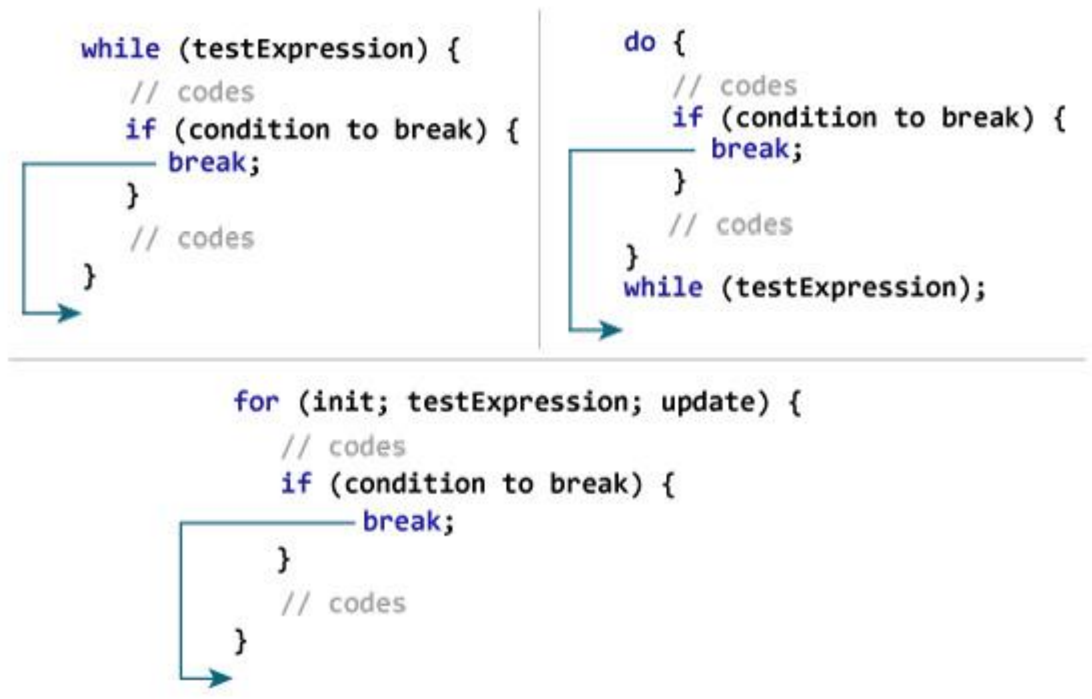
```
// Program to add numbers until the user enters zero
#include <stdio.h>
int main()
{
   double number, sum = 0;
   // the body of the loop is executed at least once
   do
   {
       printf("Enter a number: ");
       scanf("%lf", &number);
       sum += number;
   }
   while(number != 0.0);
   printf("Sum = %.2lf",sum);
   return 0;
}
```

Output

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

4) **Break statement**
The break statement ends the loop immediately when it is encountered. Its syntax is:
break;

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



```
while (testExpression) {
   // codes
   if (condition to break) {
      break;
   }
   // codes
}
```

```
do {
   // codes
   if (condition to break) {
      break;
   }
   // codes
}
while (testExpression);
```

```
for (init; testExpression; update) {
   // codes
   if (condition to break) {
      break;
   }
   // codes
}
```

Example 1: break statement

// Program to calculate the sum of a maximum of 10 numbers

// If a negative number is entered, the loop terminates

```
# include <stdio.h>
int main()
{
   int i;
   double number, sum = 0.0;
   for(i=1; i <= 10; ++i)
   {
      printf("Enter a n%d: ",i);
      scanf("%lf",&number);

      // If the user enters a negative number, the loop ends
      if(number < 0.0)
```

```
        {
            break;
        }
        sum += number; // sum = sum + number;
    }
    printf("Sum = %.2lf",sum);
    return 0;
}
```

Output

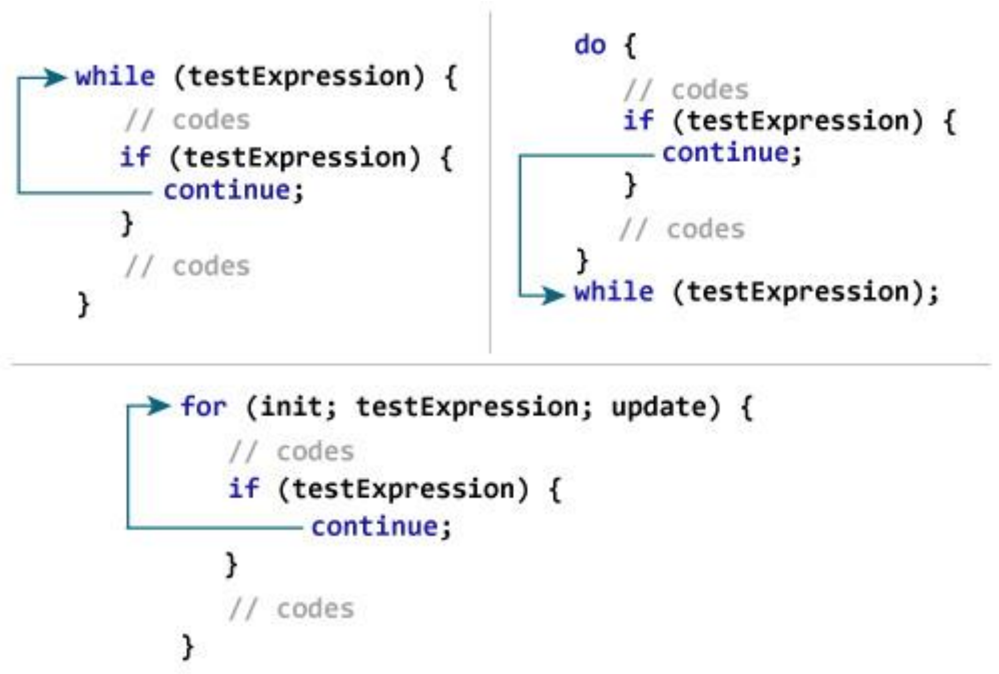
```
Enter a n1: 2.4
Enter a n2: 4.5
Enter a n3: 3.4
Enter a n4: -3
Sum = 10.30
```

5) Continue statement

The continue statement skips the current iteration of the loop and continues with the next iteration. Its syntax is:

```
continue;
```

The continue statement is almost always used with the if...else statement.



```
while (testExpression) {
    // codes
    if (testExpression) {
        continue;
    }
    // codes
}
```

```
do {
    // codes
    if (testExpression) {
        continue;
    }
    // codes
}
while (testExpression);
```

```
for (init; testExpression; update) {
    // codes
    if (testExpression) {
        continue;
    }
    // codes
}
```

Example 2: continue statement

// Program to calculate the sum of a maximum of 10 numbers

// Negative numbers are skipped from the calculation

```
# include <stdio.h>
int main()
{
    int i;
    double number, sum = 0.0;
    for(i=1; i <= 10; ++i)
    {
        printf("Enter a n%d: ",i);
        scanf("%lf",&number);

        if(number < 0.0)
        {
            continue;
        }
        sum += number; // sum = sum + number;
    }

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

    return 0;
}
```

Output

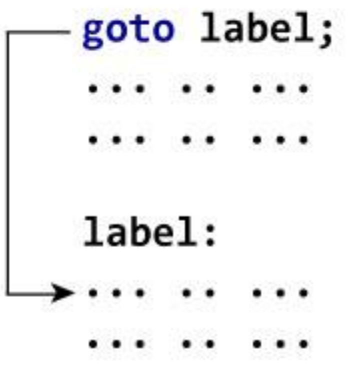
```
Enter a n1: 1.1
Enter a n2: 2.2
Enter a n3: 5.5
Enter a n4: 4.4
Enter a n5: -3.4
Enter a n6: -45.5
Enter a n7: 34.5
Enter a n8: -4.2
Enter a n9: -1000
Enter a n10: 12
Sum = 59.70
```

6) Goto statement

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

Syntax of goto Statement

```
goto label;
... .. ...
... .. ...
label:
statement;
```



Example:

```
// Program to calculate the sum and average of positive numbers
// If the user enters a negative number, the sum and average are displayed.
#include <stdio.h>
int main() {
  const int maxInput = 100;
  int i;
  double number, average, sum = 0.0;
  for (i = 1; i <= maxInput; ++i) {
    printf("%d. Enter a number: ", i);
    scanf("%lf", &number);
    // go to jump if the user enters a negative number
    if (number < 0.0) {
      goto jump;
```

```
        }
        sum += number;
    }

jump:
    average = sum / (i - 1);
    printf("Sum = %.2f\n", sum);
    printf("Average = %.2f", average);
    return 0;
}
```

Output

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

Reasons to avoid goto

The use of goto statement may lead to code that is buggy and hard to follow. For example,

```
one:
for (i = 0; i < number; ++i)
{
     test += i;
     goto two;
}
two:
if (test > 5) {
  goto three;
}
... .. ...
```

Also, the goto statement allows you to do bad stuff such as jump out of the scope.

That being said, goto can be useful sometimes. For example: to break from nested loops.

7) Difference

1) Break and continue

break	**continue**
A break can appear in both switch and loop (for, while, do) statements.	A continue can appear only in loop (for, while, do) statements.
A break causes the switch or loop statements to terminate the moment it is executed. Loop or switch ends abruptly when break is encountered.	A continue doesn't terminate the loop, it causes the loop to go to the next iteration. All iterations of the loop are executed even if continue is encountered. The continue statement is used to skip statements in the loop that appear after the continue.
The break statement can be used in both switch and loop statements.	The continue statement can appear only in loops. You will get an error if this appears in switch statement.
When a break statement is encountered, it terminates the block and gets the control out of the switch or loop.	When a continue statement is encountered, it gets the control to the next iteration of the loop.
A break causes the innermost enclosing loop or switch to be exited immediately.	A continue inside a loop nested within a switch causes the next loop iteration.

2) While and do while

WHILE	DO-WHILE
Condition is checked first then statement(s) is executed.	Statement(s) is executed atleast once, thereafter condition is checked.
It might occur statement(s) is executed zero times, If condition is false.	At least once the statement(s) is executed.
No semicolon at the end of while. while(condition)	Semicolon at the end of while. while(condition);
If there is a single statement, brackets are not required.	Brackets are always required.
Variable in condition is initialized before the execution of loop.	variable may be initialized before or within the loop.
while loop is entry controlled loop.	do-while loop is exit controlled loop.
while(condition) { statement(s); }	do { statement(s); } while(condition);

3) For loop,while loop.do while loop

Sr. No	**For loop**	**While loop**	**Do while loop**
1.	Syntax: For(initialization; condition;updating), { . Statements; }	Syntax: While(condition), { . Statements; . }	Syntax: Do { . Statements; } While(condition);
2.	It is known as entry controlled loop	It is known as entry controlled loop.	It is known as exit controlled loop.
3.	If the condition is not true first time than control will never enter in a loop	If the condition is not true first time than control will never enter in a loop.	Even if the condition is not true for the first time the control will enter in a loop.
4.	There is no semicolon; after the condition in the syntax of the for loop.	There is no semicolon; after the condition in the syntax of the while loop.	There is semicolon; after the condition in the syntax of the do while loop.
5.	Initialization and updating is the part of the syntax.	Initialization and updating is not the part of the syntax.	Initialization and updating is not the part of the syntax

4) If else and switch

BASIS FOR COMPARISON	**IF-ELSE**	**SWITCH**
Basic	Which statement will be executed depend upon the output of the expression inside if statement.	Which statement will be executed is decided by user.
Expression	if-else statement uses multiple statement for multiple choices.	switch statement uses single expression for multiple choices.
Testing	if-else statement test for equality as well as for logical expression.	switch statement test only for equality.
Evaluation	if statement evaluates integer, character, pointer or floating-point type or boolean type.	switch statement evaluates only character or integer value.
Sequence of Execution	Either if statement will be executed or else	switch statement execute one case after another till a

BASIS FOR COMPARISON	IF-ELSE	SWITCH
	statement is executed.	break statement is appeared or the end of switch statement is reached.
Default Execution	If the condition inside if statements is false, then by default the else statement is executed if created.	If the condition inside switch statements does not match with any of cases, for that instance the default statements is executed if created.
Editing	It is difficult to edit the if-else statement, if the nested if-else statement is used.	It is easy to edit switch cases as, they are recognized easily.

9. Functions

Introduction to functions-

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

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

1) create a circle function

2) create a color function

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

Types of function

There are two types of function in C programming:

1) Standard library functions
2) User-defined functions

1) Standard library functions

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

These functions are defined in header files. For example,

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

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

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

Printf():

```
#include <stdio.h>

int main()

{

   printf("Catch me if you can.");

}
```

Math.h header file standard library function-

Sqrt():

```
#include <stdio.h>

#include <math.h>

int main()

{

   float num, root;

   printf("Enter a number: ");

   scanf("%f", &num);

   // Computes the square root of num and stores in root.

   root = sqrt(num);

   printf("Square root of %.2f = %.2f", num, root);

   return 0;

}
```

Pow():

The pow() function computes the power of a number.

