C Programming

1. C Introduction

Introduction

- The C programming language is a Procedure oriented programming language, developed at Bell Laboratories in 1972 by Dennis Ritchie

- C programming language features were derived from an earlier language called “B” (Basic Combined Programming Language – BCPL)
- C language was invented for implementing UNIX operating system
- In 1978, Dennis Ritchie and Brian Kernighan published the first edition “The C Programming Language”
- In 1983, the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive definition of C.

- Computer Application and telecommunications applications.

All programming languages can be divided into two categories

1. Problem oriented language or High level language: It will give better programming efficiency, i.e. faster program development.
   Ex: Basic, Pascal, C++, Java

2. Machine oriented language or Low level language: It will give better machine efficiency, i.e. faster program execution
   Ex: Assembly language and Machine language

But C stands between two categories so it is called middle level language.

Application of C Programming are listed below –

1. C language is used for creating **computer applications**
2. Used in writing **Embedded softwares**
3. C is used for many different types of software, but it is particularly popular for system software, such as operating systems, device drivers,
4. C is widely used because it runs very fast.
5. **UNIX kernel** is completely developed in C Language.

C Basic Program:

```
#include <stdio.h>
int main()
{
/* Our first simple C basic program */
printf(“Hello World! “);
return 0;
}
```

A C program basically consists of the following parts −

S.no	Command	Explanation
1	#include <stdio.h>	This is a preprocessor command that includes standard input output header file(stdio.h) from the C library before compiling a C program
2	int main()	This is the main function from where execution of any C program begins.
3	{	This indicates the beginning of the main function.
4	/*_some_comments_*/	Whatever is given inside the command “/* */” in any C program, won’t be considered for compilation and execution.
5	printf(“Hello_World! “);	printf command prints the output onto the screen.
6	getch();	This command waits for any character input from keyboard.
7	return 0;	This command terminates C program (main function) and returns 0.
8	}	This indicates the end of the main function.

2. C Input and Output

Printf and Scanf

The printf() and scanf() functions are used for input and output in C language. Both functions are inbuilt library functions, defined in stdio.h (header file).

printf() function

The **printf() function** is used for output. It prints the given statement to the console.

- printf() function is used to print the "character, string, float, integer, octal and hexadecimal values" onto the output screen.
- We use printf() function with %d format specifier to display the value of an integer variable.
- To generate a newline, we use "\n" in C printf() statement.

The syntax of printf() function is given below:

1. printf("format string",argument_list);

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

scanf() function

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

- scanf() function is used to read character, string, numeric data from keyboard

1. scanf("format string",argument_list);

scanf() and printf() functions

```
#include<stdio.h>
int main()
{
 int i;
 printf("Enter a value\n");
 scanf("%d",&i);
 printf( "You entered: %d",i);
 return 0;
}
```

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

The **printf("You entered: %d ",i)** statement prints the number on the console.

getchar() & putchar() functions

The getchar() function reads a character from the terminal and returns it as an integer. This function reads only single character at a time. You can use this method in the loop in case you want to read more than one characters.

The putchar() function prints the character passed to it on the screen and returns the same character. This function puts only single character at a time. In case you want to display more than one characters, use putchar() method in the loop.

```
#include <stdio.h>

void main( )
{
 int c;
 printf("Enter a character");
 c=getchar();
 putchar(c);

}
```

gets() & puts() functions

The gets() function reads a line from **stdin** into the buffer pointed to by **s** until either a terminating newline or EOF (end of file).

The puts() function writes the string **s** and a trailing newline to stdout.

```
#include<stdio.h>

void main()
{
 char str[100];
 printf("Enter a string");
 gets( str );
 puts( str );

}
```

Difference between scanf() and gets()

The main difference between these two functions is that scanf() stops reading characters when it encounters a space, but gets() reads space as character too.
If you enter name as **C Programming** using scanf() it will only read and store **C** and will leave the part after space. But gets() function will read it complete.

fflush(stdin) in c

the use of fflush(stdin) is to clear the input buffer bcoz when ever we take an input the space is allocated to it but after its scope gets over, the input is removed but the space allocated is as it is. it can be used only by using headerfile

```
#include <stdio.h>
int main()
{
    int i;
    char c;
    printf("Enter the value:");
    scanf("%d",&i);
    fflush (stdin);
    printf("Enter Charecter:");
    scanf("%c",&c);
    printf("Num=%d\n",i);
    printf("Char=%c",c);
    return 0;
}
```

```
#include <stdio.h>
int main()
{
    int i;
    char c[12];
    float j;
    printf("Enter the roll_no:");
    scanf("%d",&i);
    fflush (stdin);
    printf("Enter Name:");
    gets(c);
    printf("Enter Marks:");
    scanf("%f",&j);
    printf("Roll_no=%d\n",i);
    printf("Name=");
    puts(c);
    printf("Marks=%.2f",j);
    return 0;
}
```

Assignment

Write a function that prompts for a name (up to 20 Characters and address (up to 30 characters) and accepts them one at a time. Finally the name and address are displayed in the following way.

Hello,

Your name is:

(name)

Your address is:

(address)

3. C Data Types

Data types in C

- Data types are the keywords, which are used for assigning a type to a variable.
- A data type specifies the type of data that a variable can store such as integer, floating, character etc.
- Data types will allocate some byte of memory for their variables.

1. **Fundamental Data Types**
- Integer types
- Floating Type
- Character types
2. **Derived Data Types**
- Arrays
- Pointers
- Structures
- Enumeration

Syntax for declaration of a variable:
data_type variable_name;

1. Basic data types in C:

Integer data type:

Integer data type allows a variable to store numeric values.
Syntax:int var1;

- “int” keyword is used to refer integer data type.
- The storage size of int data type is 2 or 4 or 8 byte.
- It varies depend upon the processor in the CPU that we use. If we are using 16 bit processor, 2 byte (16 bit) of memory will be allocated for int data type.
- Like wise, 4 byte (32 bit) of memory for 32 bit processor and 8 byte (64 bit) of memory for 64 bit processor is allocated for int datatype.
- If you want to use the integer value that crosses the above limit, you can go for “long int” and “long long int” for which the limits are very high.
- The following table provides the details of standard integer types with their storage sizes and value ranges −

Type	**Storage size**	**Value range**
Char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
Int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
Short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
Long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

Character data type:

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

Floating point data type:

Floating point data type consists of 2 types. They are,

1. float
2. double

1. float:

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

2. double:

- Double data type is also same as float data type which allows up-to 10 digits after decimal.
- The range for double datatype is from 1E–37 to 1E+37.
- The following table provide the details of standard floating-point types with storage sizes and value ranges and their precision −

Type	**Storage size**	**Value range**	**Precision**
Float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
Double	8 byte	2.3E-308 to 1.7E+308	10 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

1.3.1. sizeof() function in C:

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

Example:

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

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

3. ***Derived data type in C:***

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

We will see this in next topics: "C – Array" , "C – Pointer" , "C – Structure" and "C – Union"

Enumeration data type in C:

- Enumeration data type consists of named integer constants as a list.
- It start with 0 (zero) by default and value is incremented by 1 for the sequential identifiers in the list.
- Enum syntax in C:**enum identifier [optional{ enumerator-list }];**
- Enum example in C:

```
enum month { Jan, Feb, Mar }; or
/* Jan, Feb and Mar variables will be assigned to 0, 1 and 2 respectively by default */
enum month { Jan = 1, Feb, Mar };
/* Feb and Mar variables will be assigned to 2 and 3 respectively by default */
enum month { Jan = 20, Feb, Mar };
/* Jan is assigned to 20. Feb and Mar variables will be assigned to 21 and 22 respectively by default */
```

C – enum example program:

```
#include <stdio.h>
int main()
{
    enum month{jan,feb,mar};

    printf("Month=%d\n",jan);

    printf("Month=%d\n",feb);

    printf("Month=%d",mar);
    return 0;
}
```

Qualifiers

Qualifiers alter the meaning of base data types to yield a new data type.

Size qualifiers:

Size qualifiers alter the size of basic data type. The keywords long and short are two size qualifiers. For example:

```
long int i;
```

The size of int is either 2 bytes or 4 bytes but, when long keyword is used, that variable will be either 4 bytes of 8 bytes. If the larger size of variable is not needed then, short keyword can be used in similar manner as long keyword.

Sign qualifiers:

Whether a variable can hold only positive value or both values is specified by sign qualifiers. Keywords signed and unsigned are used for sign qualifiers.

```
unsigned int a;
// unsigned variable can hold zero and positive values only
```

%u Unsigned Integer

It is not necessary to define variable using keyword signed because, a variable is signed by default. Sign qualifiers can be applied to only int and char data types. For a int variable of size 4 bytes it can hold data from -2^{31} to $2^{31}-1$ but, if that variable is defined unsigned, it can hold data from 0 to $2^{32}-1$.

Constant qualifiers

Constant qualifiers can be declared with keyword const. An object declared by const cannot be modified.

```
const int p=20;
```

The value of *p* cannot be changed in the program.

The % Format Specifiers

The **%** specifiers that you can use in ANSI C are:

Usual variable type Display

%c char single character

%d (%i) int signed integer

%e (%E) float or double exponential format

%f float or double signed decimal

%g (%G) float or double use %f or %e as required

%o int unsigned octal value

%p pointer address stored in pointer

%s array of char sequence of characters

%u int unsigned decimal

%x (%X) int unsigned hex value

4. Keyword and Identifiers

C tokens:

- C tokens are the basic buildings blocks in C language which are constructed together to write a C program.
- Each and every smallest individual unit in a C program is known as C tokens.
- C tokens are of six types. They are,

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

C tokens example program:

```
int main()
{
int x, y, total;
x = 10, y = 20;
total = x + y;
Printf (“Total = %d \n”, total);
}
                .
```

Keywords in C

- A keyword is a **reserved word** used in programming. Each keyword has fixed meaning and that cannot be changed by user.
- Each keyword is meant to perform a specific function in a C program.
- You cannot use it as a variable name, constant name etc. There are only 32 reserved words (keywords) in C language.
- For example: int money; //Here, int is a keyword

As, C programming is case sensitive, all keywords must be written in lowercase. Here is the list of all keywords predefined by ANSI C.

Keywords in C Language

Auto	Double	Int	struct
Break	Else	Long	switch
Case	Enum	register	typedef
Char	extern	return	union
continue	For	signed	void
Do	If	static	while
Default	Goto	sizeof	volatile
Const	Float	short	unsigned

Identifiers

Identifier is a user defined name.

In C programming, identifiers are names given to C entities, such as variables, functions, structures etc. Identifiers are created to give unique name to C entities to identify it during the execution of program. For example:

```
int money;

int mango_tree;
```

Here, *money* is a identifier which denotes a variable of type integer. Similarly, *mango_tree* is another identifier, which denotes another variable of type integer.

Rules for writing identifier

1. An identifier can be composed of letters (both uppercase and lowercase letters), digits and underscore '_' only.
2. The first letter of identifier should be either a letter or an underscore. But, it is discouraged to start an identifier name with an underscore though it is legal. It is because, identifier that starts with underscore can conflict with system names. In such cases, compiler will complain about it. Some system names that start with underscore are *_fileno*, *_iob*, *_wfopen* etc.
3. There is no rule for the length of an identifier. However, the first 31 characters of an identifier are discriminated by the compiler. So, the first 31 letters of two identifiers in a program should be different.

5. Constants and Variables

Variables

Variables are memory location in computer's memory to store data. To indicate the memory location, each variable should be given a unique name called identifier. Variable names are just the symbolic representation of a memory location. Examples of variable name: *sum*, *car_no*, *count* etc.

- The value of the C variable may get change in the program.
- C variable might be belonging to any of the data type like int, float, char etc.

Example:

```
int num;
```

Here, *num* is a variable of integer type.

Rules for naming C variable:

1. Variable name must begin with letter or underscore.
2. Variables are case sensitive
3. They can be constructed with digits, letters.
4. No special symbols are allowed other than underscore.
5. sum, height, _value are some examples for variable name

Declaring & initializing C variable:

- Variables should be declared in the C program before to use.
- Memory space is not allocated for a variable while declaration. It happens only on variable definition.
- Variable initialization means assigning a value to the variable.

S.No	Type	Syntax	Example
1	Variable declaration	data_type variable_name;	int x, y, z; char flat, ch;
2	Variable initialization	data_type variable_name = value;	int x = 50, y = 30; char flag = ‘x’, ch=’l’;

There are two types of variables in C program they are,

1. Local variable
2. Global variable

1. Local variable in C:

- Local variables are declared inside a function, and can be used only inside that function.
- The scope of local variables will be within the function only.
- A local variable is a variable which is either a variable declared within the function or is an argument passed to a function.

```
#include<stdio.h>
int main()
{
int m = 22, n = 44; // m, n are local variables of main function
printf("\nvalues : m = %d and n = %d", m, n);
 }
```

2. ***Global variable in C:***

- Global variables are declared outside any function, and they can be accessed (used) on any function in the program.
- The scope of global variables will be throughout the program. These variables can be accessed from anywhere in the program.
- A global variable (DEF) is a variable which is accessible in multiple scopes.

```
#include<stdio.h>
int m = 22, n = 44;
int main()
{
printf("All variables are accessed from main function");
printf("\nvalues: m=%d:n=%d", m,n);
return 0;
}
```

Example 1:

```
#include <stdio.h>
int main()
{
   int m=10;
    printf("m=%d\n",m);
    {
       int n=20;
       printf("n=%d\n",n);
    }
    printf("m=%d,n=%d",m,n);

    return 0;
}
```

Example 2:

```
#include <stdio.h>
int m=30, n=40;
int main()
{
   int m=10;
    printf("m=%d\n",m);
```

```
     {
        int n=20;
        printf("n=%d\n",n);
     }
     printf("m=%d,n=%d",m, n);
return 0;
}
```

```
#include <stdio.h>
int a=20;
int main()
{
   int a;
   printf("%d\n", a);
   return 0;
}
```

Difference between variable declaration & definition in C:

S.no	Variable declaration	Variable definition
1	Declaration tells the compiler about data type and size of the variable.	Definition allocates memory for the variable.
2	Variable can be declared many times in a program.	It can happen only one time for a variable in a program.

