PROGRAMMING IN “C”

Syllabus

1 Fundamentals of C Programming History of C programming language and its features

1.1 Algorithm & Flowchart : Three construct of Algorithm and flowchart: Sequence, Decision (Selection) and Repetition

1.2 Character Set, Identifiers and keywords, Data types, Constants, Variables.

1.3 Operators-Arithmetic, Relational and logical, Assignment, Unary, Conditional, Bitwise, Comma, other operators. Expression, statements, Preprocessor, Structure of basic C program.

2 Control Flow Statements

2.1 Decision making statements- if statement, if-else statement , ifelse-if ladder, nested if-else, switch statement

2.2 Looping – while , do-while, for

2.3 Jump Statements- break, continue, goto, return, exit

3 Functions

3.1 Introduction to Functions, declaring and defining function, calling function, passing arguments to a function, recursion and its application.

3.2 Library functions – getchar( ), putchar( ), gets( ), puts( ), Math function, Ctype functions

3.3 Storage classes in C-auto, extern, static, register.

4 Arrays and Strings

4.1 Array Introduction, Declaration, Initialization, Accessing array element, One and Two-dimensional array.

4.2 Strings Introduction, String using char array, String handling functions

5 Structures

5.1 Structure Introduction, Declaration, Initialization, operations on structure.

5.2 Nested structure, Array of Structure.

6 Pointers

6.1 Pointer: Introduction, Definition, Pointer Variables, Referencing and Dereferencing operator, Pointer Arithmetic, Pointers to Pointers, void Pointer,

6.2 Pointers to Array and Strings, Passing Arrays to Function, Accessing structure using pointers, Array of Pointers, call by value and call by reference.

6.3 Dynamic Memory Allocation using malloc, calloc, realloc, free

History of C programming language and its features

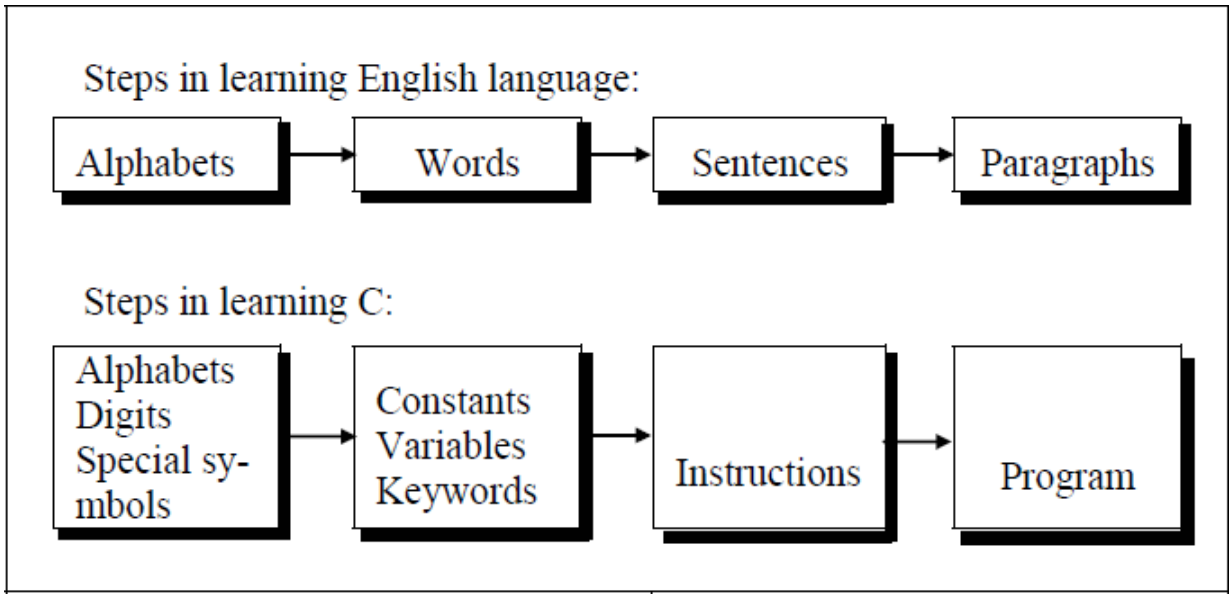
C is a programming language developed at AT & T's Bell Laboratories of USA in 1972. It was designed and written by a man named Dennis Ritchie. In the late seventies C began to replace the more familiar languages of that time like PL/I, ALGOL, etc.

ANSI C standard emerged in the early 1980s, this book was split into two titles: The original was still called ***Programming in C***, and the title that covered ANSI C was called ***Programming in ANSI C***. This was done because it took several years for the compiler vendors to release their ANSI C compilers and for them to become ubiquitous. It was initially designed for programming UNIX operating system. Now the software tool as well as the C compiler is written in C. Major parts of popular operating systems like Windows, UNIX, Linux is still written in C. This is because

even today when it comes to performance (speed of execution) nothing beats C. Moreover, if one is to extend the operating system towork with new devices one needs to write device driver programs. These programs are exclusively written in C. C seems so popular is because it is **reliable**,**simple** and **easy** to use

Program

There is a close analogy between learning English language and learning C language. The classical method of learning English is to first learn the alphabets used in the language, then learn to combine these alphabets to form words, which in turn are combined to form sentences and sentences are combined to form paragraphs. Learning C is similar and easier. Instead of straight-away learning how to write programs, we must first know what alphabets, numbers and special symbols are used in C, then how using them constants, variables and keywords areconstructed, and finally how are these combined to form an **instruction**. A group of instructions would be combined later on to form a **program**. So a computer ***program*** is just a collection of the instructions necessary to solve a specific problem. The basic operations of a computer system form what is known as the computer's ***instruction set.*** And the approach or method that is used to solvethe problem is known as an ***algorithm***.



Features of C language

1) Simple

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

2) Machine Independent or Portable

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

3) Mid-level programming language

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

4) Structured programming language

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

5) Rich Library

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

6) Memory Management

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

7) Speed

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

8) Pointer

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

9) Recursion

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

10) Extensible

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

1.2 Algorithm and Flowchart

- An algorithm is a procedure used for solving a problem.
- It a sequence of finite steps to solve a particular problem. Each step results in an action.
- Well-designed algorithm is guaranteed to terminate.

Conventions for Algorithm

- Each algorithm is enclosed by two statement START and STOP.
- INPUT / READ- accept data from user

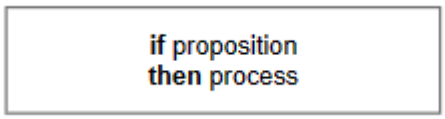
- PRINT- display message enclosed inn " " (quotes)
- <-- assignment operator
- arithmetic operator = +,-,*,/
- Relational Operators = >,<,>=,<=,!,=(equality)
- Logical Operator = AND, OR, NOT
- e.g x=2 AND y=0 , x=2 OR y=0 , NOT X=2

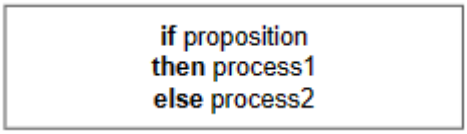
Three constructs of algorithm

1) Sequence - Each step or process in algorithm is executed in specified order .

Most programming languages simply execute instructions one after another

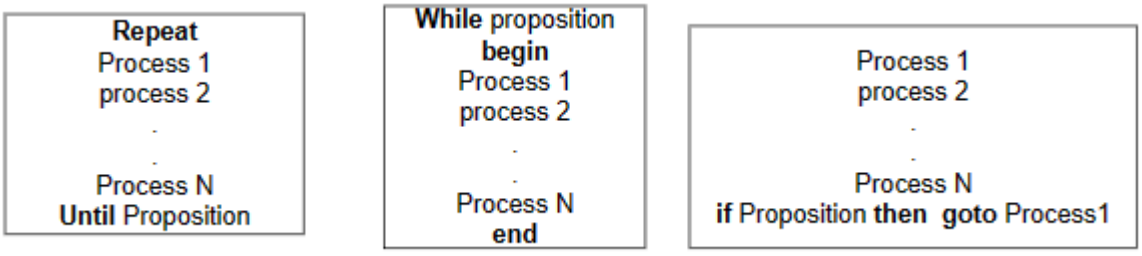
2) Selection / decision - if .... then, if.....then.....else





In an algorithm the algorithm is either true or false, there is no state in between. The decision can be stated as

3) Iteration / Repetition - Repeat, While, if.. then.. go



1. **Write the algorithm to find the sum and product of two given numbers.**

Step 1: START

Step 2: PRINT "ENTER TWO NUMBER"

Step3: READ A, B

Step 4: Sum<-- A+B

Step 5: Product<--A*B

Step 6: PRINT Sum, Product

Step 7: STOP

2. **Write the algorithm to check the given number by user odd or even.**

Step 1: START

Step 2: PRINT "ENTER THE NUMBER"

Step3: READ N

Step 4: R← N%2

Step 5: IF R=0 THEN

PRINT “N IS EVEN”

ELSE

PRINT “N IS ODD”

Step 6: STOP

3. **Write an algorithm for calculating the factorial of a given number**

   Step 1: START

   Step 2: PRINT "ENTER THE NUMBER"

   Step3: READ N

   Step 4: F←1

   Step 5: C←1

   Step 6: WHILE C<=N

   Step 7: BEGIN

   Step 8: F←F*C

   Step 9: C←C+1

   Step 10: END

   Step 11: PRINT F

   Step 12: STOP.

4. **Write an algorithm for computing the sum of digits in a number**

   Step 1: START

Step 2: PRINT "ENTER THE NUMBER"

Step3: READ N

Step 4: S←0

Step 5: Q←N/10

Step 6: R← N-Q*10

Step 7: S←S+R

Step 8: N←Q

Step 9: IF N>0 THEN GOTO 5

Step 10: PRINT S

Step 12: STOP.

5. **Write an algorithm for calculating the factorial of a given number**

Step 1: START

Step 2: PRINT "ENTER THE NUMBER"

Step3: READ N

Step 4: F←1

Step 5: C←1

Step 6: REPEAT STEP 7 THROUGH 8 UNTIL C<=N

Step 7: F←F*C

Step 8: C←C+1

Step 9: PRINT F

Step 10: STOP.

Flow-chart

(i) A flowchart is a diagrammatic representation of algorithm to plan the solution to the problem.

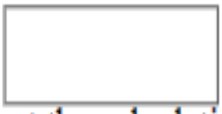
(ii) Constructed by using special geometrical symbols where each symbol represents an activity. The activity would be input/output of data, computation/processing of data etc.

Terminal symbol 

- It is used to represent the start, end of the program logic.

Input/Output 

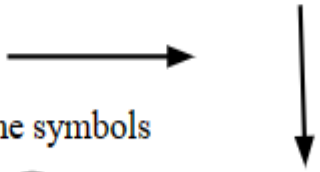
- It is used for input or output.

Process Symbol 

- It is used to represent the calculations, data movements, initialization operations etc.

Decision Symbol 

- It is used to denote a decision to be made at that point

Flow lines 

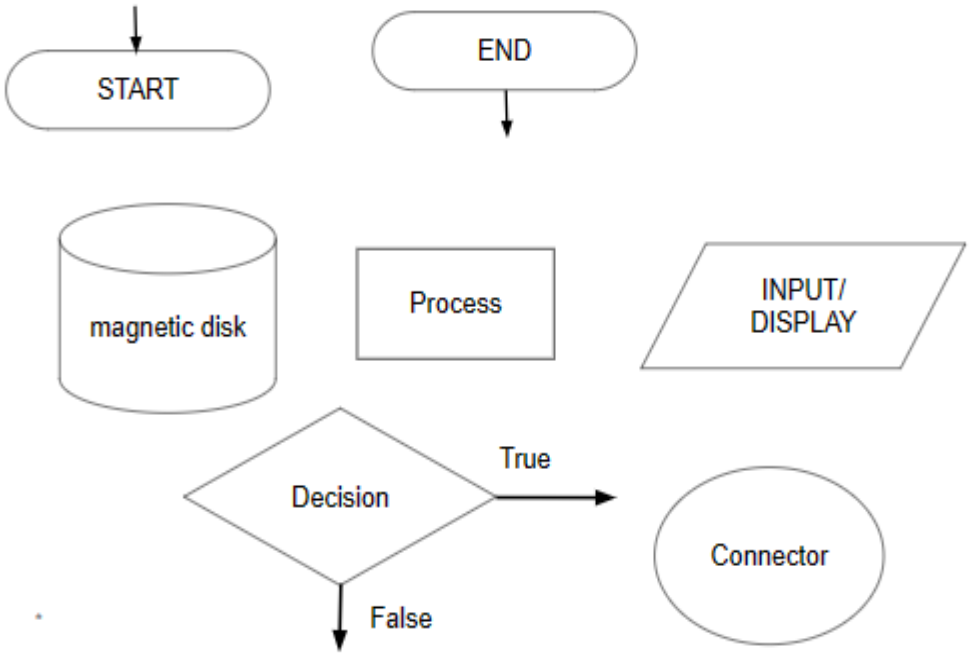
- It is used to connect the symbols

Connectors 

- It is used to connect the flow lines.

Loop 

- It is used to denote loops involved in the process.



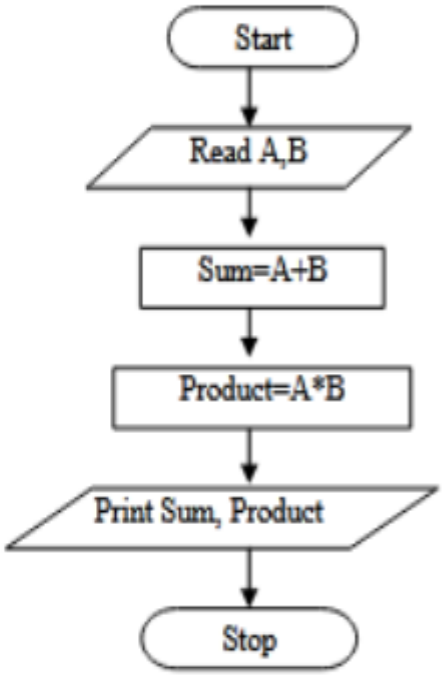
Advantages of using flowchart:

1. Communication: They are good visual aid to understand the program.
2. Quicker grasps of relationships.
3. Effective analysis.
4. Synthesis
5. Proper program documentation
6. Efficient coding
7. Orderly debugging & testing of programs.
8. Efficient program maintenance.

Limitations of using flowchart:

1. Complex logic
2. Alterations & Modifications
3. Reproduction
4. Loss of Objective

Flowchart: To find the sum and product of two given numbers:



Flowchart: To find the biggest number of three given numbers:

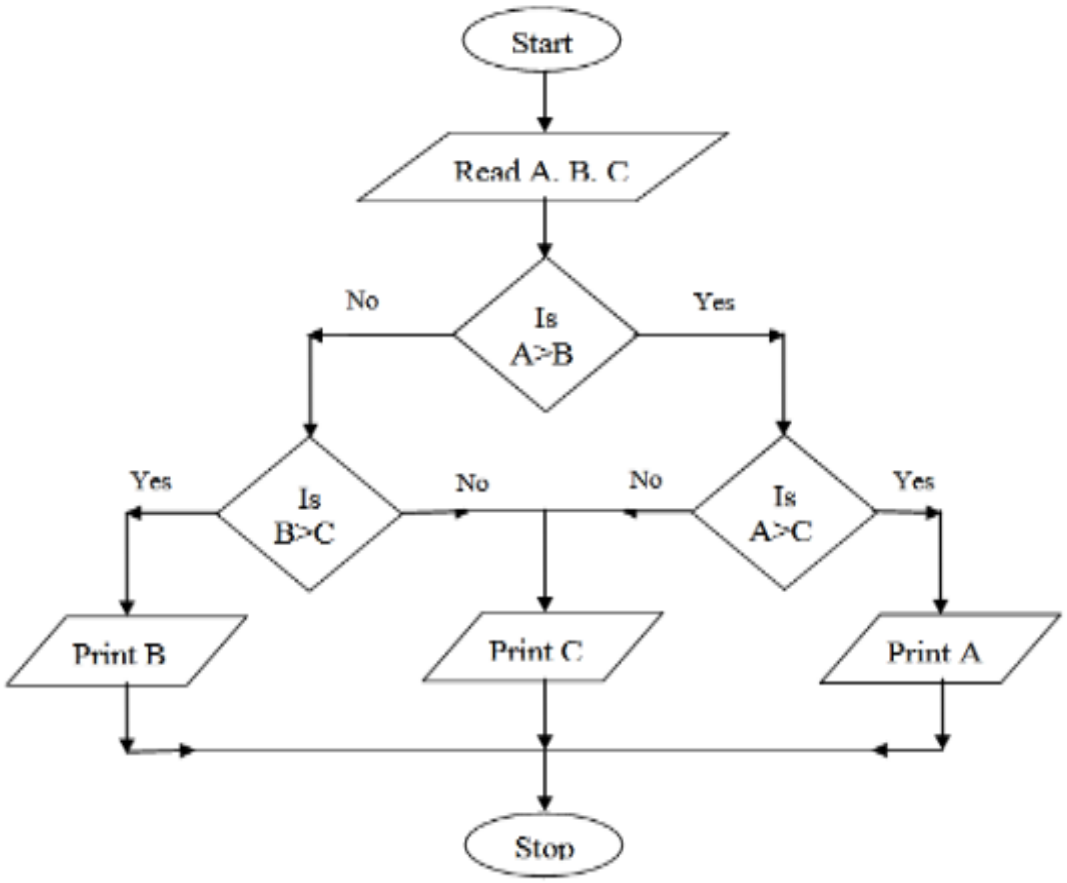
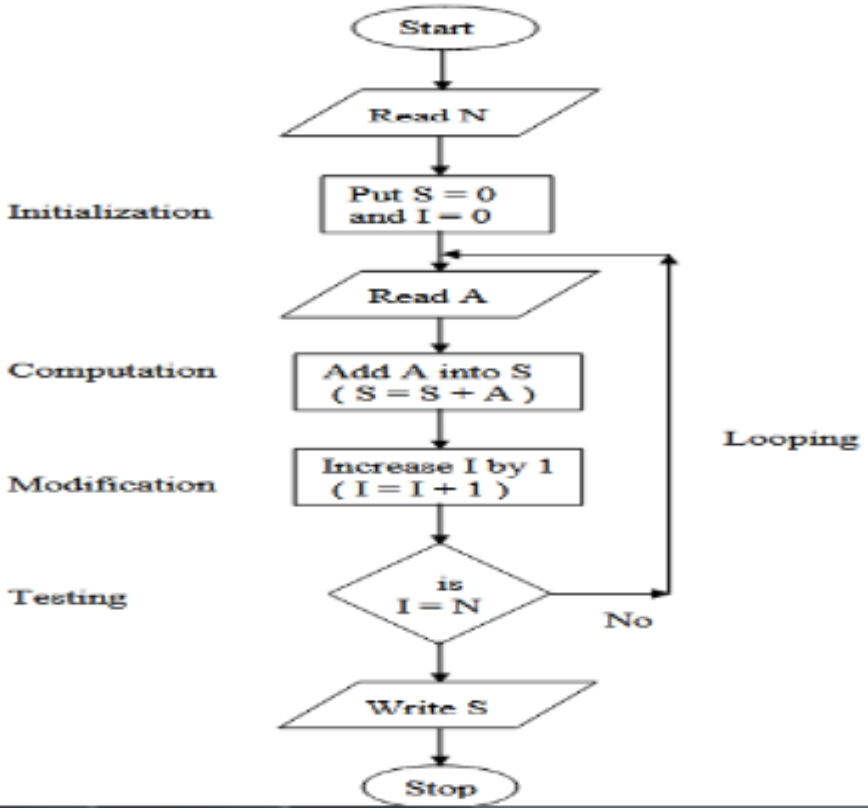


Fig: Summing of N given numbers.