The pow() function takes two arguments (base value and power value) and, returns the power raised to the base number. For example,

[Mathematics] xy = pow(x, y) [In programming]

The pow() function is defined in math.h header file.

```
#include <stdio.h>

#include <math.h>

int main()

{

    double base, power, result;

    printf("Enter the base number: ");

    scanf("%lf", &base);

    printf("Enter the power raised: ");
    scanf("%lf",&power);

    result = pow(base,power);

    printf("%.1lf^%.1lf = %.2lf", base, power, result);

    return 0;

}
```

Ceil():

The ceil() function computes the nearest integer greater than the argument passed.

```
#include <stdio.h>

#include <math.h>

int main()

{

   double num = 8.33;

   int result;

   result = ceil(num);

   printf("Ceiling integer of %.2f = %d", num, result);

   return 0;

}
```

Output

```
Ceiling integer of 8.33 = 9
```

Floor():

The floor() function calculates the nearest integer less than the argument passed.

```
#include <stdio.h>

#include <math.h>

int main()

{

   double num = -8.33;
   int result;

   result = floor(num);

   printf("Floor integer of %.2f = %d", num, result);

   return 0;

}
```

Output

```
Floor integer of -8.33 = -9
```

Fabs():

The fabs() function returns the absolute value of a number.

```
#include <stdio.h>

#include <math.h>

int main()

{

    double x, result;

    x = -1.5;

    result = fabs(x);

    printf("|%.2lf| =  %.2lf\n", x, result);


    x = 11.3;

    result = fabs(x);

    printf("|%.2lf| =  %.2lf\n", x, result);

    x = 0;

    result = fabs(x);

    printf("|%.2lf| =  %.2lf\n", x, result);

    return 0;

}
```

Output

```
|-1.50| =  1.50

|11.30| =  11.30

|0.00| =  0.00
```

2) User defined functions:

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

How user-defined function works?

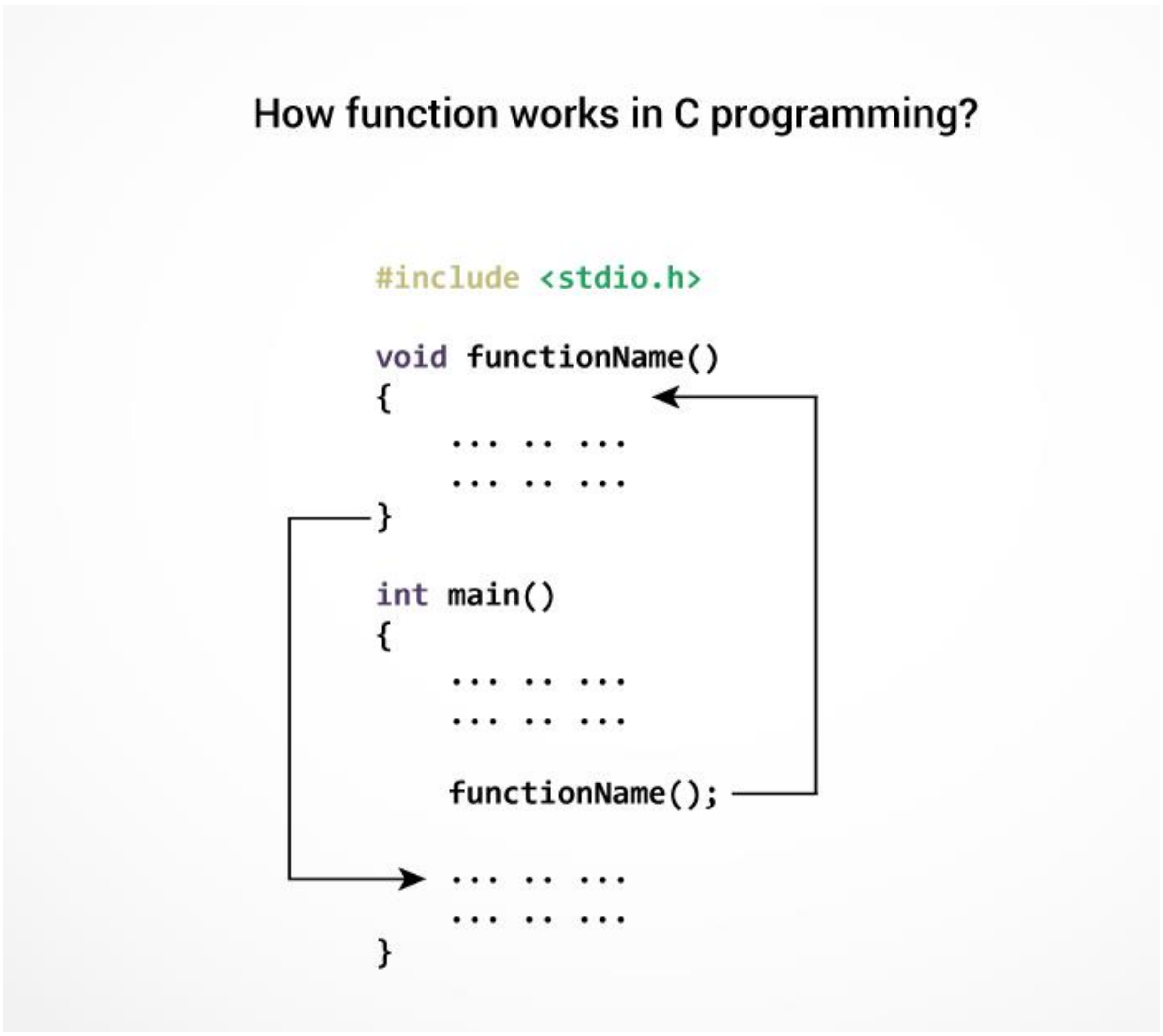
```
#include <stdio.h>
void functionName()
{
    ... .. ...
    ... .. ...
}

int main()
{
    ... .. ...
    ... .. ...

    functionName();

    ... .. ...
    ... .. ...
}
```

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

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

void functionName()

And, the compiler starts executing the codes inside functionName().
The control of the program jumps back to the main() function once code inside the function definition is executed.



```
#include <stdio.h>

void functionName()
{
    ... .. ...
    ... .. ...
}

int main()
{
    ... .. ...
    ... .. ...

    functionName();

    ... .. ...
    ... .. ...
}
```

```
#include <stdio.h>

int addNumbers(int a, int b);         // function prototype

int main()

{

    int n1,n2,sum;

    printf("Enters two numbers: ");

    scanf("%d %d",&n1,&n2);

    sum = addNumbers(n1, n2);        // function call

    printf("sum = %d",sum);

    return 0;
}


int addNumbers(int a, int b)         // function definition

{

    int result;

    result = a+b;

    return result;                   // return statement

}
```

Function prototype

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

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

Syntax of function prototype

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

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

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

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

Calling a function

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

Syntax of function call

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

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

Function definition

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

Syntax of function definition

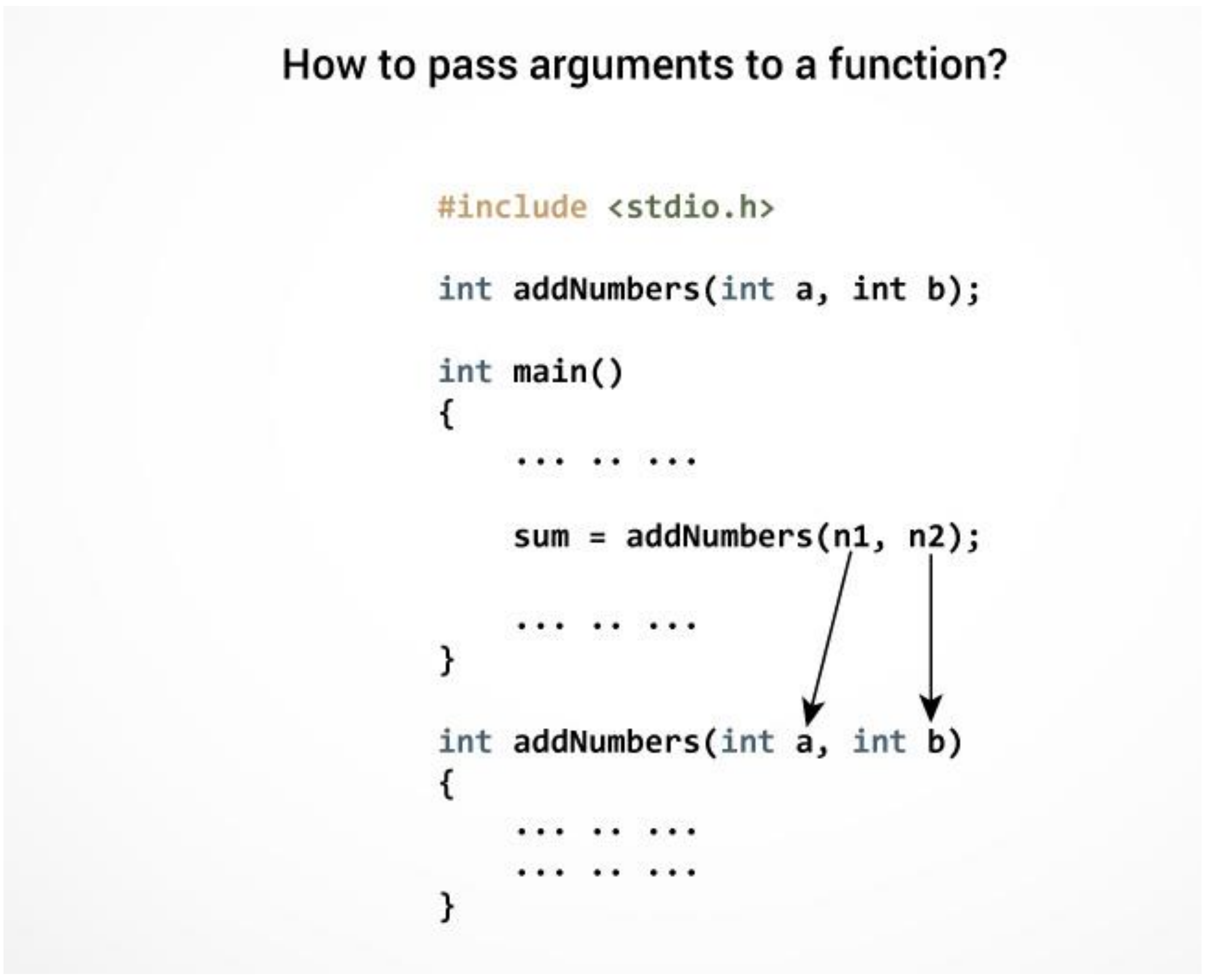
```
returnType functionName(type1 argument1, type2 argument2, ...)

{

   //body of the function

}
```

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

Passing arguments to a function

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

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



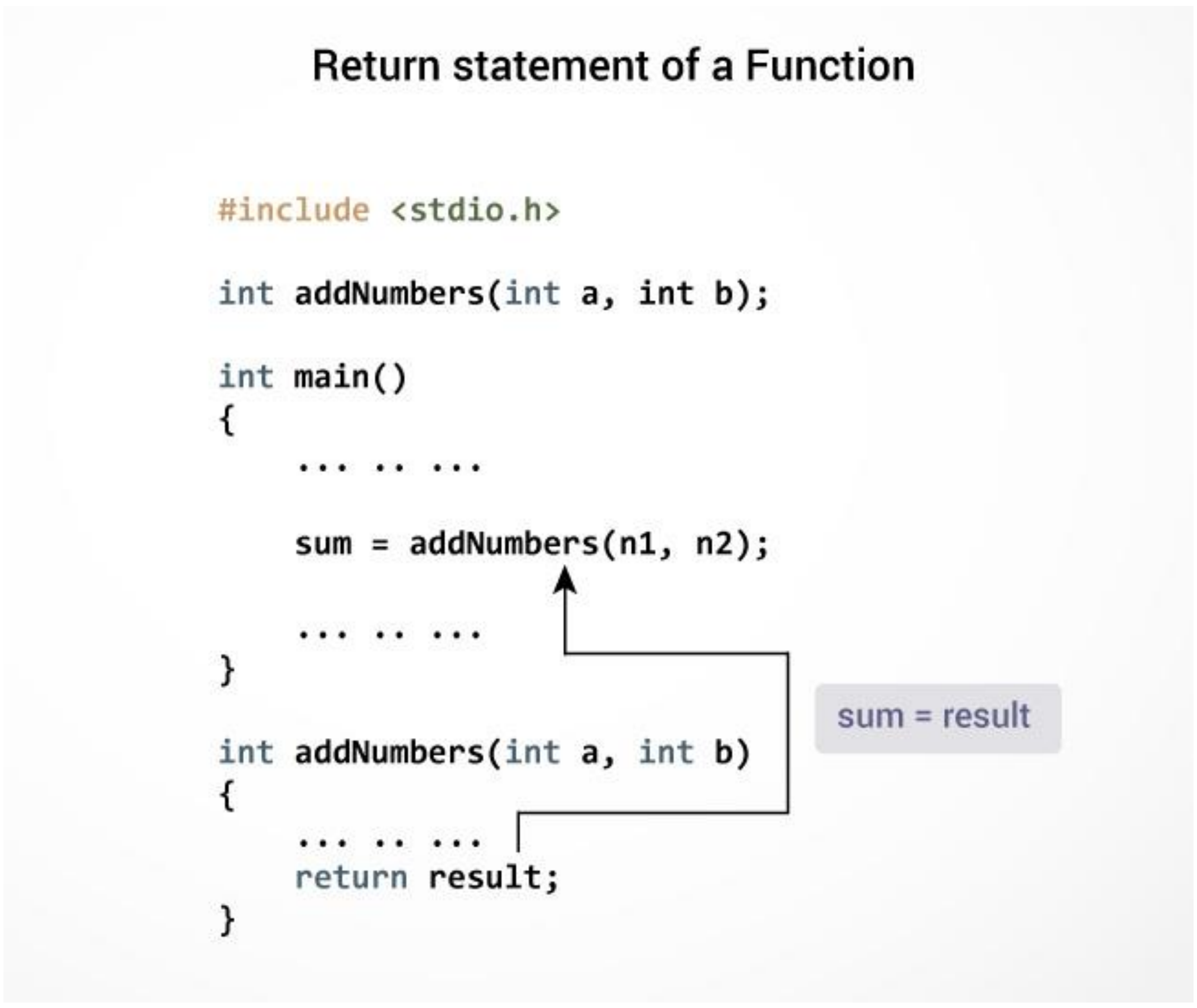
How to pass arguments to a function?

