CS25C01. Computer Programming: C

Problem Solving:: Introduction to C

* Problem Solving is a systematic approach to define a problem and create multiple possible solutions.
* It involves logical thinking, planning and structured execution.
* In programming, it means converting a real-world task into a working program.
* The process starts with problem specifications and ends with a correct tested program

Problem Solving process::

Stage	Description
Problem Specification	Understand the task, inputs and expected outputs
Analysis	Identify constraints, edge cases and logic needed
Algorithm Development	Design step-by-step instructions
Flowchart & Pseudocode	Visualize and write logic in readable format
Coding in C	Translate logic into C syntax
Testing and Debugging	Run, verify and fix errors.

Steps for problem Solving::

There are six simple steps to solve

problem effectively.

1) Identify the problem:-

The first step is to figure out what the problem is.

Eg: You are feeling hungry in the evening. The problem is: what should I eat? If you don't know the actual problem, you can't solve it.

2) Understand the problem:-

This step is all about getting the full picture. Before solving, need to understand the situation, including what you know and what the people involved know and the limits.

Eg: If you want to cook, do you have the ingredients at home?

3) Identify Alternatives (possible solutions):-

List all possible ways to solve the problem. Think outside the box and don't rule anything out.

Eg: If you are hungry, your alternatives could be:

a) Cook noodles at home
b) order a pizza

c) Eat fruits
d) Go out to a restaurant.

4) Select the Best Solution:

Look at the pros and cons of each option based on what is most important

Eg: If you are trying to choose what to eat when you are hungry, you might think:
a) Pizza may taste good, but it costs more
b) Fruits are healthy but may not be very filling
c) Cooking noodles is cheap and quick.

5) List the steps to implement the Solution:

Once have Solution, create a step-by-step plan to make it happen. These instructions should be clear and simple enough for everyone involved to understand

Eg: (Cooking noodles): Your plan might be:
a) Boil water
b) Add noodles and spice packet
c) Cook for 5 minutes
d) Serve and Eat

6) Evaluate the Solution:

After trying it, check whether the solution worked.

Eg: After eating noodles, ask yourself: Am I full and satisfied?

a) If yes → problem solved

b) If no → Try another option (may be next time order pizza

Problem Analysis Chart:

* A problem analysis chart is a structured tool used to break down a problem into key components.

* It helps to systematically understand the problem, identify necessary data and develop a structured approach to solve it.

Key components include:

* Inputs
* processes
* outputs
* Alternative solutions

Identifier	UseCase Example	Input	output
Given Data	Radius of the circle	Radius (r)	Area (A)
Processing	Formula for area	r	$A = \pi r^2$
Required Result	Calculate the area	Radius (r)	Area ($A = \pi r^2$)
Solution Alternatives	use different units	inches, metres	Area in given unit

Eg: Find area of a rectangle

Input	Process	output
Length, Breadth	area = length * breadth	area

Developing an Algorithm:

An algorithm is a step-by-step procedure or set of rules used to solve a problem.

Five Essential steps of an algorithm:

1) Define the problem
2) Analyze the problem and gather requirements
3) Design the Algorithm
4) Implement the Algorithm
5) Test and optimize the algorithm.

Properties of Algorithms:

1) Should be written in simple English
2) Should be precise and unambiguous
3) Should conclude after a finite number of steps
4) Should have a start point and endpoint
5) Derived results should be obtained only after the algorithm terminates.

Qualities of a good Algorithm:

1) **Time:**
The lesser is the time required, the better is the algorithm.

2) **Memory:**
The lesser is the memory required, the better is the algorithm.

3) **Accuracy:**
Multiple algorithms may provide suitable or correct solutions to a given problem, some of these may provide more accurate results than others and such algorithms may be suitable.

Eg: Find area of a Rectangle

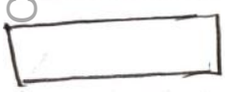
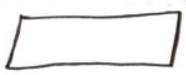
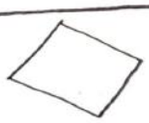
Area of Rectangle (A) = length (l) * breadth (b)

Step 1: Start
Step 2: Get the l and b values
Step 3: Calculate A = l * b
Step 4: Display A
Step 5: Stop

Developing a Flowchart:

* Flowchart is defined as a graphical representation of the logic for problem solving.
* It is used to design and document simple processes or programs.

Building Blocks of flowchart:

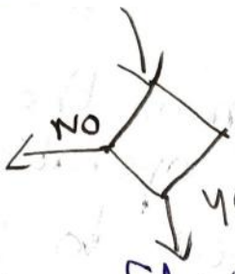
Symbol	Name	Description
	Flow Lines	Shows the process's order of operation
	Terminal	Used to start, pause or halt in the program logic
	processing	Represent the arithmetic and logic instructions
	Input/output	Indicates the process of inputting and outputting data
	Decision	Shows a conditional operation that determines which one of the two paths the program will take
	Connector	Used to join different flow lines
	Sub Function	Used to call functions

Rules for drawing a flowchart:

1) The flowchart should be clear, neat and easy to follow.

2) The flowchart must have a logical & start and finish.
3) Only one flow line should come out from a process symbol.

4) only one flow line should enter a decision symbol. However two or three flow lines may leave the decision symbol.