Constants

- Constants are the fixed values or terms that can't be changed during the execution of a program.
- These fixed values are also called **literals**.
- Syntax: const data_type variable_name; (or) const data_type *variable_name;
- For example: 1, 2.5, "Programming is easy." etc. In C, constants can be classified as:

1. Integer constants

Integer constants are the numeric constants (constant associated with number) without any fractional part or exponential part. There are three types of integer constants in C language: decimal constant (base 10), octal constant (base 8) and hexadecimal constant (base 16).

Decimal digits: 0 1 2 3 4 5 6 7 8 9

Octal digits: 0 1 2 3 4 5 6 7

Hexadecimal digits: 0 1 2 3 4 5 6 7 8 9 A B C D E F.

For example:

Decimal constants: 0, -9, 22 etc

Octal constants: 021, 077, 033 etc

Hexadecimal constants: 0x7f, 0x2a, 0x521 etc

Notes:

1. You can use small caps *a*, *b*, *c*, *d*, *e*, *f* instead of uppercase letters while writing a hexadecimal constant.
2. Every octal constant starts with 0 and hexadecimal constant starts with 0x in C programming.

2. Floating-point constants

Floating point constants are the numeric constants that has either fractional form or exponent form. For example:

-2.0

0.0000234

-0.22E-5

Note: Here, E-5 represents 10^{-5}. Thus, -0.22E-5 = -0.0000022.

```
int main()
{
    float f;
    f=-0.22E-5;
    printf("F=%f",f);
   return 0;
}
```

3. Character constants

Character constants are the constant which use single quotation around characters. For example: 'a', 'l', 'm', 'F' etc.

Escape Sequences

Escape Sequences	
Escape Sequences	Character
\b	Backspace
\n	Newline
\r	Return
\t	Horizontal tab
\\	Backslash
\'	Single quotation mark
\"	Double quotation mark
\?	Question mark

4. String constants

String constants are the constants which are enclosed in a pair of double-quote marks. For example:

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

5. Enumeration constants

Keyword enum is used to declare enumeration types. For example:

```
enum color {yellow, green, black, white};
```

Here, the variable name is color and yellow, green, black and white are the enumeration constants having value 0, 1, 2 and 3 respectively by default.

1. Example program using const keyword in C:

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

```
{
const int height = 100;             /*int constant*/
const float number = 3.14;   /*Real constant*/
const char letter = 'A';           /*char constant*/
const char letter_sequence[10] = "ABC";           /*string constant*/
const char backslash_char = '\?';                         /*special char cnst*/
printf("value of height :%d \n", height );
printf("value of number : %f \n", number );
printf("value of letter : %c \n", letter );
printf("value of letter_sequence : %s \n", letter_sequence);
printf("value of backslash_char : %c \n", backslash_char);
}
```

Program to Print ASCII Value

```
#include <stdio.h>
int main()
{
   char c;
   printf("Enter a character: ");
   scanf("%c", &c);  // Reads character input from the user

   printf("ASCII value of %c = %d", c, c);  // %d displays the integer value of a character
                                                           // %c displays the actual character
   return 0;
}
```

6. Operators

C Operators

- An operator is symbol that is used to perform operations.
- Operators are the symbol which operates on value or a variable. For example: + is an operator to perform addition.

Types of C operators:

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

1. Arithmetic Operators in C:

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

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

Example of working of arithmetic operators

```
#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;
}
```

2. Assignment Operators

The most common assignment operator is =. This operator assigns the value in right side to the left side. For example:

```
var=5  //5 is assigned to var

a=c;   //value of c is assigned to a

5=c;   // Error! 5 is a constant.
```

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

Example program for C assignment operators:

- In this program, values from 0 – 9 are summed up and total “45″ is displayed as output.
- Assignment operators such as “=” and “+=” are used in this program to assign the values and to sum up the values.

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

3. Relational Operator

Relational operator's checks relationship between two operands. If the relation is true, it returns value 1 and if the relation is false, it returns value 0. For example:

```
a>b
```

Here, > is a relational operator. If *a* is greater than *b*, *a*>*b* returns 1 if not then, it returns 0.

Relational operators are used in decision making and loops in C programming.

Operator	Meaning of Operator	Example
==	Equal to	5==3 returns false (0)
>	Greater than	5>3 returns true (1)
<	Less than	5<3 returns false (0)
!=	Not equal to	5!=3 returns true(1)
>=	Greater than or equal to	5>=3 returns true (1)
<=	Less than or equal to	5<=3 return false (0)

Example1:

```
#include <stdio.h>
 int main()
{
int m=40,n=20,z;
z=m > n;
{
   printf("Z=%d",z);
}
return 0;
}
```

Example2:

```
#include <stdio.h>
int main()
{
int m=40,n=20;
if (m == n)
{
printf("m and n are equal");
}
else
{
printf("m and n are not equal");
}
}
```

4. Logical Operators

Logical operators are used to combine expressions containing relation operators. In C, there are 3 logical operators:

Operator	Meaning of Operator	Example
&&	Logial AND	c=5 and d=2 ((c==5) && (d>5)) returns false.
||	Logical OR	c=5 and d=2 ((c==5) || (d>5)) returns true.
!	Logical NOT	c=5 !(c==5) returns false.

Example program for logical operators in C:

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

5. Bit wise operators in C:

- These operators are used to perform bit operations. Decimal values are converted into binary values which are the sequence of bits and bit wise operators work on these bits.
- **Truth table for bit wise operation Bit wise operators**

X	y	x|y	x & y	x ^ y	Operator_symbol	Operator_name
0	0	0	0	0	&	Bitwise_AND
0	1	1	0	1	|	Bitwise OR
1	0	1	0	1	~	Bitwise_NOT
1	1	1	1	0	^	XOR
					<<	Left Shift
					>>	Right Shift

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

    int a = 6;
   int b = 2;
  int c = 0;

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

  c = a | b;
  printf("Line 2 - Value of c is %d\n", c );

  c = a ^ b;
  printf("Line 3 - Value of c is %d\n", c );

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

  c = a << 1;
  printf("Line 5 - Value of c is %d\n", c );

  c = a >> 1;
  printf("Line 6 - Value of c is %d\n", c );
  return 0;
}
```

6. Conditional Operator

Conditional operator takes three operands and consists of two symbols ? and : Conditional operators are used for decision making in C. For example:

```
c=(c>0)?10:-10;
```

If *c* is greater than 0, value of *c* will be 10 but, if *c* is less than 0, value of *c* will be -10.

Example program for conditional/ternary operators in C:

```
#include <stdio.h>
 int main()
{
int m=40,n=20,z;
z=(m > n)?m:n;
{
   printf("Z=%d",z);
}
return 0;
}
```

7. Increment and decrement operators

In C, ++ and -- are called increment and decrement operators respectively. Both of these operators are unary operators, i.e, used on single operand. ++ adds 1 to operand and -- subtracts 1 to operand respectively. For example:

```
Let a=5 and b=10

a++;  //a becomes 6

a--;  //a becomes 5

++a;  //a becomes 6

--a;  //a becomes 5
```

```
#include<stdio.h>
int main(){
int c=2,d=2;

   printf("%d\n",c++);
   printf("%d",++c);
return0;
}
```

Difference between pre/post increment & decrement operators in C:

- Below table will explain the difference between pre/post increment and decrement operators in C.

S.no	Operator type	Operator	Description
1	Pre increment	++i	Value of i is incremented before assigning it to variable i.
2	Post-increment	i++	Value of i is incremented after assigning it to variable i.

8. Special Operators in C:

S.no	**Operators**	**Description**
1	&	This is used to get the address of the variable. Example : &a will give address of a.
2	*	This is used as pointer to a variable. Example : * a where, * is pointer to the variable a.
3	Sizeof ()	This gives the size of the variable. Example: size of (char) will give us 1.

Example program for & and * operators in C:

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

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

3. Program to show swap of two no's without using third variable

7. Decision Control Statements

The if statement in C language is used to perform operation on the basis of condition. By using if-else statement, you can perform operation either condition is true or false.

There are many ways to use if statement in C language:

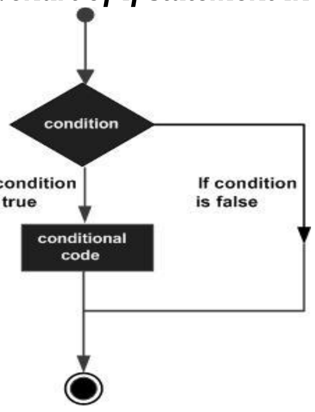
- If statement
- If-else statement
- If else-if ladder or Nested if

If Statement

The single if statement in C language is used to execute the code if condition is true. The syntax of if statement is given below:

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

Flowchart of if statement in C



Example:Program to check whether the given number is even or not.

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

C if...else statement

The if...else statement is used if the programmer wants to execute some statement/s when the test expression is true and execute some other statement/s if the test expression is false.

1. **if**(expression){
2. //code to be executed if condition is true
3. }**else**{
4. //code to be executed if condition is false
5. }

Flowchart of if-else statement in C

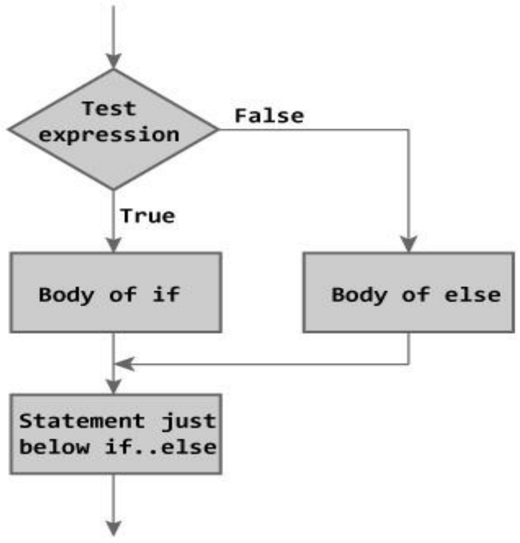


Figure: Flowchart of if...else Statement

Example 2: C if...else statement

Write a C program to check whether a number entered by user is even or odd

```
#include<stdio.h>
int main(){
int num;
     printf("Enter a number you want to check.\n");
     scanf("%d",&num);
if((num%2)==0)//checking whether remainder is 0 or not.
          printf("%d is even.",num);
else
          printf("%d is odd.",num);
return 0;
}
```

Example: Program to Check Alphabet or Not

```
#include <stdio.h>
int main()
{
   char c;
   printf("Enter a character: ");
   scanf("%c",&c);
   if( (c>='a' && c<='z') || (c>='A' && c<='Z'))
      printf("%c is an alphabet.",c);
   else
      printf("%c is not an alphabet.",c);
   return 0;
}
```

If else-if ladder Statement

The if else-if statement is used to execute one code from multiple conditions. The syntax of if else-if statement is given below:

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

Example 3: C nested if else statement

Write a C program to relate two integers entered by user using = or > or < sign.

```
#include<stdio.h>
int main(){
int numb1, numb2;
    printf("Enter two integers to check\n");
    scanf("%d %d",&numb1,&numb2);
if(numb1==numb2)//checking whether two integers are equal.
        printf("Result: %d = %d",numb1,numb2);
else
if(numb1>numb2)//checking whether numb1 is greater than numb2.
        printf("Result: %d > %d",numb1,numb2);
else
        printf("Result: %d > %d",numb2,numb1);
return 0;
}
```

3. Write a program to find largest number in 3 numbers

4. Write a program to check whether entered character is Vowel or Consonants or not a character.

8. Loop Control Statements

C Programming Loops

Loops cause program to execute the certain block of code repeatedly until test condition is false. Consider these scenarios:

- You want to execute some code/s 100 times.
- You want to execute some code/s certain number of times depending upon input from user.

These types of task can be solved in programming using loops.

There are 3 types of loops in C programming:

1. for loop
2. while loop
3. do...while loop

1. for loop in C

It iterates the code until condition is false. Here, initialization, condition and increment/decrement is given before the code. So code may be executed 0 or more times.

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

```
for(initialization;condition;incr/decr){
//code to be executed
}
```

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

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

```
      main()
    {
     int i,j;
     for(i=1;i<=5;i++)
     {
       for(j=1;j<=i;j++)
       {
             printf("%d",i);
       }
       printf("\n");
     }
    }
```

1. **C program to find power of a number using for loop**

```
main()
{
  int i,base,pow,res=1;
  printf("Enter the base: ");
  scanf("%d",&base);
  printf("Enter the power: ");
  scanf("%d",&pow);
  for(i=1;i<=pow;i++)
  {
        res=res*base;
  }
  printf("Result is: %d",res);
}
```

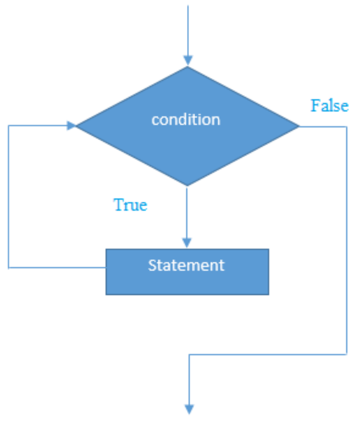
2. while loop in C

Loops causes program to execute the certain block of code repeatedly until some conditions are satisfied

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

```
1. while(condition){
2. //code to be executed
3. }
```

Flowchart of while loop in C



Example of while loop in C language

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

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

Example of while loop

Write a C program to find the factorial of a number, where the number is entered by user. (Hints: factorial of n = 1*2*3*...*n

```
#include<stdio.h>
int main(){
int number,factorial;
    printf("Enter a number.\n");
    scanf("%d",&number);
    factorial=1;
while(number>0){/* while loop continues util test condition number>0 is true */
        factorial=factorial*number;
--number;
}
printf("Factorial=%d",factorial);
return 0;
}
```

1. **Write a function that accept 5 characters and at the end, displays the smallest and the largest of all characters.**

```
    main ()
{
int  i = 0;
char chr, max, min;
printf("Enter a character :  ");
chr = getchar();
fflush (stdin);
max = min = chr;        /* set max and min to first character input */
while (i < 4)
{printf ("Enter a character : ");
chr = getchar();
fflush (stdin);
if (chr > max)
        max = chr;
if (chr < min)
        min = chr;
i =i + 1;
        }
        printf("smallest character : "),
        putchar(min);
        putchar ('\n') ;
printf ("largest character : ");
putchar(max);
putchar('\n');
 }
```

do while loop in C

To execute a part of program or code several times, we can use do-while loop of C language. The code given between the do and while block will be executed until condition is true.

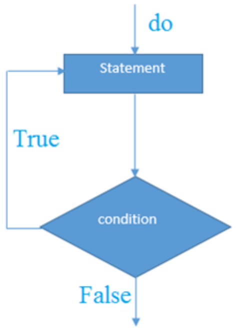
In do while loop, statement is given before the condition, so *statement or code will be executed at lease one time*. In other words, we can say it is executed 1 or more times.

do while loop syntax

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

1. **do**{
2. //code to be executed
3. }**while**(condition);

Flowchart of do while loop



do while example

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

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

Example of do...while loop

Write a C program to add all the numbers entered by a user until user enters 0.