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... .. ...

    sum = addNumbers(n1, n2);

    ... .. ...
}

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

Return Statement

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

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



```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... .. ...

    sum = addNumbers(n1, n2);

    ... .. ...
}

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

Syntax of return statement

return (expression);

For example,

return a;

return (a+b);

Example-

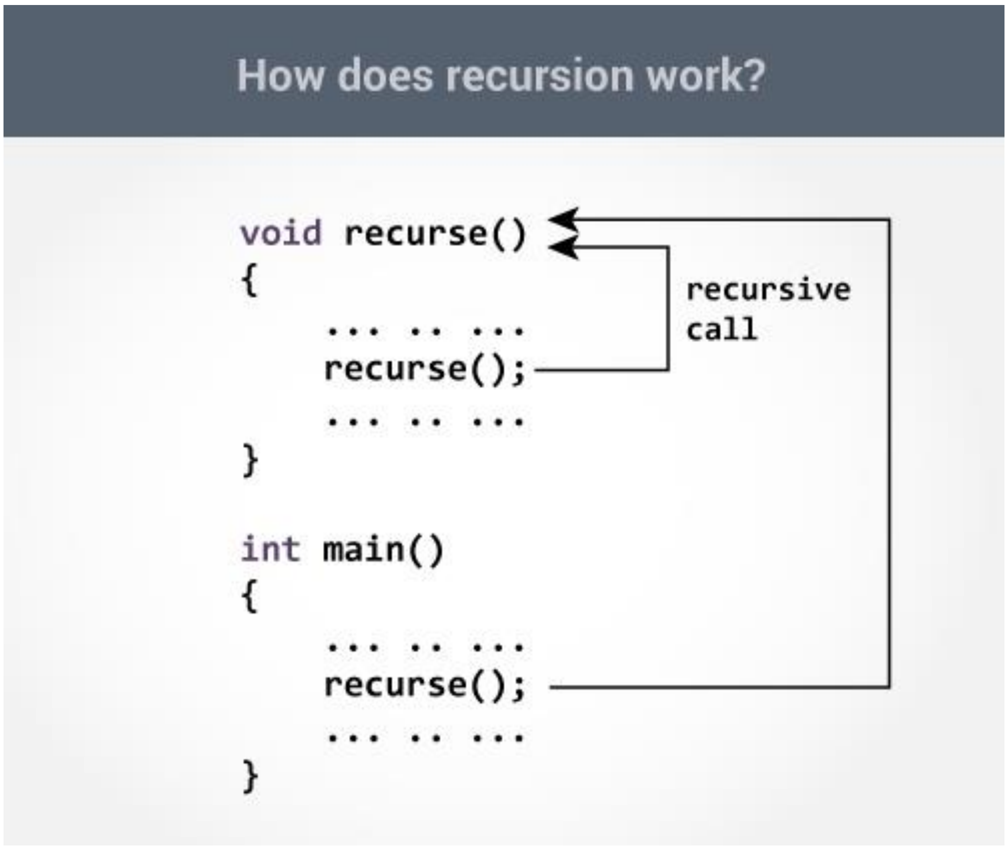
```
#include <stdio.h>

int checkPrimeNumber(int n);

int main()

{

    int n, flag;

    printf("Enter a positive integer: ");

    scanf("%d",&n);

    // n is passed to the checkPrimeNumber() function

    // the returned value is assigned to the flag variable

    flag = checkPrimeNumber(n);

    if(flag == 1)

        printf("%d is not a prime number",n);

    else

        printf("%d is a prime number",n);

    return 0;

}

// int is returned from the function

int checkPrimeNumber(int n)

{

    int i;

    for(i=2; i <= n/2; ++i)

    {

        if(n%i == 0)

            return 1

    return 0;

}
```

10.Recursion

A function that calls itself is known as a recursive function. And, this technique is known as recursion.

How recursion works?

```
void recurse()

{

    ... .. ...

    recurse();

    ... .. ...

}

int main()

{

    ... .. ...

    recurse();

    ... .. ...

}
```



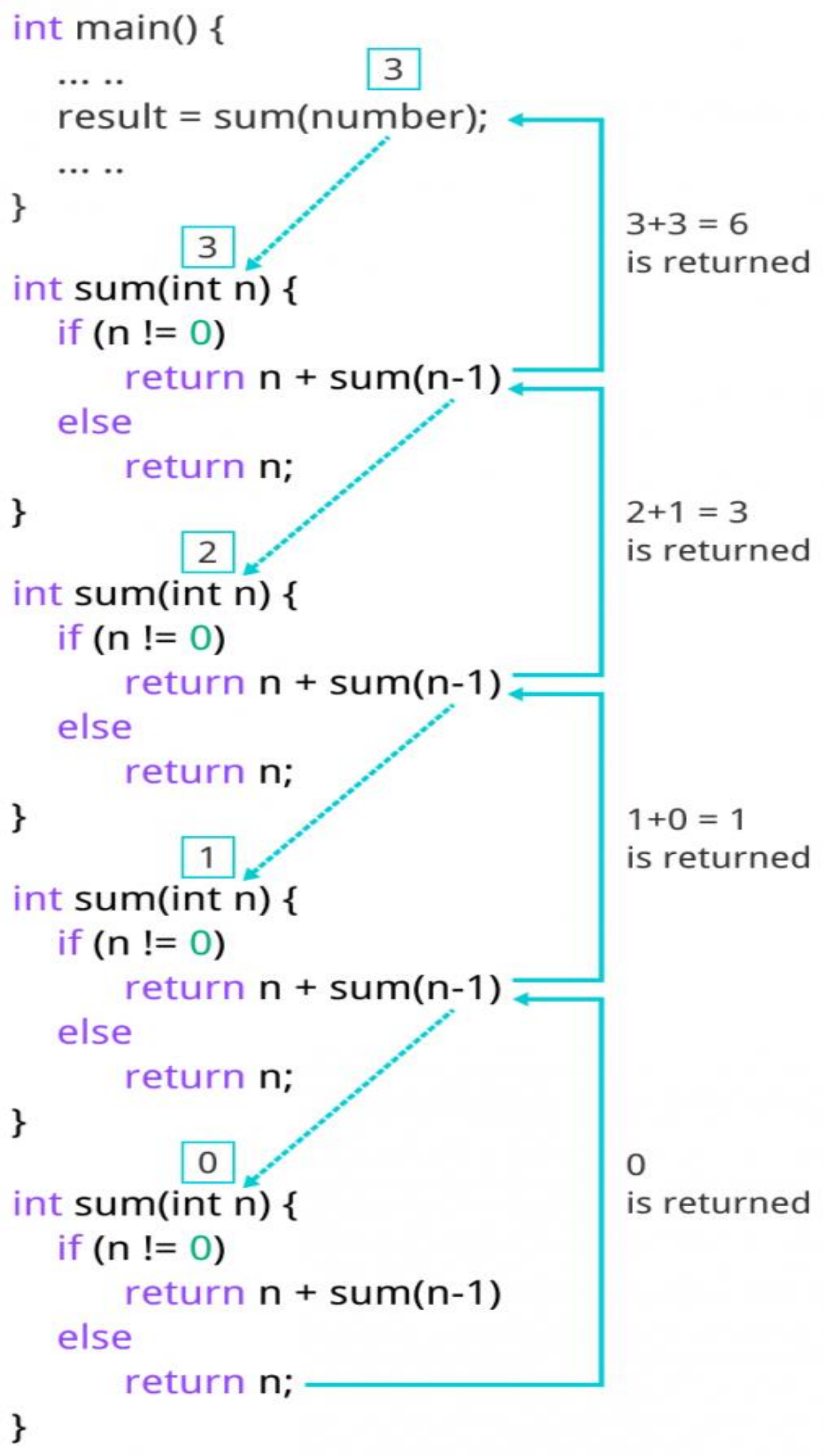
```
void recurse()
{
    ... .. ...
    recurse();
    ... .. ...
}

int main()
{
    ... .. ...
    recurse();
    ... .. ...
}
```

Example: Sum of Natural Numbers Using Recursion

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


int main() {
    int number, result;


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


    result = sum(number);


    printf("sum = %d", result);
    return 0;
}


int sum(int n) {
    if (n != 0)
        // sum() function calls itself
        return n + sum(n-1);
    else
        return n;
}
```

Output

```
Enter a positive integer:3
sum = 6
```



```
int main() {
   ... ..
   result = sum(number);
   ... ..
}

int sum(int n) {
   if (n != 0)
       return n + sum(n-1)
   else
       return n;
}

int sum(int n) {
   if (n != 0)
       return n + sum(n-1)
   else
       return n;
}

int sum(int n) {
   if (n != 0)
       return n + sum(n-1)
   else
       return n;
}

int sum(int n) {
   if (n != 0)
       return n + sum(n-1)
   else
       return n;
}
```

Advantages

1) Reduce unnecessary calling of function.
2) Through Recursion one can Solve problems in easy way while its iterative solution is very big and complex.

Disdvantages

1) Recursive solution is always logical and it is very difficult to trace.(debug and understand).
2) In recursive we must have an if statement somewhere to force the function to return without the recursive call being executed, otherwise the function will never return.
3) Recursion takes a lot of stack space, usually not considerable when the program is small and running on a PC.
4) Recursion uses more processor time.

11.Storage classes in C

A storage class represents the visibility and a location of a variable. It tells from what part of code we can access a variable. A storage class is used to describe the following things:

1) The variable scope.
2) The location where the variable will be stored.
3) The initialized value of a variable.
4) A lifetime of a variable.
5) Who can access a variable?

Storage classes

Auto-It is a default storage class.

Extern-It is a global variable.

Static-It is a local variable which is capable of returning a value even when control is transferred to the function call.

Register-It is a variable which is stored inside a Register.

Auto storage class

The variables defined using auto storage class are called as local variables. Auto stands for automatic storage class. A variable is in auto storage class by default if it is not explicitly specified.

The scope of an auto variable is limited with the particular block only. Once the control goes out of the block, the access is destroyed. This means only the block in which the auto variable is declared can access it.

A keyword auto is used to define an auto storage class. By default, an auto variable contains a garbage value.

Example, auto int age;

The program below defines a function with has two local variables

```
int add(void) {

   int a=13;

   auto int b=48;

return a+b;}
```

We take another program which shows the scope level "visibility level" for auto variables in each block code which are independently to each other:

```
#include <stdio.h>

int main( )

{

  auto int j = 1;

  {

    auto int j= 2;

    {

      auto int j = 3;

      printf ( " %d ", j);

    }

    printf ( "\t %d ",j);

  }

  printf( "%d\n", j);}
```

OUTPUT:

```
 3 2 1
```

Extern storage class

Extern stands for external storage class. Extern storage class is used when we have global functions or variables which are shared between two or more files.

Keyword extern is used to declaring a global variable or function in another file to provide the reference of variable or function which have been already defined in the original file.

The variables defined using an extern keyword are called as global variables. These variables are accessible throughout the program. Notice that the extern variable cannot be initialized it has already been defined in the original file

Example, extern void display();

First File: main.c

```
#include <stdio.h>

extern i;
```

```
main() {
   printf("value of the external integer is = %d\n", i);
   return 0;}
```

Second File: original.c

```
#include <stdio.h>
i=48;
```

Result:

```
 value of the external integer is = 48
```

Static storage class

The static variables are used within function/ file as local static variables. They can also be used as a global variable

Static local variable is a local variable that retains and stores its value between function calls or block and remains visible only to the function or block in which it is defined.

Static global variables are global variables visible only to the file in which it is declared.

Example: static int count = 10;

Keep in mind that static variable has a default initial value zero and is initialized only once in its lifetime.

```
#include <stdio.h> /* function declaration */
void next(void);
static int counter = 7; /* global variable */
main() {
 while(counter<10) {
      next();
      counter++;   }
return 0;}
void next( void ) {    /* function definition */
   static int iteration = 13; /* local static variable */
   iteration ++;

   printf("iteration=%d and counter= %d\n", iteration, counter);}
```

Result:

```
iteration=14 and counter= 7

iteration=15 and counter= 8

iteration=16 and counter= 9
```

Global variables are accessible throughout the file whereas static variables are accessible only to the particular part of a code.

The lifespan of a static variable is in the entire program code. A variable which is declared or initialized using static keyword always contains zero as a default value.

Register storage class

You can use the register storage class when you want to store local variables within functions or blocks in CPU registers instead of RAM to have quick access to these variables. For example, "counters" are a good candidate to be stored in the register.

Example: register int age;

The keyword register is used to declare a register storage class. The variables declared using register storage class has lifespan throughout the program.

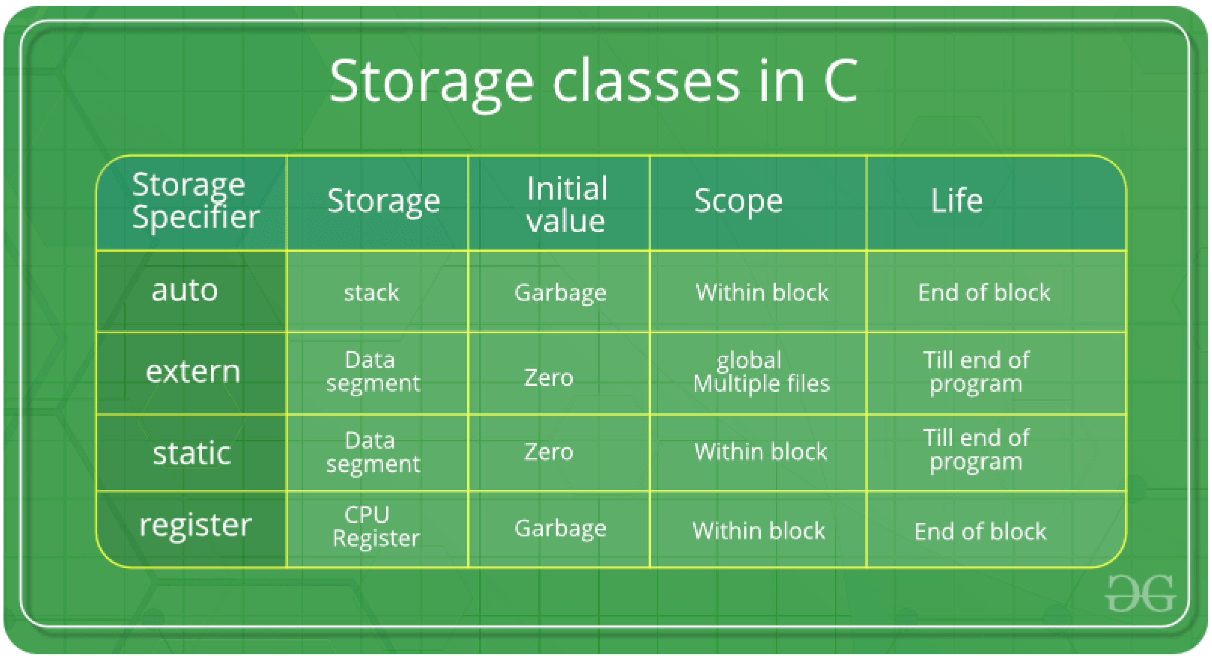
It is similar to the auto storage class. The variable is limited to the particular block. The only difference is that the variables declared using register storage class are stored inside CPU registers instead of a memory. Register has faster access than that of the main memory.