5) only one flow line is used with a terminal symbol.

6) Within Standard symbols, write briefly and precisely.
7) Intersection of flow lines should be avoided.

Advantages of Flowchart:-

1) Flowcharts are better way of communicating the logic of a system to all concerned.
2) With the help of flowchart, problem can be analyzed in more effective way.
3) program flowcharts serve as a good

documentation which is needed for various purposes.

4) The flowcharts act as a guide or blueprint during the system analysis and program development phase.

5) The flowchart helps in debugging process and maintenance is easy.

Disadvantages of Flowchart:

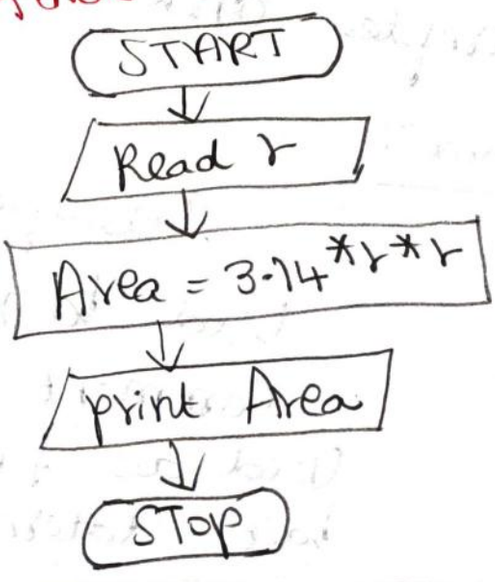
1) If the program logic is quite complicated, flowchart becomes complex and clumsy.

2) If alterations are required the flowchart may require re-drawing completely.

3) As the flowchart symbols cannot be typed, reproduction of flowchart becomes a problem.

4) For large applications, the time and cost of flowchart drawing becomes costly.

Eg: To find area of the circle



Developing a pseudocode:

* A pseudocode is defined as a step-by-step description of an algorithm.
* It uses the structural conventions of a programming language but is intended for human reading rather than machine reading.
* It is the intermediate state between an idea and its implementation in a high-level language.

Algorithm	pseudocode	program

Guidelines for writing pseudocode:

1) Write one statement per line.
2) Use capitalized keywords
3) Use indent to show hierarchy.
4) End multiline structure clearly.
5) Avoid language specific syntax.
6) Keep it simple and human readable.

Common keywords:

Keyword	Description
//	Used to represent a Comment
BEGIN, END	Used as first and last statements

INPUT, GIET, READ	Used for inputting data
COMPUTE, CALCULATE	Used for Calculation of result
ADD, SUBTRACT	Used for addition and Subtraction
OUTPUT, PRINT, DISPLAY	Used for outputting the data
IF, ELSE, ENDIF	Used to make decision
WHILE, ENDWHILE, FOR, ENDFOR	Used for iterative Statements

Algorithm Vs pseudocode

Algorithm	pseudocode
It is a step by step procedure to solve a problem	It is a step by step description of an algorithm.
There are no rules.	There are certain rules to be followed
It uses simple english words	It uses reserved keywords
Focused on solving a problem	Focused on how to implement a solution
Does not include error handling explicitly	Can include error checks

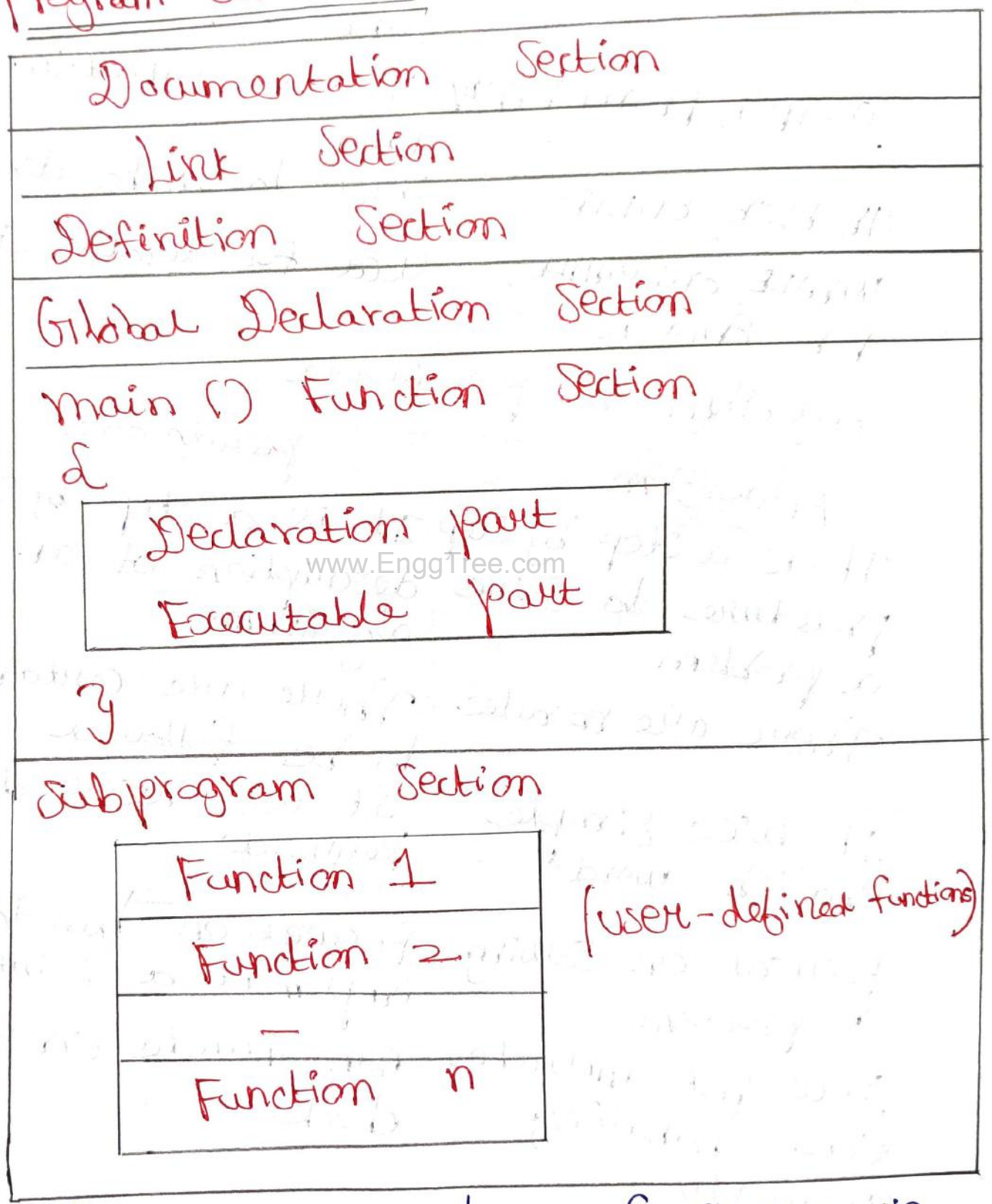
eg: pseudocode to find area of the circle

```
BEGIN
    READ radius
```