Character set

A character denotes any alphabet, digit or special symbol used to represent information. Valid alphabets, numbers and special symbols allowed in C are

Alphabets	A, B, ….., Y, Z a, b, ……, y, z
Digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Special symbols	~ ‘ ! @ # % ^ & * ( ) _ - + = | \ { } [ ] : ; " ' < > , . ? /

The alphabets, numbers and special symbols when properly combined form constants, variables and keywords.

Identifiers

Identifiers are user defined word used to name of entities like variables, arrays, functions, structures etc. Rules for naming identifiers are:

1) Name should only consists of alphabets (both upper and lower case), digits and underscore (_) sign.

2) first characters should be alphabet or underscore

3) name should not be a keyword

4) Since C is a case sensitive, the upper case and lower case considered differently, for example code, Code, CODE etc. are different identifiers.

5) Identifiers are generally given in some meaningful name such as value, net_salary, age, data etc. An identifier name may be long, some implementation recognizes only first eight characters, most recognize 31 characters

Keyword

There are certain words reserved for doing specific task, these words are known as **reserved word** or **keywords.** These words are predefined and always written in lower case or small letter. These keywords can't be used as a variable name as it assigned with fixed meaning.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsign

			ed
contin ue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Variable

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C is case-sensitive.

Valid Names		Invalid Name	
a	// Valid but poor style	$sum	// $ is illegal
student_name		2names	// First char digit
_aSystemName		sum-salary	// Contains hyphen
_Bool	// Boolean System id	stdnt Nmbr	// Contains spaces
INT_MIN	// System Defined Value	int	// Keyword

Constants

- A constant is a value or an identifier whose value cannot be altered in a program.

e.g. 1, 2.5,

- As mentioned, an identifier also can be defined as a constant.
  e.g. const double PI = 3.14
- Here, PI is a constant. Basically what it means is that, PI and 3.14 is same for this program.

1. Integer constants

- A integer constant is a numeric constant (associated with number) without any fractional or exponential part. There are three types of integer constants in C programming:
  - decimal constant(base 10)
  - octal constant(base 8)
  - hexadecimal constant(base 16)

2. **A floating point constant** is a numeric constant that has either a fractional form or an exponent form. For example: 2.0,0.0000234,-0.22E-5

3. Character constants

A character constant is a constant which uses single quotation around characters. For example: 'a', 'l', 'm', 'F'

4. String constants

String constants are the constants which are enclosed in a pair of double-quote marks. For example: "good" ,"x","Earth is round\n"

Escape Sequence in C

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

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

  List of Escape Sequences in C

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

1. **Example of Escape Sequence using '\n'**

```
#include<stdio.h>
int main()
{
printf("TechVidvan\n");
printf("Tutorial\n");
printf("Escape Sequences\n");
return 0;
}
```

2. **Horizontal Tab('\t') in C**

```
#include <stdio.h>
int main ()
{
printf("TechVidvan Tutorial: Horizontal Tab!\n");
printf("45\t693\t145\t526");
return 0;
}
```

3. **Backspace('\b') in C**

Word which precedes the '\b' will be removed.

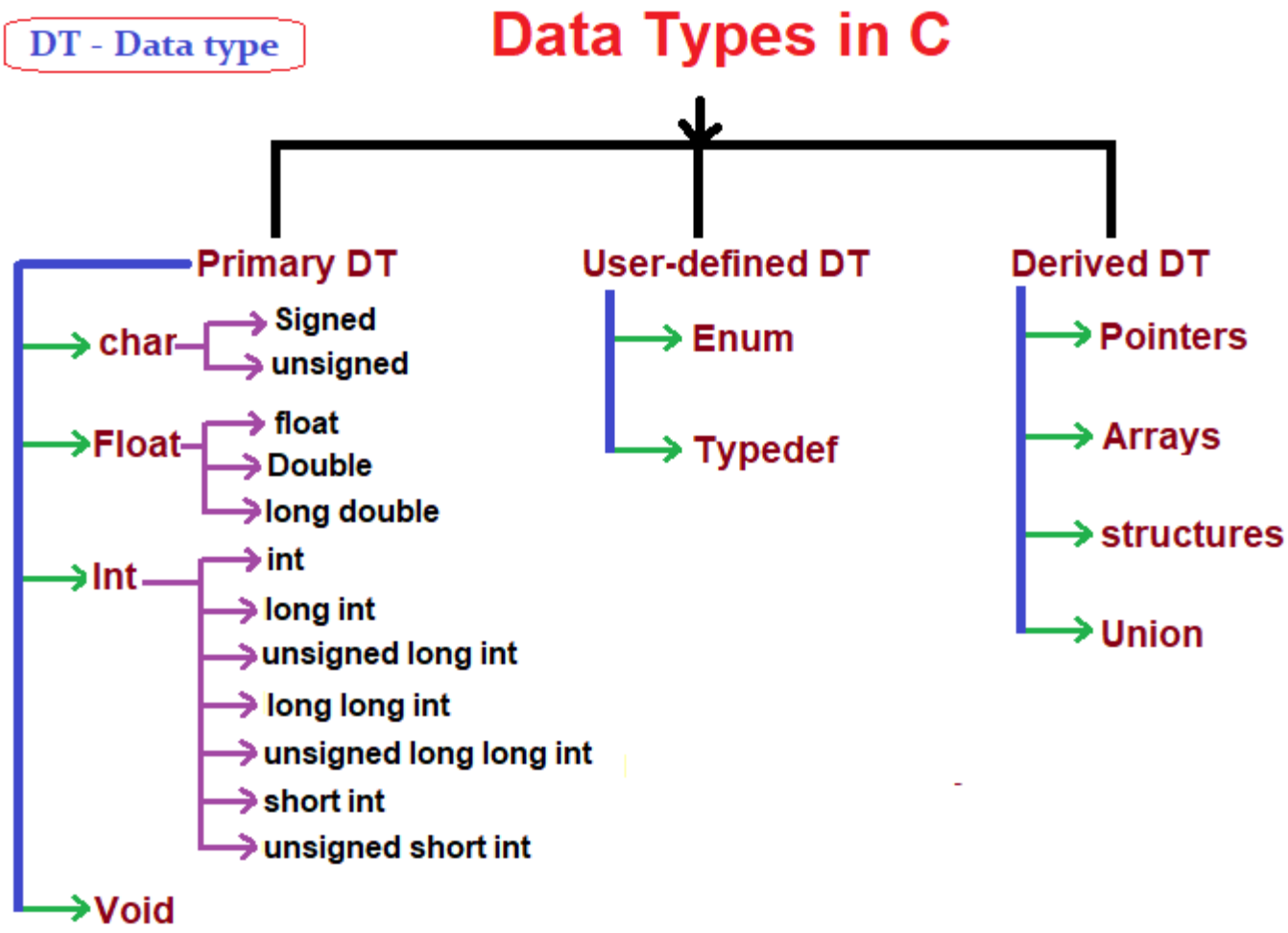
```
#include <stdio.h>
int main ()
{
printf("TechVidvan Tutoriall\b: Backspace!\n");
return 0;
}
```

Output:-

```
TechVidvan Tutorial: Backspace!
```

Data types

Data types refer to an extensive system used for declaring variables or functions of different types before its use. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted. The value of a variable can be changed any time.



C has the following 4 types of data types

Basic built-in data types: int, float, double, char

User defined : enum, typedef

Derived data type: pointer, array, structure, union

Void data type: void

Declaration of Variable

A variable declared to be of type int can be used to contain integral values only—that is, values that do not contain decimal places. A variable declared to be of type float can be used for storing floating- point numbers (values containing decimal places). The double type is the same as type float, only with roughly twice the precision. The char data type can be used to store a single character, such as the letter *a*, the digit character *6*, or a semicolon similarly A variable declared char canonly store character type value.

- It tells compiler what the variable name is.
- It specifies what type of data the variable will hold.
- Declaration of variables must be done before they are used in the program.
- Declaration of variable without initializing any value to it

  data_type variable_name;

  e.g:- char Final_Grade; // Final_Grade is a variable of type char, and no value is assigned to it.
- Declaration of variable with initializing some value to it

  data_type variable_name = val;

  e.g:- int age = 22; // age is a variable of type int and holds the value 22.
- Here, data_type specifies the type of variable like int, char, etc.
- variable_name specifies the name of the variable. val is the value for which we are initializing the variable.

User Defined Data Types – typedef

- typedef declaration do not introduce new type but introduces new name or creating synonym (or alias) for existing type.
- To construct shorter or more meaningful names for types already defined by the language or for types that you have declared.

- A typedef declaration does not reserve storage.
- The name space for a typedef name is the same as other ordinary identifiers.
- Therefore, a program can have a typedef name and a local-scope identifier by the same name.
- The syntax is,

  typedef type-declaration the_synonym

- You cannot use the typedef specifier inside a function definition.
- When declaring a local-scope identifier by the same name as a typedef, or when declaring a member of a structure or union in the same scope or in an inner scope, the type specifier must be specified.

  e.g.

  typedef float TestType;

User Defined Data Types – enum

enum is another user-defined type consisting of a set of named constants called enumerators.

Using a keyword enum, it is a set of integer constants represented by identifiers.

The syntax is shown below, ([is optional])

// for definition of enumerated type

enum [tag]

{

enum-list

}

[declarator];

And

// for declaration of variable of type tag

enum tag declarator;

These enumeration constants are, in effect, symbolic constants whose values can be set automatically.

The values in an enum start with 0, unless specified otherwise, and are incremented by 1. For example, the following enumeration,

enum days {Mon, Tue, Wed, Thu, Fri, Sat, Sun};

Creates a new data type, enum days, in which the identifiers are set automatically to the integers 0 to 6.

To number the days 1 to 7, use the following enumeration,

enum days {Mon = 1, Tue, Wed, Thu, Fri, Sat, Sun};

Or we can re-arrange the order,

enum days {Mon, Tue, Wed, Thu = 7, Fri, Sat, Sun};

These enumeration constants are, in effect, symbolic constants whose values can be set automatically.

Data type	Size (in bytes)	Range	Format Specifier
short int	2	-32,768 to 32,767	%hd
unsigned short int	2	0 to 65,535	%hu
unsigned int	4	0 to 4,294,967,295	%u
int	4	-2,147,483,648 to 2,147,483,647	%d
long int	4	-2,147,483,648 to 2,147,483,647	%ld
unsigned long int	4	0 to 4,294,967,295	%lu
long long int	8	-(2^63) to (2^63)-1	%lld
unisgned long long int	8	0 to 18,446,744,073,709,551,615	%llu
signed char	1	-128 to +127	%c
unsigned char	1	0 to 255	%c
float	4		%f
double	8		%lf
long double	16		%LF

Operators in C

C programming has wide range of operators to perform various operations. For better understanding of operators, these operators can be classified as:

Arithmetic Operators

Increment and Decrement Operators

Assignment Operators

Relational Operators

Logical Operators

Conditional Operators

Bitwise Operators

Special Operators

Arithmetic Operators

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

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

Example 1: 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;
}
```

Increment and Decrement Operators

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

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

Example 2: Increment and Decrement Operators

```
// Working of increment and decrement operators
#include <stdio.h>
int main()
{
    int a = 10, b = 100;
    float c = 10.5, d = 100.5;

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

```
    return 0;
}
```

Run Code

Output

```
++a = 11
--b = 99
++c = 11.500000
--d = 99.500000
```

Here, the operators `++` and `--` are used as prefixes. These two operators can also be used as postfixes like `a++` and `a--`.

Assignment Operators

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

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

Operator	Example	Same as
/=	a /= b	a = a/b
%=	a %= b	a = a%b

Example 3: Assignment Operators

```
// Working of assignment operators
#include <stdio.h>
int main()
{
    int a = 5, c;

    c = a;      // c is 5
    printf("c = %d\n", c);
    c += a;     // c is 10
    printf("c = %d\n", c);
    c -= a;     // c is 5
    printf("c = %d\n", c);
    c *= a;     // c is 25
    printf("c = %d\n", c);
    c /= a;     // c is 5
    printf("c = %d\n", c);
    c %= a;     // c = 0
    printf("c = %d\n", c);

    return 0;
}
```

Output

```
c = 5
c = 10
c = 5
c = 25
c = 5
c = 0
```

C Relational Operators

A relational operator checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0.Relational operators are used in decision making and loops.

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

Example 4: Relational Operators

```
// Working of relational operators
#include <stdio.h>
```

```
int main()
{
   int a = 5, b = 5, c = 10;
   printf("%d == %d is %d \n", a, b, a == b);
   printf("%d == %d is %d \n", a, c, a == c);
   printf("%d > %d is %d \n", a, b, a > b);
   printf("%d > %d is %d \n", a, c, a > c);
   printf("%d < %d is %d \n", a, b, a < b);
   printf("%d < %d is %d \n", a, c, a < c);
   printf("%d != %d is %d \n", a, b, a != b);
   printf("%d != %d is %d \n", a, c, a != c);
   printf("%d >= %d is %d \n", a, b, a >= b);
   printf("%d >= %d is %d \n", a, c, a >= c);
   printf("%d <= %d is %d \n", a, b, a <= b);
   printf("%d <= %d is %d \n", a, c, a <= c);

   return 0;
}R
```

Output

```
5 == 5 is 1
5 == 10 is 0
5 > 5 is 0
5 > 10 is 0
5 < 5 is 0
5 < 10 is 1
5 != 5 is 0
5 != 10 is 1
```

```
5 >= 5 is 1
5 >= 10 is 0
5 <= 5 is 1
5 <= 10 is 1
```

C Logical Operators

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

Operator	Meaning	Example
&&	Logical AND. True only if all operands are true	If c = 5 and d = 2 then, expression ((c==5) && (d>5)) equals to 0.
||	Logical OR. True only if either one operand is true	If c = 5 and d = 2 then, expression ((c==5) || (d>5)) equals to 1.
!	Logical NOT. True only if the operand is 0	If c = 5 then, expression! (c==5) equals to 0.

Example 5: Logical Operators

```c
#include <stdio.h>
int main()
{
    int a = 5, b = 5, c = 10, result;

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

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

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

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

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

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

    return 0;
```

```
}
```

Output

```
(a == b) && (c > b) is 1
(a == b) && (c < b) is 0
(a == b) || (c < b) is 1
(a != b) || (c < b) is 0
!(a != b) is 1
!(a == b) is 0
```

Explanation of logical operator program

- `(a == b) && (c > 5)` evaluates to 1 because both operands `(a == b)` and `(c > b)` is 1 (true).
- `(a == b) && (c < b)` evaluates to 0 because operand `(c < b)` is 0 (false).
- `(a == b) || (c < b)` evaluates to 1 because `(a = b)` is 1 (true).
- `(a != b) || (c < b)` evaluates to 0 because both operand `(a != b)` and `(c < b)` are 0 (false).
- `!(a != b)` evaluates to 1 because operand `(a != b)` is 0 (false). Hence, !(a != b) is 1 (true).
- `!(a == b)` evaluates to 0 because `(a == b)` is 1 (true). Hence, `!(a == b)` is 0 (false).

C Bitwise Operators

During computation, mathematical operations like: addition, subtraction, multiplication, division, etc are converted to bit-level which makes processing faster and saves power.Bitwise operators are used in C programming to perform bit-level operations.

Operators	Meaning of operators
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise complement
<<	Shift left
>>	Shift right

Comma Operator

Comma operators are used to link related expressions together. For example:

```
Value=(x=10,y=5,x+y)
```

First assign 10 to x then assign 5 to y and finally assign 14 to value.

Parenthesis are necessary.

The sizeof operator

The `sizeof` is a unary operator that returns the size of data (constants, variables, array, structure, etc).