The variables declared using register storage class has no default value. These variables are often declared at the beginning of a program.

```
#include <stdio.h> /* function declaration */

main() {

{

register int  weight;

int *ptr=&weight ;/*it produces an error when the compilation occurs ,we
cannot get a memory location when dealing with CPU register*/}

}
```

OUTPUT:

```
error: address of register variable 'weight' requested
```



Storage classes in C

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

12 Arrays

An array is a variable that can store multiple values of same data type. For example, if you want to store 100 integers, you can create an array for it.

int data[100];

How to declare an array?

dataType arrayName[arraySize];

For example,

float mark[5];

Here, we declared an array, mark, of floating-point type. And its size is 5. Meaning, it can hold 5 floating-point values.

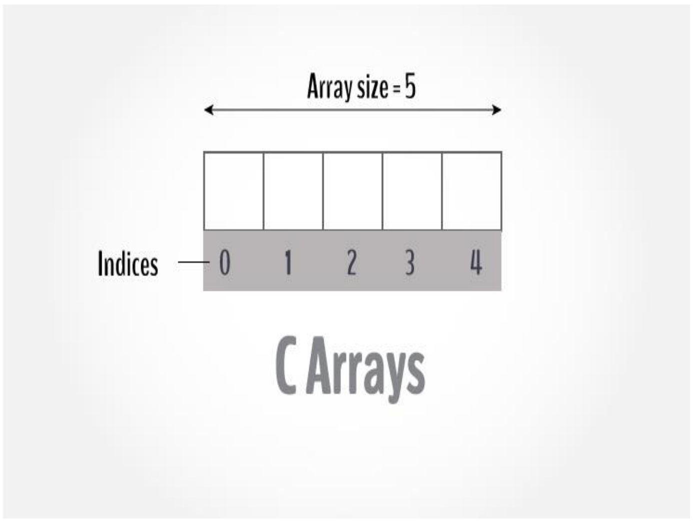
It's important to note that the size and type of an array cannot be changed once it is declared.

Types of array:-

1)Single dimensional array

2)Multidimentional array

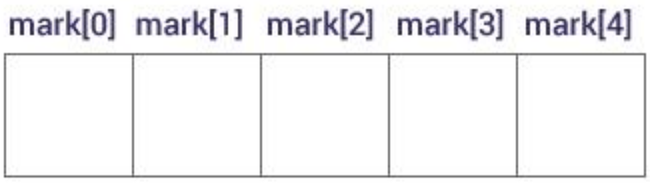
1)Single dimension array:-



Access Array Elements

You can access elements of an array by indices.

Suppose you declared an array mark as above. The first element is mark[0], the second element is mark[1] and so on.



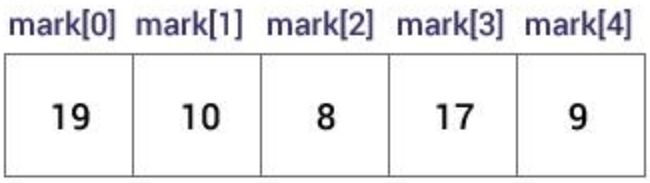
How to initialize an array?

It is possible to initialize an array during declaration. For example,

int mark[5] = {19, 10, 8, 17, 9};

You can also initialize an array like this.

int mark[] = {19, 10, 8, 17, 9};



Example-

```
// Program to take 5 values from the user and store them in an array

// Print the elements stored in the array

#include <stdio.h>

int main() {

  int values[5];

  printf("Enter 5 integers: ");

  // taking input and storing it in an array

  for(int i = 0; i < 5; ++i) {

     scanf("%d", &values[i]);

  }

  printf("Displaying integers: ");

  // printing elements of an array

  for(int i = 0; i < 5; ++i) {

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

Output

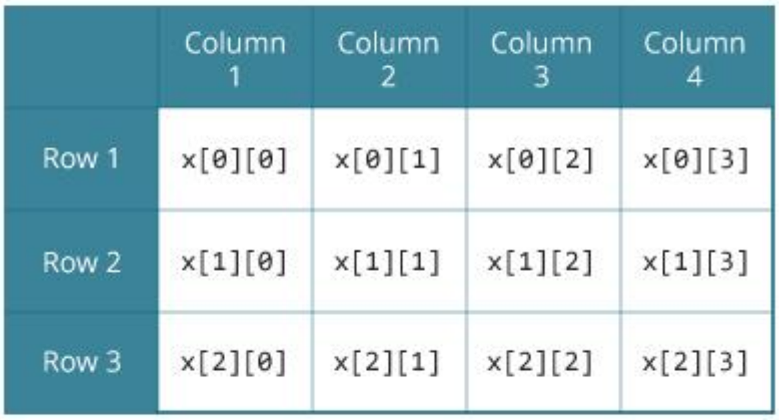
```
Enter 5 integers: 1
-3
34
0
3
Displaying integers: 1
-3
34
0
3
```

2)Multidimension array(2D,3D….)

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

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

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



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

Initialization of a 2d array

```
// Different ways to initialize two-dimensional array

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

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

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

Initialization of a 3d array

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

```
int test[2][3][4] = {

    {{3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2}},

    {{13, 4, 56, 3}, {5, 9, 3, 5}, {3, 1, 4, 9}}};
```

Example 2D array-

```
// C program to store temperature of two cities of a week and display it.

#include <stdio.h>

const int CITY = 2;

const int WEEK = 7;

int main()

{

  int temperature[CITY][WEEK];

  // Using nested loop to store values in a 2d array

  for (int i = 0; i < CITY; ++i)

  {

    for (int j = 0; j < WEEK; ++j)

    {

      printf("City %d, Day %d: ", i + 1, j + 1);

      scanf("%d", &temperature[i][j]);

    }
  }
  printf("\nDisplaying values: \n\n");

  // Using nested loop to display vlues of a 2d array
  for (int i = 0; i < CITY; ++i)
  {
    for (int j = 0; j < WEEK; ++j)
    {
      printf("City %d, Day %d = %d\n", i + 1, j + 1, temperature[i][j]);
    }
  }
  return 0;
}
```

Example 3D array

```
// C Program to store and print 12 values entered by the user

#include <stdio.h>
int main()
{
  int test[2][3][2];

  printf("Enter 12 values: \n");

  for (int i = 0; i < 2; ++i)
  {
    for (int j = 0; j < 3; ++j)
    {
      for (int k = 0; k < 2; ++k)
      {
        scanf("%d", &test[i][j][k]);
      }
    }
  }

  // Printing values with proper index.

  printf("\nDisplaying values:\n");
  for (int i = 0; i < 2; ++i)
  {
    for (int j = 0; j < 3; ++j)
    {
      for (int k = 0; k < 2; ++k)
      {
        printf("test[%d][%d][%d] = %d\n", i, j, k, test[i][j][k]);
      }
    }
  }
  return 0;
}
```

Passing an array element to function-

```
#include <stdio.h>
void display(int age1, int age2)
{
    printf("%d\n", age1);
    printf("%d\n", age2);
}

int main()
{
    int ageArray[] = {2, 8, 4, 12};
    // Passing second and third elements to display()
    display(ageArray[1], ageArray[2]);
    return 0;
}
```

Output

```
8
4
```

Passing entire element to function-

```
// Program to calculate the sum of array elements by passing to a function
#include <stdio.h>
float calculateSum(float age[]);

int main() {
    float result, age[] = {23.4, 55, 22.6, 3, 40.5, 18};

    // age array is passed to calculateSum()
    result = calculateSum(age);
    printf("Result = %.2f", result);
    return 0;
}
```

```
float calculateSum(float age[]) {


  float sum = 0.0;


  for (int i = 0; i < 6; ++i) {
                 sum += age[i];
  }
  return sum;
}
```

Output

```
Result = 162.50
```

Passing two D array to function

```
#include <stdio.h>
void displayNumbers(int num[2][2]);
int main()
{
    int num[2][2];
    printf("Enter 4 numbers:\n");
    for (int i = 0; i < 2; ++i)
        for (int j = 0; j < 2; ++j)
            scanf("%d", &num[i][j]);

    // passing multi-dimensional array to a function
    displayNumbers(num);
    return 0;
}
```

```
void displayNumbers(int num[2][2])
{
    printf("Displaying:\n");
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 2; ++j) {
           printf("%d\n", num[i][j]);
        }
    }
}
```

13 Pointer in C

Pointers are powerful features of C and C++ programming. Before we learn pointers, let's learn about addresses in C programming.

Address in C

If you have a variable var in your program, &var will give you its address in the memory.

We have used address numerous times while using the scanf() function.

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

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

```
#include <stdio.h>

int main()

{

  int var = 5;

  printf("var: %d\n", var);

  // Notice the use of & before var

  printf("address of var: %p", &var);

  return 0;

}
```

Output

```
var: 5

address of var: 2686778
```

C Pointers

Pointers (pointer variables) are special variables that are used to store addresses rather than values.

Pointer Syntax

Here is how we can declare pointers.

```
int* p;
```

Here, we have declared a pointer p of int type.

You can also declare pointers in these ways.

```
int *p1;

int * p2;
```

Assigning addresses to Pointers

Let's take an example.

```
int* pc, c;

c = 5;

pc = &c;
```

Here, 5 is assigned to the c variable. And, the address of c is assigned to the pc pointer.

Get Value of Thing Pointed by Pointers

To get the value of the thing pointed by the pointers, we use the * operator. For example:

```
int* pc, c;

c = 5;

pc = &c;

printf("%d", *pc);   // Output: 5
```

Example: Working of Pointers

Let's take a working example.

```
#include <stdio.h>

int main()

{

   int* pc, c;



   c = 22;

   printf("Address of c: %p\n", &c);

   printf("Value of c: %d\n\n", c);  // 22
   pc = &c;
   printf("Address of pointer pc: %p\n", pc);
   printf("Content of pointer pc: %d\n\n", *pc); // 22

   c = 11;
   printf("Address of pointer pc: %p\n", pc);
   printf("Content of pointer pc: %d\n\n", *pc); // 11

   *pc = 2;
   printf("Address of c: %p\n", &c);
   printf("Value of c: %d\n\n", c); // 2
   return 0;
}
```

Output

```
Address of c: 2686784
Value of c: 22
Address of pointer pc: 2686784
Content of pointer pc: 22
Address of pointer pc: 2686784
Content of pointer pc: 11
Address of c: 2686784
Value of c: 2
```

Example: Working of Pointers

Let's take a working example.

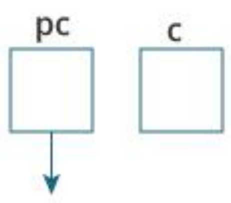
```
#include <stdio.h>

int main()

{

   int* pc, c;

   c = 22;

   printf("Address of c: %p\n", &c);

   printf("Value of c: %d\n\n", c);  // 22

   pc = &c;

   printf("Address of pointer pc: %p\n", pc);

   printf("Content of pointer pc: %d\n\n", *pc); // 22

   c = 11;

   printf("Address of pointer pc: %p\n", pc);

   printf("Content of pointer pc: %d\n\n", *pc); // 11

   *pc = 2;

   printf("Address of c: %p\n", &c);

   printf("Value of c: %d\n\n", c); // 2

   return 0;

}
```

Output

```
Address of c: 2686784

Value of c: 22

Address of pointer pc: 2686784

Content of pointer pc: 22

Address of pointer pc: 2686784

Content of pointer pc: 11

Address of c: 2686784

Value of c: 2
```

Explanation of the program

int* pc, c;

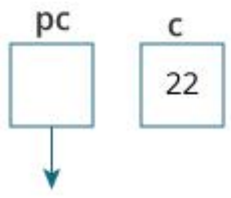


A pointer variable and a normal variable is created.

Here, a pointer pc and a normal variable c, both of type int, is created.

Since pc and c are not initialized at initially, pointer pc points to either no address or a random address. And, variable c has an address but contains random garbage value.

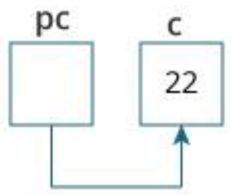
c = 22;



22 is assigned to variable c.

This assigns 22 to the variable c. That is, 22 is stored in the memory location of variable c.

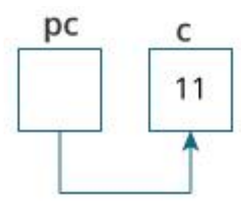
pc = &c;



Address of variable c is assigned to pointer pc.

This assigns the address of variable c to the pointer pc.

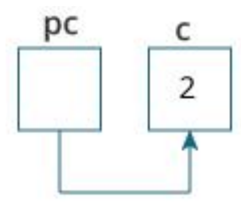
c = 11;



11 is assigned to variable c.

This assigns 11 to variable c.

*pc = 2;



This change the value at the memory location pointed by the pointer pc to 2.