COMPUTE area as 3.14*radius*radius
DISPLAY area

END

Program Structure:



* The structure of a C program is divided into 6 parts which makes it easy to read, modify, document and

/understand in a particular format. 13

* C program must follow the outlines in order to successfully compile and execute.
* Debugging is easier in a well-structured C program.

1) Documentation Section:

* The documentation section consists of set of comment lines giving the name of the program, the author and other details which the programmer would like to use later.
* Anything written as comments will be treated as documentation of the program and this will not interfere with the given code.

// filename, authorname, description etc.,
(or)
/* filename, authorname, description etc.,*/

2) Link Section:

* The link section provides instructions to the compiler to link functions from the system library.
* All the header files of the program will be declared here.

* Header files help us to access other's improved code into our code.

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

3) Definition Section:-

* The definition section defines all symbolic constants.
* The #define preprocessor is used to create a constant throughout the program.
* Whenever this name is encountered by the compiler, it is replaced by the actual piece of defined code.

```
#define number 10
```

4) Global Declaration Section:-

* There are some variables that are used in more than one function.
* Such variables are called global variables and are declared in the global declaration section that is outside of all the functions.
* This section also declares all the user-defined functions.
* Variables and functions which are declared in this scope can be used

anywhere in the program.

```
int number =18;
```

5) Main() Function Section:

* Every C program must have one main() function section.
* This section contains declaration part and executable part.
* The declaration part declares all the variables used in the executable part.
* There is atleast one statement in the executable part.
* These two parts must appear between the opening and closing braces.
* The program execution begins at the opening brace and ends at the closing brace.
* The closing brace of the main function is the logical end of the program.
* All statements in the declaration and executable parts end with a semicolon.
* Void main() tells the compiler that the program will not return any value.
* int main() tells the compiler that the program will return an integer value.

6) Subprogram Section:

* It contains all the user-defined functions that are called in the main function.
* User-defined functions are generally placed immediately after the main function, although may appear in any order.

```
int sum (int x, int y)
{
    return x+y;
}
```