```
#include<stdio.h>
int main(){
int sum=0,num;
do/* Codes inside the body of do...while loops are at least executed once. */
{
        printf("Enter a number\n");
        scanf("%d",&num);
        sum+=num;
}
while(num!=0);
   printf("sum=%d",sum);
return 0;
}
```

2. **Write a program, which allows the user to enter characters until the input is ‘0’. Each time a character is entered, the program should display whether it is greater/lesser than or equal to the previous character entered.**

```
      main ()
{
            char ch, prev;
            int  i = 0;
            do
            {
                        printf("Enter a character :  ");
                        ch        = getchar();
                        fflush(stdin);
                        if        (i == 0)
                  i = i + 1;
                else    /* do this only from second character */
                {
                   if  (ch == prev)
            puts("new character is equal to previous character");
                   else if (ch > prev)
            puts("new character is greater than previous character");
                   else
            puts("new character is lesser than previous character");
            }
                prev = ch;
            }  while ( ch != '0');
}
```

Assignment:
Write a function to accept ten characters from the character set, and to display whether the number of lower-case character is greater than, less than, or equal to number of upper-case characters. Display an error message if the input is not an alphabet.

Difference between while & do while loops in C:

S.no	**While**	**do while**
1	Loop is executed only when condition is true.	Loop is executed for first time irrespective of the condition. After executing while loop for first time, then condition is checked.

The Infinite Loop

A loop becomes an infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the 'for' loop are required, you can make an endless loop by leaving the conditional expression empty.

```
#include <stdio.h>
int main () {
  for( ; ; )
  {
     printf("This loop will run.\n");
  }
  return 0;
}
```

1. Write a Program to draw a box with a dimension of 10*30(i.e. 10 rows and 30 coloumn)

9. Case Control Statement

The statements which are used to execute only specific block of statements in a series of blocks are called case control statements.
There are 4 types of case control statements in C language. They are,

1. switch
2. break
3. continue
4. goto

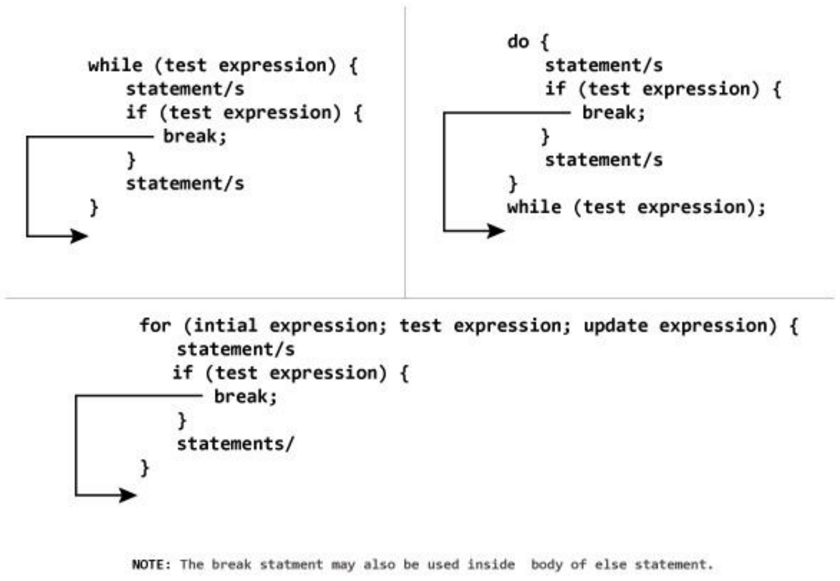
1. break statement in C:

The **break statement** in C language is used to break the execution of loop (while, do while and for) and switch case.

Syntax of break statement
break;

The break statement can be used in terminating all three loops for, while and do...while loops.

The figure below explains the working of break statement in all three type of loops.



```
while (test expression) {
    statement/s
    if (test expression) {
        break;
    }
    statement/s
}
```

```
do {
    statement/s
    if (test expression) {
        break;
    }
    statement/s
}
while (test expression);
```

```
for (intial expression; test expression; update expression) {
    statement/s
    if (test expression) {
        break;
    }
    statements/
}
```

NOTE: The break statment may also be used inside body of else statement.

Example of break statement

Write a C program to find average of maximum of *n* positive numbers entered by user. But, if the input is negative, display the average (excluding the average of negative input) and end the program.

```
# include <stdio.h>
int main(){
float num,average,sum=0;
int i,n;
  printf("Maximum no. of inputs\n");
  scanf("%d",&n);
for(i=1;i<=n;++i){
     printf("Enter n%d: ",i);
     scanf("%f",&num);
if(num<0.0)
break;//for loop breaks if num<0.0
     sum=sum+num;
}
  average=sum/(i-1);
  printf("Average=%.2f",average);
return 0;
}
```

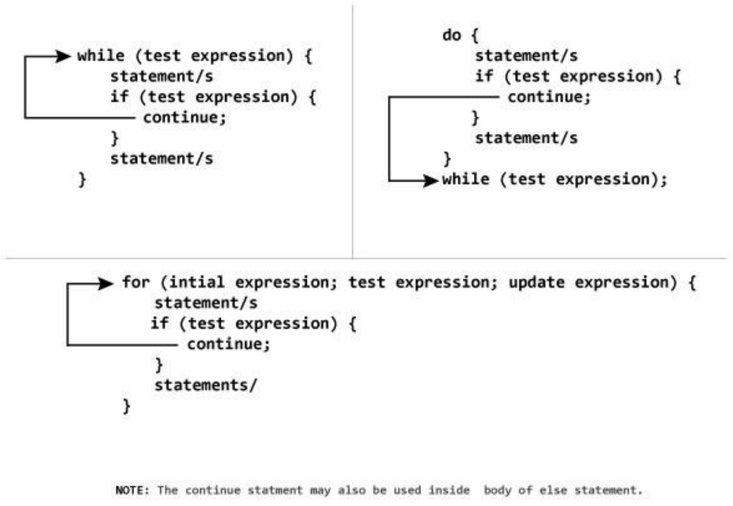
2. Continue Statement

Continue statement is used to continue the next iteration of for loop, while loop and do-while loops.
It is sometimes desirable to skip some statements inside the loop. In such cases, continue statements are used.

Syntax of continue Statement
continue;

Just like break, continue is also used with conditional if statement.

For better understanding of how continue statements works in C programming. Analyze the figure below which bypasses some code/s inside loops using continue statement.



```
while (test expression) {
    statement/s
    if (test expression) {
        continue;
    }
    statement/s
}
```

```
do {
    statement/s
    if (test expression) {
        continue;
    }
    statement/s
}
while (test expression);
```

```
for (intial expression; test expression; update expression) {
    statement/s
    if (test expression) {
        continue;
    }
    statements/
}
```

NOTE: The continue statment may also be used inside body of else statement.

Example of continue statement

Write a C program to find the product of 4 integers entered by a user. If user enters 0 skip it.

```
# include <stdio.h>
int main(){
int i,num,product;
for(i=1,product=1;i<=4;++i){
      printf("Enter num%d:",i);
      scanf("%d",&num);
if(num==0)
continue;
      product*=num;
}
```

```
    printf("product=%d",product);
return 0;
}
```

3. switch case statement in C:

The switch statement in C language is used *to execute the code from multiple conditions*.

- Switch case statements are used to execute only specific case statements based on the switch expression.
- Below is the syntax for switch case statement.

```
switch (expression)
{
case label1:   statements;
break;
case label2:   statements;
break;
default:    statements;
break;
}
```

Example of switch...case statement

```
/* C Program to create a simple calculator for addition, subtraction,
   multiplication and division */
# include <stdio.h>
int main(){
char ch;
float num1,num2;
    printf("Select an operator either + or - or * or / \n");
    scanf("%c",&ch);
    printf("Enter two operands: ");
    scanf("%f%f",&num1,&num2);
switch(ch){
case'+':
            printf("%.1f + %.1f = %.1f",num1, num2, num1+num2);
break;
case'-':
            printf("%.1f - %.1f = %.1f",num1, num2, num1-num2);
break;
case'*':
            printf("%.1f * %.1f = %.1f",num1, num2, num1*num2);
break;
case'/':
            printf("%.1f / %.1f = %.1f",num1, num2, num1/num2);
break;
default:
```

```
/* If operator is other than +, -, * or /, error message is shown */
          printf("Error! operator is not correct");
break;
}
return 0;
}
```

The break statement at the end of each case cause switch statement to exit. If break statement is not used, all statements below that case statement are also executed.

Example: Program to accept a number from 0 to 9, along with a string. The string should then be displayed the number of times specified.

```
main ()
          {
                    char str [15], inp;
                    puts ("Enter number of times to display a string (0 to 9)");
                    inp = getchar ();  fflush (stdin);
                    puts ("Enter string to display");
                    gets (str);
                    switch (inp)
                    {
                              case '9'      :      puts (str);
                              case '8'      :      puts (str);
                              case '7'      :      puts (str);
                              case '6'      :      puts (str);
                              case '5'      :      puts (str);
                              case '4'      :      puts (str);
                              case '3'      :      puts (str);
                              case '2'      :      puts (str);
                              case '1'      :      puts (str);
                              case '0'      :      break;
                    }
          }
```

Write a program to display the following menu and accept a choice number. If an invalid choice is entered then an appropriate error message must be displayed, else the choice number entered must be displayed.

```
Menu
1.Create a Directory
2.Delete a Directory
3.Show a Directory
4.Exit
Your Choice:
```

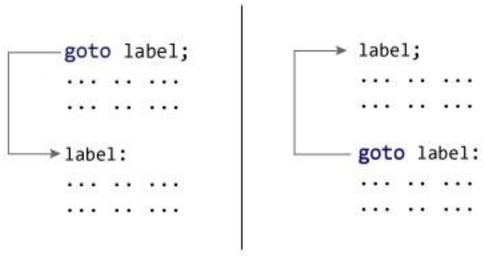
4. goto statement in C:

- goto statements is used to transfer the normal flow of a program to the specified label in the program.
- Below is the syntax for goto statement in C.

Syntax of goto statement

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

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



Example of goto statement

/* This program calculates the average of numbers entered by user. If user enters negative number, it ignores that number and calculates the average of number entered before it.*/

```
# include <stdio.h>
int main(){
float num,average,sum=0;
int i,n;
   printf("Maximum no. of inputs: ");
   scanf("%d",&n);
for(i=1;i<=n;++i){
       printf("Enter n%d: ",i);
       scanf("%f",&num);
if(num<0.0)
goto jump;/* control of the program moves to label jump */
       sum=sum+num;
}
jump:
  average=sum/(i-1);
  printf("Average: %.2f",average);
return 0;
}
```

11. Functions

1. WHAT IS C FUNCTION?

- C program is a collection of one or more functions.
- A function is a block of code that performs a specific task.
- A function is a set of statements that take inputs, do some specific computation and produces output.
- A large C program is divided into basic building blocks called C function. C function contains set of instructions enclosed by "{ }" which performs specific operation in a C program. Actually, Collection of these functions creates a C program.

2. USES OF C FUNCTIONS:

- C functions are used to avoid rewriting same logic/code again and again in a program.
- There is no limit in calling C functions to make use of same functionality wherever required.
- We can call functions any number of times in a program and from any place in a program.
- A large C program can easily be tracked when it is divided into functions.
- The core concept of C functions are, re-usability, dividing a big task into small pieces to achieve the functionality and to improve understandability of very large C programs.

3. Advantage of functions in C

There are many advantages of functions.
1) Code Reusability:
Writing functions avoids rewriting the same code over and over.

2) Code optimization:
Using functions it becomes easier to write programs and keep track of what they are doing. If the operation of a program can be divided into separate activities, and each activity placed in a different function, then each could be written and checked more or less independently.

4. Types of C functions

There are two types of functions in C programming:

- Library function
- User defined function

1. Library function

The standard library functions are built-in functions in C programming to handle tasks such as mathematical computations, I/O processing, string handling etc.

These functions are defined in the header file. When you include the header file, these functions are available for use. 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 **"stdio.h"** header file.

There are other numerous library functions defined under **"stdio.h"**, such as **scanf()**, **fprintf()**, **getchar()** etc. Once you include **"stdio.h"** in your program, all these functions are available for use.

LIST OF MOST USED HEADER FILES IN C:

S.No	Header file	Description
1	**stdio.h**	This is standard input/output header file in which Input/Output functions are declared
2	**conio.h**	This is console input/output header file
3	**string.h**	All string related functions are defined in this header file
4	**stdlib.h**	This header file contains general functions used in C programs
5	**math.h**	All maths related functions are defined in this header file
6	**time.h**	This header file contains time and clock related functions

<ctype.h>:Character type functions

isalpha():checks whether character is alphabetic