Example 6: sizeof Operator

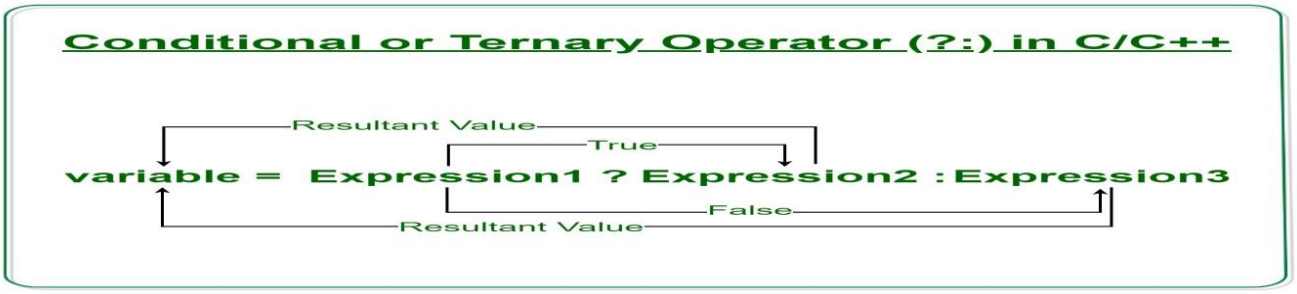
```
#include <stdio.h>
int main()
{
   int a;
   float b;
   double c;
   char d;
   printf("Size of int=%lu bytes\n",sizeof(a));
   printf("Size of float=%lu bytes\n",sizeof(b));
   printf("Size of double=%lu bytes\n",sizeof(c));
   printf("Size of char=%lu byte\n",sizeof(d));

   return 0;
}
```

Output

```
Size of int = 4 bytes
Size of float = 4 bytes
Size of double = 8 bytes
Size of char = 1 byte
```

Conditional or Ternary Operator (?:) in C

The conditional operator is kind of similar to the if-else statement as it does follow the same algorithm as of if-else statement but the conditional operator takes less space and helps to write the if-else statements in the shortest way possible.



Syntax:

The conditional operator is of the form

Variable = Expression1? Expression2: Expression3

Or syntax will also be in this form

(Condition)? (Variable = Expression2): (variable = Expression3)

It can be visualized into if-else statement as:

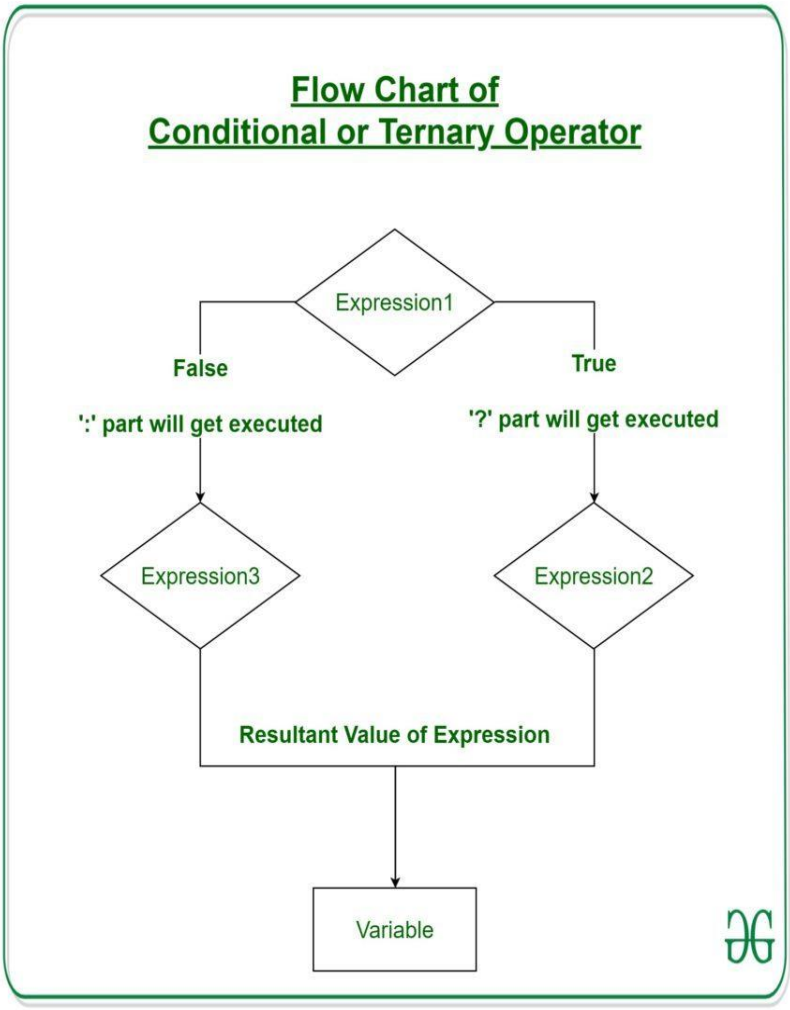
```
if (Expression1)
{
   Variable = Expression2;
}
else
{
```

```
    Variable = Expression3;
}
```

Since the Conditional Operator ‘?:’ takes three operands to work, hence they are also called ternary operators.

Working:

Here, Expression1 is the condition to be evaluated. If the condition(Expression1) is True then Expression2 will be executed and the result will be returned. Otherwise, if the condition(Expression1) is false then Expression3 will be executed and the result will be returned.




```
#include <stdio.h>
 int main()
{
    int m = 5, n = 4;

    (m > n) ? printf("m is greater than n that is %d > %d",
                m, n)
          : printf("n is greater than m that is %d > %d",
                n, m);
```

```
    return 0;
}
```

Evaluation of an expression in C is very important.

Consider the following arithmetic operation:

- left to right

6 / 2 * 1 + 2 = 5

- right to left

6/2 * 1 + 2 = 1

- using parentheses

= 6 / (2 * 1) + 2

= (6 / 2) + 2

= 3 + 2

= 5

Precedence of operator

Precedence	Operator	Description	Associativity
1	`::`	Scope resolution	Left-to-right
2	`++` `--`	Suffix/postfix increment and decrement	
	`()`	Function call	
	`[]`	Array subscripting	
	`.`	Element selection by reference	
	`->`	Element selection through pointer	
3	`++` `--`	Prefix increment and decrement	Right-to-left
	`+` `-`	Unary plus and minus	
	`!` `~`	Logical NOT and bitwise NOT	
	`(`*type*`)`	Type cast	
	`*`	Indirection (dereference)	
	`&`	Address-of	
	`sizeof`	Size-of	
	`new`, `new[]`	Dynamic memory allocation	
	`delete`, `delete[]`	Dynamic memory deallocation	
4	`.*` `->*`	Pointer to member	Left-to-right
5	`*` `/` `%`	Multiplication, division, and remainder	
6	`+` `-`	Addition and subtraction	
7	`<<` `>>`	Bitwise left shift and right shift	
8	`<` `<=`	For relational operators < and ≤ respectively	
	`>` `>=`	For relational operators > and ≥ respectively	
9	`==` `!=`	For relational = and ≠ respectively	
10	`&`	Bitwise AND	
11	`^`	Bitwise XOR (exclusive or)	
12	`|`	Bitwise OR (inclusive or)	
13	`&&`	Logical AND	
14	`||`	Logical OR	
15	`?:`	Ternary conditional	Right-to-left
	`=`	Direct assignment (provided by default for C++ classes)	
	`+=` `-=`	Assignment by sum and difference	
	`*=` `/=` `%=`	Assignment by product, quotient, and remainder	
	`<<=` `>>=`	Assignment by bitwise left shift and right shift	
	`&=` `^=` `|=`	Assignment by bitwise AND, XOR, and OR	
16	`throw`	Throw operator (for exceptions)	
17	`,`	Comma	Left-to-right

Structure of C Language program

1. Comment line
2. Preprocessor directive
3. Global variable declaration
4. main function( )

```
{

        Local variables;
        Statements;
}
        User defined function
{
}
```

Comment line

It indicates the purpose of the program. It is represented as

/*……..*/

Comment line is used for increasing the readability of the program. It is useful in explaining the program and generally used for documentation. It is enclosed within the decimeters. Comment line can be single or multiple line but should not be nested. It can be anywhere in the program except inside string constant & characterconstant.

Preprocessor Directive:

#include<stdio.h> tells the compiler to include information about the standard input/output library. It is also used in symbolic constant such as #define PI 3.14(value). The stdio.h (standard input output header file) contains definition &declaration of

system defined function such as printf( ), scanf( ), pow( ) etc. Generally printf() function used to display and scanf() function used to read value

Global Declaration:

This is the section where variable are declared globally so that it can be access byall the functions used in the program. And it is generally declared outside the function:

main()

It is the user defined function and every function has one main() function from where actually program is started and it is encloses within the pair of curly braces.

The main( ) function can be anywhere in the program but in general practice it is placed in the first position.

Syntax :main()

```
{
……..
……..
……..
}
```

The main( ) function return value when it declared by data type as int

```
main( )
{
return 0;
}
```

The main function does not return any value when void (means null/empty) as

```
void  main(void ) or  void main()
{
printf (“C language”);
}
```

Output: C language

The program execution start with opening braces and end with closing brace.

And in between the two braces declaration part as well as executable part is mentioned.

And at the end of each line, the semi-colon is given which indicates statement termination.

```
/*First  c program with  return  statement*/
#include  <stdio.h>  int
main (void)
{
printf ("welcome to c Programming language.\n");return 0;
}
```

Output: welcome to c programming language.

```
#include  <stdio.h>  int
main (void)
{
int  v1, v2, sum;                    //v1,v2,sum are variables and int is data type declared
v1 = 150;
v2 = 25;
sum = v1 + v2;
```

```
printf ("The sum of %i and %i is= %i\n", v1, v2, sum);
return 0;
}
```

Output:

The sum of 150 and 25 is=17

Control Flow Statements

2.1 Decision making statements- if statement, if-else statement, if else-if ladder, nested if-else, switch statement

2.2 Looping – while, do-while, for

2.3 Jump Statements- break, continue, goto, return, exit

Control Flow Statement

Generally C program statement is executed in a order in which they appear in the program. But sometimes we use decision making condition for execution only a part of program that is called control statement. Control statement defined how the control is transferred from one part to the other part of the program. Thereare several control statement like if...else, switch, while, do. ..while, for loop, break, continue, goto etc.

if statement

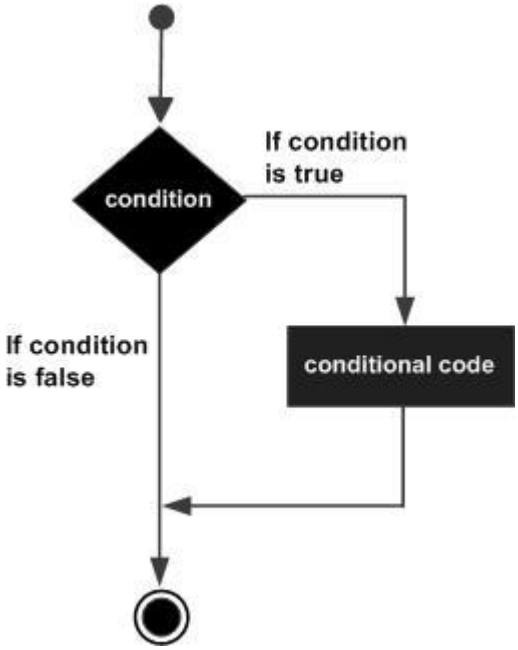
Statement execute set of command like when condition is true and its syntax is

```
if (condition)
{
Statement;
}
```

The statement is executed only when condition is true. If the if statement body is

consists of several statement then better to use pair of curly braces. Here in case condition is false then compiler skip the line within the if block.

Flow chart




```
void main()
{
int n;
printf (" enter a number:");
scanf("%d",&n);
If (n>10)
printf(" number is grater");
}
```

Output:

Enter a number:12

Number is greater

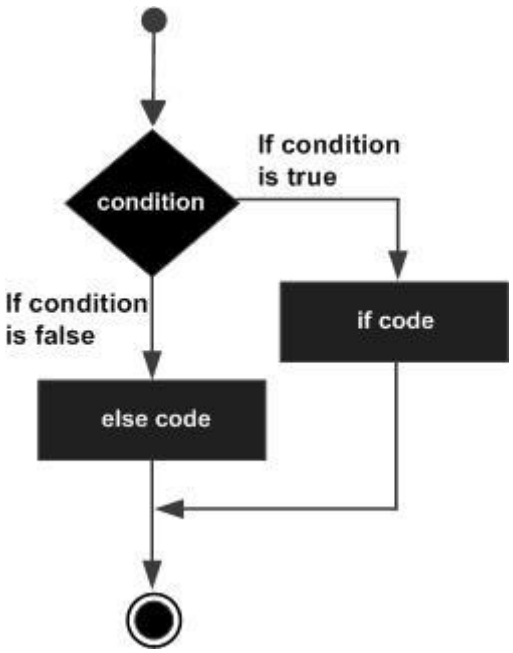
if…..else ... Statement

it is bidirectional conditional control statement that contains one condition & two possible action. Condition may be true or false, where non-zero value regarded astrue & zero value regarded as false. If condition are satisfy true, then a single or block of statement executed otherwise another single or block of statement is executed.

Its syntax is:-

```
if (condition)
{
Statement1;Statement2;
}
else
{
Statement1;Statement2;
}
```

Flow Chart



Else statement cannot be used without if or no multiple else statement are allowed within one if statement. It means there must be an if statement with in an else

statement.

Example:-

```
/* To check a number is eve or odd */
void main()
{

    int n;
   printf (“enter a number:”);
   scanf (“%d”, &n);
  if (n%2==0)
  printf (“even number”);
  else
  printf(“odd number”);

}
```

Output: enter a number:121

odd number

Nesting of if …else

When there are another if else statement in if-block or else-block, then it is called nesting of if-else statement.

Syntax is :-

```
if (condition)
{
        if (condition)

        Statement1;
        else
         Statement2;
         Statement3;

}
```

else if LADDER

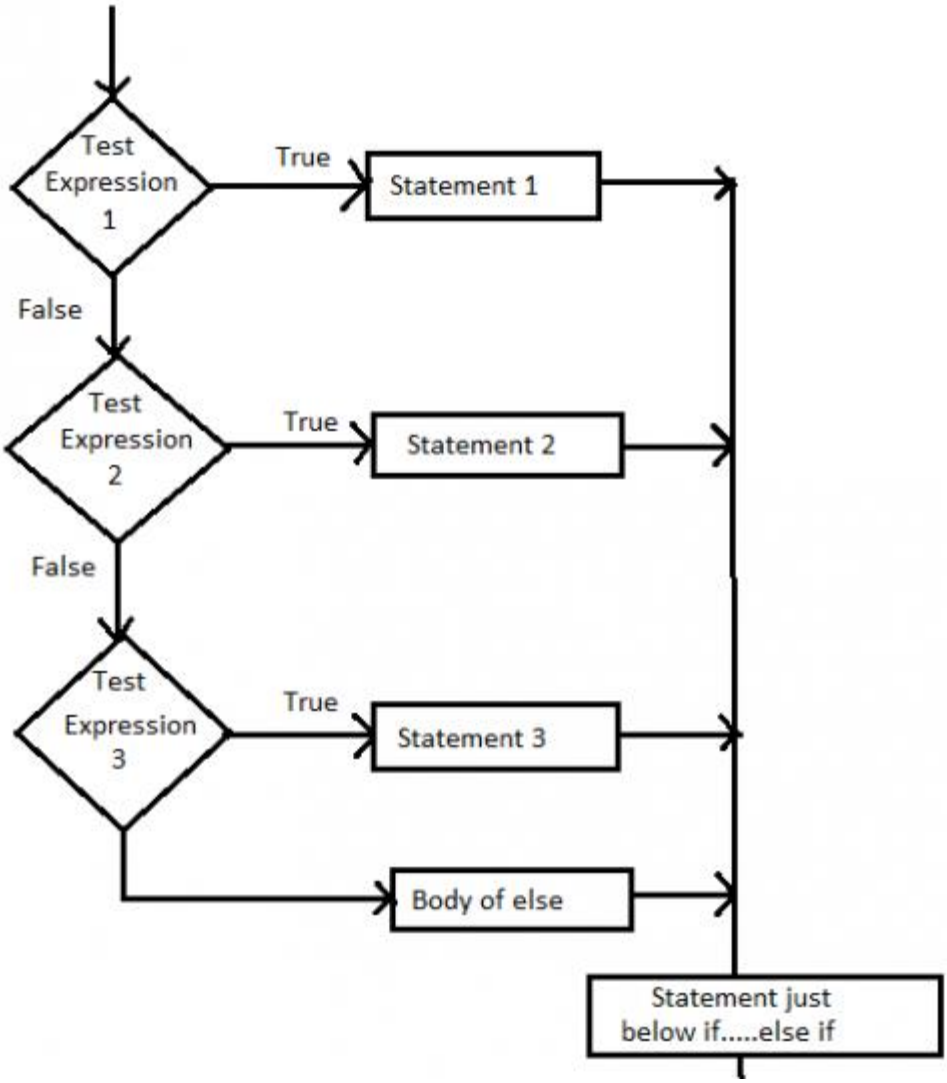
In this type of nesting there is an if else statement in every else part except the lastpart. If condition is false control pass to block where condition is again checked with its if statement.

Syntax is:-

```
if (condition)
        Statement1;
else if (condition)
```

```
        statement2;
else if (condition)
        statement3;
else
        statement4;
```

Flow chart



This process continue until there is no if statement in the last block. if one of the condition is satisfy the condition other nested "else if" would not executed.

But it has disadvantage over if else statement that, in if else statement whenever the condition is true, other condition are not checked. While in this case, all condition are checked.

```
#include<stdio.h>
void main (){
  int a,b,c,d;
  printf("Enter the values of a,b,c,d: ");
  scanf("%d%d%d%d",&a,&b,&c,&d);
  if(a>b && a>c && a>d){
    printf("%d is the largest",a);
  }else if(b>c && b>a && b>d){
    printf("%d is the largest",b);
  }else if(c>d && c>a && c>b){
    printf("%d is the largest",c);
  }else{
    printf("%d is the largest",d);
  }
}
```

Output

Run 1:Enter the values of a,b,c,d: 2 4 6 8

8 is the largest

Run 2: Enter the values of a,b,c,d: 23 12 56 23

56 is the largest

Switch statement

The switch statement allows us to execute one code block among many alternatives.

You can do the same thing with the if...else..if ladder. However, the syntax of the switch statement is much easier to read and write.