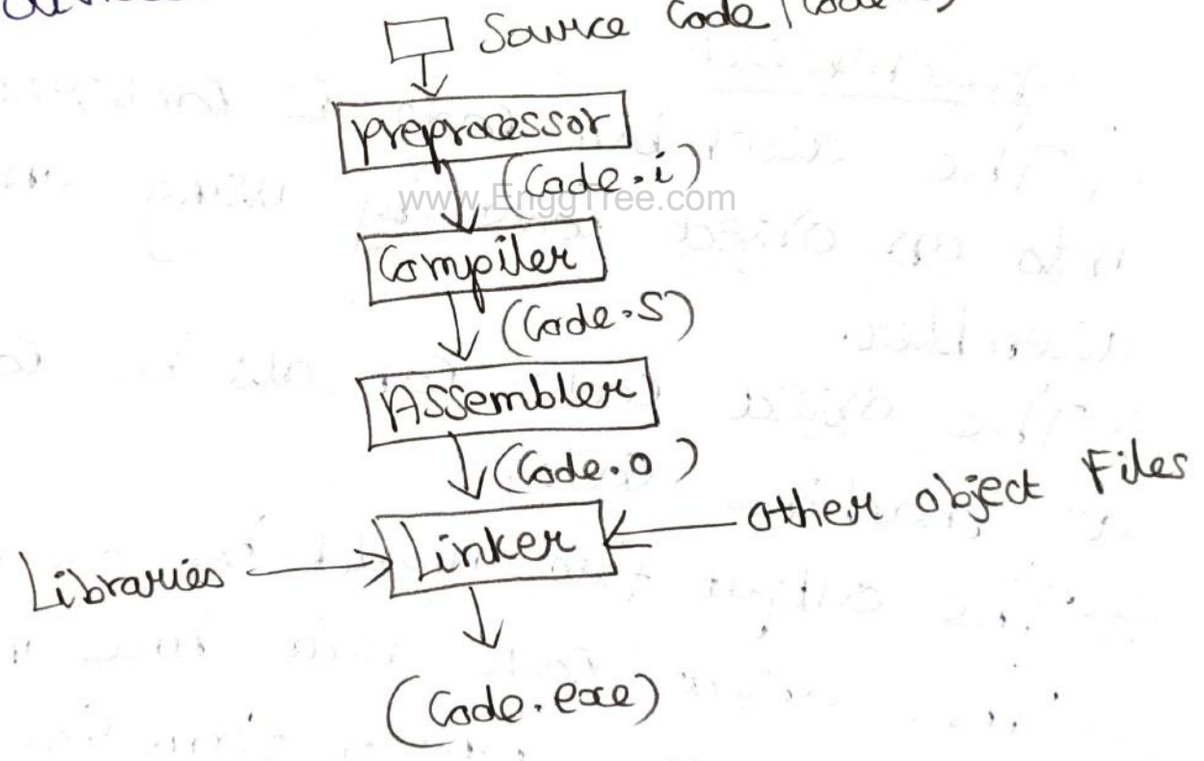

* All sections, except the main function Section may be absent when they are not required.
* Main Function Section is mandatory in any C program.

Eg:

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

Compilation and Execution process:

* The compilation is a process of converting the source code into object code or machine code.
* The compiler checks the source code for errors and if the source code is error-free then it generates the object code.
* The compilation process can be divided into 4 steps.



1) Preprocessor:

* Removal of comments
* Expansion of macros
* File inclusion
* Conditional compilation

The pre-processed output is stored in

the filename.i

2) Compiler:-

* The code which is expanded by the preprocessor is passed to the compiler.
* The compiler converts this code into assembly code.
* The output file would be Code.S
* It also checks for syntax errors.

3) Assembler:-

* The assembly code is converted into an object code by using an assembler.
* The object code can also be called as machine code.
* The output file would be Code.O
* The object code will have instructions represented in binary form consisting of sequences of 0s and 1s, which is the only language the processor can directly understand.

4) Linker:-

The linker combines the object code

of library files with the object code of program.

* It allocates memory space for various code and data sections.
* It produces the final file that the computer can execute.

Execution:

* Once the compilation process successfully generates an executable file, the program can run.
* Loading: The operating system loads the executable file into the computer's memory.
* Execution: The CPU then begins executing the machine code instructions within the loaded program.

Interactive and Script mode:

* In interactive mode, the program prompts the user for input during runtime and responds immediately.

Eg: A calculator program that asks for numbers and operations one by one.

* C doesn't have a native script mode like python, but can simulate it

by writing a full program and compiling it before execution.

* In script mode, need to write the program in .C file, compile it using a compiler and then run the .exe file.

Comments:

* Comments are non-executable texts.
* They are ignored by the compiler so they don't affect the program output.
* Comments are used for documentation, reminders or temporarily disabling code.

Types of Comments in C:

1) Single-line Comments
2) Multi-line Comments

1) Single-line Comments:

A Single-line comment is used to annotate a single line of code, or provide brief explanations. It begins with two forward slashes (//) and everything following // on that line is treated as a Comment.

2) Multi-line Comments:

A multi-line comment spans

across multiple lines and is used for detailed documentation or to comment out blocks of code. It begins with (/*) and ends with (*/). All text between these markers is treated as a comment.

Indentation:

* Indentation in C refers to the practice of formatting code by adding consistent horizontal spacing (using tabs or spaces) to visually separate blocks of code.
* While indentation is not syntactically required in C, it is a critical coding convention that enhances readability, maintainability and debugging efficiency.

Purpose of Indentation:

* Improves readability by visually organizing code structure.
* Clarifies logical flow, especially in nested control structures.
* Facilitates collaboration, making code easier to understand for other developers.
* Supports debugging and maintenance by clearly showing block boundaries.

Common Indentation practices:

* Use Consistent Spacing (typically 4 spaces or 1 tab per level)
* Indent Code inside:
  - → Control Structures (if, else, for, while, switch)
  - → Function Definitions
  - → Loops and nested blocks
* Align opening { and closing } braces properly to reflect block hierarchy.

Eg:

```
#include <stdio.h>
int main()
{
    int a=10, b=20;
    if(a>b)
    {
        printf("a is greater than b\n");
    }
    else
    {
        printf("b is greater than a\n");
    }
    return 0;
}
```

Error Messages:

* In C programming error messages are diagnostic outputs that inform the programmer about issues encountered during compilation or runtime execution.
* These messages help identify syntax errors, logical mistakes or system-level failures, enabling effective debugging and correction.
* Error messages often include the file name, line number and a brief description of the issue, which helps programmers quickly locate and resolve the problem within the source code.