Arrays and pointer

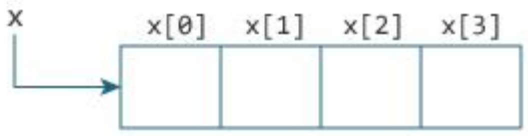
Relationship Between Arrays and Pointers

An array is a block of sequential data. Let's write a program to print addresses of array elements.

```
#include <stdio.h>

int main() {

  int x[4];

  int i;


  for(i = 0; i < 4; ++i) {

    printf("&x[%d] = %p\n", i, &x[i]);

  }

  printf("Address of array x: %p", x);

  return 0;

}
```

Output

```
&x[0] = 1450734448

&x[1] = 1450734452

&x[2] = 1450734456

&x[3] = 1450734460

Address of array x: 1450734448
```



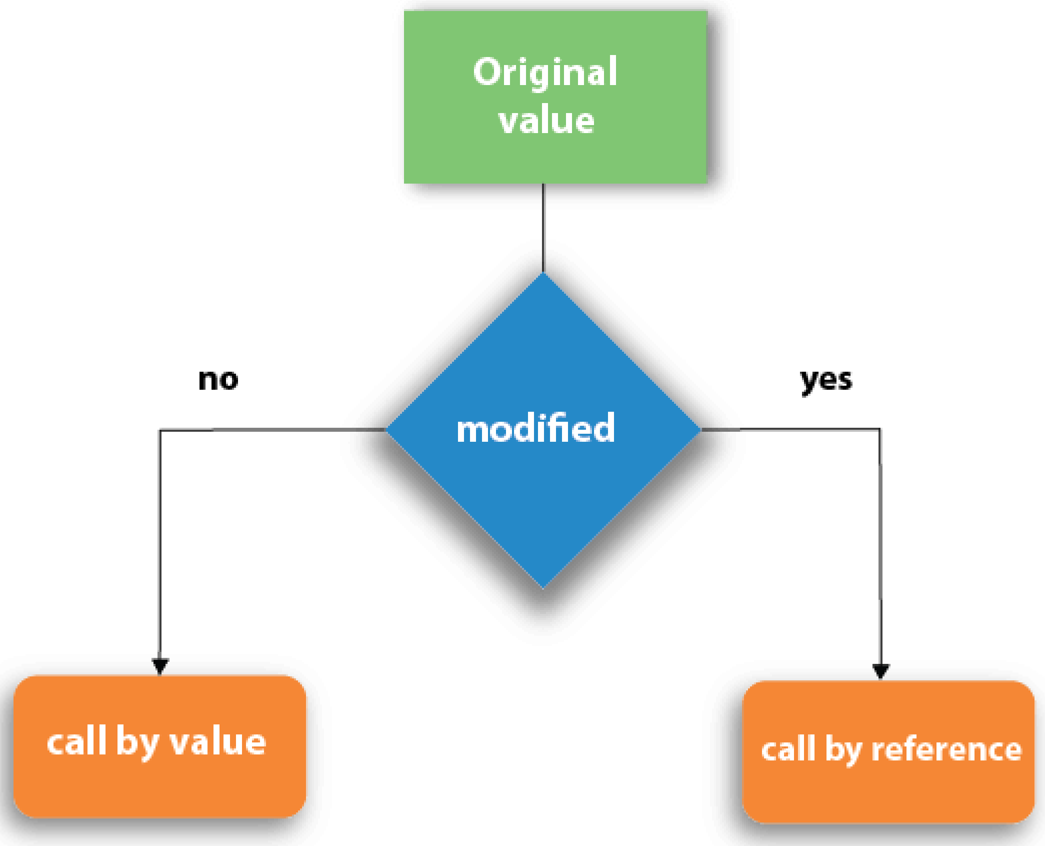
Example 1: Pointers and Arrays

```
#include <stdio.h>

int main() {

  int i, x[6], sum = 0;

  printf("Enter 6 numbers: ");

  for(i = 0; i < 6; ++i) {

  // Equivalent to scanf("%d", &x[i]);

      scanf("%d", x+i);

  // Equivalent to sum += x[i]

      sum += *(x+i);

  }

  printf("Sum = %d", sum);

  return 0;

}
```

When you run the program, the output will be:

```
Enter 6 numbers:  2

 3

 4

 4
 12

 4

Sum = 29
```

Pointer and function



1) Call by value

    1) In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.
    2) In call by value method, we can not modify the value of the actual parameter by the formal parameter.
    3) In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
    4) The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

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

```
#include <stdio.h>

void swap(int , int); //prototype of the function

int main()

{

    int a = 10;

    int b = 20;

    printf("Before swapping the values in main a = %d, b = %d\n",a,b); //
printing the value of a and b in main

    swap(a,b);

    printf("After swapping values in main a = %d, b = %d\n",a,b); // The value
of actual parameters do not change by changing the formal parameters in call by
value, a = 10, b = 20

}

void swap (int a, int b)

{

    int temp;

    temp = a;

    a=b;

    b=temp;

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

}
```

Output

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

After swapping values in function a = 20, b = 10

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

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

2) The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
3) In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

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

```
#include <stdio.h>

void swap(int *, int *); //prototype of the function

int main()

{

    int a = 10;

    int b = 20;

    printf("Before swapping the values in main a = %d, b = %d\n",a,b); //
printing the value of a and b in main

    swap(&a,&b);

    printf("After swapping values in main a = %d, b = %d\n",a,b); // The values
of actual parameters do change in call by reference, a = 10, b = 20

}

void swap (int *a, int *b)

{

    int temp;

    temp = *a;

    *a=*b;

    *b=temp;

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

}
```

Output

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

After swapping values in function a = 20, b = 10

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

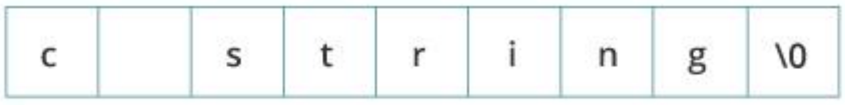
No.	Call by value	Call by reference
1	A copy of the value is passed into the function	An address of value is passed into the function
2	Changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters.	Changes made inside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

14 Strings in C

In C programming, a string is a sequence of characters terminated with a null character \0. For example:

char c[] = "c string";

When the compiler encounters a sequence of characters enclosed in the double quotation marks, it appends a null character \0 at the end by default.

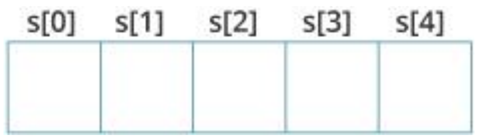


c		s	t	r	i	n	g	\0

How to declare a string?

Here's how you can declare strings:

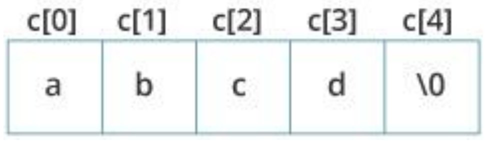
char s[5];



How to initialize strings?

You can initialize strings in a number of ways.

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

char c[50] = "abcd";

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

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



Assigning Values to Strings

Arrays and strings are second-class citizens in C; they do not support the assignment operator once it is declared. For example,

```
char c[100];

c = "C programming";  // Error! array type is not assignable.
```

Read String from the user

You can use the scanf() function to read a string.

The scanf() function reads the sequence of characters until it encounters whitespace (space, newline, tab, etc.).

Example 1: scanf() to read a string

```
#include <stdio.h>

int main()

{

    char name[20];

    printf("Enter name: ");

    scanf("%s", name);

    printf("Your name is %s.", name);
    return 0;

}
```

Output

```
Enter name: Dennis Ritchie

Your name is Dennis.
```

gets() and puts()

Functions gets() and puts() are two string functions to take string input from the user and display it respectively as mentioned in the previous chapter.

```
#include<stdio.h>

int main()

{

    char name[30];

    printf("Enter name: ");

    gets(name);     //Function to read string from user.

    printf("Name: ");

    puts(name);    //Function to display string.

    return 0;

}
```

String Library functions-

Commonly Used String Functions

1)strlen() - calculates the length of a string

2)strcpy() - copies a string to another

3)strcmp() - compares two strings

4)strcat() - concatenates two strings

Strings handling functions are defined under "string.h" header file.

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

1) **Strlen-**
   The strlen() function takes a string as an argument and returns its length. The returned value is of type long int.

   It is defined in the <string.h> header file.

```
#include <stdio.h>
#include <string.h>
int main()
{
   char a[20]="Program";
   char b[20]={'P','r','o','g','r','a','m','\0'};

   printf("Length of string a = %ld \n",strlen(a));
   printf("Length of string b = %ld \n",strlen(b));

   return 0;
}
```

   Output

```
Length of string a = 7
Length of string b = 7
```

2) **strcpy():-**

The function prototype of strcpy() is:

char* strcpy(char* destination, const char* source);

The strcpy() function copies the string pointed by source (including the null character) to the destination.

The strcpy() function also returns the copied string.

The strcpy() function is defined in the string.h header file.

Example: C strcpy()

```
#include <stdio.h>

#include <string.h>

int main() {

  char str1[20] = "C programming";

  char str2[20];
```

```
   // copying str1 to str2

   strcpy(str2, str1);

   puts(str2); // C programming

   return 0;

}
```

Output

```
C programming
```

3)strcpy:-

The function definition of strcat() is:

```
char *strcat(char *destination, const char *source)
```

It is defined in the string.h header file.

strcat() arguments
As you can see, the strcat() function takes two arguments:

destination - destination string
source - source string

The strcat() function concatenates the destination string and the source string, and the result is stored in the destination string.

Example: C strcat() function

```
#include <stdio.h>
#include <string.h>
int main() {
   char str1[100] = "This is ", str2[] = "Vineet.com";

   // concatenates str1 and str2
   // the resultant string is stored in str1.
   strcat(str1, str2);
   puts(str1);
   puts(str2);
   return 0;
}
```

Output

```
This is Vineet.com
Vineet.com
```

4) strcmp()

The strcmp() function compares two strings and returns 0 if both strings are identical.

C strcmp() Prototype

int strcmp (const char* str1, const char* str2);

The strcmp() function takes two strings and returns an integer.

The strcmp() compares two strings character by character.

If the first character of two strings is equal, the next character of two strings are compared. This continues until the corresponding characters of two strings are different or a null character '\0' is reached.

It is defined in the string.h header file.

Return Value from strcmp()

Return ValueRemarks

0	if both strings are identical (equal)
negative	if the ASCII value of the first unmatched character is less than second.
positive integer than second.	if the ASCII value of the first unmatched character is greater

Example: C strcmp() function

```
#include <stdio.h>

#include <string.h>

int main()

{

   char str1[] = "abcd", str2[] = "abCd", str3[] = "abcd";

   int result;

   // comparing strings str1 and str2

   result = strcmp(str1, str2);

   printf("strcmp(str1, str2) = %d\n", result);
```

```
    // comparing strings str1 and str3

    result = strcmp(str1, str3);

    printf("strcmp(str1, str3) = %d\n", result);

    return 0;

}
```

Output

```
strcmp(str1, str2) = 32

strcmp(str1, str3) = 0
```

The first unmatched character between string str1 and str2 is third character. The ASCII value of 'c' is 99 and the ASCII value of 'C' is 67. Hence, when strings str1 and str2 are compared, the return value is 32.

When strings str1 and str3 are compared, the result is 0 because both strings are identical.

15.Structure and Unions

In C programming, a struct (or structure) is a collection of variables (can be of different types) under a single name.

How to define structures?

Before you can create structure variables, you need to define its data type. To define a struct, the struct keyword is used.

Syntax of struct

```
struct structureName

{

    dataType member1;

    dataType member2;

    ...

};
```

Here is an example:

```
struct Person

{

    char name[50];

    int citNo;

    float salary;

};
```

Here, a derived type struct Person is defined. Now, you can create variables of this type.

Ways of creating structures:-

1) **struct Person person1**

```
        struct Person

        {

    char name[50];

    int citNo;
    float salary;
};
int main()
{
    struct Person person1, person2, p[20];
    return 0;
}
```

2) struct Person

```
{
    char name[50];
    int citNo;
    float salary;
} person1, person2, p[20];
```

Accessing struct variables-

```
// Program to add two distances (feet-inch)
#include <stdio.h>
struct Distance
{
    int feet;
    float inch;
} dist1, dist2, sum;
int main()
{
    printf("1st distance\n");
    printf("Enter feet: ");
    scanf("%d", &dist1.feet);
    printf("Enter inch: ");
    scanf("%f", &dist1.inch);
    printf("2nd distance\n");
    printf("Enter feet: ");
    scanf("%d", &dist2.feet);
    printf("Enter inch: ");
    scanf("%f", &dist2.inch);
    // adding feet
    sum.feet = dist1.feet + dist2.feet;
    // adding inches
    sum.inch = dist1.inch + dist2.inch;
    // changing to feet if inch is greater than 12
    while (sum.inch >= 12)
    {
        ++sum.feet;
        sum.inch = sum.inch - 12;
    }
    printf("Sum of distances = %d\'-%.1f\"", sum.feet, sum.inch);
    return 0;
}
```

Why structs in C?

Suppose, you want to store information about a person: his/her name, citizenship number, and salary. You can create different variables name, citNo and salary to store this information.

What if you need to store information of more than one person? Now, you need to create different variables for each information per person: name1, citNo1, salary1, name2, citNo2, salary2, etc.

A better approach would be to have a collection of all related information under a single name Person structure and use it for every person.

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.

```
#include<stdio.h>

struct address

{

    char city[20];

    int pin;

    char phone[14];

};

struct employee

{

    char name[20];

    struct address add;

};

void main ()

{

    struct employee emp;

    printf("Enter employee information?\n");

    scanf("%s %s %d %s",emp.name,emp.add.city, &emp.add.pin,
emp.add.phone);

    printf("Printing the employee information....\n");

    printf("name: %s\nCity: %s\nPincode: %d\nPhone:
%s",emp.name,emp.add.city,emp.add.pin,emp.add.phone);
}
```

The structure can be nested in the following ways.