Syntax of switch...case

*

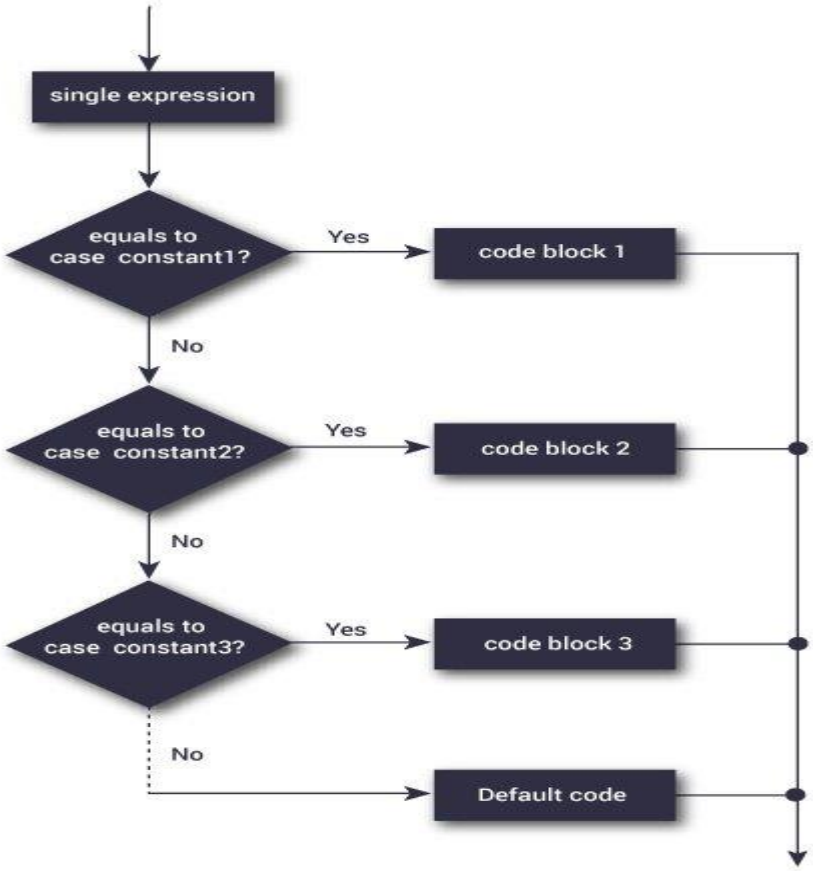
```
switch (expression)
{
   case constant1:
     // statements
     break;
   case constant2:
     // statements
     break;
   .
   .
   .
   default:
     // default statements
       }
```

How does the switch statement work?

- The `expression` is evaluated once and compared with the values of each `case` label.
- If there is a match, the corresponding statements after the matching label are executed. For example, if the value of the expression is equal to `constant2`, statements after `case constant2:` are executed until `break` is encountered.
- If there is no match, the default statements are executed.

switch Statement Flowchart

*



Example: Simple Calculator

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

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

    printf("Enter an operator (+, -, *, /): ");
    scanf("%c", &operation);
    printf("Enter two operands: ");
    scanf("%lf %lf",&n1, &n2);

    switch(operation)
    {
```

*

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

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

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

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

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

    return 0;
}
```

Run Code

Output

```
Enter an operator (+, -, *, /): -
Enter two operands: 32.5
12.4
32.5 - 12.4 = 20.1
```

*

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

}
```

Output

```
hi
```

- Finally, the break statement terminates the `switch` statement.
- If we do not use the `break` statement, all statements after the matching label are also executed.
- The `default` clause inside the `switch` statement is optional.

*

Differences b/w if-else and switch statement :

	If-else	Switch
Definition	Depending on the condition in the 'if' statement, 'if' and 'else' blocks are executed.	The user will decide which statement is to be executed.
Expression	It contains either logical or equality expression.	It contains a single expression which can be either a character or integer variable.
Evaluation	It evaluates all types of data, such as integer, floating-point, character or Boolean.	It evaluates either an integer, or character.
Sequence of execution	First, the condition is checked. If the condition is true then 'if' block is executed otherwise 'else' block	It executes one case after another till the break keyword is not found, or the default statement is executed.
Default execution	If the condition is not true, then by default, else block will be executed.	If the value does not match with any case, then by default, default statement is executed.
Editing	Editing is not easy in the 'if-else' statement.	Cases in a switch statement are easy to maintain and modify. Therefore, we can say that the removal or editing of any case will not interrupt the execution of other cases.
Speed	If there are multiple choices implemented through 'if-else', then the speed of the execution will be slow.	If we have multiple choices then the switch statement is the best option as the speed of the execution will be much higher than 'if-else'.

Loops in C

Loop:-it is a block of statement that performs set of instructions. In loops

Repeating particular portion of the program either a specified number of time oruntil a particular no of condition is being satisfied.

There are three types of loops in c

1. While loop

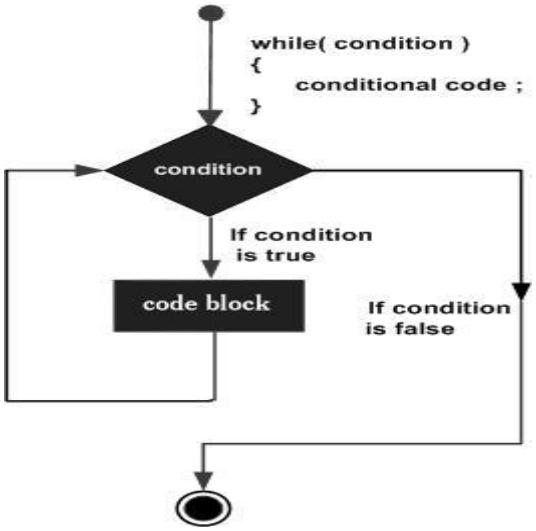
2. do while loop

3.for loop

*

While loop syntax

```
while(condition)
        {
                statement(s);
        }
```



The test condition may be any expression .when we want to do something a fixedno of times but not known about the number of iteration, in a program then whileloop is used.

Here first condition is checked if,it is true body of the loop is executed else, If condition is false control will be come out of loop.

Example:-

```
/* wap to print 5 times welcome to C" */
#include<stdio.h>
void main()
{
int p=1;
While(p<=5)
{
printf("Welcome to C\n");
p=p+1;
}
```

Output:

Welcome to C

Welcome to C

Welcome to C

Welcome to C

Welcome to C

So as long as condition remains true statements within the body of while loop will get executed repeatedly.

do while loop

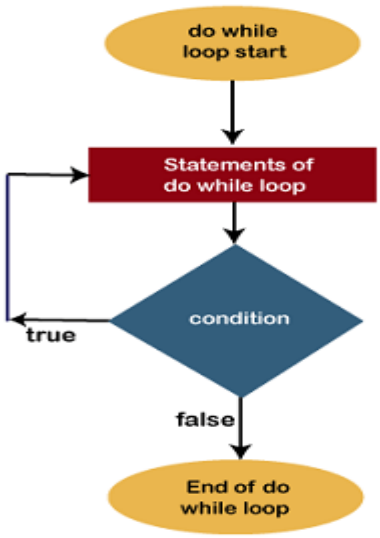
This (do while loop) statement is also used for looping. The body of this loop may contain single statement or block of statement. The syntax for writing this statement is:

Syntax:-

```
do
{
Statement;
}
while(condition);
```

Flow chart

*




```
Example:- // Print numbers from 1 to 5

#include <stdio.h>
int main() {
  int i = 1;

  while (i <= 5) {
    printf("%d\n", i);
    ++i;
  }

  return 0;
}
```

Output

```
1
```

```
2
3
4
5
```

Here firstly statement inside body is executed then condition is checked. If the condition is true again body of loop is executed and this process continue until the condition becomes false. Unlike while loop semicolon is placed at the end of while.

There is minor difference between while and do while loop, while loop test the condition before executing any of the statement of loop. Whereas do while loop test condition after having executed the statement at least one within the loop.

If initial condition is false while loop would not executed it's statement on other hand do while loop executed it's statement at least once even If condition fails for first time. It means do while loop always executes at least once.

for loop

In a program, for loop is generally used when number of iteration are known in advance. The body of the loop can be single statement or multiple statements. Its syntax for writing is:

Syntax:-

```
for(exp1;exp2;exp3)
{
Statement;
}
```

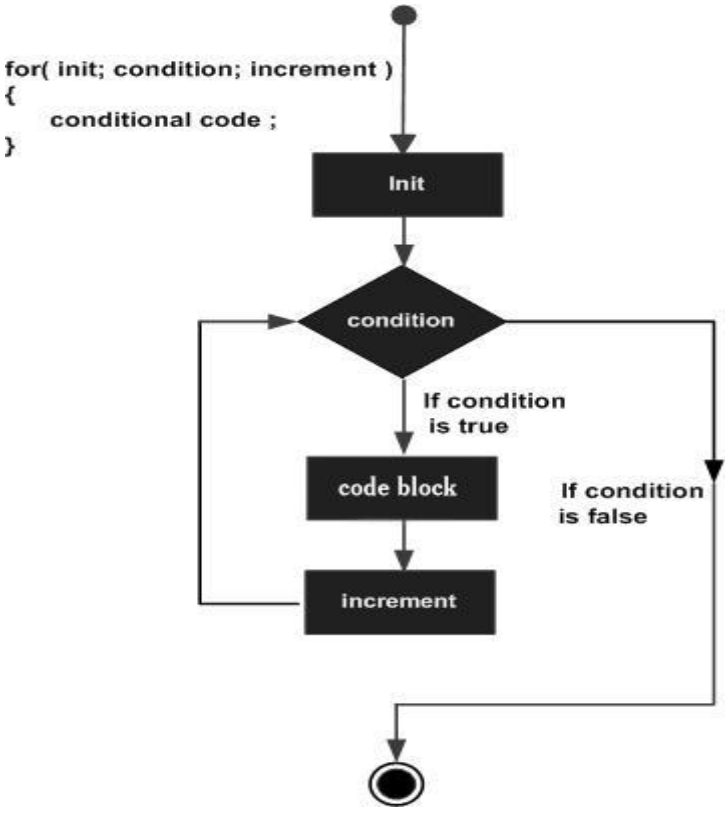
*

Or

```
for(initialized counter; test counter; update counter)
{
Statement;
}
```

Here exp1 is an initialization expression, exp2 is test expression or condition and exp3 is an update expression. Expression 1 is executed only once when loop started and used to initialize the loop variables. Condition expression generally uses relational and logical operators. And updation part executed only when after body of the loop is executed.

Flow chart



*

```c
// Program to add numbers until the user enters zero

#include <stdio.h>
int main() {
  double number, sum = 0;

  // the body of the loop is executed at least once
  do {
    printf("Enter a number: ");
    scanf("%lf", &number);
    sum += number;
  }
  while(number != 0.0);

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

  return 0;
}
```

Output

```
Enter a number: 1.5
Enter a number: 2.4
Enter a number: -3.4
Enter a number: 4.2
Enter a number: 0
```

*

Nesting of loop

When a loop written inside the body of another loop then, it is known as nesting of loop. Any type of loop can be nested in any type such as while, do while, for. For example nesting of for loop can be represented as :

Syntax of Nested loop

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

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

*

```
        }
        printf("n");
        i++;
    }
    return 0;
}
```

Output:

1

1 2

1 2 3

1 2 3 4

1 2 3 4 5

Difference between for, while and for loop

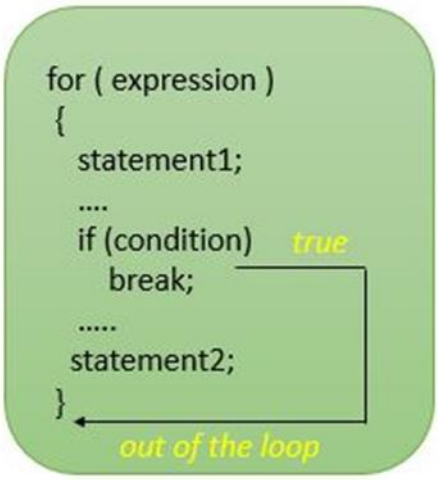
Sr. No	For loop	While loop	Do while loop
1.	Syntax: For(initialization; condition;updating), { . Statements; }	Syntax: While(condition), { . Statements; . }	Syntax: Do { . Statements; } While(condition);
2.	It is known as entry controlled loop	It is known as entry controlled loop.	It is known as exit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

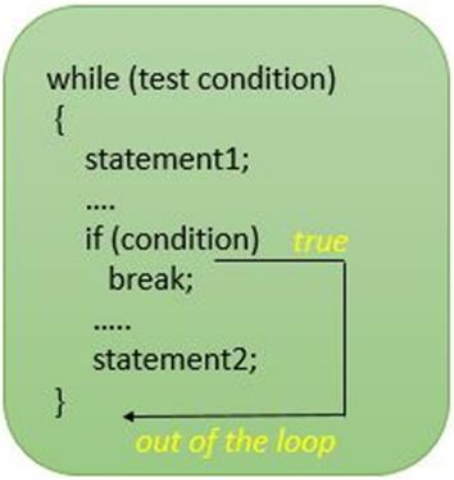
*

break statement(break)

Sometimes it becomes necessary to come out of the loop even before loop condition becomes false then break statement is used. Break statement is used inside loop and switch statements. It cause immediate exit from that loop in whichit appears and it is generally written with condition. It is written with the keywordas **break.** When break statement is encountered loop is terminated and control is transferred to the statement, immediately after loop or situation where we want to jump out of the loop instantly without waiting to get back to conditional state.

When break is encountered inside any loop, control automatically passes to thefirst statement after the loop. The break statement is almost always used with if...else statement inside the loop.





*

Example 1: break statement

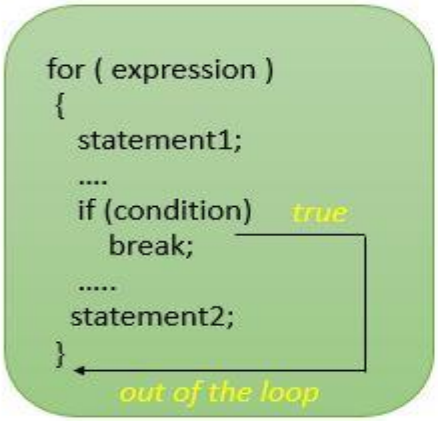
```
// Program to calculate the sum of numbers (10 numbers max)
// If the user enters a negative number, the loop terminates

#include <stdio.h>

int main() {
  int i;
  double number, sum = 0.0;

  for (i = 1; i <= 10; ++i) {
    printf("Enter n%d: ", i);
    scanf("%lf", &number);

    // if the user enters a negative number, break the loop
    if (number < 0.0) {
      break;
    }

    sum += number; // sum = sum + number;
  }

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

  return 0;
}
```

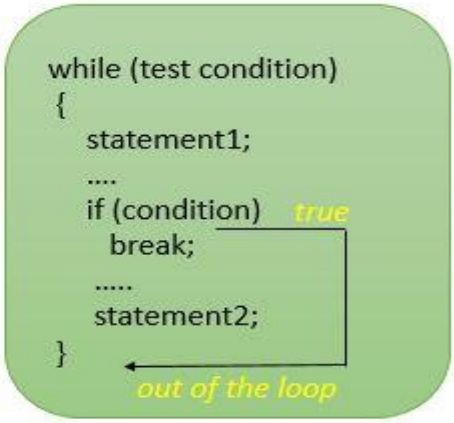
Output

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

*

How does break statement works?





Continue statement (key word continue)

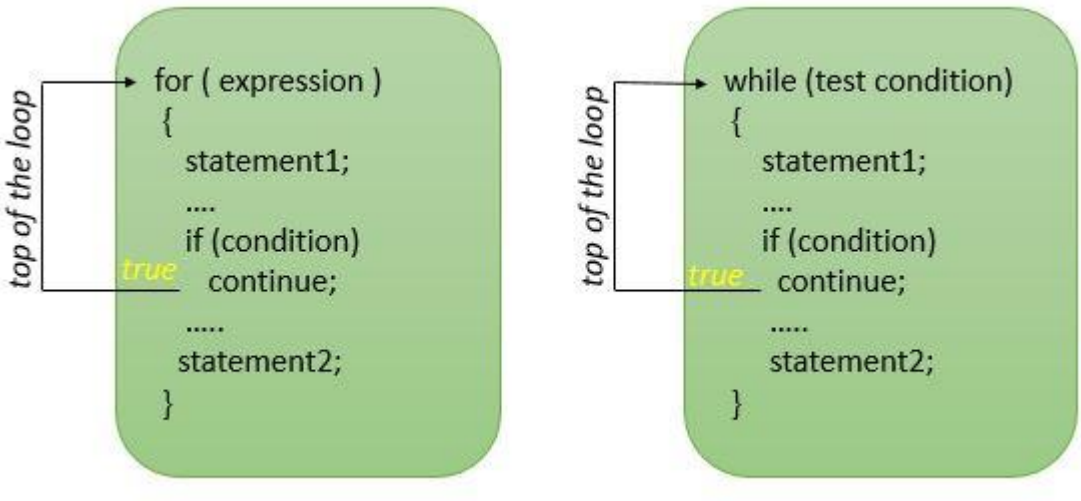
Continue statement is used for continuing next iteration of loop after skipping some statement of loop. When it encountered control automatically passes through the beginning of the loop. It is usually associated with the if statement. It isuseful when we want to continue the program without executing any part of the program.

The difference between break and continue is, when the break encountered loop is terminated and it transfer to the next statement and when continue is encounter control come back to the beginning position.

In a while and do while loop after continue statement control transfer to the test condition and then loop continue where as in, for loop after continue control transferred to the updating expression and condition is tested.

*

How continue statement work?



Example : continue statement

```c
// Program to calculate the sum of numbers (10 numbers max)
// If the user enters a negative number, it's not added to the result

#include <stdio.h>
int main() {
   int i;
   double number, sum = 0.0;

   for (i = 1; i <= 10; ++i) {
      printf("Enter a n%d: ", i);
      scanf("%lf", &number);

      if (number < 0.0) {
         continue;
      }

      sum += number; // sum = sum + number;
   }

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

   return 0;
}
```

*

Output

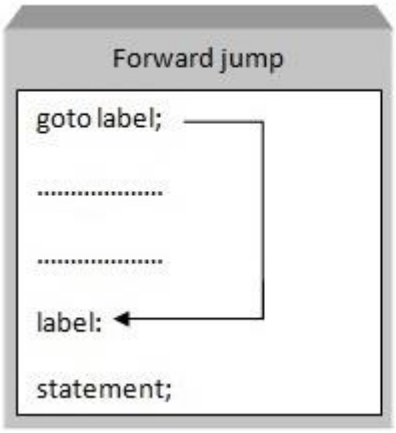
```
Enter n1: 1.1
Enter n2: 2.2
Enter n3: 5.5
Enter n4: 4.4
Enter n5: -3.4
Enter n6: -45.5
Enter n7: 34.5
Enter n8: -4.2
Enter n9: -1000
Enter n10: 12
Sum = 59.70
```

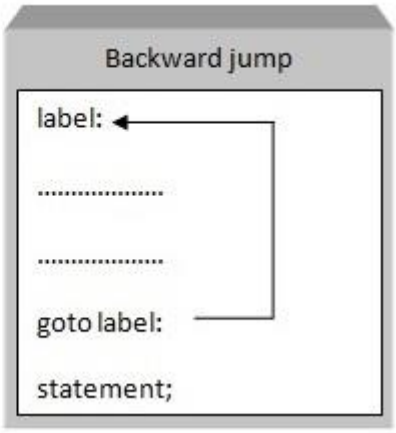
Syntax of goto Statement

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

The `label` is an identifier which specifies the place where the flow is to be jumped and it must be followed by colon. When the `goto` statement is encountered, the control of the program jumps to `label:` and starts executing the code.

*





Example: goto Statement

```
// Program to calculate the sum and average of positive numbers
// If the user enters a negative number, the sum and average are
displayed.