Types of Errors in C:

Error Type	Description
Compile time Error	occurs during compilation due to syntax violations, undeclared variables, etc
Linker Error	Happens when external references (Eg., functions or libraries) are unresolved
Runtime Error	Arises during program execution (Eg., Division by zero, file not found)
Logical Error	Produces incorrect output due to flawed logic, even though the program runs.

Eg C program for interactive mode

```
#include<stdio.h>
int main()
{
    int a,b;
    printf("Enter two numbers:");
    scanf("%d %d", &a, &b);
    printf("Sum = %d\n", a+b);
    return 0;
}
```

output:

```
Enter two numbers: 2
                   3
        Sum = 5
```

Eg: C Program for Error messages.

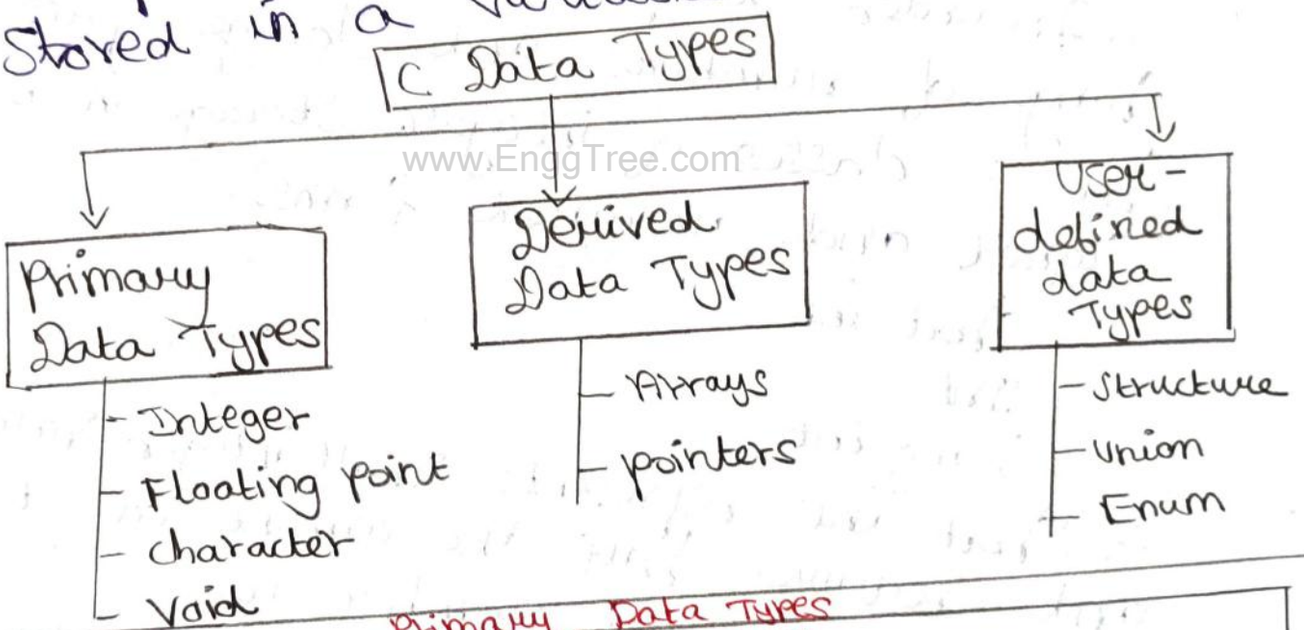
```
#include<stdio.h>
//int add(int a, int b);   (Add this line as // comment, then Linker Error)
int main()
{
    int x=10, y=0;   (change y value as 3, & result change a+b as a-b then, logical error bcoz it did subtraction but did addition)
    int result = add(x,y);
    printf("Result: %d\n", result);
    return 0;   (Remove ; then Compile time Error)
}
int add(int a, int b)
{
    return a+b;   (change Add as Div & + symbol as /, then runtime Error)
}
```