1) By separate structure

2) By Embedded structure

1) Separate structure

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

```
struct Date

{

   int dd;

   int mm;

   int yyyy;

};

struct Employee

{

   int id;

   char name[20];

   struct Date doj;

}emp1;
```

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

2) Embedded structure

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

```
struct Employee
```

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

Accessing Nested Structure

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

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

C Nested Structure example

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

```
#include <stdio.h>
#include <string.h>
struct Employee
{
   int id;
   char name[20];
   struct Date
    {
      int dd;
      int mm;

      int yyyy;

    }doj;

}e1;

int main( )

{

   //storing employee information

   e1.id=101;

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

   e1.doj.dd=10;

   e1.doj.mm=11;

   e1.doj.yyyy=2014;


   //printing first employee information

   printf( "employee id : %d\n", e1.id);

   printf( "employee name : %s\n", e1.name);

   printf( "employee date of joining (dd/mm/yyyy) : %d/%d/%d\n",
e1.doj.dd,e1.doj.mm,e1.doj.yyyy);

   return 0;

}
```

Passing structure to functions-

```
#include <stdio.h>

struct student {

   char name[50];

   int age;

};


// function prototype
```

```
void display(struct student s);


int main() {
   struct student s1;


   printf("Enter name: ");


   // read string input from the user until \n is entered
   // \n is discarded
   scanf("%[^\n]%*c", s1.name);


   printf("Enter age: ");
   scanf("%d", &s1.age);


   display(s1); // passing struct as an argument


   return 0;
}


void display(struct student s) {
   printf("\nDisplaying information\n");
   printf("Name: %s", s.name);
   printf("\nAge: %d", s.age);
}
```

Returning structure from function-

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

```
{
    char name[50];
    int age;
};

// function prototype
struct student getInformation();

int main()
{
    struct student s;

    s = getInformation();

    printf("\nDisplaying information\n");
    printf("Name: %s", s.name);
    printf("\nRoll: %d", s.age);

    return 0;
}
struct student getInformation()
{
  struct student s1;

  printf("Enter name: ");
  scanf ("%[^\n]%*c", s1.name);
  printf("Enter age: ");

  scanf("%d", &s1.age);

  return s1;

}
```

Unions:

A union is a user-defined type similar to structs in C programming.

How to define a union?

We use the union keyword to define unions. Here's an example:

```
union car

{

  char name[50];

  int price;

};
```

The above code defines a derived type union car.

Create union variables

When a union is defined, it creates a user-defined type. However, no memory is allocated. To allocate memory for a given union type and work with it, we need to create variables.

Here's how we create union variables.

1) 1st way

```
union car

{

  char name[50];

  int price;

};

int main()

{

  union car car1, car2, *car3;
  return 0;
}
```

2) Another way of creating union variables is:

```
union car
{
  char name[50];
  int price;
} car1, car2, *car3;
```

Access members of a union

We use the . operator to access members of a union.

In the above example,

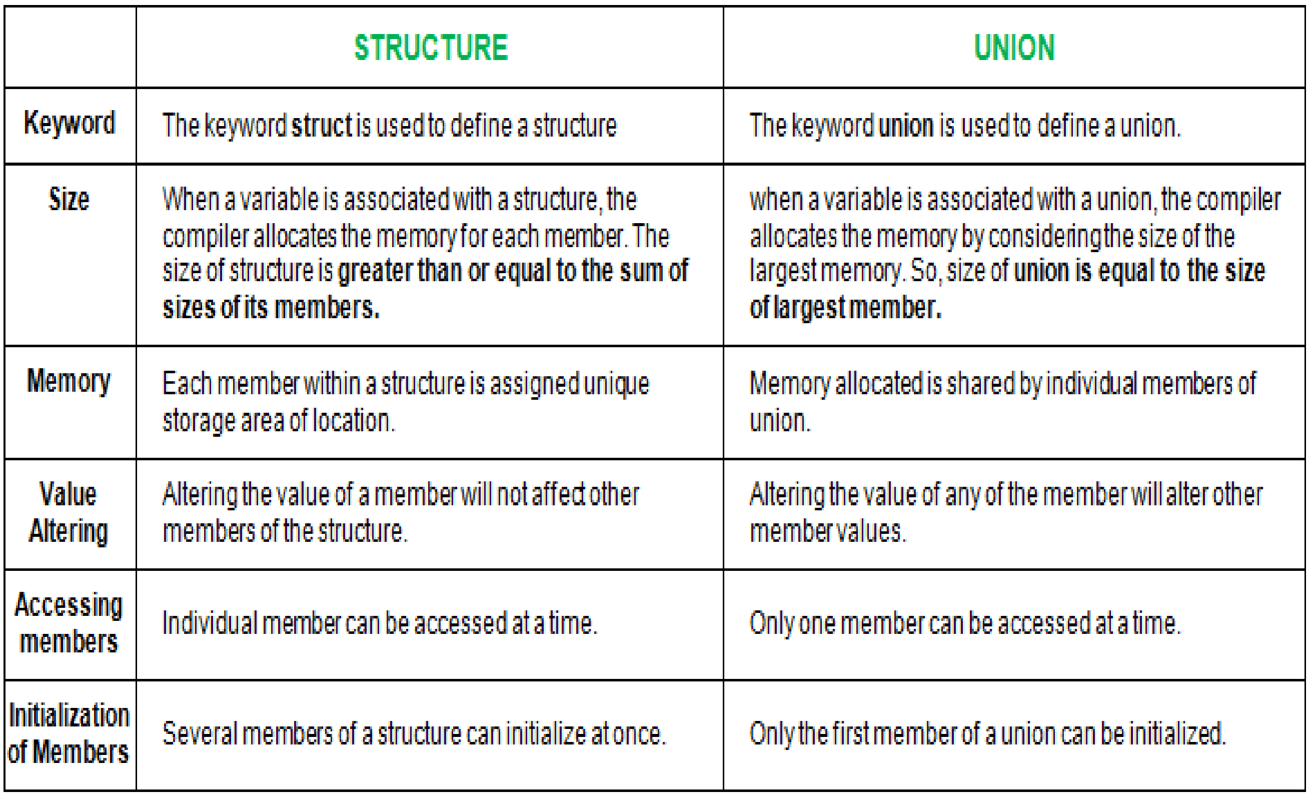
To access price for car1, car1.price is used.

Difference between structure and Unions-

```
#include <stdio.h>
union unionJob
{
   //defining a union
   char name[32];
   float salary;
   int workerNo;
} uJob;

struct structJob
{
   char name[32];
   float salary;

   int workerNo;

} sJob;


int main()

{

   printf("size of union = %d bytes", sizeof(uJob));

   printf("\nsize of structure = %d bytes", sizeof(sJob));

   return 0;

}
```

Output

```
size of union = 32

size of structure = 40
```



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

16.File Handling

Why files are needed?

When a program is terminated, the entire data is lost. Storing in a file will preserve your data even if the program terminates.

If you have to enter a large number of data, it will take a lot of time to enter them all.

However, if you have a file containing all the data, you can easily access the contents of the file using a few commands in C.

You can easily move your data from one computer to another without any changes.

Types of Files

When dealing with files, there are two types of files you should know about:

1) Text files

2) Binary files

1. Text files

Text files are the normal .txt files. You can easily create text files using any simple text editors such as Notepad.

When you open those files, you'll see all the contents within the file as plain text. You can easily edit or delete the contents.

They take minimum effort to maintain, are easily readable, and provide the least security and takes bigger storage space.

2. Binary files

Binary files are mostly the .bin files in your computer.

Instead of storing data in plain text, they store it in the binary form (0's and 1's).

They can hold a higher amount of data, are not readable easily, and provides better security than text files.

File Operations

In C, you can perform four major operations on files, either text or binary:

1)Creating a new file

2)Opening an existing file

3)Closing a file

4) Reading from and writing information to a file

Working with files

When working with files, you need to declare a pointer of type file. This declaration is needed for communication between the file and the program.

FILE *fptr;

Opening a file - for creation and edit

Opening a file is performed using the fopen() function defined in the stdio.h header file.

The syntax for opening a file in standard I/O is:

ptr = fopen("fileopen","mode");

For example,

fopen("E:\\cprogram\\newprogram.txt","w");

fopen("E:\\cprogram\\oldprogram.bin","rb");

Let's suppose the file newprogram.txt doesn't exist in the location E:\cprogram. The first function creates a new file named newprogram.txt and opens it for writing as per the mode 'w'.

The writing mode allows you to create and edit (overwrite) the contents of the file.

Now let's suppose the second binary file oldprogram.bin exists in the location E:\cprogram. The second function opens the existing file for reading in binary mode 'rb'.

The reading mode only allows you to read the file, you cannot write into the file.

Mode	Meaning of Mode	During Inexistence of file
`r`	Open for reading.	If the file does not exist, `fopen()` returns NULL.
`rb`	Open for reading in binary mode.	If the file does not exist, `fopen()` returns NULL.
`w`	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
`wb`	Open for writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
`a`	Open for append. Data is added to the end of the file.	If the file does not exist, it will be created.
`ab`	Open for append in binary mode. Data is added to the end of the file.	If the file does not exist, it will be created.
`r+`	Open for both reading and writing.	If the file does not exist, `fopen()` returns NULL.
`rb+`	Open for both reading and writing in binary mode.	If the file does not exist, `fopen()` returns NULL.
`w+`	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
`wb+`	Open for both reading and writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
`a+`	Open for both reading and appending.	If the file does not exist, it will be created.
`ab+`	Open for both reading and appending in binary mode.	If the file does not exist, it will be created.

Closing a File

The file (both text and binary) should be closed after reading/writing.

Closing a file is performed using the fclose() function.

```
fclose(fptr);
```

Here, fptr is a file pointer associated with the file to be closed.

Example 1: Write to a text file

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int num;
   FILE *fptr;
   // use appropriate location if you are using MacOS or Linux
   fptr = fopen("C:\\program.txt","w");
   if(fptr == NULL)
   {
      printf("Error!");
      exit(1);
   }
   printf("Enter num: ");
   scanf("%d",&num);
   fprintf(fptr,"%d",num);
   fclose(fptr);
   return 0;
}
```

Example 2: Read from a text file

```
#include <stdio.h>

#include <stdlib.h>

int main()

{

   int num;

   FILE *fptr;

   if ((fptr = fopen("C:\\program.txt","r")) == NULL){

       printf("Error! opening file");

       // Program exits if the file pointer returns NULL.

       exit(1);

   }

   fscanf(fptr,"%d", &num);

   printf("Value of n=%d", num);

   fclose(fptr);

   return 0;

}
```

Example 3: Write to a binary file using fwrite()

```
#include <stdio.h>

#include <stdlib.h>

struct threeNum

{

   int n1, n2, n3;

};

int main()

{

   int n;
   struct threeNum num;

   FILE *fptr;


   if ((fptr = fopen("C:\\program.bin","wb")) == NULL){

       printf("Error! opening file");


       // Program exits if the file pointer returns NULL.

       exit(1);

   }


   for(n = 1; n < 5; ++n)

   {

      num.n1 = n;

      num.n2 = 5*n;

      num.n3 = 5*n + 1;

      fwrite(&num, sizeof(struct threeNum), 1, fptr);

   }

   fclose(fptr);

   return 0;

}
```

Example 4: Read from a binary file using fread()

```
#include <stdio.h>

#include <stdlib.h>


struct threeNum

{
   int n1, n2, n3;
};

int main()
{
   int n;
   struct threeNum num;
   FILE *fptr;
   if ((fptr = fopen("C:\\program.bin","rb")) == NULL){
       printf("Error! opening file");
       // Program exits if the file pointer returns NULL.
       exit(1);
   }
   for(n = 1; n < 5; ++n)
   {
      fread(&num, sizeof(struct threeNum), 1, fptr);
      printf("n1: %d\tn2: %d\tn3: %d", num.n1, num.n2, num.n3);
   }
   fclose(fptr);
   return 0;
}
```

17 Command line Argument-

Command line argument is a parameter supplied to the program when it is invoked. Command line argument is an important concept in C programming. It is mostly used when you need to control your program from outside. Command line arguments are passed to the main() method.

Syntax:

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

Here argc counts the number of arguments on the command line and argv[ ] is a pointer array which holds pointers of type char which points to the arguments passed to the program.

Example for Command Line Argument

```
#include <stdio.h>

#include <conio.h>

void main( int argc, char *argv[] )  {

  clrscr();

  if( argc == 2 )

  {

    printf("The argument supplied is %s\n", argv[1]);

  }

  else if( argc > 2 )

  {

    printf("Too many arguments supplied.\n");

  }

  else

  {

    printf("One argument expected.\n");