#include <stdio.h>

int main() {

   const int maxInput = 100;
   int i;
   double number, average, sum = 0.0;

   for (i = 1; i <= maxInput; ++i) {
      printf("%d. Enter a number: ", i);
      scanf("%lf", &number);

      // go to jump if the user enters a negative number
      if (number < 0.0) {
         goto jump;
      }
      sum += number;
   }

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

```
    return 0;
}
```

Output

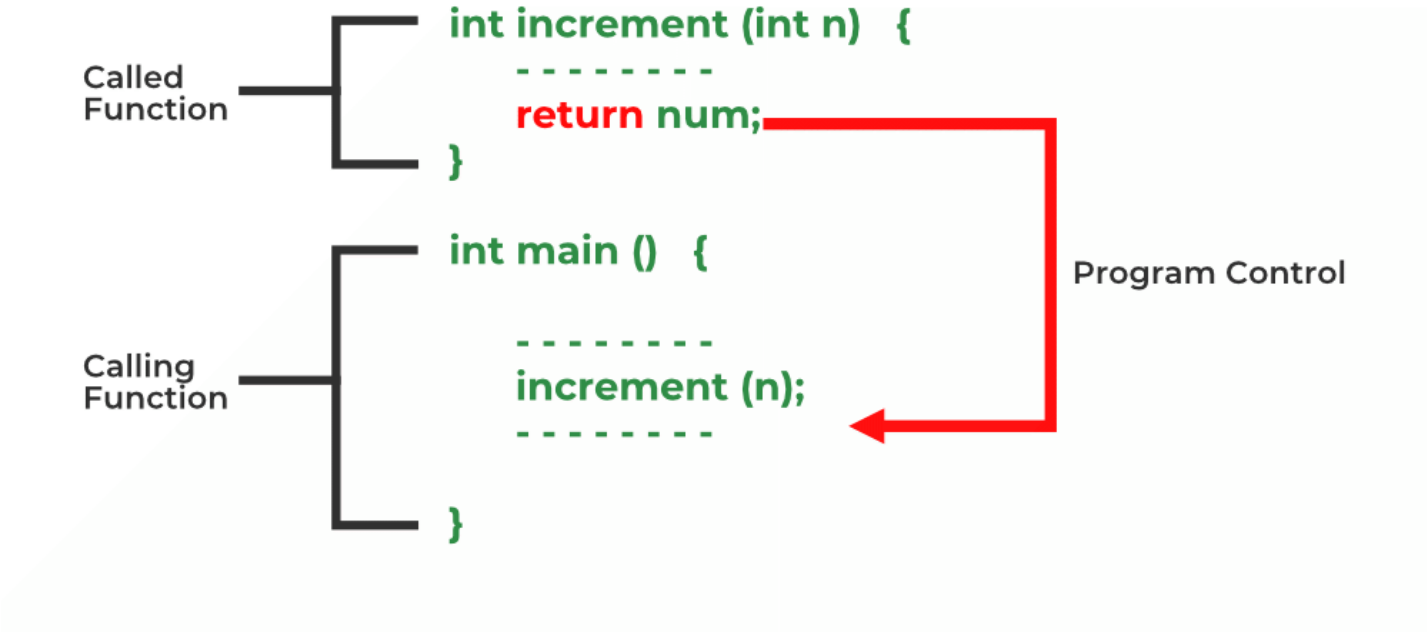
```
1. Enter a number: 3
2. Enter a number: 4.3
3. Enter a number: 9.3
4. Enter a number: -2.9
Sum = 16.60
Average = 5.53
```

return statement (keyword return) :

C return statement ends the execution of a function and returns the control to the function from where it was called. The return statement may or may not return a value depending upon the return type of the function.

In C, we can only return a single value from the function using the return statement and we have to declare the data_type of the return value in the function definition/declaration.

*



Here is the syntax and examples:

Returning control from function that does not return value:

return;

Returning control from function that returns value:

return <value>;

The return value could be any valid expression that returns a value:

a constant

a variable

a calculation, for instance (a + b) * c

call to another function that returns a value

*

The value must be of the same (or compatible) type that the function was defined.

For example, an int function can’t return a float value.

```
#include <stdio.h>
  // non-void return type function to calculate sum
int SUM(int a, int b)
{
    int s1 = a + b;
      // method using the return statement to return a value
    return s1;
}

int main()
{
    int num1 = 10;
    int num2 = 10;
    int sum_of = SUM(num1, num2);
    printf("The sum is %d", sum_of);
    return 0;
}
```

Output:

The sum is 20

exit() statement in C:

C exit() function is a standard library function defined in the <stdlib.h> header file used to terminate C program execution immediately with an error code.

*

void exit(int status)

Exit Success is indicated by **exit(0)** statement which means successful termination of the program, i.e. program has been executed without any error or interrupt.

Exit Failure is indicated by **exit(1)** which means the abnormal termination of the program, i.e. some error or interrupt has occurred.

```
#include <stdlib.h>
int main(){
   for(int i=0;i<=7;i++){
      if(i==5)
         exit(0);
      else
         printf("%d\n",i);
   }
   return 0;
}
```

Output

```
0
1
2
3
4
```

*

Module 4

*

ARRAY

Array is the collection of similar data types or collection of similar entity stored in contiguous memory location. Array of character is a **string**. Each data item of an array is called an element. And each element is unique and located in separated memory location. Each of elements of an array share a variable but each element having different index no. known as subscript.

An array can be a single dimensional or multi-dimensional and number of subscripts determines its dimension. And number of subscript is always starts with zero. **One dimensional array is known as vector and two dimensional arrays are known as matrix.**

DECLARATION OF AN ARRAY:

Its syntax is :

Data type array name [size];

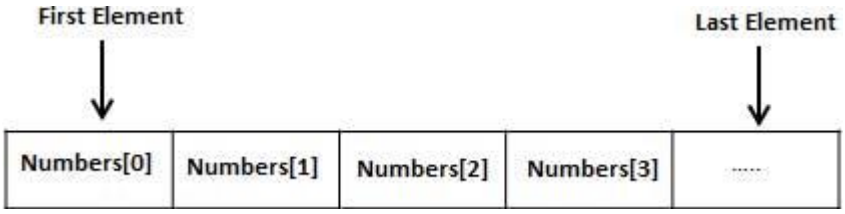
The size should be integer value.

Symbolic constant can also be used to specify the size of the array as:

```
#define SIZE 10;


int arr[100];
int mark[SIZE];
int number[5]={10,20,30,100,5}
```

The declaration of an array tells the compiler that, the data type, name of the array,size of the array and for each element it occupies memory space. Like for int data type, it occupies 2 bytes for each element and for float it occupies 4 byte for each element etc. The size of the array operates the number of elements that can be stored in an array and it may be an int constant or constant int expression.

*



All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

We can represent individual array as:

int ar[5];

ar[0], ar[1], ar[2], ar[3], ar[4];

INITIALIZATION OF AN ARRAY:

After declaration element of local array has garbage value. If it is global or static array then it will be automatically initialize with zero. An explicitly it can be initialize that

Data type array name [size] = {value1, value2, value3…}

Example:

int ar[5]={20,60,90, 100,120};

Array subscript always start from zero which is known as lower bound and upper value is known as upper bound and the last subscript value is one less than the size of array. Subscript can be an expression i.e. integer value. It can be any integer, integer constant, integer variable, integer expression or return value from functional call that yield integer value.

The array elements are standing in continuous memory locations and the amount of storage required for hold the element depend in its size & type.

Total size in byte for 1D array is:

Total bytes=size of (data type) * size of array.

*

Example : if an array declared is:

int [20];

Total byte= 2 * 20 =40 byte.

ACCESSING OF ARRAY ELEMENT:

We can access any array element using array name and subscript/index written inside pair of square brackets [].

Remember array indexing starts from 0. Nth element in array is at index N-1.

For Example:

Suppose we have an integer array of length 5 whose name is marks.

```
int marks[5] = {5,2,9,1,1};
```

Now we can access elements of array marks using subscript followed by array name.

marks[0] = First element of array marks = 5

marks[1] = Second element of array marks = 2

marks[2] = Third element of array marks = 9

marks[3] = Fourth element of array marks = 1

marks[4] = Last element of array marks = 1

Change Value of Array elements

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

// make the value of the third element to -1
mark[2] = -1;

// make the value of the fifth element to 0
```

```
mark[4] = 0;
```

Input and Output Array Elements

Here's how you can take input from the user and store it in an array element.

```
// take input and store it in the 3rd element
scanf("%d", &mark[2]);

// take input and store it in the ith element
scanf("%d", &mark[i-1]);
```

Here's how you can print an individual element of an array.

```
// print the first element of the array
printf("%d", mark[0]);

// print the third element of the array
printf("%d", mark[2]);

// print ith element of the array
printf("%d", mark[i-1]);
```

Example 1: Array Input/Output

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

#include <stdio.h>

int main() {

  int values[5];

  printf("Enter 5 integers: ");
```

*

```c
  // taking input and storing it in an array
  for(int i = 0; i < 5; ++i) {
     scanf("%d", &values[i]);
  }

  printf("Displaying integers: ");

  // printing elements of an array
  for(int i = 0; i < 5; ++i) {
     printf("%d\n", values[i]);
  }
  return 0;
}
```
Run C

Output

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

Here, we have used a `for` loop to take 5 inputs from the user and store them in an array. Then, using another `for` loop, these elements are displayed on the screen.

Example 2: Calculate Average

```c
// Program to find the average of n numbers using arrays
#include <stdio.h>
int main()
{

  int marks[10], i, n, sum = 0;
  double average;

  printf("Enter number of elements: ");
```

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

  for(i=0; i < n; ++i) {
    printf("Enter number%d: ",i+1);
    scanf("%d", &marks[i]);

    // adding integers entered by the user to the sum variable
    sum += marks[i];
  }

  // explicitly convert sum to double
  // then calculate average
  average = (double) sum / n;

  printf("Average = %.2lf", average);

  return 0;
}
```

Run Co

Output

```
Enter number of elements: 5
Enter number1: 45
Enter number2: 35
Enter number3: 38
Enter number4: 31
Enter number5: 49
Average = 39.60
```

Here, we have computed the average of n numbers entered by the user.

Access elements out of its bound!

Suppose you declared an array of 10 elements. Let's say,

```
int testArray[10];
```

*

You can access the array elements from `testArray[0]` to `testArray[9]`.

Now let's say if you try to access `testArray[12]`. The element is not available. This may cause unexpected output (undefined behavior). Sometimes you might get an error and some other time your program may run correctly.

Hence, you should never access elements of an array outside of its bound.

C Program to print all Array Elements

```
#include <stdio.h>
 int main()
{
   int value[7] = {1,2,3,4,5,6,7};
   int i;
   /* Printing array elements using loop */
      for(i = 0; i < 7; i++)
        {
                printf("Element at index %d is %d\n", i, value[i]);
        }
       return 0;
}
```

Output

```
Element at index 0 is 1

Element at index 1 is 2

Element at index 2 is 3
```

```
Element at index 3 is 4

Element at index 4 is 5

Element at index 5 is 6

Element at index 6 is 7
```

```
/* Write a program to add 10 array elements */
#include<stdio.h>
void main()
{
        int i ;
        int arr [10];
        int sum=0;
        for (i=0; i<=9; i++)
                {
                printf (“enter the %d element \n”, i+1);scanf (“%d”, &arr[i]);
                }
                for (i=0; i<=9; i++)
                {
                        sum = sum + a[i];
                }
                        printf (“the sum of 10 array elements is %d”, sum);
        }
```

OUTPUT:

```
Enter a value for arr[0] =5
Enter a value for arr[1] =10
Enter a value for arr[2] =15
```

*

Enter a value for arr[3] =20

Enter a value for arr[4] =25

Enter a value for arr[5] =30

Enter a value for arr[6] =35

Enter a value for arr[7] =40

Enter a value for arr[8] =45

Enter a value for arr[9] =50

the sum of 10 array elements is 275

While initializing a single dimensional array, it is optional to specify the size of array. If the size is omitted during initialization then the compiler assumes the size of array equal to the number of initializers.

For example:-

int marks[]={99,78,50,45,67,89};

If during the initialization of the number the initializers is less then size of array, then all the remaining elements of array are assigned value zero.

For example:-

int marks[5]={99,78};

Here the size of the array is 5 while there are only two initializers so after this initialization, the value of the rest elements are automatically occupied by zerossuch as

Marks[0]=99 ,

Marks[1]=78 ,

Marks[2]=0,

Marks[3]=0,

*

Marks[4]=0

Again if we initialize an array like
int array[100]={0};

Then the all the element of the array will be initialized to zero. If the number of initializers is more than the size given in brackets then the compiler will show an error.

For example:-
int arr[5]={1,2,3,4,5,6,7,8};//error

we cannot copy all the elements of an array to another array by simply assigning it to the other array like, by initializing or declaring as

int a[5] ={1,2,3,4,5};
int b[5];
b=a;//not valid

(Note:-here we will have to copy all the elements of array one by one, using for loop.)

Single dimensional arrays and functions

/*program to pass array elements to a function*/
#include<stdio.h>
void main()

*

```
{
        int arr[10],i;
        printf(“enter the array elements\n”);
        for(i=0;i<10;i++)
                {
                scanf(“%d”,&arr[i]);
                check(arr[i]);
                }
}

void check(int num)
{
        if(num%2==0)
        {
                printf(”%d is even \n”,num);
        }
        else
        {
                printf(”%d is odd \n”,num);
        }
}
```

Multidimensional Arrays

In this tutorial, you will learn to work with multidimensional arrays (two-dimensional and three-dimensional arrays) with the help of examples.

*

Two dimensional arrays

Two dimensional array is known as matrix. The array declaration in both the arrayi.e.in single dimensional array single subscript is used and in two dimensional array two subscripts are is used.

Its syntax is

Data-type array name[row][column];

Or we can say 2-d array is a collection of 1-D array placed one below the other.

Total no. of elements in 2-D array is calculated as **row*column**

Example:-

int a[2][3];

Total no of elements=row*column is 2*3 =6

It means the matrix consist of 2 rows and 3 columns For example:-

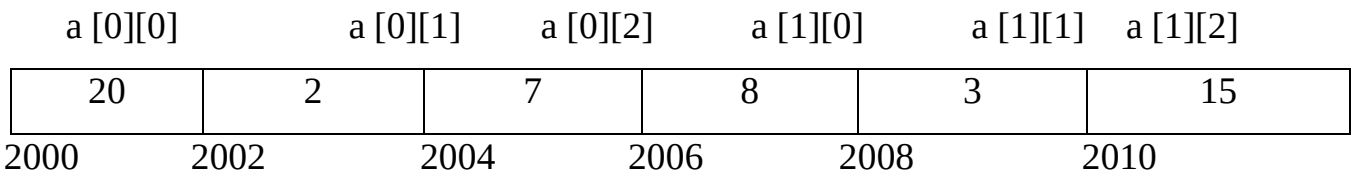
20 2 7

8 3 15

Positions of 2-D array elements in an array are as below

00 01 02

10 11 12



Accessing 2-d array /processing 2-d arrays

For processing 2-d array, we use two nested for loops. The outer for loopcorresponds to

*

the row and the inner for loop corresponds to the column.

For example

int a[4][5];

For reading value:-

```
for(i=0;i<4;i++)
{
for(j=0;j<5;j++)
{
        scanf("%d",&a[i][j]);
}
}
```

For displaying value:-

```
for(i=0;i<4;i++)
        {
        for(j=0;j<5;j++)
        {
                printf("%d",a[i][j]);
        }
        }
```

Initialization of 2-d array:

2D array can be initialized in a way similar to that of 1-D array.

for example:-

int mat[4][3]={11,12,13,14,15,16,17,18,19,20,21,22};

These values are assigned to the elements row wise, so the values of elements after this initialization are

*

Mat[0][0]=11, Mat[1][0]=14, Mat[2][0]=17 Mat[3][0]=20

Mat[0][1]=12, Mat[1][1]=15, Mat[2][1]=18 Mat[3][1]=21

Mat[0][2]=13, Mat[1][2]=16, Mat[2][2]=19 Mat[3][2]=22

While initializing we can group the elements row wise using inner braces.

for example:-

int mat[4][3]={{11,12,13},{14,15,16},{17,18,19},{20,21,22}};

And while initializing, it is necessary to mention the 2nd dimension where 1st dimension is optional.

int mat[][3];

int mat[2][3];

int mat[][];

int mat[2][]; } invalid

If we **initialize an array** as

int mat[4][3]={{11},{12,13},{14,15,16},{17}};

Then the compiler will assume its all rest value as 0, which are not defined.

Mat[0][0]=11,	Mat[1][0]=12,	Mat[2][0]=14,	Mat[3][0]=17
Mat[0][1]=0,	Mat[1][1]=13,	Mat[2][1]=15	Mat[3][1]=0

*

Mat[0][2]=0, Mat[1][2]=0, Mat[2][2]=16, Mat[3][2]=0

In memory map whether it is 1-D or 2-D, elements are stored in onecontiguous manner.

We can also give the size of the 2-D array by using symbolic constant such as

#define ROW 2;

#define COLUMN 3;

int mat[ROW][COLUMN];