```
#include <stdio.h>
int main()
{
char ch;
printf("Enter any charactern");
scanf("%c", &ch);
if ( isalpha ( ch ) )
printf ( "nEntered character is alphabetic" ) ;
else
printf ( "nEntered character is not alphabetic" ) ;
}
```

isdigit: Tests whether a character is digit or not

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

```
int main()
{
   char c;
   c='5';
   printf("Result when numeric character is passed: %d", isdigit(c));

   c='+';
   printf("\nResult when non-numeric character is passed: %d", isdigit(c));

   return 0;
}
```

```
#include <stdio.h>
#include <math.h>
int main(){
int num;
float squareRoot;
    printf("Enter a number: ");
    scanf("%d",&num);
    squareRoot = sqrt(num);
    printf("Square root of %d=%.2f ",num,squareRoot);
return 0;
}
```

User defined function

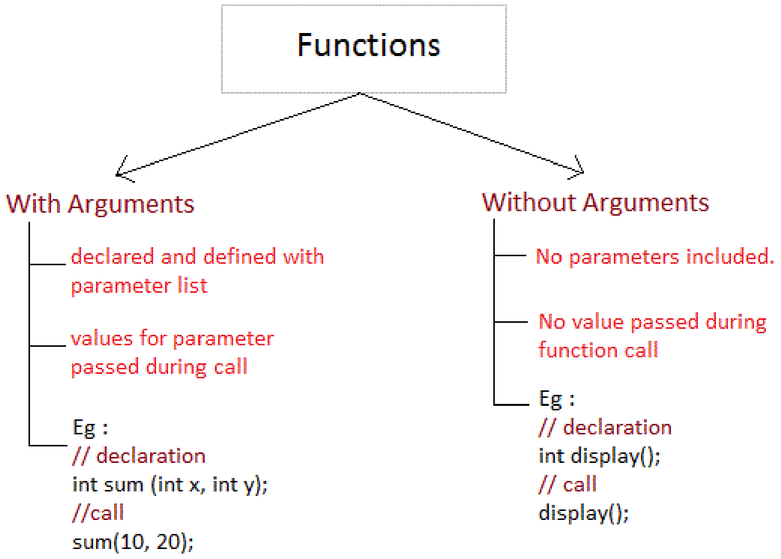
- C allows programmer to define their own function according to their requirement. These types of functions are known as user-defined functions.
- C allow programmers to define functions. Such functions created by the user are called user-defined functions.
- Depending upon the complexity and requirement of the program, you can create as many user-defined functions as you want.

C FUNCTION DECLARATION, FUNCTION CALL AND FUNCTION DEFINITION:

There are 3 aspects in each C function. They are,

- Function declaration or prototype – This informs compiler about the function name, function parameters and return value’s data type.
- Function call – This calls the actual function
- Function definition – This contains all the statements to be executed.

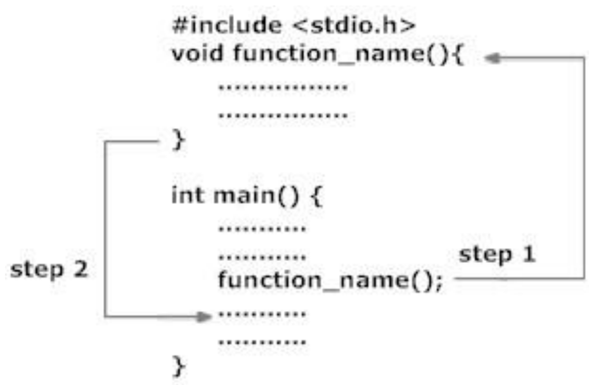
S.no	C function aspects	Syntax
1	function definition	return_type function_name (arguments list) { Body of function; }
2	function call	function_name (arguments list);
3	function declaration	return_type function_name (argument list);



How user-defined function works in C Programming?

```
#include <stdio.h>
void function_name();
int main()
{
Function _name();
}
void function_name()
{
……………….
……………….
}
```

OR



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

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

Fig: Working of Functions

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 function call.

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

Example of user-defined function

Write a C program to add two integers. Make a function add to add integers and display sum inmain() function.

```
#include<stdio.h>
int add(int a,int b);//function prototype(declaration)
int main(){
int num1,num2,sum;
    printf("Enters two number to add\n");
    scanf("%d %d",&num1,&num2);
    sum=add(num1,num2);//function call
    printf("sum=%d",sum);
return 0;
}
int add(int a,int b)//function declarator
{
/* Start of function definition. */
int add;
    add=a+b;
return add;//return statement of function
}
```

Assignment:

Write a program to perform Addition, Substraction, Multiplication and Division using different functions.

Example:

```
main( )
{
printf ( "\nI am in main" ) ;
Mysore( ) ;
Mandya( ) ;
Banglore( ) ;
}
void Mysore( )
{
printf ( "\nI am inMysore" ) ;
}
void Mandya( )
{
printf ( "\nI am in Mandya" ) ;
}
void Banglore( )
{
printf ( "\nI am in Banglore" ) ;
}
```

Example:

```
main( )
{
printf ( "\nI am in main" ) ;
Mysore( ) ;
}
void Mysore ( )
{
printf ( "\nI am in Mysore " ) ;
Mandya( ) ;
}
void Mandya ( )
{
printf ( "\nI am in Mandya " ) ;
Banglore( ) ;
}
void Banglore ( )
{
printf ( "\nI am in Banglore " ) ;
}
```

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 return statement.
- Return statement is used for returning a value from function definition to calling function.

Syntax of return statement

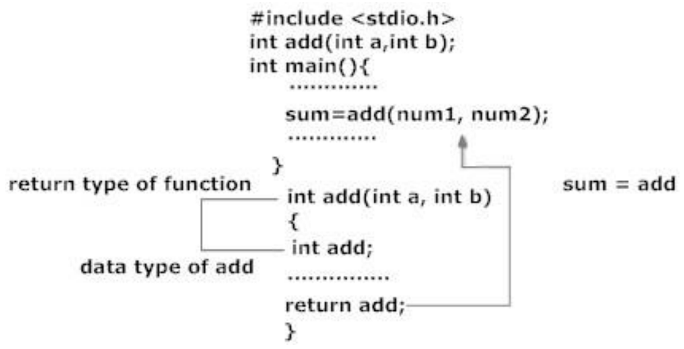
```
return (expression);
```

The **return** statement serves two purposes:

(1) On executing the **return** statement it immediately transfers the control back to the calling program.
(2) It returns the value present in the parentheses after **return**, to the calling program. In the above program the value of sum of three numbers is being returned.

For example:

```
return a;
return (a+b);
```



```
#include <stdio.h>
int add(int a,int b);
int main(){
   .............
   sum=add(num1, num2);
   .............
}
   int add(int a, int b)
   {
    int add;
   ...............
   return add;
   }
```

- There is no restriction on the number of **return** statements that may be present in a function. Also, the **return** statement need not always be present at the end of the called function. The following program illustrates these facts.

Example : Function that return some value

```
#include<stdio.h>

int larger(int a,int b);    // function declaration

void main()
{
 int i,j,k;
 i=99;
 j=112;
 k=larger(i,j);       // function call
 printf("%d",k);
}
```

```
int larger(int a,int b)     // function declaration
{
 if(a>b)
 return a;
 else
 return b;
}

int main(){
    char s;
    s=fun();
    printf("%c",s);
}
int fun( )
{
char ch ;
printf ( "\nenter any alphabet " ) ;
scanf ( "%c", &ch ) ;
if ( ch >= 65 && ch <= 90 )
return ( ch ) ;
else
return ( ch + 2 ) ;
}
```

- A function can return only one value at a time. Thus, the following statements are invalid.

```
 return ( a, b ) ;
return ( x, 12 ) ;
```

Calling Convention

Calling convention indicates the order in which arguments are passed to a function when a function call is encountered. There are two possibilities here:

Arguments might be passed from left to right.
Arguments might be passed from right to left.

C language follows the second order.

Example:

```
int a = 1 ;
printf ( "%d %d %d", a, ++a, a++ ) ;
```

It appears that this **printf( )** would output 1 2 3.

This however is not the case. Surprisingly, it outputs 3 3 1. This is because C’s calling convention is from right to left.

HOW TO CALL C FUNCTIONS IN A PROGRAM?

There are two ways that a C function can be called from a program. They are,

1. Call by value
2. Call by reference

1. CALL BY VALUE:

In call by value, **original value is not modified**.

- In call by value method, the value of the variable is passed to the function as parameter.
- The value of the actual parameter can not be modified by formal parameter.
- Different Memory is allocated for both actual and formal parameters. Because, value of actual parameter is copied to formal parameter.

Note:

- Actual parameter – This is the argument which is used in function call.
- Formal parameter – This is the argument which is used in function definition

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

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

2. CALL BY REFERENCE:

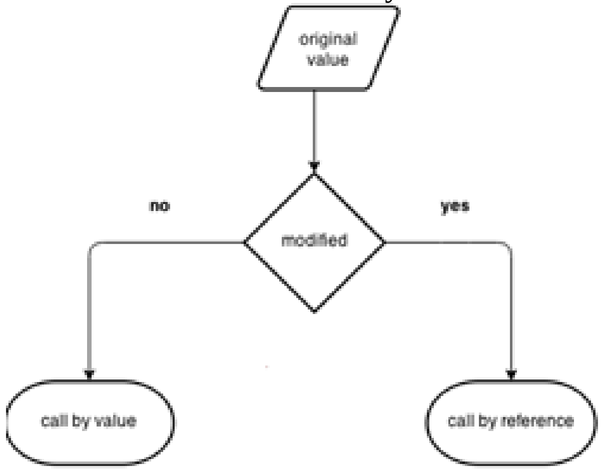
- In call by reference, **original value is modified** because we pass reference (address).
- In call by reference method, the address of the variable is passed to the function as parameter.
- The value of the actual parameter can be modified by formal parameter.
- Same memory is used for both actual and formal parameters since only address is used by both parameters.

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

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

Call by value and call by reference in C

There are two ways to pass value or data to function in C language: *call by value* and *call by reference*. Original value is not modified in call by value but it is modified in call by reference.



No.	Call by value	Call by reference
1	A copy of value is passed to the function	An address of value is passed to the function
2	Changes made inside the function is not reflected on other functions	Changes made inside the function is reflected outside the function also
3	Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location

Types of User-defined Functions in C Programming

All C functions can be called either with arguments or without arguments in a C program. These functions may or may not return values to the calling function. Now, we will see simple example C programs for each one of the below.

1. C function with arguments (parameters) and with return value
2. C function with arguments (parameters) and without return value
3. C function without arguments (parameters) and without return value
4. C function without arguments (parameters) and with return value

S.no	C function	syntax

1	with arguments and with return values	int function (int); // function declaration
		function (a); // function call
		int function(int a) // function definition
		{statements; return a;}

2	with arguments and without return values	void function (int); // function declaration
		function(a); // function call
		void function(int a) // function definition
		{statements;}

3	without arguments and without return values	void function(); // function declaration
		function(); // function call
		void function() // function definition
		{statements;}

4	without arguments and with return values	int function ( ); // function declaration
		function ( ); // function call
		int function( ) // function definition
		{statements; return a;}

Note:

- If the return data type of a function is “void”, then, it can’t return any values to the calling function.
- If the return data type of the function is other than void such as “int, float, double etc”, then, it can return values to the calling function.

Function with no arguments and no return value.