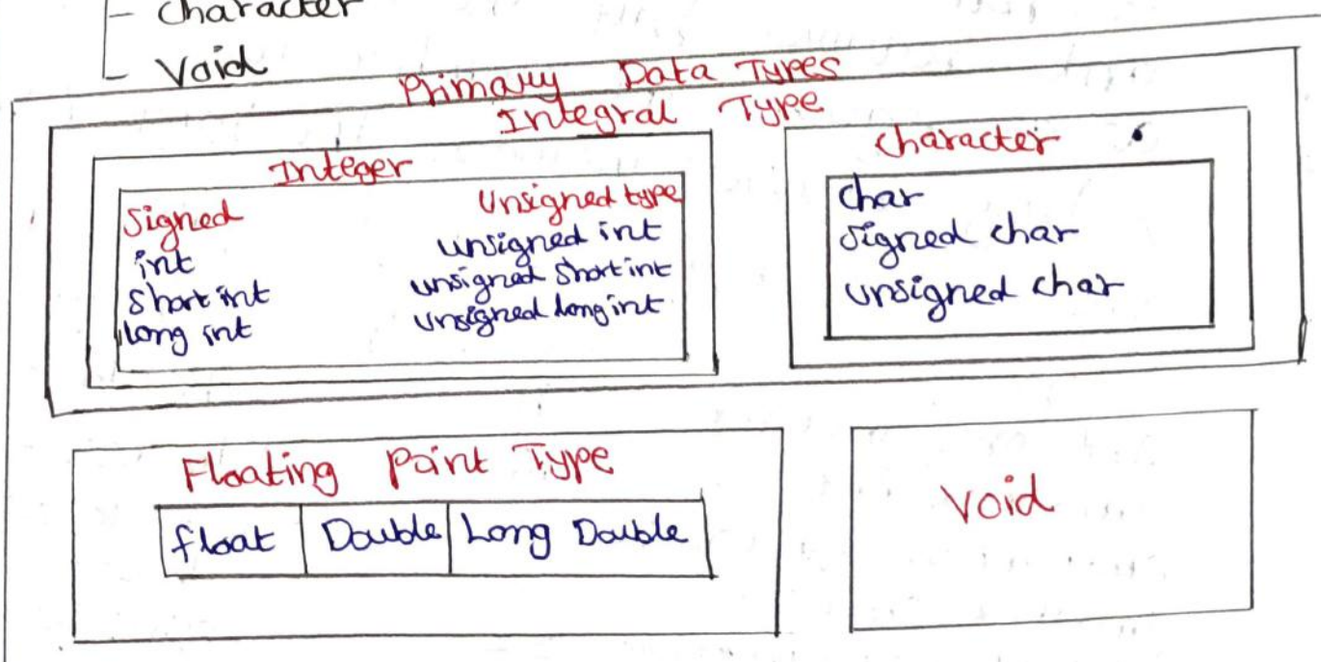
output:

```
Result: 10
```

Primitive Data Types:

* It refers to the size and type of data associated with variables and functions.
* It is a classification that specifies
  → What kind of value a variable can store
  → How much memory it occupies
  → What operations can be performed on it
* It is like a label that tells the compiler how to interpret the data stored in a variable.





1) Integer Type:

* Integers are whole numbers.
* The size of an integer that can be stored depends upon the compiler.
* For a 16 bit word length, the integer value range is limited to -32768 to +32767
* For a 32 bit word length, the integer value range is limited to -2,147,483,648 to 2,147,483,647
* In order to provide control over the range of numbers and storage space, C has 3 classes of integer storage in both signed and unsigned forms:
  - → Short int
  - → Int
  - → Long int
* Short int represents small integer values and requires half the amount of storage as a regular int number uses.
* Long integers increase the range of values.

Type and Range of integer data Types:

Type	Size(bits)	Range
Int or signed int	16	-32768 to 32767
Unsigned int	16	0 to 65535
Short int or signed short int	8	-128 to 127
Unsigned short int	8	0 to 255
Long int or unsigned long int	32	-2,147,483,648 to 2,147,483,648
Unsigned long int	32	0 to 4,294,967,295

Syntax:

```
int <Variable name>;
Short int <Variable name>;
long int <Variable name>;
```

Eg:

```
int a;
Short int b;
long int c;
```

2) Floating point Type:

* It is used to store real numbers with 6 digits of precision.
* It is defined by the keyword float
* When the accuracy provided by a float number is not sufficient, the type double can be used to define the number.
* The double data type represents the same data type that float represents but with a greater precision.
* To extend the precision further, may use long double uses 80 bits.

Size and range of floating point Types:

Type	Size (bits)	Range
float	32	3.4E -38 to 3.4E+38
double	64	1.7E-308 to 1.7E+308
long double	80	3.4E-4932 to 1.1E+4932

Syntax:

float <Variable name>;
double <Variable name>;
long double <Variable name>;

3) Character Type:

* A single character can be defined as a character (char) type data.
* Characters are usually stored in 8 bits of internal storage.
* The qualifier signed or unsigned may be explicitly applied to char.

Syntax:

char <Variable name>;

Eg:

```
char ch = 'J';
char ab = '?';
```

Size and range of character Type:

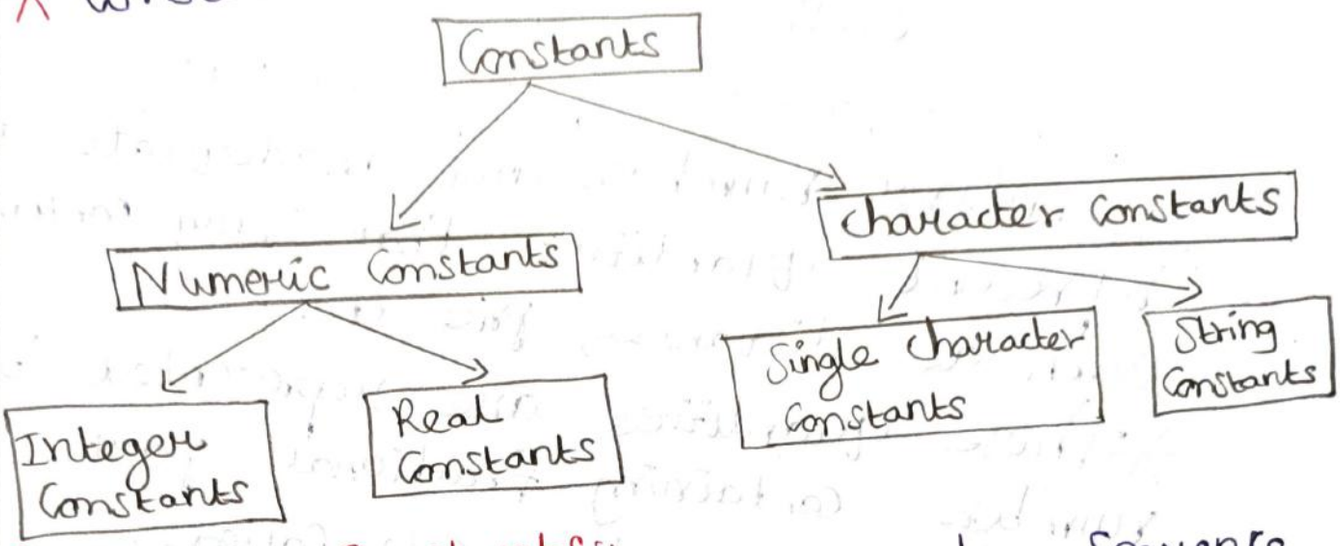
Type	Size(bits)	Range
char	8	-128 to 127
Signed char	8	-128 to 127
Unsigned char	8	0 to 255