```c
#include <stdio.h>
int main() {
  int a[10][10], transpose[10][10], r, c;
  printf("Enter rows and columns: ");
  scanf("%d %d", &r, &c);

  // asssigning elements to the matrix
  printf("\nEnter matrix elements:\n");
  for (int i = 0; i < r; ++i)
  for (int j = 0; j < c; ++j) {
    printf("Enter element a%d%d: ", i + 1, j + 1);
    scanf("%d", &a[i][j]);
  }

  // printing the matrix a[][]
  printf("\nEntered matrix: \n");
  for (int i = 0; i < r; ++i)
  for (int j = 0; j < c; ++j) {
    printf("%d  ", a[i][j]);
    if (j == c - 1)
    printf("\n");
  }

  // computing the transpose
  for (int i = 0; i < r; ++i)
  for (int j = 0; j < c; ++j) {
    transpose[j][i] = a[i][j];
  }
```

*

```
 // printing the transpose
 printf("\nTranspose of the matrix:\n");
 for (int i = 0; i < c; ++i)
 for (int j = 0; j < r; ++j) {
   printf("%d  ", transpose[i][j]);
   if (j == r - 1)
   printf("\n");
 }
 return 0;
}
```

Output:

```
Enter rows and columns: 2
3

Enter matrix elements:
Enter element a11: 1
Enter element a12: 4
Enter element a13: 0
Enter element a21:-5
Enter element a22: 2
Enter element a23: 7

Entered matrix:
1  4  0
-5  2  7

Transpose of the matrix:
1  -5
4  2
0  7
```

```
#include <stdio.h>
int main() {
  int r, c, a[100][100], b[100][100], sum[100][100], i, j;
```

*

```
  printf("Enter the number of rows (between 1 and 100): ");
  scanf("%d", &r);
  printf("Enter the number of columns (between 1 and 100): ");
  scanf("%d", &c);

  printf("\nEnter elements of 1st matrix:\n");
  for (i = 0; i < r; ++i)
    for (j = 0; j < c; ++j) {
      printf("Enter element a %d%d: ", i + 1, j + 1);
      scanf("%d", &a[i][j]);
    }

  printf("Enter elements of 2nd matrix:\n");
  for (i = 0; i < r; ++i)
    for (j = 0; j < c; ++j) {
      printf("Enter element b %d%d: ", i + 1, j + 1);
      scanf("%d", &b[i][j]);
    }

  // adding two matrices
  for (i = 0; i < r; ++i)
    for (j = 0; j < c; ++j) {
      sum[i][j] = a[i][j] + b[i][j];
    }

  // printing the result
  printf("\nSum of two matrices: \n");
  for (i = 0; i < r; ++i)
    for (j = 0; j < c; ++j) {
      printf("%d   ", sum[i][j]);
      if (j == c - 1) {
        printf("\n\n");
      }
    }

  return 0;
}
```

Output

```
Enter the number of rows (between 1 and 100): 2
```

```
Enter the number of columns (between 1 and 100): 3

Enter elements of 1st matrix:
Enter element a11: 2
Enter element a12: 3
Enter element a13: 4
Enter element a21: 5
Enter element a22: 2
Enter element a23: 3
Enter elements of 2nd matrix:
Enter element b11: -4
Enter element b12: 5
Enter element b13: 3
Enter element b21: 5
Enter element b22: 6
Enter element b23: 3

Sum of two matrices:
-2   8   7
10   8   6
```

String

String in C programming is a sequence of characters terminated with a null character '\0'. Strings are defined as an array of characters. The difference between a character array and a string is the string is terminated with a unique character '\0'.String is a one dimensional array of character.

Declaration of Strings

Declaring a string is as simple as declaring a one-dimensional array. Below is the basic syntax for declaring a string.

char str_name[size];

Initializing a String

*

4 Ways to Initialize a String in C

1. Assigning a string literal without size: String literals can be assigned without size. Here, the name of the string str acts as a pointer because it is an array.

```
char str[] = "GeeksforGeeks";
```

2. Assigning a string literal with a predefined size: String literals can be assigned with a predefined size. But we should always account for one extra space which will be assigned to the null character. If we want to store a string of size n then we should always declare a string with a size equal to or greater than n+1.

```
char str[50] = "GeeksforGeeks";
```

3. Assigning character by character with size: We can also assign a string character by character. But we should remember to set the end character as '\0' which is a null character.

```
char str[14] = { 'G','e','e','k','s','f','o','r','G','e','e','k','s','\0'};
```

4. Assigning character by character without size: We can assign character by character without size with the NULL character at the end. The size of the string is determined by the compiler automatically.

```
char str[] = { 'G','e','e','k','s','f','o','r','G','e','e','k','s','\0'};
```

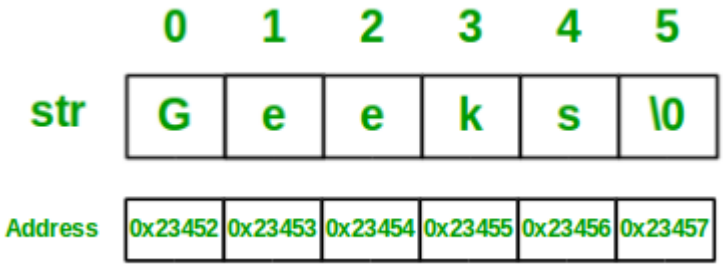
We can initialize the string as

```
char name[]={'G','e','e','k','s','\0'};
```

Here each character occupies 1 byte of memory and last character is always NULL character. Where '\0' and 0 (zero) are not same, where **ASCII** value of '\0' is 0 and ASCII value of 0 is 48. Array elements of character array are also stored in contiguous memory allocation.

From the above we can represent as;

*



The terminating NULL is important because it is only the way that the function that work with string can know, where string end.

String can also be **initialized** as;

char name [] =”John”;

Here the NULL character is not necessary and the compiler will assume it automatically.

Read String from the user

1) scanf() function

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

Example 1: scanf() to read a string

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

Output

```
Enter name: Dennis Ritchie
```

```
Your name is Dennis.
```

2) Using *getchar*() Functions

getchar() function reads one character at a time. We can use getchar function inside a loop to read characters one by one till we don't read newline character (\n). Once we read newline character we break loop and add '\0' character at the end of string.

```
#include <stdio.h>
int main()
{
        char str[50], ch;
        int i;
        printf("Enter a string: ");
        i = 0;
        ch = getchar ();
        while(ch!='\n')
        {
        str[i] = ch;
        i++;
        ch = getchar();
        }
        str[i] ='\0';
        printf("Entered string is: %s", str);
        return 0;
}
```

Output:

```
Enter a string: Where have you been?
Entered string is: Where have you been?
```

3) **Using gets() function in C :**

Using gets(), string can be read as follows:

```
gets(str);
```

1. gets() overcomes the shortcomings of scanf(). Gets stands for get string.
2. In this case, the entire string “Hello Word” will be read by the function.
3. The function takes starting address of the string which will hold the input and automatically appends the null character at the end of the string.
4. The gets function takes one parameter, the string in which to store the data which is read. It reads characters from standard input up to the next newline character (that is, when the user presses Enter button), discards the newline character, and copies the rest of the data into the string. If there was no error, it returns the same string ; otherwise, if there was an error, it returns a null pointer.
5. **gets** is the easiest to use, however should be avoided as far as possible to avoid unreliable behaviour.
6. We shall see going ahead our first string operation example using gets().

```
#include <stdio.h>
int main ()
 {
        char str[50];
        printf("Enter a string : ");
        gets(str);
        printf("You entered: %s", str);
        return(0);
}
```

Output:

```
Enter a string : tutorialspoint.com
```

*

You entered: tutorialspoint.com

Writing String from the user

There are three methods using which a string can be written / displayed on screen.

Consider the

string; str[] = “Hello World”;

1. **Using printf() function:(Most Preferred):**

1. Using printf, a string can be displayed as follows : %s s the format specifier used to print a string.
2. This function prints a text string to the terminal i.e stdout screen. It is one of the preferred modes to output a string.
3. The entire string “Hello Word” will be printed by the function.
4. It enables us to print formatted output; thus giving us flexibility.

```
 #include<stdio.h>
#include<conio.h>
int main()
{
   char str[20];
   printf("Enter your first name: ");
   scanf("%s", str);
   printf("Hello, %s", str);
   getch();
   return 0;
}
```

Output:

*

```
Enter any sentence: we love codescracker

You have entered:
we
```

2. **Using puts() function:**

Using puts(), string can be displayed as follows: It just takes its parameter as the string to be printed.

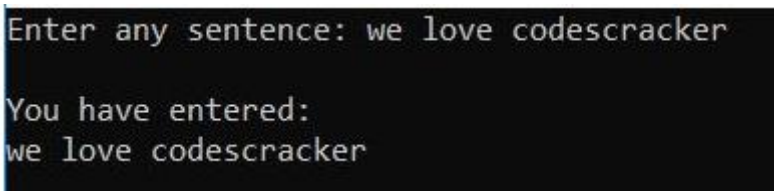
puts(str);

1. In this case too, the entire string “Hello Word” will be printed by the function.
2. The most convenient function for printing a simple message on standard output is puts. It is even simpler than printf, since you do not need to include a newline character — puts does that for you.
3. The puts function is safe and simple, but not very flexible as it does not give us an option of formatting our string.
4. Similar to puts you have fputs() (“file put string”) function ; except that it accepts a second parameter, a stream to which to write the string. It does not add a newline character, however; it only writes the characters in the string. It returns EOF if an error occurs; otherwise it returns a non-negative integer value.

```
#include<stdio.h>
#include<conio.h>
int main()
{
   char str[100];
   printf("Enter any sentence: ");
   gets(str);
   printf("\nYou have entered:\n");
   puts(str);
   getch();
```

```
    return 0;
}
```

Output:



3. **Using putchar() function:**

- putchar() as the name states prints only one character at a time.
- In order to print a string, we have to use this function repeatedly until a terminating character is encountered.
- Using putchar(), string can be read as follows:

```
/*
* C Program to read and print string using getchar and putchar
*/
#include <stdio.h>
#include <conio.h>

int main()
{
      char inputString[100], c;
      int index = 0;
    printf("Enter a string\n");
    while((c = getchar()) != '\n')
{
       inputString[index] = c;
       index++;
```

```
    }
    inputString[index] = '\0';
     index = 0;
    while(inputString[index] != '\0')
   {
        putchar(inputString[index]);
        index++;
  }
    getch();
    return 0;
}
```

Output

Enter a string

C Programming

C Programming

String library function

There are several string library functions used to manipulate string and the prototypes for these functions are in header file **“string.h”.** Several string functionsare

No.	Function	Description
1)	strlen(string_name)	returns the length of string name.
2)	strcpy(destination, source)	copies the contents of source string to destination string.
3)	strcat(first_string, second_string)	concats or joins first string with second string.

*

		The result of the string is stored in first string.
4)	strcmp(S1, S2)	compares the first string with second string. If S1 & S2 strings are same, it returns 0 returns less than 0 if S1<S2 returns greater than 0 if S1>S2
5)	strchr(S1, ch)	Returns pointer to first occurrence ‘ch‘ in S1
6)	strstr(S1,S2)	Returns pointer to first occurrence ‘S2‘ in S1

Function	Built in Function	User defined function

strlen()	`#include <stdio.h>`
	`#include <string.h>`
	`int main() {`
	
	`  char string1[20] = "ScalerAcademy";`
	`  printf("Length of string string1: %ld", strlen(string1));`
	`  return 0;`
	`}`
		`#include<stdio.h>`
		`/* Function Prototype */`
		`int mystrlen(char str[30]);`
		`/* Main Function */`
		`int main()`
		`{`
		`char str[30];`
		`int i, len;`
		`printf("Enter string:\n");`
		`gets(str);`
		`len = mystrlen(str); /* Function Call */`
		`printf("Length of given string is: %d", len);`
		`return 0;`

		}
		/* Function Definition */
		int mystrlen(char str[30])
		{
		int i, len=0;
		for(i=0;str[i]!='\0';i++)
		{
		len++;
		}
		return(len);
		}

strcmp()	#include <stdio.h>
	#include <string.h>
	int main() {
	
	char s1[20] = "ScalerAcademy";
	char s2[20] = "ScalerAcademy.COM";
	if (strcmp(s1, s2) == 0)
	{
	printf("string 1 and string 2 are equal");
	}
	else
	{
	printf("string 1 and 2 are different");
	}
	}
		#include<stdio.h>
		/* Function Protptype*/
		int mystrcmp(char str1[40], char str2[40]);
		/* Main Function */
		int main()
		{
		char str1[40], str2[40];
		int d;
		printf("Enter first string:\n");
		gets(str1);
		printf("Enter second string:\n");
		gets(str2);
		/* Function Call */
		d = mystrcmp(str1, str2);

*

```
 if(d==0)
 {
  printf("Given strings are same.");
 }
 else
 {
  printf("Given strings are different.");
 }
 return 0;
}
/* Function Definition */
int mystrcmp(char str1[40], char str2[40])
{
 int d,i, len1=0, len2=0, flag=0;
 /* Finding length of first string */
 for(i=0; str1[i]!='\0'; i++)
 {
  len1++;
 }

 /* Finding length of first string */
 for(i=0; str2[i]!='\0'; i++)
 {
```

*

		len2++;
		}
		if(len1!=len2)
		{
		return(1);
		}
		else
		{
		for(i=0;i< len1;i++)
		{
		if(str1[i]!=str2[i])
		{
		flag=1;
		break;
		}
		}
		if(flag==0)
		{
		return(0);
		}
		else
		{
		return(1);
		}
		}
		}

strcat()	#include <stdio.h> #include <string.h> int main() { char string1[10] = "Hello"; char string2[10] = "World";	#include<stdio.h> /* Function Prototype*/ void mystrcat(char

*

```
  strcat(string1, string2);
  printf("Output string after
concatenation: %s", string1);

}
```

```
str1[40], char str2[40]);

/* Main Function */

int main()

{

 char str1[50], str2[50];

 int i, len=0;

 printf("Enter first
string:\n");

 gets(str1);

 printf("Enter second
string:\n");

 gets(str2);

 mystrcat(str1, str2);

 printf("Concatenated
string is: %s", str1);

 return 0;

}

void mystrcat(char
str1[40], char str2[40])

{

 int i, len=0;

 /* Calculating length of
first string */
for(i=0;str1[i]!='\0';i++)

 {

  len++;

 }

 /* Concatenating second
string to first string */


for(i=0;str2[i]!='\0';i++)
```

*

		{
		str1[len+i] = str2[i];
		}
		str1[len+i]='\0';
		}

strcpy()	#include <stdio.h>
	#include <string.h>
	
	int main() {
	
	char s1[35] = "string 1"; // string1
	char s2[35] = "I'll be copied to string 1."; // string2
	strcpy(s1, s2); // copying string2 to string1
	printf("String s1 is: %s", s1); // printing string1
	
	}
		/* Function prototype */
		void mystrcpy(char str2[30], char str1[30]);
		/* Main function */
		int main()
		{
		char str1[30], str2[30];
		int i;
		printf("Enterstring:\n");
		gets(str1);
		mystrcpy(str2, str1);
		printf("Copied string is: %s", str2);
		return 0;
		}
		/* Function Definition*/
		void mystrcpy(char str2[30], char str1[30])
		{
		int i;
		for(i=0;str1[i]!='\0';i++)
		{
		str2[i] = str1[i];
		}
		str2[i] = '\0';

*

		}

strchr()	#include <stdio.h>
	#include <string.h>
	
	int main() {
	char string1[30] = "I love to write.";
	printf("%s", strchr(string1, 'w'));
	}
	
	Output : write.	

strstr()	#include <stdio.h>
	#include <string.h>
	
	int main() {
	
	char string1[70] = "You are reading string functions.";
	printf("Output string is: %s", strstr(string1, "string"));
	}
	Output: Output string is: string functions.
		#include<stdio.h>
		int fStrStr(char* str, char* strSub)
		{
		int i=0, j=0;
		int nTemp = i;
		int nStrLen = strlen(str);
		int nStrSubLen = strlen(strSub);
		for(i=0;i<nStrLen-nStrSubLen; i++)
		{
		nTemp = i;
		for(j=0; j<nStrSubLen; j++)
		{
		if(str[nTemp]==strSub[j])
		{
		if(j==nStrSubLen-1)
		return 1;
		nTemp++;
		}

```
            else
                break;
        }
    }
    return 0;
}
int main()
{
    char str[] = "CSEStack";
    char strSub[] = "SES";
     if(fStrStr(str, strSub))
        printf("Sub-string found.");
    else
       printf("Sub-string not found.");
}
```

FUNCTION

A function is a self contained block of codes or sub programs with a set of statements that perform some specific task or coherent task when it is called.

It is something like to hiring a person to do some specific task like, every six months servicing a bike and hand over to it.

Any 'C' program contain at least one function i.e main().

There are basically two types of function those are

1. **Library function**
2. **User defined function**

The user defined functions defined by the user according to its requirement

System defined function can't be modified, it can only read and can be used.These function are supplied with every C compiler

Source of these library function are pre complied and only object code get used bythe user by linking to the code by linker

Here in system defined function description:

Function definition : predefined, precompiled, stored in the library

*

Function declaration : In header file with or function prototype.

Function call : By the programmer

User defined function

Syntax:-

Return type name of function (type 1 arg 1, type2 arg2, type3 arg3)

Return type function name argument list of the above syntax

So when user gets his own function three thing he has to know, these are.

Function declaration

Function definition

Function call

These three things are represented like