```
/*C program to check whether a number entered by user is prime or not using function with no arguments and no return value*/
#include<stdio.h>
void prime();
int main(){
   prime();//No argument is passed to prime().
return 0;
}
```

```
void prime(){
/* There is no return value to calling function main(). Hence, return type of prime() is void */
int num,i,flag=0;
   printf("Enter positive integer to check:\n");
   scanf("%d",&num);
for(i=2;i<=num/2;++i){
if(num%i==0){
          flag=1;
}
}
if(flag==1)
      printf("%d is not prime",num);
else
     printf("%d is prime",num);
}
```

Function with no arguments but return value

```
/*C program to check whether a number entered by user is prime or not using function with no arguments but having return value */
#include<stdio.h>
int input();
int main(){
int num,i,flag =0;
   num=input();/* No argument is passed to input() */
for(i=2; i<=num/2;++i){
if(num%i==0){
      flag =1;
break;
}
}
if(flag ==1)
      printf("%d is not prime",num);
else
      printf("%d is prime", num);
return 0;
}
int input(){/* Integer value is returned from input() to calling function */
int n;
   printf("Enter positive integer to check:\n");
   scanf("%d",&n);
return n;
}
```

Function with arguments and no return value

```
/*Program to check whether a number entered by user is prime or not using function with arguments and no
return value */

#include<stdio.h>
void check_display(int n);
int main(){
int num;
   printf("Enter positive number to check:\n");
   scanf("%d",&num);
   check_display(num);/* Argument num is passed to function. */
return 0;
}
void check_display(int n){
/* There is no return value to calling function. Hence, return type of function is void. */
int i, flag =0;
for(i=2; i<=n/2;++i){
if(n%i==0){
      flag =1;
break;
}
}
if(flag ==1)
      printf("%d is not prime",n);
else
      printf("%d is prime", n);
}
```

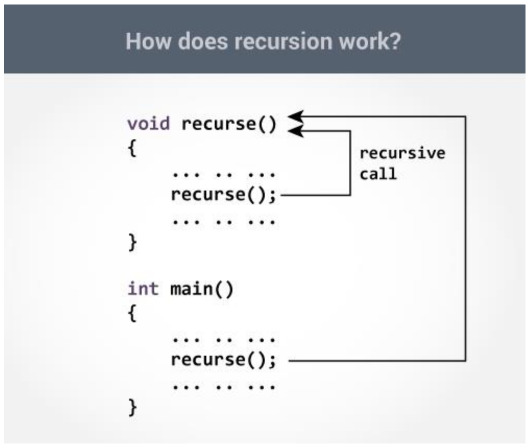
Function with argument and a return value

```
/* Program to check whether a number entered by user is prime or not using function with argument and return
value */

#include<stdio.h>
int check(int n);
int main(){
int num,num_check=0;
   printf("Enter positive number to check:\n");
   scanf("%d",&num);
   num_check=check(num);/* Argument num is passed to check() function. */
if(num_check==1)
     printf("%d is not prime",num);
else
     printf("%d is prime",num);
return 0;
}
int check(int n){
/* Integer value is returned from function check() */
int i;
for(i=2;i<=n/2;++i){
if(n%i==0)
return1;
}
return 0;
}
```

C Programming Recursion

When *function is called within the same function*, it is known as **recursion** in C. The function which calls the same function, is known as **recursive function**.

A function that calls itself, and doesn't perform any task after function call, is know as **tail recursion**. In tail recursion, we generally call the same function with return statement. An example of tail recursion is given below.



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

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

Let's see a simple example of recursion.

```
recursion_function(){
recursion_function();//calling self function
}
```

Example of recursion in C programming

Write a C program to find sum of first n natural numbers using recursion. Note: Positive integers are known as natural number i.e. 1, 2, 3....n

```
#include<stdio.h>
int sum(int n);
int main(){
int num,add;
    printf("Enter a positive integer:\n");
    scanf("%d",&num);
    add=sum(num);
    printf("sum=%d",add);
}
int sum(int n){
if(n==0)
return n;
else
return n+sum(n-1);/*self call  to function sum() */
}
```

For better visualization of recursion in this example:

```
sum(5)
=5+sum(4)
=5+4+sum(3)
=5+4+3+sum(2)
=5+4+3+2+sum(1)
=5+4+3+2+1+sum(0)
=5+4+3+2+1+0
=5+4+3+2+1
=5+4+3+3
=5+4+6
=5+10
=15
```

Every recursive function must be provided with a way to end the recursion. In this example when, *n* is equal to 0, there is no recursive call and recursion ends.

Example: Factorial of a Number Using Recursion

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

int main()
{
   int n;
   printf("Enter a positive integer: ");
   scanf("%d", &n);
   printf("Factorial of %d = %ld", n, multiplyNumbers(n));
   return 0;
}
long int multiplyNumbers(int n)
{
   if (n >= 1)
      return n*multiplyNumbers(n-1);
   else
      return 1;
}
```

Advantages and Disadvantages of Recursion

Recursion is more elegant and requires few variables which make program clean. Recursion can be used to replace complex nesting code by dividing the problem into same problem of its sub-type.

In other hand, it is hard to think the logic of a recursive function. It is also difficult to debug the code containing recursion.

/*Program to convertsinput character to upper- case */

```
main()
{
char c, x;
printf("Enter a charecter to convert from lower case to upper case:");
scanf("%c",&c);
                if ((c>='a') &&(c<= 'z'))
                x = convert (c);
printf("%c", x);
}

convert(z)
{
return z-32; /* 32 less from z for ASCII value of uppercase */
}
```

C – Command line arguments

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 **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>
int main( int argc, char *argv[] )
{
  int i;
  if( argc >= 2 )
   {
    printf("The arguments supplied are:\n");
    for(i=1;i< argc;i++)
    {
     printf("%s\t",argv[i]);
    }
}
   else
{
     printf("argument list is empty.\n");
   }
 return 0;
}
```

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 one.

C – Type Casting functions

- Type casting is a way to convert a variable from one data type to another data type.
- For example, if you want to store a long value into a simple integer then you can type cast long to int. You can convert values from one type to another explicitly using the cast operator.

Type Conversion in C

- A type cast is basically a conversion from one type to another. There are two types of type conversion:

1.**Implicit Type Conversion** Also known as 'automatic type conversion'.

- Done by the compiler on its own, without any external trigger from the user.
- Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data.
- All the data types of the variables are upgraded to the data type of the variable with largest data type.

```
char -> short int -> int ->
    unsigned int -> long -> unsigned ->
    long long -> float -> double -> long double
```

Example of Type Implicit Conversion:

```
#include<stdio.h>
int main()
{
   int x = 10;    // integer x
   char y = 'a';  // character c

   // y implicitly converted to int. ASCII
   // value of 'a' is 97
   x = x + y;

   float z = x + 1.0;    // x is implicitly converted to float

   printf("x = %d, z = %f", x, z);
   return 0;
}
```

2. Explicit Type Conversion– This process is also called type casting and it is user defined. Here the user can type cast the result to make it of a particular data type.

The syntax in C:(type) expression

```
// C program to demonstrate explicit type casting
#include<stdio.h>
int main()
{
   double x = 1.2;
      int sum = (int)x + 1;// Explicit conversion from double to int
   printf("sum = %d", sum);
   return 0;
}
```

INBUILT TYPECAST FUNCTIONS IN C:

- There are many inbuilt typecasting functions available in C language which performs data type conversion from one type to another.

S.no	Typecast function	Description
1	**atof()**	Converts string to float
2	**atoi()**	Converts string to int
3	**atol()**	Converts string to long
4	**itoa()**	Converts int to string
5	**ltoa()**	Converts long to string

C – atof() function

- atof() function in C language converts string data type to float data type. Syntax for atof() function is given below.

double atof (const char* string);

- "stdlib.h" header file supports all the type casting functions in C language.

EXAMPLE PROGRAM FOR ATOF() FUNCTION IN C:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
char a[10] = "3.14";
float pi = atof(a);
printf("Value of pi = %f\n", pi);
return 0;
}
```

10. Storage Class Specifiers

Storage class specifiers in C language tells the compiler where to store a variable, how to store the variable.

There are 4 types of storage class:

1. automatic
2. external
3. static
4. register

S. No.	Storage Specifier	Storage place	Initial / default value	Scope	Life
1	**Auto**	CPU Memory	Garbage value	local	Within the function only.
2	**Extern**	CPU memory	Zero	Global	Till the end of the main program. Variable definition might be anywhere in the C program
3	**Static**	CPU memory	Zero	local	Retains the value of the variable between different function calls.
4	**Register**	Register memory	Garbage value	local	Within the function

Note:

- For faster access of a variable, it is better to go for register specifiers rather than auto specifiers.
- Because, register variables are stored in register memory whereas auto variables are stored in main CPU memory.
- Only few variables can be stored in register memory. So, we can use variables as register that are used very often in a C program.

Automatic variables

A variable declared inside a function without any storage class specification, is by default an automatic variable. They are created when a function is called and are destroyed automatically when the function exits.

```
void main()
{
 int detail;
 or
 auto int detail;    //Both are same
}
```

The following C program demonstrates the visibility level of auto variables.

```
#include <stdio.h>
int main( )
{
  auto int i = 1;
  {
    auto int i = 2;
    {
      auto int i = 3;
      printf ( "\n%d ", i);
    }
   printf ( "%d ", i);
  }
  printf( "%d\n", i);
}
```

External Variable

An **external variable** is a variable defined outside any function block.

Example 1:

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

Analysis: This program is compiled successfully. Here var is declared only. Notice var is never used so no problems.

Example 2:

```
extern int var;
int main()
{
 var = 10;
printf("%d",var);
 return 0;
}
```

Analysis: This program throws error in compilation. Because var is declared but not defined anywhere. Essentially, the var isn’t allocated any memory. And the program is trying to change the value to 10 of a variable that doesn’t exist at all.

Example 3:

```
extern int var = 0;
int main()
{
 var = 10;
printf("%d",var);
 return 0;
}
```

Analysis: Guess this program will work? Well, here comes another surprise from C standards. They say that..if a variable is only declared and an initializer is also provided with that declaration, then the memory for that variable will be allocated i.e. that variable will be considered as defined. Therefore, as per the C standard, this program will compile successfully and work.
So that was a preliminary look at "extern" keyword in C.

```
int number;
void main()
{
 number=10;
}
fun1()
{
 number=20;
}
fun2()
{
 number=30;
}
```

Here the global variable number is available to all three functions.

Static variables

A static variable tells the compiler to persist the variable until the end of program.

static is initialized only once and remains into existence till the end of program. A static variable can either be internal or external depending upon the place of declaraction.

They are assigned 0 (zero) as default value by the compiler.

```
void test();
main()
{
 test();
 test();
 test();
}
void test()
{
 static int a = 0;        //Static variable
 a = a+1;
 printf("%d\t",a);
}
```

Example 2:

```
void staticDemo()
{
  static int i;
  {
    static int i = 1;
    printf("%d ", i);
    i++;
  }
  printf("%d\n", i);
  i++;
}

int main()
{
  staticDemo();
  staticDemo();
}
```

Register variable

A `register` declaration is equivalent to an `auto` declaration, Register variable inform the compiler to store the variable in register instead of memory. Register variable has faster access than normal variable. Frequently used variables are kept in register. Only few variables can be placed inside register.

NOTE: We can never get the address of such variables.

Syntax:

```
register int number;
```

```
#include <stdio.h>
#include <stdlib.h>
 int main()
{
  register int i = 10;
  int *p = &i; //error: address of register variable requested

  printf("Value of i: %d", *p);
  printf("Address of i: %u", p);

}
```

12. Array

- C Array is a collection of variables belongings to the same data type. You can store group of data of same data type in an array.
- An array is a collection of data that holds fixed number of values of same type. For example: if you want to store marks of 100 students, you can create an array for it.

  float marks[100];

- An array is a sequence of data item of homogeneous value (same type).

Arrays are of two types:

1. One-dimensional arrays
2. Multidimensional arrays

1. One-dimensional arrays

Declaration of an array

data_type array_name[array_size];

For example,

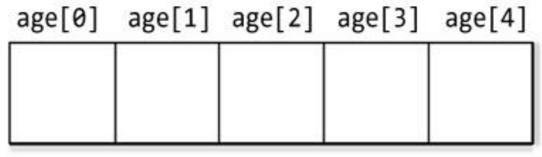
float mark[5];

Elements of an Array

You can access elements of an array by indices.

The maximum number of elements an array can hold depends upon the size of an array. Consider this code below:
int age[5];

This array can hold 5 integer elements.



Few key notes:

Arrays have 0 as the first index not 1. In this example, age[0]

If the size of an array is n, to access the last element, (n-1) index is used.

Initialization of an array in C

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

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

Another method to initialize array during declaration:

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

Example:

```
#include <stdio.h>
int main()
{
    int i, a[5]={1,2,3,4,5};
    for(i=0;i<5;i++)
        printf("a[%d]=%d\n",i,a[i]);
    return 0;
}
```

Example of array in C programming

```
/* C program to find the sum marks of n students using arrays */
#include<stdio.h>
int main(){
int marks[10],i,n,sum=0;
    printf("Enter number of students: ");
    scanf("%d",&n);
for(i=0;i<n;++i){
        printf("Enter marks of student%d: ",i+1);
        scanf("%d",&marks[i]);
        sum+=marks[i];
}
    printf("Sum= %d",sum);
return 0;
}
```

Assignment:
Program to Read & Display the marks of 6 sub and calculate average of 6 sub

2. Multidimensional Arrays

- In C programming, you can create array of an array known as multidimensional array. 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 table with 3 row and each row has 4 column.

Three-dimensional (3d) array.

For example: float y[2][3][4];

Here, The array y can hold 24 elements.

You can think this example as: Each 2 elements have 4 elements, which makes 8 elements and each 8 elements can have 3 elements. Hence, the total number of elements is 24.

Initialization of a two dimensional 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 three dimensional 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 of Multidimensional Array in C

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

```
int main()
{
    int i,j;
    int arr[2][2]={1,2,3,4};
    for(i=0;i<2;i++)
    {
        for(j=0;j<2;j++)
        {
        printf("arr[%d][%d]=%d\n",i,j,arr[i][j]);
        }
    }
    return 0;
}
```

Write a C program to find sum of two matrix of order 2*2 using multidimensional arrays where, elements of matrix are entered by user.

```
#include<stdio.h>
int main(){
float a[2][2], b[2][2], c[2][2];
int i,j;
   printf("Enter the elements of 1st matrix\n");

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

       c[i][j]=a[i][j]+b[i][j];/* Sum of corresponding elements of two arrays. */
}
   printf("\nSum Of Matrix:\n");
for(i=0;i<2;++i)
for(j=0;j<2;++j){
       printf("%.1f\t",c[i][j]);
if(j==1)/* To display matrix sum in order. */
            printf("\n");
}
return 0;
}
```

Example 3: Three Dimensional Array

C Program to store values entered by the user in a three-dimensional array and display it.

```
#include <stdio.h>
int main()
{
   // this array can store 12 elements

   int i, j, k, test[2][3][2];

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

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

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

Two dimensional char array

Two-Dimentional array are typically used to create an array of strings.