  }

  getch();
```

}Remember that argv[0] holds the name of the program and argv[1] points to the first command line argument and argv[n] gives the last argument. If no argument is supplied, argc will be 1.

How to Run-

1. Open DOS Shell

2. First use cd.. for coming back to Turbo C++ main directory

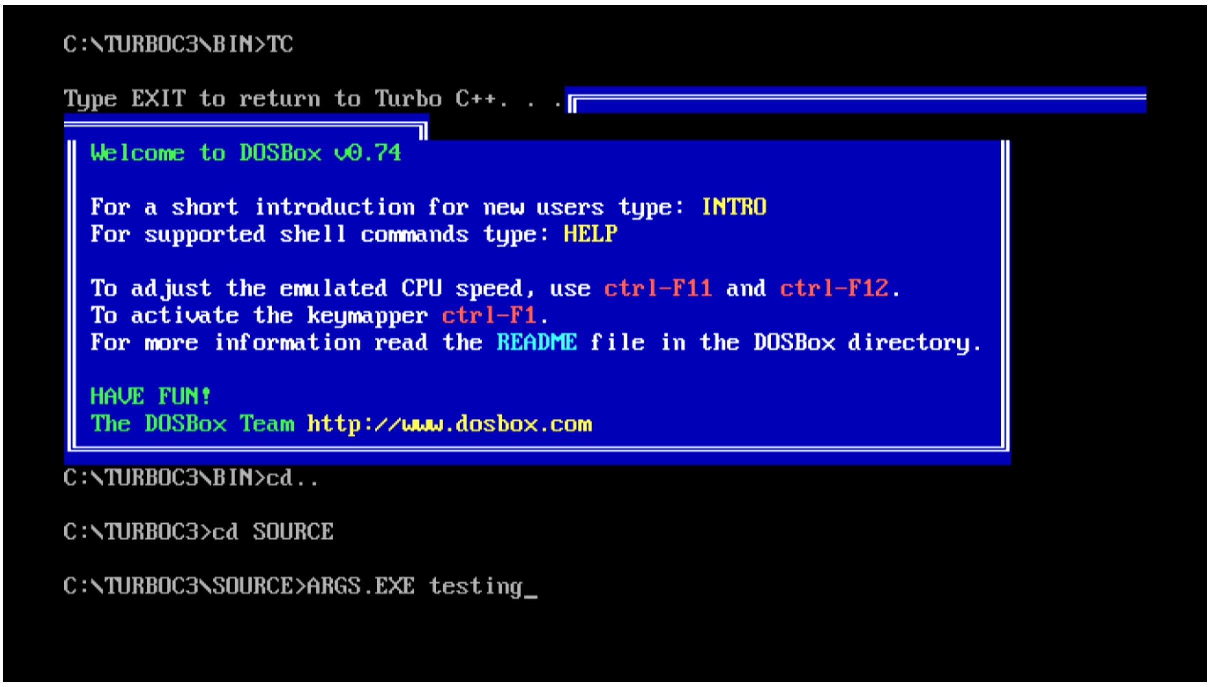
  cd..

3. Now use cd SOURCE to access the SOURCE directory

  cd SOURCE

4.Execute Program with Command Line Arguments

ARGS.EXE testing



```
C:\TURBOC3\BIN>TC

Type EXIT to return to Turbo C++. . .
Welcome to DOSBox v0.74

For a short introduction for new users type: INTRO
For supported shell commands type: HELP

To adjust the emulated CPU speed, use ctrl-F11 and ctrl-F12.
To activate the keymapper ctrl-F1.
For more information read the README file in the DOSBox directory.

HAVE FUN!
The DOSBox Team http://www.dosbox.com
C:\TURBOC3\BIN>cd..

C:\TURBOC3>cd SOURCE

C:\TURBOC3\SOURCE>ARGS.EXE testing_
```

18.Error Handing in C

C language does not provide any direct support for error handling. However a few methods and variables defined in error.h header file can be used to point out error using the return statement in a function. In C language, a function returns -1 or NULL value in case of any error and a global variable errno is set with the error code. So the return value can be used to check error while programming.

What is errno?

Whenever a function call is made in C language, a variable named errno is associated with it. It is a global variable, which can be used to identify which type of error was encountered while function execution, based on its value. Below we have the list of Error numbers and what does they mean.

Errno	Error
1	Operation not permitted
2	No such file or directory
3	No such process
4	Interrupted system call
5	I/O error
6	No such device or address
7	Argument list too long
8	Exec format error
9	Bad file number
10	No child processes
11	Try again
12	Out of memory
13	Permission denied

C language uses the following functions to represent error messages associated with errno:

perror(): returns the string passed to it along with the textual represention of the current errno value.

strerror() is defined in string.h library. This method returns a pointer to the string representation of the current errno value.

Example

```
#include <stdio.h>

#include <errno.h>

#include <string.h>

int main ()

{

   FILE *fp;

   /*

      If a file, which does not exists, is opened,

      we will get an error

   */

   fp = fopen("IWillReturnError.txt", "r");

   printf("Value of errno: %d\n ", errno);

   printf("The error message is : %s\n", strerror(errno));

   perror("Message from perror");

   return 0;

}
```

```
Value of errno: 2

The error message is: No such file or directory

Message from perror: No such file or directory
```

Divide by Zero Errors

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

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

Live Demo

```
#include <stdio.h>

#include <stdlib.h>

main() {

   int dividend = 20;

   int divisor = 0;

   int quotient;

   if( divisor == 0){

      fprintf(stderr, "Division by zero! Exiting...\n");

      exit(-1);

   }

   quotient = dividend / divisor;

   fprintf(stderr, "Value of quotient : %d\n", quotient );


   exit(0);

}
```

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

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

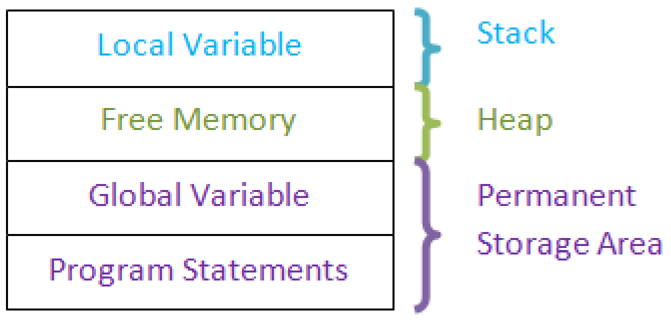
19.Memory Management in C

Every programming language deals with memory in the system. Each and every variable needs a specified amount of memory, the program itself require memory to store its own program, some temporary memory to store intermediate values etc. Hence it is required to manage the memory with utmost care. Memory locations assigned to one program or variable should not be used by another program or variable. Hence C provides 2 methods of allocating memory to the variables and programs. They are static and dynamic memory allocations. In static memory allocation, memory is allocated at the time of compilation and will be same throughout the program. There will not be any changes to the amount of memory nor the location in the memory. But in the case of dynamic memory allocation, memory is allocated at the run time and we can increase/decrease the amount of memory allocated or completely release the memory when not in use. We can reallocate the memory when it is required. Hence dynamic memory allocation gives the flexibility to use the memory efficiently.

Before proceeding to memory allocation, let us understand types of variables, types of memory and methods of allocating memory to the various variables and programs. In a program, we will have different types of variables and memory requirement. The global variables are the ones which will be used throughout the program by different functions and blocks. Hence memory area allocated to them needs to exist throughout the program. Hence they get memory allocated at the internal memories of the system, which are known as permanent storage area. Similarly the program and their statements also need to exist throughout when system is on. Hence they also need to occupy permanent storage area.

Local variables are the one which need to exist in the particular block or function where they are declared. If we store them in permanent storage area, it will be waste of memory as we keep the memory allocate which are not in use. Hence we use stack memory to store the local variables and remove them from the stack as the use of local variable is over.

There is a free memory space between this stack memory and permanent storage area called heap memory. This memory is flexible memory area and keeps changing the size. Hence they are suitable for allocating the memory during the execution of the program. That means dynamic memory allocations use these heap memories.



Static Memory Allocation

Suppose we need to add two integer numbers and display the result. Here we know how many variables and which type of variables are involved in calculations. i.e.; we need two integer variables to store two numbers and one integer variable to store result. Thus we need three integer variables. This implies that at compile time itself we know that there are 3 integer variables. Hence it is easy for the compiler to reserve the memory for these variables. Such reserved variables will have same size and memory address till the end of the program. There will not be any change in size, type and memory location for those variables.

This kind of memory allocation for the variables is known as static memory allocation. Here no need to explicitly allocate memory to the variables. When we declare the variables, memory will be automatically assigned to them. These variables can be local or global variables. But we need to know in advance the size and type of the variable. They need not be simple variables; but they can be array or structure too provided we know their size.

int intX; // needs to be initialized or assigned some value at run time

int intExample = 0; //normal variable

const int intConstant = 10; // constant, read-only variable

Dynamic Memory Allocation

This is in contrast to the static memory allocation. Here program will not know the size and even sometimes type of the variable. It is determined only at the execution time. In such case, we cannot assign any memory at compilation time. It can be assigned only at the run time.

Suppose we need to add any number of numbers that are entered by the user. Here we are not sure about how many numbers are entered by the user. We only know that it he is entering only integers. In this case we cannot pre-assign any memory to the variables. He can enter only 2 numbers or 100s of numbers. If user enters less numbers then the program should be flexible enough to assign to those less number of numbers and as the numbers increase memory allocation space also should increase. But this can be determined only at the run time – depends on the user who enters the value. Thus we need to allocate space at the run time which is done by using dynamic memory allocation methods.

malloc ()

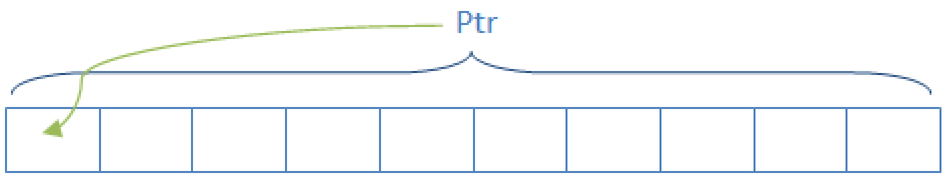
this is the most common method of allocating memory at run time. This function allocates requested amount of memory to the variables at run time and returns the void pointer to the first memory address. That means it allocates the requested amount of memory in bytes and it does not points/ defines datatype for the variable. It considers the variable as void and moves its pointer to the first byte in the allocated memory. In case it cannot allocate memory, then it returns the NULL pointer. When memory is allocated using malloc, variables not initialized t

The general syntax for allocating memory using malloc is:

(cast_type *) malloc (size_in_bytes);

ptr =   malloc(10); // allocates 10 bytes of memory

Here ptr is a pointer variable and it allocates 10 bytes of memory. Here we have not defined datatype of the variable and ptr is a void pointer now. It will now point to the first byte in the allocated memory.



ptr = (int*)malloc(10); //returns integer pointer to ptr pointing to first byte of allocated memory

ptr = (int*)malloc(10* sizeof(int)); //allocates memory sufficient for 10 integer values and returns integer pointer to ptr

arr = (float*)malloc(10 * sizeof(float));

struct student *std = (struct student *)malloc(sizeof(struct student));

calloc ()

This function is similar to malloc. But this function is usually used to allocate memories to arrays and structures. When calloc () is used to allocate memory, it automatically initializes the variable to zero. Suppose we need to allocate memory for 50 students. In malloc we multiply 50 with the size of student structure to get the total memory size. But in calloc, we pass 50 and size of student as two arguments as shown below. Apart from this, it allocates memory in the same way as malloc.

(cast_type *) calloc (blocks , size_of_block);

struct student *std = (struct student *)calloc(sizeof(struct student));// single student

struct student *std = (struct student *)malloc(50, sizeof(struct student));// 50 students

realloc ()

Suppose we need to increase or decrease the memory size of already allocated variable. In such case we can use realloc function to re-define memory size of the variable.

(cast_type *) realloc (blocks, size_of_block);

It will allocate totally new memory location with new block size.

free ()

It is always good practice to release the allocated memory once it is no longer required. This is because whenever the memory is allocated dynamically, they will consume lot of memory space to the variables. It will be available to same or different programs only when it is released. However, all the memories held by the program will be released automatically, once the program is complete.

free (variable_name);

free (std);

20 Programs:-

1) C "Hello, World!" Program
2) C Program to Print an Integer (Entered by the User)
3) C Program to Add Two Integers
4) C Program to Multiply Two Floating-Point Numbers
5) C Program to Find ASCII Value of a Character
6) C Program to Compute Quotient and Remainder
7) C Program to Find the Size of int, float, double and char
8) C Program to Demonstrate the Working of Keyword long
9) C Program to Swap Two Numbers
10) C Program to Check Whether a Number is Even or Odd
11) C Program to Check Whether a Character is a Vowel or Consonant
12) C Program to Find the Largest Number Among Three Numbers
13) C Program to Find the Roots of a Quadratic Equation
14) C Program to Check Leap Year
15) C Program to Check Whether a Number is Positive or Negative
16) C Program to Check Whether a Character is an Alphabet or not
17) C Program to Calculate the Sum of Natural Numbers
18) C Program to Find Factorial of a Number
19) C Program to Generate Multiplication Table
20) C Program to Display Fibonacci Sequence
21) C Program to Find GCD of two Numbers
22) C Program to Find LCM of two Numbers
23) C Program to Display Characters from A to Z Using Loop
24) C Program to Count Number of Digits in an Integer
25) C Program to Reverse a Number
26) C Program to Calculate the Power of a Number
27) C Program to Check Whether a Number is Palindrome or Not
28) C Program to Check Whether a Number is Prime or Not
29) C Program to Display Prime Numbers Between Two Intervals
30) C Program to Check Armstrong Number
31) C Program to Display Armstrong Number Between Two Intervals
32) C Program to Display Factors of a Number
33) C Programming Code To Create Pyramid and Structure
34) C Program to Make a Simple Calculator Using switch...case
35) C Program to Display Prime Numbers Between Intervals Using Function
36) C Program to Check Prime or Armstrong Number Using User-defined Function
37) C Program to Check Whether a Number can be Expressed as Sum of Two Prime Numbers
38) C Program to Find the Sum of Natural Numbers using Recursion
39) C Program to Find Factorial of a Number Using Recursion
40) C Program to Find G.C.D Using Recursion

41) C Program to Convert Binary Number to Decimal and vice-versa
42) C Program to Convert Octal Number to Decimal and vice-versa
43) C Program to Convert Binary Number to Octal and vice-versa
44) C program to Reverse a Sentence Using Recursion
45) C program to calculate the power using recursion
46) C Program to Calculate Average Using Arrays
47) C Program to Find Largest Element in an Array
48) C Program to Calculate Standard Deviation
49) C Program to Add Two Matrices Using Multi-dimensional Arrays
50) C Program to Multiply Two Matrices Using Multi-dimensional Arrays
51) C Program to Find Transpose of a Matrix
52) C Program to Multiply two Matrices by Passing Matrix to a Function
53) C Program to Access Array Elements Using Pointer
54) C Program Swap Numbers in Cyclic Order Using Call by Reference
55) C Program to Find Largest Number Using Dynamic Memory Allocation
56) C Program to Find the Frequency of Characters in a String
57) C Program to Count the Number of Vowels, Consonants and so on
58) C Program to Remove all Characters in a String Except Alphabets
59) C Program to Find the Length of a String
60) C Program to Concatenate Two Strings
61) C Program to Copy String Without Using strcpy()
62) C Program to Sort Elements in Lexicographical Order (Dictionary Order)
63) C Program to Store Information of a Student Using Structure
64) C Program to Add Two Distances (in inch-feet system) using Structures
65) C Program to Add Two Complex Numbers by Passing Structure to a Function
66) C Program to Calculate Difference Between Two Time Periods
67) C Program to Store Information of Students Using Structure
68) C Program to Store Data in Structures Dynamically
69) C Program to Write a Sentence to a File
70) C Program to Read a Line From a File and Display it
71) C Program to Display its own Source Code as Output
72) C Program to Print Pyramids and Patterns

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

1
1 2
1 2 3
```

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

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

* * * * *
* * * *
* * *
* *
*
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1

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

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

C Question and Answer-

Q1. What are the basic Datatypes supported in C Programming Language?