```
int function(int, int, int);                    /*function declaration*/
main()                    /* calling function*/
{
function(arg1,arg2,arg3);
}
int function(type 1 arg 1,type2 arg2,type3, arg3)          /*function definition/*
{
  Local   variable   declaration;
Statement;
Return value;
}
```

*

Function declaration:-

Function declaration is also known as function prototype. It inform the compilerabout three thing, those are name of the function, number and type of argument received by the function and the type of value returned by the function.

While declaring the name of the argument is optional and the function prototype always terminated by the semicolon.

Function definition:-

Function definition consists of the whole description and code of the function. It tells about what function is doing what are its inputs and what are its out put It consists of two parts function header and function body

Syntax:-

return type function(type 1 arg1, type2 arg2, type3 arg3) /*function header*/

{

Local variable declaration; Statement

1;

Statement 2;Return value

}

The return type denotes the type of the value that function will return and it is optional and if it is omitted, it is assumed to be int by default. The body of the function is the compound statements or block which consists of local variable declaration statement and optional return statement.

*

The local variable declared inside a function is local to that function only. It can't be used anywhere in the program and its existence is only within this function.

The arguments of the function **definition** are known as **formal arguments**.

Function Call

When the function get called by the calling function then that is called, function call. The compiler execute these functions when the semicolon is followed by the function name.

Example:- function(arg1,arg2,arg3);

The argument that are used inside the function call are called **actual argument**

Ex:-

int S=sum(a, b); //actual arguments

Actual argument

The arguments which are mentioned or used inside the function call is knows as actual argument and these are the original values and copy of these are actually sent to the called function

e written as constant, expression or any function call likeFunction (x);

Function (20, 30); Function (a*b, c*d);

Function(2,3,sum(a, b));

Formal Arguments

The arguments which are mentioned in function definition are called formal arguments or dummy arguments.

*

These arguments are used to just hold the copied of the values that are sent by the calling function through the function call.

These arguments are like other local variables which are created when the functioncall starts and destroyed when the function ends.

The basic difference between the formal argument and the actual argument are

1) The formal argument are declared inside the parenthesis where as the local variable declared at the beginning of the function block.

2). The **formal argument** are automatically initialized when the copy of actual arguments are passed while other local variable are assigned values through the statements.

Order number and type of actual arguments in the function call should be matchwith the order number and type of the formal arguments.

Return type

sed to return value to the calling function. It can be used in two way asreturn

Or return(expression);

Ex:- return (a);

return (a*b); return (a*b+c);

Here the 1st return statement used to terminate the function without returning anyvalue

Ex:- /*summation of two values*/

int sum (int a1, int a2);

main()

*

```
{
int a,b;
printf("enter two no");
scanf("%d%d",&a,&b); int
S=sum(a,b);
printf("summation is = %d",s);
}
int sum(intx1,int y1)
{
int z=x1+y1;
Return z;
}
```

Advantage of function

By using function large and difficult program can be divided in to sub programs and solved. When we want to perform some task repeatedly or some code is to beused more than once at different place in the program, then function avoids this repeatition or rewritten over and over.

Due to reducing size, modular function it is easy to modify and test

Notes:-

C program is a collection of one or more function.

A function is get called when function is followed by the semicolon.

A function is defined when a function name followed by a pair of curly braces

*

Any function can be called by another function even main() can be called by otherfunction.

```
main()
{
function1()
}
function1()
{
Statement;
function2;
}
function 2()
{



}
```

So every function in a program must be called directly or indirectly by the main() function. A function can be called any number of times.

A function can call itself again and again and this process is called **recursion**.

A function can be called from other function **but** a function can't be defined in another function

Lecture Note: 15

Category of Function based on argument and return type

i) Function with no argument & no return value

*

Function that have no argument and no return value is written as:- void function(void);

```
main()
{
void function()
{
Statement;
}
```

Example:- void me();

```
main()

{
me();
printf("in main");

}
void me()
{
printf("come on");
}
```

Output: come on

inn main

*

ii) Function with no argument but return value

Syntax:-

```
int                    fun(void);
main()
{

int r; r=fun();
}
int                        fun()
{

reurn(exp);
}
```

Example:- int sum();

```
main()
{
int                  b=sum();
printf("entered   %d\n, b");
}
int                 sum()
{
int  a,b,s;
```

*

s=a+b; return s;

}

Here called function is independent and are initialized. The values aren't passed bythe calling function .Here the calling function and called function are communicated partly with each other.

Lecture Note: 16

iii) function with argument but no return value

Here the function have argument so the calling function send data to the calledfunction but called function dose n't return value.

Syntax:-

```
void                    fun (int,int);
main()
{

int (a,b);
}
```

```
void                fun(int x, int y);
{
Statement;
}
```

*

Here the result obtained by the called function.

iv) function with argument and return value

Here the calling function has the argument to pass to the called function and the called function returned value to the calling function.

Syntax:-

```
fun(int,int);main()
{
int r=fun(a,b);
}
int  fun(intx,inty)
```

Example:

```
{
```

*

{

}

main()

*

```
return(exp);

int fun(int);int  a,num;
printf(“enter value:\n”);scanf(“%d”,&a)
```

*

```
int num=fun(a);
}
int  fun(int x)
{

++x;



}
```

*

return x;

Call by value and call by reference

There are two way through which we can pass the arguments to the function such as **call by** value and **call by reference**.

1. **Call by value**

In the call by value copy of the actual argument is passed to the formal argument and the operation is done on formal argument.

When the function is called by 'call by value' method, it doesn't affect content of the actual argument.

Changes made to formal argument are local to block of called function so when the control back to calling function the changes made is vanish.

Example:-

```
main()

{
 int x,y; change(int,int);
```

```
printf("enter two values:\n");
scanf("%d%d",&x,&y); change(x ,y);
printf("value of x=%d and y=%d\n",x ,y);
}
change(int a,int b);
{
int k;k=a; a=b; b=k;
}
```

Output: enter two values: 1223

Value of x=12 and y=23

2. Call by reference

Instead of passing the value of variable, address or reference is passed and the function operate on address of the variable rather than value.

Here formal argument is alter to the actual argument, it means formal arguments calls the actual arguments.

Example:-

```
void main()
```

*

```
{
int a,b;
change(int *,int*); printf("enter two
values:\n"); scanf("%d%d",&a,&b);
change(&a,&b);
printf("after changing two value of a=%d and b=%d\n:"a,b);
}
change(int *a, int *b)
{
int k; k=*a;
*a=*b;
*b= k;

printf("value in this function a=%d and b=%d\n",*a,*b);
}
```

Output: enter two values: 1232

Value in this function a=32 and b=12

After changing two value of a=32 and b=12

So here instead of passing value of the variable, directly passing address of the variables. Formal argument directly access the value and swapping is possible evenafter calling a function.

*

Lecture Note: 17

Local, Global and Static variable

Local variable:-

variables that are defined with in a body of function or block. The local variables can be used only in that function or block in which they are declared.

Same variables may be used in different functions such asfunction()

{

int a,b; function 1();

}

function2 ()

{

int a=0;b=20;

}

*

Global variable:-

*

the variables that are defined outside of the function is called global variable. All functions in the program can access and modify global variables. Global variables are automatically initialized at the time of initialization.

Example:

```
#include<stdio.h>          void
function(void);            void
function1(void);           void
function2(void);int a, b=20;
void main()
{

    printf("inside main a=%d,b=%d \n",a,b);
function();
function1();
function2();
}
function()
{

Prinf("inside function a=%d,b=%d\n",a,b);
}

function 1()
{
```

```
prinf(“inside function a=%d,b=%d\n”,a,b);
}

function 2()
{

prinf(“inside function a=%d,b=%d\n”,a,);
}
```

Static variables: static variables are declared by writing the key word static.

-syntax:-

static data type variable name;static int a;

-the static variables initialized only once and it retain between the function call. If its variable is not initialized, then it is automatically initialized to zero.

Example:

```
void fun1(void); void
fun2(void);void main()
{

fun1();
fun2();
}

void fun1()
{
```

*

```
int a=10, static int b=2; printf("a=%d,
b=%d",a,b);a++;
b++;
}
```

Output:a= 10 b= 2

a=10 b= 3

Recursion

When function calls itself (inside function body) again and again then it is called as recursive function. In recursion calling function and called function are same. It is powerful technique of writing complicated algorithm in easiest way. According to recursion problem is defined in term of itself. Here statement with in body of the function calls the same function and same times it is called as circular definition. In other words recursion is the process of defining something in form ofitself.

Syntax:

```
main ()
{

rec(); /*function call*/rec();
rec();
```

Ex:- /*calculate factorial of a no.using recursion*/ int fact(int);

```
void main()
```

*

```
{
int num;
printf(“enter a number”); scanf(“%d”,&num);
f=fact(num); printf(“factorial is =%d\n”f);
}

fact (int num)
{

If  (num==0||num==1)
return 1;else
return(num*fact(num-1));
}
```

Lecture Note: 18

Monolithic Programming

The program which contains a single function for the large program is called monolithic program. In monolithic program not divided the program, it is huge long pieces of code that jump back and forth doing all the tasks like single thread of execution, the program requires. Problem arise in monolithic program is that, when the

program **size** increases it leads inconvenience and difficult to maintain

*

such as testing, debugging etc. Many disadvantages of monolithic programmingare:

1. Difficult to check error on large programs size.
2. Difficult to maintain because of huge size.
3. Code can be specific to a particular problem. i.e. it cannot be reused.

Many early languages (FORTRAN, COBOL, BASIC, C) required one huge workspace with labelled areas that may does specific tasks but are not isolated.

Modular Programming

The process of subdividing a computer program into separate sub-programs such as functions and subroutines is called Modular programming. **Modular programming sometimes also called as structured programming.** It enables multiple programmers to divide up the large program and debug pieces of program independently and tested. . Then the linker will link all these modules to form the complete program. This principle dividing software up into parts, or modules, where a module can be changed, replaced, or removed, with minimal effect on the other software it works with. Segmenting the program into modules clearly defined functions, it can determine the source of program errors more easily. Breaking down program functions into modules, where each of which accomplishes one function and contains all the source code and variables needed to accomplish that function.

Modular program is the solution to the problem of very large program that are difficult to debug, test and maintain. A program module may be rewritten while its inputs and outputs remain the same. The person making a change may only understand a small portion of the original program.

*

Object-oriented programming (OOP) is compatible with the modular programming concept to a large extent.

. , Less code has to be written that makes shorter.

*

- A single procedure can be developed for reuse, eliminating the need to retype the code many times.
- Programs can be designed more easily because a small team deals with only a small part of the entire code.
- Modular programming allows many programmers to collaborate on the same application.
- The code is stored across multiple files.
- Code is short, simple and easy to understand and modify, make simple to figure out how the program is operate and reduce likely hood of bugs.
- Errors can easily be identified, as they are localized to a subroutine or function or isolated to specific module.
- The same code can be reused in many applications.
- The scoping of variables and functions can easily be controlled.

Disadvantages

However it may takes longer to develop the program using this technique.

Storage Classes

Storage class in c language is a specifier which tells the compiler where and how to store variables, its initial value and scope of the variables in a program. Or attributes of variable is known as storage class or in compiler point of view a variable identify some physical location within a computer where its string of bits value can be stored is known as storage class.

*

The kind of location in the computer, where value can be stored is either in the memory or in the register. There are various storage class which determined, in which of the two location value would be stored.

Syntax of declaring storage classes is:-

storageclass datatype variable name;

There are four types of storage classes and all are keywords:-**1)**

Automatic (auto)

*

2) Register (register)

3) Static (static)

4) External (extern)

Examples:-

auto float x; or float x;extern int x; register char c; static int y;

Compiler assume different storage class based on:-

1) Storage class:- tells us about storage place(where variable would be stored).

2) Intial value :-what would be the initial value of the variable.

If initial value not assigned, then what value taken by uninitialized variable.

3) Scope of the variable:-what would be the value of the variable of the program.

4) Life time :- It is the time between the creation and distribution of a variable or how long would variable exists.

1. Automatic storage class

The keyword used to declare automatic storage class is auto. Its features:-

Storage-memory location

Default initial value:-unpredictable value or garbage value.

*

Scope:-local to the block or function in which variable is defined.

Life time:-Till the control remains within function or block in which it is defined. It terminates when function is released.

The variable without any storage class specifier is called automatic variable.

Example:-

```
main( )
{
auto    int    i;
printf("i=",i);
}
```

Lecture Note: 19

2. Register storage class

The keyword used to declare this storage class is register. The features are:-

Storage:-CPU register.

Default initial value :-garbage value

Scope :-local to the function or block in which it is defined.

Life time :-till controls remains within function or blocks in which it is defined.

Register variable don't have memory address so we can't apply address operator on it. CPU register generally of 16 bits or 2 bytes. So we can apply storage classes only for integers, characters, pointer type.

*

Variable stored in register storage class always access faster than,which is alwaysstored in the memory. But to store all variable in the CPU register is not possible because of limitation of the register pair.

And when variable is used at many places like loop counter, then it is better todeclare it as register class.

Example:-

```
main( )
{
register int i;
for(i=1;i<=12;i++)
printf("%d",i);
}
```

3 Static storage class

The keyword used to declare static storage class is static. Its feature are:-

Storage:-memory location

Default initial value:- zero

Scope :- local to the block or function in which it is defined.

Life time:- value of the variable persist or remain between different function call.

Example:-

```
main( )
```

*

```
{
reduce( );
reduce( );
reduce ( );
}
reduce( )
{
static int x=10;
printf("%d",x);x++;
}
```

Output:-10,11,12

External storage classes

The keyword used for this class is extern.

Features are:-

Storage:- memory area

Default initial value:-zero

Scope :- global

Life time:-as long as program execution remains it retains.

*

Declaration does not create variables, only it refer that already been created at somewhere else. So, memory is not allocated at a time of declaration and the external variables are declared at outside of all the function.

Example:-

```
int i,j;
void main( )

{
printf( “i=%d”,i );receive( ); receive ( ); reduce( );
reduce( );
}
receive( )
{
i=i+2;
printf(“on increase i=%d”,i);
}
reduce( )
{
i=i-1;
printf(“on reduce i=%d”,i);
}
```

*

Output:-i=0,2,4,3,2.

When there is large program i.e divided into several files, then external variableshould be preferred. External variable extend the scope of variable.

Lecture Note: 20

POINTER

A pointer is a variable that store memory address or that contains address of another variable where addresses are the location number always contains whole number. So, pointer contain always the whole number. It is called pointer because it points to a particular location in memory by storing address of that location.

Syntax-

Data type *pointer name;

Here * before pointer indicate the compiler that variable declared as a pointer.e.g.

int *p1; //pointer to integer type float *p2;

//pointer to float type char *p3; //pointer to

character type

*

When pointer declared, it contains garbage value i.e. it may point any value in the memory.

*

Two operators are used in the pointer i.e. **address operator(&)** and **indirection operator or dereference operator (*).**

Indirection operator gives the values stored at a particular address. Address operator cannot be used in any constant or any expression.Example:

```
void main()
{

int  i=105;  int  *p;
p=&i;
t

printf(“value  of  i=%d”,*p);
printf(“value  of  i=%d”,*/&i);
printf(“address  of  i=%d”,&i);
printf(“address  of  i=%d”,p);
printf(“address of p=%u”,&p);
}
```

Pointer Expression

Pointer assignment int i=10;

int *p=&i;//value assigning to the pointer

*

Here declaration tells the compiler that P will be used to store the address of integer value or in other word P is a pointer to an integer and *p reads the **value atthe address contain in p.**

P++;

printf(“value of p=%d”);

We can assign value of 1 pointer variable to other when their base type and datatype is same or both the pointer points to the same variable as in the array.

Int *p1,*p2;

P1=&a[1];

P2=&a[3];

We can assign constant 0 to a pointer of any type for that symbolic constant‘**NULL**’ is used such as

*p=NULL;

It means pointer doesn’t point to any valid memory location.

Pointer Arithmetic

Pointer arithmetic is different from ordinary arithmetic and it is perform relative tothe data type(base type of a pointer).

Example:-

If integer pointer contain address of 2000 on incrementing we get address of 2002 instead of 2001, because, size of the integer is of 2 bytes.

Note:-

When we move a pointer, somewhere else in memory by incrementing or decrement or adding or subtracting integer, it is not necessary that, pointer still pointer to a variable of same data, because, memory allocation to the variable aredone by the compiler.

*

But in case of array it is possible, since there data are stored in a consecutivemanner.

Ex:-

```
void main( )
{
static int a[ ]={20,30,105,82,97,72,66,102};
int *p,*p1;
P=&a[1];
P1=&a[6];
printf("%d",*p1-*p);
printf("%d",p1-p);
}
```

Arithmetic operation never perform on pointer are:

addition, multiplication and division of two pointer. multiplication between the pointer by any number.division of pointer by any number -add of float or double value to the pointer.

Operation performed in pointer are:-

/* Addition of a number through pointer */

Example

```
int  i=100;  int
*p;
```

*

p=&i;

p=p+2;

p=p+3;

p=p+9;

ii /* Subtraction of a number from a pointer'*/Ex:-

int i=22;

*p1=&a;

p1=p1-10;

p1=p1-2;

iii- Subtraction of one pointer to another is possible when pointer variable point to an element of same type such as an array.

Ex:-

in tar[ ]={2,3,4,5,6,7};

int *ptr1,*ptr1; ptr1=&a[3];

//2000+4 ptr2=&a[6];

//2000+6

Lecture Note: 21

*

Precedence of dereference (*) Operator and increment operator and decrement operator

The precedence level of difference operator increment or decrement operator is same and their associatively from right to left.

Example :- int

x=25; int

*p=&x;

Let us calculate int y=*p++;

Equivalent to *(p++)

Since the operator associate from right to left, increment operator will applied to the pointer p.

i) int y=*p++; equivalent to *(p++)

p =p++ or p=p+1

ii) *++p;→*(++p)→p=p+1

y=*p

iii) int y=++*p

equivalent to ++(*p)

p=p+1 then *p

iv) y=(*p)++→equivalent to *p++ y=*p

then

P=p+1 ;

Since it is postfix increment the value of p.

Pointer Comparison

*

Pointer variable can be compared when both variable, object of same data typeand it is useful when both pointers variable points to element of same array.

Moreover pointer variable are compared with zero which is usually expressed asnull, so several operators are used for comparison like the relational operator.

==,!=,<=,<,>,>=, can be used with pointer. Equal and not equal operators used to compare two pointer should finding whether they contain same address or not andthey will equal only if are null or contains address of same variable.

Ex:-