```
main()
{
char name[5][10];
int i;
for(i=0;i<5;i++)
{
   printf(" Enter a name which you want to register\n");
   scanf("%s",name[i]);
}
for(i=0;i<5;i++)
{
   printf("  Registerd Name%d: ",i+1);
   printf("%s\n",name[i]);
}
}
```

C Programming Arrays and Functions

In C programming, a single array element or an entire array can be passed to a function. Also, both one-dimensional and multi-dimensional array can be passed to function as argument.

1. Passing One-dimensional Array in Function

C program to pass a single element of an array to function

```
#include<stdio.h>
void display(int a)
{
  printf("%d",a);
}
int main(){
int c[]={2,3,4};
  display(c[2]);//Passing array element c[2] only.
return 0;
}
```

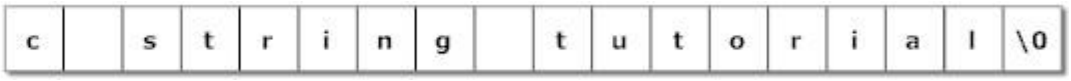
Passing an entire one-dimensional array to a function

```
float average(float marks[]);

main()

{

    float avg, marks[] = { 23.4, 55, 22.6, 3, 40.5, 18 };

    avg = average(marks); /* Only name of array is passed as argument. */

    printf("Average age=%.2f", avg);

    return 0;

}

float average(float marks[])

{

    int i;

    float avg, sum = 0.0;

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

        sum += marks[i];

    }

    avg = (sum / 6);

    return avg;

}
```

13. Strings

In C programming, array of character are called strings. A string is terminated by null character \0. For example:
"c string tutorial"

Here, "c string tutorial" is a string. When, compiler encounters strings, it appends null character at the end of string.

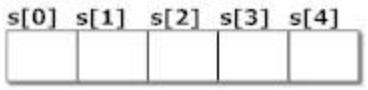


Declaration of strings

Strings are declared in a similar manner as arrays. Only difference is that, strings are of chartype.

Using arrays

char s[5];



Using pointers

Strings can also be declared using pointer.

char *p;

Initialization of strings

In C, string can be initialized in a number of different ways.

For convenience and ease, both initialization and declaration are done in the same step.

Using arrays

char c[] = "abcd";

OR,

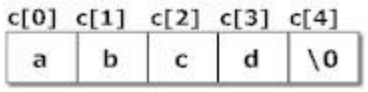
char c[50] = "abcd";

OR,

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

OR,

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



Using pointers

String can also be initialized using pointers as:

```
char *c = "abcd";
```

Reading Strings from user

Example #1: Using scanf() to read a string

Write a C program to illustrate how to read string from terminal.

```
#include<stdio.h>
int main(){
char name[20];
   printf("Enter name: ");
   scanf("%s",name);
   printf("Your name is %s.",name);
return 0;
}
```

Reading a line of text

An approach to reading a full line of text is to read and store each character one by one.

Example #2: Using getchar() to read a line of text

1. C program to read line of text character by character.

```
#include <stdio.h>
int main()
{
   char name[30], ch;
   int i = 0;
   printf("Enter name: ");
   while(ch != '\n')    // terminates if user hit enter
   {
      ch = getchar();
```

```
        name[i] = ch;
        i++;
    }
    name[i] = '\0';       // inserting null character at end
    printf("Name: %s", name);
    return 0;
}
```

Example #3: Using standard library function to read a line of text

2. C program to read line of text using gets() and puts()

To make life easier, there are predefined functions gets() and puts in C language to read and display string respectively.

```
#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;
}
```

Pointers and Strings

Suppose we wish to store "Hello". We may either store it in a string or we may ask the C compiler to store it at some location in memory and assign the address of the string in a char pointer. This is shown below:

```
char str[ ] = "Hello" ;
char *p = "Hello" ;
```

There is a difference in usage of these two forms. For example, we cannot assign a string to another, whereas, we can assign a char pointer to another char pointer. This is shown in the following program.

```
main( )

{

char str1[ ] = "Hello" ;

char str2[10] ;

char *s = "Good Morning" ;

char *q ;

str2 = str1 ; /* error */

q = s ; /* works */

printf("String1=%s\n",str1);

printf("String2=%s",q);

}
```

Also, once a string has been defined it cannot be initialized to another set of characters. Unlike strings, such an operation is perfectly valid with char pointers.

```
main( )

{

char str1[ ] = "Hello" ;

char *p = "Hello" ;

str1 = "Bye" ; /* error */

p = "Bye" ; /* works */

printf("String1:%s\n",str1);

printf("String2:%s",p);

}
```

String handling functions

- There are various string operations you can perform manually like: finding the length of a string, concatenating (joining) two strings etc.
- But, for programmer's ease, many of these library functions are already defined under the header file <string.h>.

String Manipulations In C Programming Using Library Functions

Few commonly used string handling functions are discussed below:

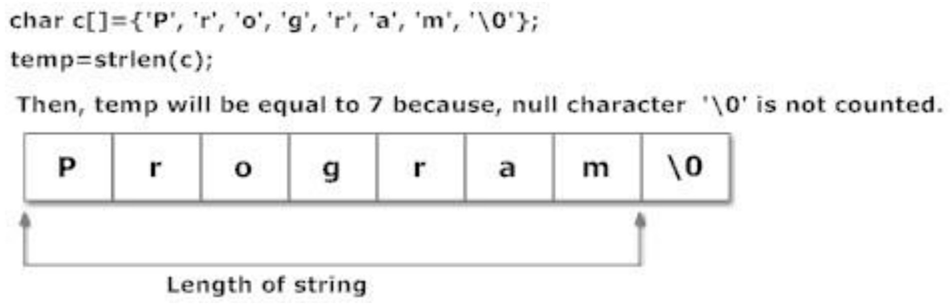
Function	Work of Function
strlen()	Calculates the length of string
strcpy()	Copies a string to another string
strcat()	Concatenates(joins) two strings
strcmp()	Compares two string
strlwr()	Converts string to lowercase
strupr()	Converts string to uppercase

C strlen() Prototype

size_t strlen(const char *str);

The function takes a single argument, i.e, the string variable whose length is to be found, and returns the length of the string passed.

The strlen() function is defined in <string.h> header file.



```
char c[]={'P', 'r', 'o', 'g', 'r', 'a', 'm', '\0'};
temp=strlen(c);
```

Then, temp will be equal to 7 because, null character '\0' is not counted.

Example: C strlen() function

```
#include <stdio.h>

#include <string.h>

int main()

{

   char a[20]="C Program";
```

```
    char b[20]={'P','r','o','g','r','a','m','\0'};
    char c[20];
    printf("Enter string: ");
    gets(c);
    printf("Length of string a = %d \n",strlen(a));
    //calculates the length of string before null charcter.
    printf("Length of string b = %d \n",strlen(b));
    printf("Length of string c = %d \n",strlen(c));
    return 0;
}
```

strcpy() Function prototype

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

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

This function returns character array destination.

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

Example: C strcpy()

```
#include <stdio.h>
#include <string.h>
int main()
{
    char str1[10]= "Mandya";
    char str2[10];
    char str3[10];
    strcpy(str2, str1);
    strcpy(str3, "Yuva Tech");
    puts(str2);
    puts(str3);
    return 0;
}
```

It is important to note that, the destination array should be large enough otherwise it may result in undefined behavior.

C strcat() Prototype

char *strcat(char *dest, const char *src)

It takes two arguments, i.e, two strings or character arrays, and stores the resultant concatenated string in the first string specified in the argument.

The pointer to the resultant string is passed as a return value.

Example: C strcat() function

```
#include <stdio.h>

#include <string.h>

int main()

{

   char str1[] = "This is ", str2[] = "C Program";

   strcat(str1,str2);

   puts(str1);

   puts(str2);

   return 0;

}
```

C strcmp() Prototype

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

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

The strcmp() compares two strings character by character. If the first character of two strings are equal, 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 string.h header file.

Return Value from strcmp()

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

Example: C strcmp() function

```
#include <stdio.h>
#include <string.h>
int main()
{
    char str1[] = "Mandya", str2[] = "Banglore", str3[] = "Mandya";
    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;
}

#include<stdio.h>
#include<string.h>
int main()
{
char a[100], b[100];
printf("Enter the first string\n");
gets(a);
printf("Enter the second string\n");
gets(b);
if( strcmp(a,b) == 0 )
printf("Entered strings are equal.\n");
else
printf("Entered strings are not equal.\n");
return 0;
}
```

C strlwr() Prototype

strlwr():Converts string to lowercase

C Program to Convert String into Lowercase Using Library Function

```
#include<stdio.h>
#include<string.h>
int main() {
   char string[100];
   printf("Enter String : ");
   gets(string);
   strlwr(string);
   printf("\nString after strlwr : %s", string);
   return (0);
}
```

C strupr() Prototype

strupr():Converts string to uppercase

Program to Convert String into Uppercase Using Library Function

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

```
int main() {
  char string[100];
  printf("Enter String : ");
  gets(string);
  strupr(string);
  printf("String after strupr : %s", string);
  return (0);
}
```

Passing Strings to Functions

String can be passed to function in similar manner as arrays as, string is also an array.

```
#include<stdio.h>
voidDisplay(char ch[]);
int main(){
char c[50];
   printf("Enter string: ");
   gets(c);
Display(c); // Passing string c to function.
return0;
}
voidDisplay(char ch[]){
   printf("String Output: ");
   puts(ch);
}
```

#define Directive (macro definition)

- The C preprocessor is a macro preprocessor (allows you to define macros) that transforms your program before it is compiled. These transformations can be inclusion of header file, macro expansions etc.
- These macro definitions allow constant values to be declared for use throughout your code.
- All preprocessing directives begins with a # symbol. For example,

Example

Let's look at how to use #define directives with numbers, strings, and expressions.

Number

The following is an example of how you use the #define directive to define a numeric constant:

```
#define NUM 10
```

In this example, the constant named NUM would contain the value of 10.

String

You can use the #define directive to define a string constant.

For example:

```
#define NAME "Google.com"
```

In this example, the constant called NAME would contain the value of "Google.com".

Below is an example C program where we define these two constants:

```
#include <stdio.h>
#define NAME "Google.com"
#define NUM 10
int main()
{
  printf("%s is over %d years old.\n", NAME, NUM);
  return 0;
}
```

This C program would print the following:

```
Google.com is over 10 years old.
```

<u>Example:</u>

```
#include <stdio.h>
#include <string.h>
#define FOUND 1
#define NOTFOUND 0
main( )
{
char Usernames[6][10] = {
```

```
"akash",
"suma",
"ramu",
"srinivas",
"gopal",
"rajesh" } ;
int i, flag, a ;
char name[10] ;
printf ( "\nEnter your name " ) ;
scanf ( "%s", name ) ;
flag = NOTFOUND ;
for ( i = 0 ; i <= 5 ; i++ )
{
a = strcmp ( Usernames[i], name ) ;
if ( a == 0 )
{
printf ( "Welcome %s",name ) ;
flag = FOUND ;
break ;
}
}
if ( flag == NOTFOUND )
printf ( "Please Check Your Username" ) ;
}
```

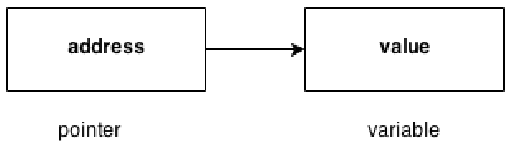
Predefined Macros

C Program to find the current time

```
#include <stdio.h>
int main()
{
  printf("Current time: %s",__TIME__);   //calculate the current time
}
```

14. Pointers

The **pointer in C language** is a variable, it is also known as locator or indicator that points to an address of a value.



- C Pointer is a variable that stores/points the address of another variable. C Pointer is used to allocate memory dynamically i.e. at run time. The pointer variable might be belonging to any of the data type such as int, float, char, double, short etc.

Syntax: data_type *var_name; Example: int *p; char *p;

- Where, * is used to denote that "p" is pointer variable and not a normal variable.

Usage of pointer

There are many usage of pointers in c language.

1) Dynamic memory allocation

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

2) Arrays, Functions and Structures

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

Symbols used in pointer

Symbol	Name	Description
& (ampersand sign)	address of operator	determines the address of a variable.
* (asterisk sign)	indirection operator	accesses the value at the address.

KEY POINTS TO REMEMBER ABOUT POINTERS IN C:

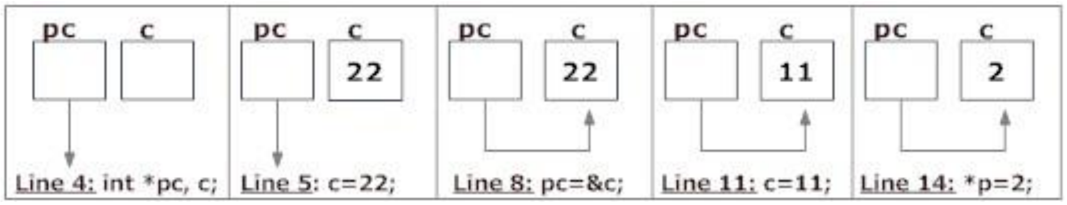
- Normal variable stores the value whereas pointer variable stores the address of the variable.
- Always C pointer is initialized to null, i.e. int *p = null.
- The value of null pointer is 0.
- If pointer is assigned to NULL, it means it is pointing to nothing.
- & symbol is used to get the address of the variable.
- * symbol is used to get the value of the variable that the pointer is pointing to.

EXAMPLE PROGRAM FOR POINTER IN C:

```
 #include <stdio.h>
int main(){
  int var=5;

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

  printf("Address: %d",&var);  //Notice, the ampersand(&) before var.

  printf ( "\nValue: %d", *( &var) ) ;

  return 0;

}
```

Example To Demonstrate Working of Pointers

```
/* Source code to demonstrate, handling of pointers in C program */
#include<stdio.h>
int main(){
int* pc;
   int c;
   c=22;
   printf("Address of c:%d\n",&c);
   printf("Value of c:%d\n\n",c);
   pc=&c;
   printf("Address of pointer pc:%d\n",pc);
   printf("Content of pointer pc:%d\n\n",*pc);
   c=11;
   printf("Address of pointer pc:%d\n",pc);
   printf("Content of pointer pc:%d\n\n",*pc);
*pc=2;
   printf("Address of c:%d\n",&c);
   printf("Value of c:%d\n\n",c);
return 0;
}
```



Common mistakes when working with pointers

Suppose, you want pointer pc to point to the address of c. Then,

```
int c, *pc;

pc = c;  // Wrong! pc is address whereas, c is not an address.