Ans: Data type chapter

Q2. What do you mean by Dangling Pointer Variable in C Programming?

Ans: A Pointer in C Programming is used to point the memory location of an existing variable. In case if that particular variable is deleted and the Pointer is still pointing to the same memory location, then that particular pointer variable is called as a Dangling Pointer Variable.

Q3. What do you mean by the Scope of the variable? What is the scope of the variables in C?

Ans: Scope of the variable can be defined as the part of the code area where the variables declared in the program can be accessed directly. In C, all identifiers are lexically (or statically) scoped.

Q4. What are static variables and functions?

Ans: The variables and functions that are declared using the keyword Static are considered as Static Variable and Static Functions. The variables declared using Static keyword will have their scope restricted to the function in which they are declared.

Q5. Differentiate between calloc() and malloc()

Ans: calloc() and malloc() are memory dynamic memory allocating functions. The only difference between them is that calloc() will load all the assigned memory locations with value 0 but malloc() will not.
Q6. What are the valid places where the programmer can apply Break Control Statement?

Ans: Break Control statement is valid to be used inside a loop and Switch control statements.

Q7. How can we store a negative integer?

Ans: To store a negative integer, we need to follow the following steps. Calculate the two’s complement of the same positive integer.

Eg: 1011 (-5)

Step-1 − One’s complement of 5: 1010

Step-2 − Add 1 to above, giving 1011, which is -5

Q8. Differentiate between Actual Parameters and Formal Parameters.

Ans: The Parameters which are sent from main function to the subdivided function are called as Actual Parameters and the parameters which are declared a the Subdivided function end are called as Formal Parameters.

Q9. Can a C program be compiled or executed in the absence of a main()?

Ans: The program will be compiled but will not be executed. To execute any C program, main() is required.

Q10. What do you mean by a Nested Structure?

Ans: When a data member of one structure is referred by the data member of another function, then the structure is called a Nested Structure.

Q11. What is a C Token?

Ans: Keywords, Constants, Special Symbols, Strings, Operators, Identifiers used in C program are referred to as C Tokens.

Q12. What is Preprocessor?

Ans: A Preprocessor Directive is considered as a built-in predefined function or macro that acts as a directive to the compiler and it gets executed before the actual C Program is executed.

Q13. Why is C called the Mother of all Languages?

Ans: C introduced many core concepts and data structures like arrays, lists, functions, strings, etc. Many languages designed after C are designed on the basis of C Language. Hence, it is considered as the mother of all languages.

Q14. Mention the features of C Programming Language.

Ans: Answer youself

Q15. What is the purpose of printf() and scanf() in C Program?

Ans: printf() is used to print the values on the screen. To print certain values, and on the other hand, scanf() is used to scan the values. We need an appropriate datatype format specifier for both printing and scanning purposes. For example,

%d: It is a datatype format specifier used to print and scan an integer value.
%s: It is a datatype format specifier used to print and scan a string.
%c: It is a datatype format specifier used to display and scan a character value.
%f: It is a datatype format specifier used to display and scan a float value.

Q16. What is an array?

Ans. The array is a simple data structure that stores multiple elements of the same datatype in a reserved and sequential manner. There are three types of arrays, namely,

One Dimensional Array
Two Dimensional Array
Multi-Dimensional Array

Q17. What is /0 character?

Ans: The Symbol mentioned is called a Null Character. It is considered as the terminating character used in strings to notify the end of the string to the compiler.

Q18. What is the main difference between the Compiler and the Interpreter?

Ans: Compiler is used in C Language and it translates the complete code into the Machine Code in one shot. On the other hand, Interpreter is used in Java Programming Langauge and other high-end programming languages. It is designed to compile code in line by line fashion.

Q19. Can I use int datatype to store 32768 value?

Ans: No, Integer datatype will support the range between -32768 and 32767. Any value exceeding that will not be stored. We can either use float or long int.

Intermediate C Programming Interview Questions
C-Programming-Interview-Questions-Edureka-Intermediate-Interview-Questions

Q20. How is a Function declared in C Language?

Ans: A function in C language is declared as follows,

```
return_type function_name(formal parameter list)
{
      Function_Body;
}
```

Q21. What is Dynamic Memory allocation? Mention the syntax.

Ans: Dynamic Memory Allocation is the process of allocating memory to the program and its variables in runtime. Dynamic Memory Allocation process involves three functions for allocating memory and one function to free the used memory.

malloc() – Allocates memory

Syntax:
ptr = (cast-type*) malloc(byte-size);
calloc() – Allocates memory

Syntax:
ptr = (cast-type*)calloc(n, element-size);
realloc() – Allocates memory

Syntax:
ptr = realloc(ptr, newsize);
free() – Deallocates the used memory

Syntax:

```
free(ptr);
```

Q22. What do you mean by Dangling Pointer Variable in C Programming?

Ans: A Pointer in C Programming is used to point the memory location of an existing variable. In case if that particular variable is deleted and the Pointer is still pointing to the same memory location, then that particular pointer variable is called as a Dangling Pointer Variable.

Q23. Where can we not use &(address operator in C)?

Ans: We cannot use & on constants and on a variable which is declared using the register storage class.

Q24. Write a simple example of a structure in C Language

Ans: Structure is defined as a user-defined data type that is designed to store multiple data members of the different data types as a single unit. A structure will consume the memory equal to the summation of all the data members.

```
struct employee
{
   char name[10];
   int age;
}e1;
int main()
{
   printf("Enter the name");
   scanf("%s",e1.name);
   printf("n");
   printf("Enter the age");
   scanf("%d",&e1.age);
   printf("n");
   printf("Name and age of the employee: %s,%d",e1.name,e1.age);
   return 0;
}
```

Q25. Differentiate between call by value and call by reference

Ans:

Pointer Chapter

```
#include<stdio.h>
void change(int,int);
int main()
{
    int a=25,b=50;
    change(a,b);
    printf("The value assigned to a is: %d",a);
    printf("n");
    printf("The value assigned to of b is: %d",b);
    return 0;
}
void change(int x,int y)
{
    x=100;
    y=200;
}
//Output
```

```
The value assigned to of a is: 25
The value assigned to of b is: 50
```

```
//Example of Call by Reference method

#include<stdio.h>
void change(int*,int*);
int main()
{
    int a=25,b=50;
    change(&a,&b);
    printf("The value assigned to a is: %d",a);
    printf("n");
    printf("The value assigned to b is: %d",b);
    return 0;
}
void change(int *x,int *y)
{
    *x=100;
    *y=200;
}
```

```
//Output

The value assigned to a is: 100
The value assigned to b is: 200
```

Q26. Differentiate between getch() and getche()

Ans: Both the functions are designed to read characters from the keyboard and the only difference is that

getch(): reads characters from the keyboard but it does not use any buffers. Hence, data is not displayed on the screen.

getche(): reads characters from the keyboard and it uses a buffer. Hence, data is displayed on the screen.

```
//Example

#include<stdio.h>
#include<conio.h>
int main()
{
    char ch;
    printf("Please enter a character ");
    ch=getch();
    printf("nYour entered character is %c",ch);
    printf("nPlease enter another character ");
    ch=getche();
    printf("nYour new character is %c",ch);
    return 0;
}
//Output

Please enter a character
Your entered character is x
Please enter another character z
Your new character is z
```

Q27. Explain toupper() with an example.

Ans. toupper() is a function designed to convert lowercase words/characters into upper case.

```
//Example

#include<stdio.h>
#include<ctype.h>
int main()
{
   char c;
   c=a;
   printf("%c after conversions  %c", c, toupper(c));
   c=B;
   printf("%c after conversions  %c", c, toupper(c));
//Output:

a after conversions A
B after conversions B
```

Q28. Write a code to generate random numbers in C Language

Ans: Random numbers in C Language can be generated as follows:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int a,b;
    for(a=1;a<=10;a++)
    {
         b=rand();
         printf("%dn",b);
    }
    return 0;
}
//Output

1987384758
2057844389
3475398489
2247357398
1435983905
```

Q29. Can I create a customized Head File in C language?

Ans: It is possible to create a new header file. Create a file with function prototypes that need to be used in the program. Include the file in the '#include' section in its name.

Q30. What do you mean by Memory Leak?

Ans: Memory Leak can be defined as a situation where programmer allocates dynamic memory to the program but fails to free or delete the used memory after the completion of the code. This is harmful if daemons and servers are included in the program.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int* ptr;
    int n, i, sum = 0;
    n = 5;
    printf("Enter the number of elements: %dn", n);
    ptr = (int*)malloc(n * sizeof(int));
    if (ptr == NULL)
    {
          printf("Memory not allocated.n");
          exit(0);
    }
    else
    {
          printf("Memory successfully allocated using malloc.n");
          for (i = 0; i<= n; ++i)
          {
               ptr[i] = i + 1;
           }
           printf("The elements of the array are: ");
           for (i = 0; i<=n; ++i)
          {
               printf("%d, ", ptr[i]);
          }
    }
    return 0;
}
//Output

Enter the number of elements: 5
Memory successfully allocated using malloc.
The elements of the array are: 1, 2, 3, 4, 5,
```

Q31. Explain Local Static Variables and what is their use?

Ans: A local static variable is a variable whose life doesn’t end with a function call where it is declared. It extends for the lifetime of the complete program. All calls to the function share the same copy of local static variables.

```
#include<stdio.h>
void fun()
{
   static int x;
   printf("%d ", x);
   x = x + 1;
}
int main()
{
   fun();
   fun();
   return 0;
}
//Output

0 1
```

Q32. What is the difference between declaring a header file with < > and ” “?

Ans: If the Header File is declared using < > then the compiler searches for the header file within the Built-in Path. If the Header File is declared using ” ” then the compiler will search for the Header File in the current working directory and if not found then it searches for the file in other locations.

Q33. When should we use the register storage specifier?

Ans: We use Register Storage Specifier if a certain variable is used very frequently. This helps the compiler to locate the variable as the variable will be declared in one of the CPU registers.

Q34. Which statement is efficient and why? x=x+1; or x++;

Ans: x++; is the most efficient statement as it just a single instruction to the compiler while the other is not.

Q35. Can I declare the same variable name to the variables which have different scopes?

Ans: Yes, Same variable name can be declared to the variables with different variable scopes as the following example.

```
int var;
void function()
{
   int variable;
}
int main()
{
   int variable;
}
```

Q36. Which variable can be used to access Union data members if the Union variable is declared as a pointer variable?

Ans: Arrow Operator(->) can be used to access the data members of a Union if the Union Variable is declared as a pointer variable.

Q37. Mention File operations in C Language.

Ans: Basic File Handling Techniques in C, provide the basic functionalities that user can perform against files in the system.

Function	Operation
fopen()	To Open a File
fclose()	To Close a File
fgets()	To Read a File
fprint()	To Write into a File

Q38. What are the different storage class specifiers in C?
Ans: The different storage specifiers available in C Language are as follows:

auto
register
static
extern

Q39. What is typecasting?

Ans: Typecasting is a process of converting one data type into another is known as typecasting. If we want to store the floating type value to an int type, then we will convert the data type into another data type explicitly.

Syntax:

```
(type_name) expression;
```

Q40. Write a C program to print hello world without using a semicolon ;

Ans:

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

```
{
     if(printf("hello world")){}
}
//Output:

hello world
```

Q41. Write a program to swap two numbers without using the third variable.

Ans:

```
#include<stdio.h>
#include<conio.h>
main()
{
    int a=10, b=20;
    clrscr();
    printf("Before swapping a=%d b=%d",a,b);
    a=a+b;
    b=a-b;
    a=a-b;
    printf("nAfter swapping a=%d b=%d",a,b);
    getch();
}
//Output

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

Write output of following program-

http://www.indiabix.com/c-programming/questions-and-answers/

Tips and Tricks-

1) There is no better way to do well in Coding interviews than practicing as many coding problems as possible. This will not only train your mind to recognize algorithmic patterns in problems but also give you the much-needed confidence to solve the problem you have never seen before.
2) My second tips are to learn about as many data structure and algorithms as possible. This is an extension of the previous tip but it also involves reading and not just practicing. For example, If you know about the hash table you can also many array and counter-based problems easily. Same is true for tree and graph.
3) Choosing the right data structure is a very important part of software development and coding interview and unless and until you know them, you won't be able to choose.
4) Time yourself — candidates who solve interview problems within the time limit and quickly are more likely to do well in the interview so you should also time yourself.
5) Think of edge cases and run your code through them. Some good edge cases might be the empty input, some weird input or some really large input to test the boundary conditions and limits.
6) After solving the problem, try explaining it to a friend or colleagues how is also interested in coding problems. This will tell you whether you have really understood the problem or not. If you can explain easily means you understood. Also, the discussion makes your mind work and you could come up with an alternative solution and able to find some flaws in your existing algorithms.

Another useful tip to excel Coding interviews is to appear in the coding interview and lots of them. You will find yourself getting better after every interview and this also helps you to get multiple offers which further allows you to better negotiate and get those extra 30K to 50K which you generally leave on a table if you just have one offer in hand.