4) Void Type:

* The void type has no values.
* It is usually used to specify the type of functions.
* It cannot be used to declare a variable.
* The type of function is said to be void when it doesn't return any value to calling function.

Constants:

* Constants refer to fixed values that do not change during the execution of the program.
* Constants are also called as literals.



i) Integer Constants:

* An integer constant refers to a sequence of digits.
* There are 3 types of integers:

i) Decimal integer:

It consists of a set of digits, 0 through 9, preceded by an optional + or − sign.

Eg: 123
321
654321
0

ii) Octal integer:

It consists of any combination of digits from the set 0 through 7, with a leading 0.

Eg: 037
0
0435
0551

iii) Hexadecimal integer:

A sequence of digits preceded by 0x or 0X is considered. It may also include alphabets A-F or a-f.

Eg: 0X2
0X9F

2) Real (floating point) Constants:

* Integer numbers are inadequate to represent quantities that vary continuously such as distances, price etc.
* These quantities are represented by numbers containing fractional parts like 17.548. Such numbers are called Real Constants

Eg: 0.0083
-0.75

* These numbers are shown in decimal notation, having whole number followed by a decimal point and the fractional part.
* It is possible to omit digits before the decimal point or digits after the decimal point.

Eg: 215.
.95

* A real number may also be expressed in exponential notation (e or E)

Eg: 215.65 may be written as 2.1565e2

3) Single character Constants:

* A single character constant contains a single character enclosed within a pair of single quote marks.

Eg: '5'
'X'
'?'
' '

* C also supports some backslash character constants that are used in output functions.

Eg: '\n'
'\t'
'\0'

4) String Constants:

* A string constant is a sequence of characters enclosed in double quotes.
* The characters may be letters, numbers, special characters and blank space.

Eg: "Hello"
"CSE"
"1947"
"172"
"a"

Variables:

* A variable is a data name that may be

used to store a data value.

* Unlike constants that remain unchanged during the execution of the program, a variable may take different values at different times during execution.
* A variable name can be chosen by the programmer in a meaningful way so as to reflect its nature in the program

Eg: Average
height
Total

Rules for declaring a variable:

The variable names may consist of letters, digits and the underscore (_) character, subject to the following conditions:

1) They must begin with a letter.
2) ANSI standard recognizes a length of 31 characters, however length should not be more than 8 characters.
3) Uppercase and lowercase are significant (Total != TOTAL)
4) It should not be a keyword.
5) Whitespace is not allowed.

Eg: // Addition of 2 numbers

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

```
{
    int number 1 = 10, number 2 = 3;
    printf ("Result : %d \n", number 1);
    return 0;
}
```

Output:

Result : 10

Eg:
```
#include <stdio.h>
int main()
{
    int number 1 = 10, NUMBER 1 = 3;
    printf ("result: %d\n", NUMBER 1);
    return 0;
}
```


Output:

Result : 3

Eg:

int 0number 1 = 10, NUMBER 1 = 3; → Error (zero)

int number 1 = 10, NUMB ER 1 = 3; → Error (Gap)

Eg:

int number 1 = 10, NUMB_ER 1 = 3; (underscore)

printf ("Result: %d\n", number 1);

Output:

Result : 10

Reserved words:

* Reserved words are also called as Keywords.
* All keywords have fixed meanings and these meanings cannot be changed.
* Keywords serve as basic building blocks for program statement.
* All keywords must be written in lowercase.
* Keywords are part of the syntax and they cannot be used as an identifier or variable.

Keywords:

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

Eg:

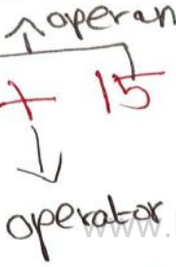
```
int int =10, NUMB_ER1 =3;        // Error
int if =10, NUMB_ER1 =3;         // Error
int number1 =10, void =3;        // Error
```

From this eg, keywords can't be used as a variable.

Operators:

* An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations.
* operators are used in programs to manipulate data and variables.
* They usually form a part of the mathematical or logical expressions.
* An expression is a sequence of operands and operators that reduces to a single value.