*pc = &c; // Wrong! *pc is the value pointed by address whereas, %amp;c is an address.

pc = &c; // Correct! pc is an address and, %amp;pc is also an address.

*pc = c;// Correct! *pc is the value pointed by address and, c is also a value.
```

NULL Pointers

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a null pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program −

```
#include <stdio.h>
int main () {
int *ptr = NULL;
   printf("The value of ptr is : %x\n", ptr  );//%x for printing Hexadecimal
   return 0;
}
```

Pointers in Detail

1. Pointer arithmetic

There are four arithmetic operators that can be used in pointers: ++, --, +, -

Incrementing a Pointer

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array −

```
#include <stdio.h>
const int MAX = 3;
int main () {
   int  var[] = {10, 100, 200};
   int  i, *ptr;
   ptr = &var;      /* let us have array address in pointer */
   for ( i = 0; i < MAX; i++) {
      printf("Address of var[%d] = %x\n", i, ptr );
      printf("Value of var[%d] = %d\n", i, *ptr );
      ptr++;       /* move to the next location */
   }
   return 0;
}
```

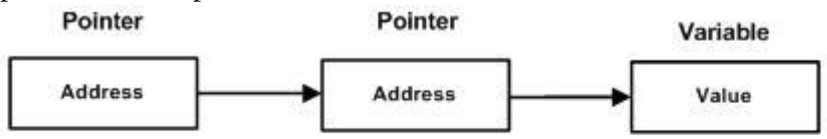
2. Array of pointers

Arrays are closely related to pointers in C programming but the important difference between them is that, a pointer variable can take different addresses as value whereas, in case of array it is fixed. This can be demonstrated by an example:

```
#include <stdio.h>
int main( )
{
   int i,val[7] = { 11, 22, 33, 44, 55, 66, 77 } ;
   /* for loop to print value and address of each element of array*/
   for (i = 0 ; i <= 6 ; i++ )
   {
         printf("val[%d]: value is %d and address is %u\n", i, val[i], &val[i]);
   }
   return 0;
}
```

3. Pointer to Pointer

A pointer to a pointer is a form of multiple indirection, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, the following declaration declares a pointer to a pointer of type int −
int **var;

```
#include <stdio.h>
int main () {
   int  var;
   int  *ptr;
   int  **pptr;
   var = 3000;
   ptr = &var;  /* take the address of var */
   pptr = &ptr;     /* take the address of ptr using address of operator & */
   /* take the value using pptr */
   printf("Value of var = %d\n", var );
   printf("Value available at *ptr = %d\n", *ptr );
   printf("Value available at **pptr = %d\n", **pptr);
   return 0;
}
```

4. Passing pointers to functions

When, argument is passed using pointer, address of the memory location is passed instead of value.

Example of Pointer and Functions

/* C Program to swap two numbers using pointers and function. */

```
#include<stdio.h>
void swap(int *a,int *b);
int main(){
int num1=5,num2=10;
  swap(&num1,&num2);/* address of num1 and num2 is passed to swap function */
  printf("Number1 = %d\n",num1);
  printf("Number2 = %d",num2);
return 0;
}
void swap(int *a,int *b){/* pointer a and b points to address of num1 and num2 respectively */
int temp;
  temp=*a;
*a=*b;
*b=temp;
}
```

Consider example with pointers –

```
#include <stdio.h>
int main( )
{
   /*Pointer variable*/
   int *p;
   /*Array declaration*/
   int val[7] = { 11, 22, 33, 44, 55, 66, 77 } ;

   /* Assigning the address of val[0] to pointer: 88820*/
```

```
  p = &val[0];

  for ( int i = 0 ; i <= 6 ; i++ )
  {
    printf("val[%d]: value is %d and address is %u", i, *p, p);
    p++;
  }
  return 0;
}
```

Pointer logic

You must have understood the logic in above code so now its time to play with few pointer arithmetic and expressions.

if p = &val[0] which means

*p =val[0]

(p+0) == &val[0] & *(p+1) == val[1]

(p+2) == &val[2] & *(p+2) == val[2]

(p+n) == &val[n] & *(p+n) == val[n]

Now its time to rewrite our old example program in a better way –

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

{

  int *p;

  int i, val[7] = { 11, 22, 33, 44, 55, 66, 77 } ;

  p = &val[0];

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

  {
    printf("val[%d]: value is %d and address is %u\n", i, *(p+i), (p+i));
  }
  return 0;
}
```

15. Dynamic Memory Allocation

- The process of allocating memory at runtime is known as dynamic memory allocation.
- It is used for dynamic memory allocation during execution of the program.
- Library routines known as "memory management functions" are used for allocating and freeing memory during execution of a program.
- These functions are defined in stdlib.h.

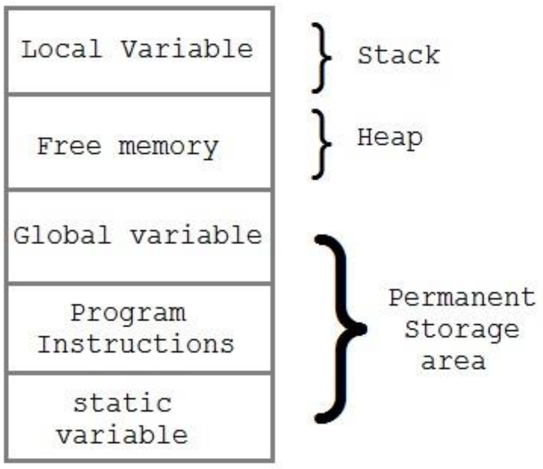
1. malloc()
2. calloc()
3. realloc()
4. free()

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

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

Memory Allocation Process

- Global variables, static variables and program instructions get their memory in permanent storage area
- Local variables are stored in area called Stack.
- The memory space between these two region is known as Heap area.
- This region is used for dynamic memory allocation during execution of the program. The size of heap keep changing.



Function	Description
malloc()	• allocates single block of requested memory. • allocates requested size of bytes and returns a void pointer pointing to the first byte of the allocated space
calloc()	• allocates multiple block of requested memory. • allocates space for an array of elements, initialize them to zero and then return a void pointer to the memory
free()	• releases previously allocated memory
realloc()	• modify the size of previously allocated space • reallocates the memory occupied by malloc() or calloc() functions.

malloc()

The name malloc stands for "memory allocation". The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form.

Syntax of malloc()

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

Here, *ptr* is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

```
ptr=(int*)malloc(100*sizeof(int));
```

This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

calloc()

The name calloc stands for "contiguous allocation". The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

Syntax of calloc()

```
ptr=(cast-type*)calloc(n,element-size);
```

This statement will allocate contiguous space in memory for an array of *n* elements. For example:

```
ptr=(float*)calloc(25,sizeof(float));
```

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

DIFFERENCE BETWEEN MALLOC() AND CALLOC() FUNCTIONS IN C:

S.no	malloc()	calloc()
1	It allocates only single block of requested memory	It allocates multiple blocks of requested memory
2	int *ptr;ptr = malloc(20 * sizeof(int));For the above, 20*4 bytes of memory only allocated in one block. Total = 80 bytes	int *ptr;Ptr = calloc(20, 20 * sizeof(int));For the above, 20 blocks of memory will be created and each contains 20*4 bytes of memory. Total = 1600 bytes
3	malloc () doesn't initializes the allocated memory. It contains garbage values	calloc () initializes the allocated memory to zero

free()

Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space.

syntax of free()

```
free(ptr);
```

This statement causes the space in memory pointer by ptr to be deallocated.

Examples of calloc() and malloc()

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using malloc() function.

```
#include<stdio.h>
#include<stdlib.h>
int main(){
int n,i,*ptr,sum=0;
   printf("Enter number of elements: ");
   scanf("%d",&n);
   ptr=(int*)malloc(n*sizeof(int));//memory allocated using malloc
if(ptr==NULL)
{
      printf("Error! memory not allocated.");
exit(0);
}
```

```
   printf("Enter elements of array: ");
for(i=0;i<n;++i)
{
      scanf("%d",ptr+i);
      sum+=*(ptr+i);
}
   printf("Sum=%d",sum);
   free(ptr);
return 0;
}
```

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using calloc() function.

```
#include<stdio.h>
#include<stdlib.h>
int main(){
int n,i,*ptr,sum=0;
   printf("Enter number of elements: ");
   scanf("%d",&n);
   ptr=(int*)calloc(n,sizeof(int));
if(ptr==NULL)
{
      printf("Error! memory not allocated.");
exit(0);
}
   printf("Enter elements of array: ");
for(i=0;i<n;++i)
{
      scanf("%d",ptr+i);
      sum+=*(ptr+i);
}
   printf("Sum=%d",sum);
   free(ptr);
return 0;
}
```

realloc()

If the previously allocated memory is insufficient or more than sufficient. Then, you can change memory size previously allocated using realloc().

Syntax of realloc()

```
ptr=realloc(ptr,newsize);
```

Here, ptr is reallocated with size of newsize.

```
#include<stdio.h>
#include<stdlib.h>
int main(){
int*ptr,i,n1,n2;
    printf("Enter size of array: ");
    scanf("%d",&n1);
    ptr=(int*)malloc(n1*sizeof(int));
    printf("Address of previously allocated memory: ");
for(i=0;i<n1;++i)
         printf("%u\t",ptr+i);
    printf("\nEnter new size of array: ");
    scanf("%d",&n2);
    ptr=realloc(ptr,n2);
for(i=0;i<n2;++i)
         printf("%u\t",ptr+i);
return 0;
}
```

16. Structure and Union

Structure

C Structure is a collection of different data types which are grouped together and each element in a C structure is called member.

- If you want to access structure members in C, structure variable should be declared.
- Many structure variables can be declared for same structure and memory will be allocated for each separately.
- It is widely used to store student information, employee information, product information, book information etc.

DIFFERENCE BETWEEN C VARIABLE, C ARRAY AND C STRUCTURE:

- A normal C variable can hold only one data of one data type at a time.
- An array can hold group of data of same data type.
- A structure can hold group of data of different data types
- Data types can be int, char, float, double and long double etc.

BELOW TABLE EXPLAINS FOLLOWING CONCEPTS IN C STRUCTURE.

Type	Using normal variable	Using pointer variabe

Syntax	struct tag_name
	{
	data type var_name1;
	data type var_name2;
	data type var_name3;
	};
		struct tag_name
		{
		data type var_name1;
		data type var_name2;
		data type var_name3;
		};

Example	struct student
	{
	int mark;
	char name[10];
	float average;
	};
		struct student
		{
		int mark;
		char name[10];
		float average;
		};

Declaring structure variable	struct student report;	struct student *report, rep;

Initializing structure variable	struct student report = {100, “Mani”, 99.5};	struct student rep = {100, “Mani”, 99.5};
		report = &rep;

Accessing structure members	report.mark report.name report.average	report -> mark report -> name report -> average

Defining structure

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

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

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

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

Declaring structure variable

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

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

1st way:

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

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

Now write given code inside the main() function.

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

2nd way:

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

```
struct employee
```

```
 {   int id;
     char name[50];
     float salary;
 }e1,e2;
```

Which approach is good

But if no. of variable are not fixed, use 1st approach. It provides you flexibility to declare the structure variable many times.

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

Declaration & Intialization

```
struct book
{
char name[10] ;
float price ;
int pages ;
} ;
int main()
{
struct book b1 = { "Basic", 130.00, 550 } ;
struct book b2 = { "Physics", 150.80, 800 } ;
printf("Book Name \t Book Price \t Book Pages\n");
printf("%s\t\t ",b1.name);
printf("%f\t ",b1.price);
printf("%d\n",b1.pages);
printf("%s\t\t ",b2.name);
printf("%f\t ",b2.price);
printf("%d\t ",b2.pages);
return 0;
}
```

Accessing members of structure

There are two ways to access structure members:

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

Example of structure

Write a C program to add two distances entered by user. Measurement of distance should be in inch and feet.(Note: 12 inches = 1 feett)

```
#include<stdio.h>
struct Distance{
```

```
int feet;
float inch;
}d1,d2,sum;
int main(){
   printf("1st distance\n");
   printf("Enter feet: ");
   scanf("%d",&d1.feet);/* input of feet for structure variable d1 */
   printf("Enter inch: ");
   scanf("%f",&d1.inch);/* input of inch for structure variable d1 */
   printf("2nd distance\n");
   printf("Enter feet: ");
   scanf("%d",&d2.feet);/* input of feet for structure variable d2 */
   printf("Enter inch: ");
   scanf("%f",&d2.inch);/* input of inch for structure variable d2 */
   sum.feet=d1.feet+d2.feet;
   sum.inch=d1.inch+d2.inch;
if(sum.inch>12){//If inch is greater than 12, changing it to feet.
++sum.feet;
      sum.inch=sum.inch-12;
}
   printf("Sum of distances=%d\'-%.1f\"",sum.feet,sum.inch);
/* printing sum of distance d1 and d2 */
return 0;
}
```

One structure can be nested within another structure.

```
main( )
{
struct address
{
char phone[15] ;
char city[25] ;
int pin ;
} ;
struct emp
{
char name[25] ;
struct address a ;
} ;
struct emp e = { "Yuva", "9880094433", "Mandya", 571401};
printf ( "\nname = %s phone = %s", e.name, e.a.phone ) ;
printf ( "\ncity = %s pin = %d", e.a.city, e.a.pin ) ;
}
```

C – Typedef

- Typedef is a keyword that is used to give a new symbolic name for the existing name in a C program. This is same like defining alias for the commands.