```
void main()

{
static  int  arr[]={20,25,15,27,105,96} int
*x,*y;
x=&a[5];
y=&(a+5); if(x==y)
printf(“same”);  else
printf(“not”);




}
```

Lecture Note: 22

*

Pointer to pointer

Addition of pointer variable stored in some other variable is called pointer to pointer variable.

Or

Pointer within another pointer is called pointer to pointer.

Syntax:-

Data type **p;int x=22;

int *p=&x; int

**p1=&p;

printf("value of x=%d",x);

printf("value of x=%d",*p);

printf("value of x=%d",*&x);

printf("value of x=%d",**p1);

printf("value of p=%u",&p);

printf("address of p=%u",p1);

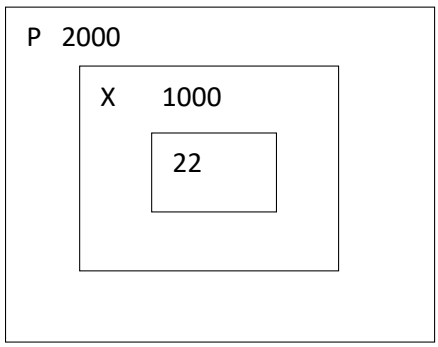
printf("address of x=%u",p);

printf("address of p1=%u",&p1);

printf("value of p=%u",p);

printf("value of p=%u",&x);

p1



*Under revision

3000

Pointer vs array

Example :- void

main()

{

static char arr[]=”Rama”;

char*p=”Rama”;

printf(“%s%s”, arr, p);

In the above example, at the first time printf( ), print the same value array andpointer.

Here array arr, as **pointer to character** and **p act as a pointer to array of character .** When we are trying to increase the value of arr it would give the error because its known to compiler about an array and its base address which is alwaysprinted to base address is known as constant pointer and the base address of array which is not allowed

by the compiler.

printf(“size of (p)”,size of (ar));size of (p) 2/4 bytes

size of(ar) 5 byes

Sructure

It is the collection of dissimilar data types or heterogenous data types grouped together. It means the data types may or may not be of same type.

Structure declaration-

```
struct tagname
{
Data type member1;Data
type member2;Data type
member3;
……
……
Data type member n;
}; OR
struct
{
Data type member1;Data
```

type member2;

Data type member3;

……

……

Data type member n;

}; OR

struct tagname

{

struct element 1;

struct element 2;

struct element 3;

……

……

struct element n;

};

Structure variable declaration;

struct student

{

int age; char name[20]; char branch[20];

}; struct student s;

Initialization of structure variable-

Like primary variables structure variables can also be initialized when they are declared. Structure templates can be defined locally or globally. If it is local it can be used within that function. If it is global it can be used by all other functions of the program.

We cant initialize structure members while defining the structure

```
struct student
{

int age=20;
char name[20]=”sona”;
}s1;
```

The above is **invalid.**

A structure can be initialized as

```
struct student
{

        int age,roll; char
name[20];
} struct student s1={16,101,”sona”}; struct
student s2={17,102,”rupa”};
```

If initialiser is less than no.of structure variable, automatically rest values are taken as zero.

Accessing structure elements-

Dot operator is used to access the structure elements. Its associativety is from left to right.

structure variable ;

s1.name[];

s1.roll;

s1.age;

Elements of structure are stored in contiguous memory locations. Value of structure variable can be assigned to another structure variable of same type using assignment operator.

Example:

```
#include<stdio.h>
#include<conio.h> void
main()
{
int roll, age; char
branch;
} s1,s2;
printf("\n enter roll, age, branch=");
scanf("%d %d %c", &s1.roll, &s1.age, &s1.branch);
s2.roll=s1.roll;
printf(" students details=\n");
printf("%d %d %c", s1.roll, s1.age, s1.branch);
printf("%d", s2.roll);
```

}

Unary, relational, arithmetic, bitwise operators are not allowed within structure variables.

Lecture Note:24

Size of structure-

Size of structure can be found out using sizeof() operator with structure variable name or tag name with keyword.

sizeof(struct student); or

sizeof(s1);

sizeof(s2);

Size of structure is different in different machines. So size of whole structure may not be equal to sum of size of its members.

Array of structures

When database of any element is used in huge amount, we prefer Array of structures.

Example: suppose we want to maintain data base of 200 students, Array of structures is used.

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

```
char name[30]; char
branch[25];int roll;
};
void main()
{
struct student s[200];int i;
s[i].roll=i+1;
printf("\nEnter information of students:");
for(i=0;i<200;i++)
{
printf("\nEnter the roll no:%d\n",s[i].roll);
printf("\nEnter the name:");
scanf("%s",s[i].name);
printf("\nEnter the branch:");
scanf("%s",s[i].branch);
printf("\n");
}
printf("\nDisplaying information of students:\n\n");
for(i=0;i<200;i++)
{
printf("\n\nInformation for roll no%d:\n",i+1);
```

```
printf("\nName:");
puts(s[i].name);     printf("\nBranch:");
puts(s[i].branch);
}
}
```

In Array of structures each element of array is of structure type as in aboveexample.

Array within structures

```
struct student
{
char name[30];
int roll,age,marks[5];
}; struct student s[200];
```

We can also initialize using same syntax as in array.

Nested structure

When a structure is within another structure, it is called Nested structure. A structure variable can be a member of another structure and it is represented as

```
struct student
```

{

element 1;

element 2;

……

……

struct student1

{

member 1;

member 2;

}variable 1;

…….

…….

element n;

}variable 2;

It is possible to define structure outside & declare its variable inside other structure.

struct date

{

int date,month;

};

struct student

{

```
 char  nm[20];  int
roll; struct date d;
};  struct  student  s1;  struct
student s2,s3;
```

Nested structure may also be initialized at the time of declaration like in aboveexample.

```
struct student s={“name”,200, {date, month}};
{“ram”,201, {12,11}};
```

Nesting of structure within itself is not valid. Nesting of structure can beextended to any level.

```
struct time
{
int hr,min;
};
struct day
{
int       date,month;
struct time t1;
};
struct student
```

```
{
char        nm[20];
struct day d;
}stud1, stud2, stud3;
```

Lecture Note: 25

Passing structure elements to function

We can pass each element of the structure through function but passing individual element is difficult when number of structure element increases. To overcome this, we use to pass the whole structure through function instead of passing individual element.

```
#include<stdio.h>
#include<string.h> void
main()
{
struct student
{
char name[30]; char
branch[25];int roll;
}struct student s; printf("\n
enter name=");
```

```
gets(s.name);       printf("\nEnter
roll:");        scanf("%d",&s.roll);
printf("\nEnter           branch:");
gets(s.branch);
display(name,roll,branch);
}
display(char name, int roll, char branch)
{
printf("\n name=%s,\n roll=%d, \n branch=%s", s.name, s.roll. s.branch);
}
```

Passing entire structure to function

```
#include<stdio.h> #include<string.h>
struct student
{
char  name[30];  int
age,roll;
};
display(struct student);                    //passing entire structure
void main()
```

```
{
struct student s1={"sona",16,101 }; struct
student s2={"rupa",17,102 };
display(s1);
display(s2);
}
display(struct student s)
{
printf("\n name=%s, \n age=%d ,\n roll=%d", s.name, s.age, s.roll);
}
```

Output: name=sona
roll=16

Lecture Note: 26

UNION

Union is derived data type contains collection of different data type or dissimilar elements. All definition declaration of union variable and accessing member is similar

to structure, but instead of keyword struct the keyword union is used, the main difference between union and structure is

Each member of structure occupy the memory location, but in the unions members share memory. Union is used for saving memory and concept is usefulwhen it is not necessary to use all members of union at a time.

Where union offers a memory treated as variable of one type on one occasion where (struct), it read number of different variables stored at different place ofmemory.

Syntax of union:

union student

{

datatype member1;

datatype member2;

};

Like structure variable, union variable can be declared with definition or separatelysuch as

union union name

{

Datatype member1;

}var1;

Example:- union student s;

Union members can also be accessed by the dot operator with union variable and ifwe have pointer to union then member can be accessed by using (arrow) operator as with structure.

Example:- struct student

struct student

{

int i;

char ch[10];

};struct student s;

Here datatype/member structure occupy 12 byte of location is memory, where as in the union side it occupy only 10 byte.

Lecture Note:27

Nested of Union

When one union is inside the another union it is called nested of union.

Example:-

union a

{

int i; int

age;

};

union b

```
{
char        name[10];
union a aa;
}; union b bb;
```

There can also be union inside structure or structure in union.

Example:-

```
void
main()
{
struct a
{
int i;
char ch[20];
};
struct  b
{
int i;
char d[10];
};
union z
{
struct     a     a1;
struct b b1;
```

```
}; union z z1;
z1.b1.j=20; z1.a1.i=10;
z1.a1.ch[10]= " i";
z1.b1.d[0]="j ";
printf(" ");
```

Dynamic memory Allocation

The process of allocating memory at the time of execution or at the runtime, is called dynamic memory location.

Two types of problem may occur in static memory allocation.

If number of values to be stored is less than the size of memory, there would be wastage of memory.

If we would want to store more values by increase in size during the execution on assigned size then it fails.

Allocation and release of memory space can be done with the help of some library function called dynamic memory allocation function. These library function are called as **dynamic memory allocation function.** These library function prototype are found in the header file, "alloc.h" where it has defined.

Function take memory from memory area is called heap and release when notrequired.

Pointer has important role in the dynamic memory allocation to allocate memory.

malloc():

This function use to allocate memory during run time, its declaration is void*malloc(size);

malloc ()

returns the pointer to the 1st byte and allocate memory, and its return type is void, which can be type cast such as:

int *p=(datatype*)malloc(size)

If memory location is successful, it returns the address of the memory chunk that was allocated and it returns null on unsuccessful and from the above declaration apointer of type**(datatype)** and size in byte.

And **datatype** pointer used to typecast the pointer returned by malloc and this typecasting is necessary since, malloc() by default returns a pointer to void.

Example int*p=(int*)malloc(10);

So, from the above pointer p, allocated IO contigious memory space address of 1stbyte and is stored in the variable.

We can also use, the size of operator to specify the the size, such as

p=(int)malloc(5*size of int) Here, 5 is the no. of data.

Moreover , it returns null, if no sufficient memory available , we should always check the malloc return such as, **if(p==null)**

printf(“not sufficient memory”);

Example:

/*calculate the average of mark*/void

main()

{

int n , avg,i,*p,sum=0;

```
printf("enter the no. of marks ”);
scanf(“%d”,&n);
p=(int *)malloc(n*size(int));
if(p==null)
printf(“not sufficient”);
exit();
}
for(i=0;i<n;i++)
scanf(“%d”,(p+i));
for(i=0;i<n;i++)
Printf(“%d”,*(p+i));
sum=sum+*p; avg=sum/n;
printf(“avg=%d”,avg);
```

Lecture Note: 28

calloc()

Similar to malloc only difference is that calloc function use to allocate multiple block of memory .

two arguments are there

1st argument specify number of blocks

2nd argument specify size of each block.

Example:-

int *p= (int*) calloc(5, 2);

int*p=(int *)calloc(5, size of (int));

Another difference between malloc and calloc is by default memory allocated by malloc contains garbage value, where as memory allocated by calloc is initialised by zero(but this initialisation) is not reliable.

realloc()

The function realloc use to change the size of the memory block and it alter the size of the memory block without loosing the old data, it is called reallocation ofmemory.

It takes two argument such as; int

*ptr=(int *)malloc(size);

int*p=(int *)realloc(ptr, new size);

The new size allocated may be larger or smaller.

If new size is larger than the old size, then old data is not lost and newly allocated bytes are uninitialized. If old address is not sufficient then starting address contained in pointer may be changed and this reallocation function moves content of old block into the new block and data on the old block is not lost.

Example:

#include<stdio.h>

#include<alloc.h> void

main()

int i,*p;

```
p=(int*)malloc(5*size of (int));
if(p==null)
{
printf("space not available");
exit();
printf("enter 5 integer");
for(i=0;i<5;i++)
{
scanf("%d",(p+i));
int*ptr=(int*)realloc(9*size of (int) );
if(ptr==null)
{
printf("not available");
exit();
}
printf("enter 4 more integer");
for(i=5;i<9;i++)
scanf("%d",(p+i));
for(i=0;i<9;i++)
printf("%d",*(p+i));
}
```

free()

Function free() is used to release space allocated dynamically, the memory released by free() is made available to heap again. It can be used for further purpose.

Syntax for free declaration .

void(*ptr)

Or

free(p)

When program is terminated, memory released automatically by the operating system.

Even we don't free the memory, it doesn't give error, thus lead to memoryleak.

We can't free the memory, those didn't allocated.

Lecture Note: 29

Dynamic array

Array is the example where memory is organized in contiguous way, in the dynamic memory allocation function used such as malloc(), calloc(), realloc() always made up of contiguous way and as usual we can access the element in twoways as:

Subscript notation

Pointer notation

Example:

```
#include<stdio.h>
#include<alloc.h> void
main()
{
printf(“enter the no.of values”);
scanf(“%d”,&n);
p=(int*)malloc(n*size of int);
If(p==null)
printf(“not available memory”);
exit();
}
for(i=0;i<n;i++)
{
printf(“enter an integer”);
scanf(“%d”,&p[i]);
for(i=0;i<n;i++)
{
printf(“%d”,p[i]);
}
}
```

File handling

File: the file is a permanent storage medium in which we can store the data permanently.

Types of file can be handled

we can handle three type of file as

(1) sequential file

(2) random access file

(3) binary file

File Operation

opening a file:

Before performing any type of operation, a file must be opened and for this fopen() function is used.

syntax:

file-pointer=fopen("FILE NAME ","Mode of open");

example:

FILE *fp=fopen("ar.c","r");

If fopen() unable to open a file than it will return NULL to the file pointer.

File-pointer: The file pointer is a pointer variable which can be store the addressof a special file that means it is based upon the file pointer a file gets opened.

Declaration of a file pointer:-

FILE* var;

Modes of open

The file can be open in three different ways as

Read mode ' r'/rt Write mode 'w'/wt

Appened Mode 'a'/at

Reading a character from a file

getc() is used to read a character into a file

Syntax:

character_variable=getc(file_ptr);

Writing acharacter into a file

putc() is used to write a character into a file

puts(character-var,file-ptr);

ClOSING A FILE

fclose() function close a file.

fclose(file-ptr);

fcloseall () is used to close all the opened file at a time

File Operation

The following file operation carried out the file

(1)creation of a new file

(3)writing a file

(4)closing a file

Before performing any type of operation we must have to open the file.c, language communicate with file using A new type called **file pointer**.

Operation with fopen()

File pointer=fopen(“FILE NAME”,”mode of open”);

If **fopen()** unable to open a file then it will return **NULL** to the file-pointer.

Lecture Note: 30

Reading and writing a characters from/to a filefgetc() is

used for reading a character from the file

Syntax:

character variable= fgetc(file pointer);

fputc() is used to writing a character to a file

Syntax:

fputc(character,file_pointer);

```
/*Program to copy a file to another*/
#include<stdio.h>
void main()
{
FILE *fs,*fd; char
ch;
If(fs=fopen("scr.txt","r")==0)
{
printf("sorry….The source file cannot be opened");
return;
}
If(fd=fopen("dest.txt","w")==0)
{
printf("Sorry…..The destination file cannot be opened");
fclose(fs);
return;
}
while(ch=fgets(fs)!=EOF)fputc(ch,fd);
fcloseall();
}
```

Reading and writing a string from/to a file getw() is used for reading a string from the file

Syntax:

gets(file pointer);

putw() is used to writing a character to a file

Syntax:

fputs(integer,file_pointer);

```
#include<stdio.h> #include<stdlib.h>
void main()
{
FILE *fp; int
word;
/*place the word in a file*/
fp=fopen("dgt.txt","wb");
If(fp==NULL)
{
printf("Error opening file");
exit(1);
}
word=94;
putw(word,fp);
If(ferror(fp))
```

```
printf("Error writing to file\n");else
printf("Successful write\n");
fclose(fp);
/*reopen the file*/
fp=fopen("dgt.txt","rb");
If(fp==NULL)
{
printf("Error opening file");
exit(1);
}


/*extract the word*/
word=getw(fp);
If(ferror(fp))
printf("Error reading file\n");else
printf("Successful read:word=%d\n",word);
/*clean up*/
fclose(fp);
}
```

Lecture Note: 31

Reading and writing a string from/to a file

fgets() is used for reading a string from the file

Syntax:

fgets(string, length, file pointer);

fputs() is used to writing a character to a file

Syntax:

fputs(string,file_pointer);

```
#include<string.h> #include<stdio.h>
void main(void)
{
FILE*stream;
char string[]=”This is a test”;char
msg[20];
/*open a file for update*/ stream=fopen(“DUMMY.FIL”,”w+”);

/*write a string into the file*/ fwrite(string,strlen(string),1,stream);
/*seek to the start of the file*/
fseek(stream,0,SEEK_SET);
```

```
/*read a string from the file*/ fgets(msg,strlen(string)+1,stream);
/*display the string*/
printf("%s",msg);
fclose(stream);
}
```


BOOKS:

1 E.Balagurusamy "Programming in C". Tata McGraw Hill2 Y. Kanetkar "Let Us C". BPB publication

3 Ashok N. Kamthane "Programming with ANSI and TURBO C". Pearson Education

4 Programming in C, a complete introduction to the programming language, Stephan G. Kocham, third edition

5 C in Depth, S.K Srivastava and Deepali Srivastava