Keyword typedef while using structure

Programmer generally uses typedef while using structure in C language. For example:

```
typedef struct complex{
  int imag;
  float real;
}comp;
Inside main:
comp c1,c2;
```

Here, typedef keyword is used in creating a type *comp* (which is of type as **struct complex**). Then, two structure variables *c1* and *c2* are created by this *comp* type.

C Programming Structure and Pointer

Consider an example to access structure's member through pointer.

```
#include<stdio.h>
struct name{
int a;
float b;
};
int main(){
struct name *ptr,p;
    ptr=&p;/* Referencing pointer to memory address of p */
    printf("Enter integer: ");
    scanf("%d",&(*ptr).a);
    printf("Enter number: ");
    scanf("%f",&(*ptr).b);
    printf("Displaying: ");
    printf("%d%f",(*ptr).a,(*ptr).b);
return 0;
}
```

In this example, the pointer variable of type struct name is referenced to the address of p. Then, only the structure member through pointer can can accessed.

Structure pointer member can also be accessed using -> operator.

(*ptr).a is same as ptr->a

(*ptr).b is same as ptr->b

Assignment:
Write a program to Store Student Information and Display it Using Structure.

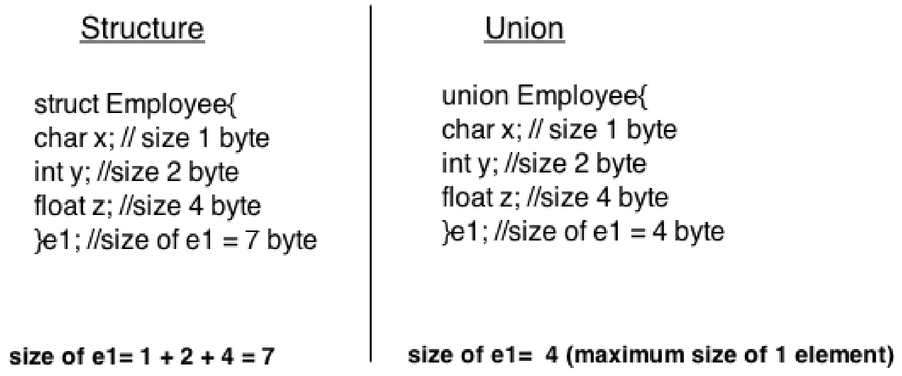
<u>Union</u>

C Union is also like structure, i.e. collection of different data types which are grouped together. Each element in a union is called member.

- Union and structure in C are same in concepts, except allocating memory for their members.
- Structure allocates storage space for all its members separately.
- Whereas, Union allocates one common storage space for all its members
- We can access only one member of union at a time. We can't access all member values at the same time in union. But, structure can access all member values at the same time. This is because, Union allocates one common storage space for all its members. Wher**e** as Structure allocates storage space for all its members separately.
- Many union variables can be created in a program and memory will be allocated for each union variable separately.

Like structure, **Union in c language** is *a user defined datatype* that is used to hold different type of elements.

But it doesn't occupy sum of all members size. It occupies the memory of largest member only. It shares memory of largest member.



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

size of e1= 1 + 2 + 4 = 7

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

size of e1= 4 (maximum size of 1 element)

Advantage of union over structure
It **occupies less memory** because it occupies the memory of largest member only.

Disadvantage of union over structure
It can **store data in one member only**.

Type	Using normal variable	Using pointer variable

Syntax	union tag_name
	{
	data type var_name1;
	data type var_name2;
	data type var_name3;
	};
		union tag_name
		{
		data type var_name1;
		data type var_name2;
		data type var_name3;
		};

Example	union student
	{
	int mark;
	char name[10];
	float average;
	};
		union student
		{
		int mark;
		char name[10];
		float average;
		};

Declaring union variable	union student report;	union student *report, rep;
Initializing union variable	union student report = {100, “Mani”, 99.5};	union student rep = {100, “Mani”, 99.5};report = &rep;
Accessing union members	report.mark report.name report.average	report -> mark report -> name report -> average

Only one union member can be accessed at a time

In the case of structure, all of its members can be accessed at any time.
But, in the case of union, only one of its members can be accessed at a time

```
#include <stdio.h>
union job
{
   char name[32];
   float salary;
   int workerNo;
} job1;

int main()
{
   printf("Enter name:\n");
   scanf("%s", &job1.name);

   printf("Enter salary: \n");
   scanf("%f", &job1.salary);

   printf("Displaying\nName :%s\n", job1.name);
   printf("Salary: %.1f", job1.salary);
   return 0;
}
```

EXAMPLE PROGRAM FOR C UNION:

```
#include <stdio.h>
#include <string.h>
union student
{
char name[20];
char subject[20];
float percentage;
};
int main()
{
union student record1;
union student record2;
// assigning values to record1 union variable
strcpy(record1.name, "Raju");
strcpy(record1.subject, "Maths");
record1.percentage = 86.50;
printf("Union record1 values example \n");
printf(" Name       : %s \n", record1.name);
printf(" Subject    : %s \n", record1.subject);
printf(" Percentage : %f \nn", record1.percentage);
// assigning values to record2 union variable
printf("Union record2 values example \n");
strcpy(record2.name, "Mani");
printf(" Name       : %s \n", record2.name);
strcpy(record2.subject, "Physics");
printf(" Subject    : %s \n", record2.subject);
record2.percentage = 99.50;
printf(" Percentage : %f \n", record2.percentage);
return 0;
}
```

DIFFERENCE BETWEEN STRUCTURE AND UNION IN C:

S.no	C Structure	C Union
1	Structure allocates storage space for all its members separately.	Union allocates one common storage space for all its members. Union finds that which of its member needs high storage space over other members and allocates that much space
2	Structure occupies higher memory space.	Union occupies lower memory space over structure.
3	We can access all members of structure at a time.	We can access only one member of union at a time.

4	Structure example:
	struct student
	{
	int mark;
	char name[6];
	double average;
	};
		Union example:
		union student
		{
		int mark;
		char name[6];
		double average;
		};

5	For above structure, memory allocation will be like below. int mark – 2B char name[6] – 6B double average – 8B Total memory allocation = 2+6+8 = 16 Bytes	For above union, only 8 bytes of memory will be allocated since double data type will occupy maximum space of memory over other data types. Total memory allocation = 8 Bytes

Difference between union and structure

Though unions are similar to structure in so many ways, the difference between them is crucial to understand. The primary difference can be demonstrated by this example:

```
#include <stdio.h>
union unionJob
{
   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", sizeof(uJob));
   printf("\nsize of structure = %d", sizeof(sJob));
   return 0;
}
```

17. Files

- In C programming, file is a place on disk where a group of related data is stored.
- File is a collection of bytes that is stored on secondary storage devices like disk.

Why files are needed?

When the program is terminated, the entire data is lost in C programming. If you want to keep large volume of data, it is time consuming to enter the entire data. But, if file is created, these information can be accessed using few commands.

File Handling in c language is *used to open, read, write, search or close file*. It is used for permanent storage.

Advantage of File

It *will contain the data even after program exit*. Normally we use variable or array to store data, but data is lost after program exit. Variables and arrays are non-permanent storage medium whereas file is permanent storage medium.

Functions for file handling

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

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

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 that you can easily create using Notepad or any simple text editors.
- 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 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 higher amount of data, are not readable easily and provides a better security than text files.
- If a large amount of numerical data it to be stored, text mode will be insufficient. In such case binary file is used.

Text File Operations

1. Creating a new file
2. Opening an existing file
3. Reading from and writing information to a file
4. Closing a file

Working with file

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

```
FILE *ptr;
```

Opening a file - for creation and edit

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

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.

Opening Modes in Standard I/O

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. i.e, Data is added to end of file.	If the file does not exists, it will be created.
Ab	Open for append in binary mode. i.e, Data is added to end of file.	If the file does not exists, 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 exists, it will be created.
ab+	Open for both reading and appending in binary mode.	If the file does not exists, 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 library function fclose().

fclose(fptr); //fptr is the file pointer associated with file to be closed.

Reading and writing to a text file

For reading and writing to a text file, we use the functions fprintf() and fscanf().

The Functions fprintf() and fscanf() functions.

Writing to a Text File : fprintf() function

The fprintf() function is used to write input into file.

```
#include <stdio.h>
int main()
{
  int n;
  FILE *fptr;
  fptr=fopen("E:\program.txt","w");
  if(fptr==NULL){
    printf("Error!");
    exit(1);
  }
  printf("Enter n: ");
  scanf("%d",&n);
  fprintf(fptr,"%d",n);
  fclose(fptr);
  return 0;
}
```

Reading from a Text File : fscanf() function

The fscanf() function is used to read value from file.

```
#include <stdio.h>
int main()
{
  int n;
  FILE *fptr;
  if ((fptr=fopen("program.txt","r"))==NULL){
      printf("Error! opening file");
      exit(1);         /* Program exits if file pointer returns NULL. */
  }
  fscanf(fptr,"%d",&n);
  printf("Value of n=%d",n);
  fclose(fptr);
  return 0;
}
```

Reading and writing to a binary file

Functions fread() and fwrite() are used for reading from and writing to a file on the disk respectively in case of binary files.

```
int main()
{
    int n;
    FILE *fptr;
    fptr=fopen("program.bin","wb");
    if(fptr==NULL){
      printf("Error!");
      exit(1);
    }
    printf("Enter n: ");
    scanf("%d",&n);
    fprintf(fptr,"%d",n);
    fclose(fptr);
      if ((fptr=fopen("program.bin","rb"))==NULL){
       printf("Error! opening file");
       exit(1);         /* Program exits if file pointer returns NULL. */
    }
    fscanf(fptr,"%d",&n);
    printf("Value of n=%d",n);
    fclose(fptr);
```

```
  return 0;
}
```

Example

The following example shows the usage of fseek() function.

```
#include <stdio.h>
int main ()
{
  FILE *fp;
fp = fopen("file.txt","w+");
  fputs("This is a File Program", fp);

  fseek( fp, 7, SEEK_SET );
  fputs(" C Programming Language", fp);
  fclose(fp);
  return(0);
}
```

The content of the above file using following program-

```
#include <stdio.h>
int main ()
{
  FILE *fp;
  int c;

  fp = fopen("file.txt","r");
  while(1)
  {
    c = fgetc(fp);
    if( feof(fp) )
    {
      break;
    }
    printf("%c", c);
  }
  fclose(fp);
  return(0);
}
```

C Program to Write a Sentence to a File

This program stores a sentence entered by user in a file.

```
#include <stdio.h>
#include <stdlib.h>  /* For exit() function */
int main()
{
   char sentence[1000];
   FILE *fptr;

   fptr = fopen("program.txt", "w");
   if(fptr == NULL)
   {
      printf("Error!");
      exit(1);
   }

   printf("Enter a sentence:\n");
   gets(sentence);

   fprintf(fptr,"%s", sentence);
   fclose(fptr);

   return 0;
}
```

C Program to Read a Line from a File and Display it

Example: Program to read text from a file

```
#include <stdio.h>
#include <stdlib.h> // For exit() function
int main()
{
    char c[1000];
    FILE *fptr;

    if ((fptr = fopen("program.txt", "r")) == NULL)
    {
        printf("Error! opening file");
        // Program exits if file pointer returns NULL.
        exit(1);
    }

    // reads text until newline
    fscanf(fptr,"%[^\n]", c);

    printf("Data from the file:\n%s", c);
    fclose(fptr);
    return 0;
}
```

Multiple Write

```
#include<stdio.h>
int main()
{
   FILE *fp1, *fp2;
   fp1=fopen("file.c", "w");
   fp2=fopen("file.c", "w");
   fputc('A', fp1);
   fputc('B', fp2);
   fclose(fp1);
   fclose(fp2);
   return 0;
}
```

Write a C program to read name and marks of n number of students from user and store them in a file

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

  char name[50];

  int marks,i,n;

  printf("Enter number of students: ");

  scanf("%d",&n);

  FILE *fptr;

  fptr=(fopen("student.txt","w"));

  if(fptr==NULL){

     printf("Error!");

     exit(1);

  }

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

  {

    printf("For student%d\nEnter name: ",i+1);

    scanf("%s",&name);

    printf("Enter marks: ");

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

```
     fprintf(fptr,"\nName: %s \nMarks=%d \n",name,marks);
  }
  fclose(fptr);
  fptr=(fopen("student.txt","r"));
  for(i=0;i<n;++i)
  {
      fscanf(fptr,"\nName: %s \nMarks=%d \n",&name,&marks);
     printf("\nDetails of student%d\n",i+1);
     printf("Name:%s\n",name);
     printf("Marks:%d",marks);
  }
  return 0;
}
```

Write a C program to write all the members of an array of strcures to a file using fwrite(). Read the array from the file and display on the screen.

```
struct s
{
char name[50];
int marks;
};
int main(){
   struct s a[10],b[10];
   FILE *fptr;
   int i,n;
   printf("Enter number of students: ");
  scanf("%d",&n);
   fptr=fopen("file.txt","w");
   for(i=0;i<n;++i)
   {
       printf("Enter name: ");
       scanf("%s", &a[i].name);
       printf("Enter marks: ");
       scanf("%d",&a[i].marks);
   }
   fwrite(a,sizeof(a),1,fptr);
   fclose(fptr);
   fptr=fopen("file.txt","r");
   fread(b,sizeof(b),1,fptr);
   for(i=0;i<n;++i)
   {
       printf("Name: %s\tmarks: %d\n",b[i].name,b[i].marks);
   }
   fclose(fptr);
}
```