Eg:



Types of operators:

1) Arithmetic operators:

The operator +, -, *, / and % are the arithmetic operators supported by C.

operator	meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo Division

Eg:
```
#include <stdio.h>
main()
{
    int a=10, b=5;
    printf("Addition = %d", a+b);
    printf("Subtraction = %d", a-b);
    printf("Multiplication = %d", a*b);
    printf("Division = %d", a/b);
    printf("Modulo Division = %d", a%b);
}
```

output:

Addition = 15
Subtraction = 5
Multiplication = 50
Division = 2
Modulo Division = 0

Eg:
```
int a=10, b=4;
printf("Division = %d", a/b);
printf("Modulo Division = %d", a%b);
```

output:

Division = 2
Modulo Division = 2

Eg:
```
int a=10, b=3;
printf("Division = %d", a/b);
printf("Modulo Division = %d", a%b);
```

output:

Division = 3
Modulo Division = 1

2) Relational operators:

The operators <, <=, >, >=, == and != are the relational operators supported by C.

operator	Meaning
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to
!=	is not equal to

Eg:

```
#include <stdio.h>
main()
{
    int a=10, b=3;
    printf("Less than = %d\n", a<b); //1 (True) (or) 0 (False)
    printf("Less than or equal to = %d\n", a<=b);
}
```

Output:

```
Less than = 0
Less than or equal to = 0
```

Eg:

```
int a=3, b=3;
printf("Less than = %d\n", a<b);
printf("Less than or equal to = %d\n", a<=b);
```

Output:

```
Less than = 0
Less than or equal to = 1
```

Eg:
```
#include <stdio.h>
main()
{
    int a=10, b=3;
    printf("Greater than = %d", a>b);
    printf("Greater than or equal to = %d", a>=b);
}
```

output:

Greater than = 1

Greater than or equal to = 1

Eg:
```
int a=3, b=3;
printf("Greater than = %d", a>b);
printf("Greater than or equal to = %d", a>=b);
```

output:

Greater than = 0

Greater than or equal to = 1

Eg:
```
#include <stdio.h>
main()
{
    int a=3, b=3;
    printf("Equal to = %d", a==b);
    printf("Not Equal to = %d", a!=b);
}
```

output:

Equal to = 1

Not Equal to = 0

Eg:
```
int a=10, b=3;
```

Equal to = 0

Not Equal to = 1

3) Logical operators:

* C has the following three logical operators.

operator	Meaning
&&	AND
||	OR
!	NOT

* The logical operators && and || are used to test more than one conditions and make decisions.

Eg: a>b && x==10

* An expression of this kind, combines two or more relational expressions is termed as a logical expression.

Eg: To check whether 2 given numbers are positive.

```
#include<stdio.h>
int main()
{
    int a=5, b=10;
    if(a>0 && b>0)
    {
        printf("Both a and b are positive");
    }
}
```

Output:

Both a and b are positive

Eg: int a = -5, b = 10;

output:

No output because 1 number is -ve

Eg: // To check whether any 1 given number is positive

```
#include <stdio.h>
int main()
{
    int a = -5, b = 10;
    if (a > 0 || b > 0)
    {
        printf("Either a or b is positive");
    }
}
```

output:

Either a or b is positive

Eg:

```
#include <stdio.h>
int main()
{
    int a = 5, b = 10;
    if (a == b)
    {
        printf("a equals b");
    }
}
```

output:

No output because a and b values are not equal

Eg:

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

```
if (!(a==b)) → Add ( ). for ! symbol
{
  printf("a equals b");
}
```

output:

a equals b

4) Bitwise operators:

* C uses bitwise operators for manipulation of data at bit level.
* These operators are used for testing the bits or shifting them right or left.
* It may not be applied to float or double.

operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
<<	Shift left
>>	Shift right

p	q	p&q	p|q	p^q
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

* The binary representations of the values can be obtained using the formula 2^n $(n = 0, 1, 2 \cdots)$

Eg: If p=7 and q=4, the binary representations are,

p = 0000 0111
q = 0000 0100
p & q = 0000 0100 (4)
p | q = 0000 0111 (7)
p ^ q = 0000 0011 (3)
p << 1 = 0000 1110 (14)
q >> 1 = 0000 0010 (2)

Eg:
```
#include <stdio.h>
int main()
{
    int p=7, q=4;
    printf("p & q = %d\n", p&q);
    printf("p | q = %d\n", p|q);
    printf("p ^ q = %d\n", p^q);
    printf("p<<1 = %d\n", p<<1);
    printf("q >>1 = %d\n", p>>1);
}
```

output: p & q = 4

p|q = 7

p^q = 3

p<<1 = 14

q>>1 = 2

5) Assignment operators:

* Assignment operators are used to assign the result of an expression to a variable.

Eg: C = a+b

* C also has a set of shorthand assignment operators.

Statement with simple assignment operator	Statement with short hand operator
a = a+1	a+ = 1
a = a-1	a- = 1
a = a*1	a* = 1
a = a|1	a| = 1
a = a^1	a^ = 1

Eg:

```
#include<stdio.h>
int main()
{
    int a=5, b=2;
    int c;
    c=a+b;
    printf("%d", c);
}
```

Output:

7

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

output:

6

Eg:
```
int a=5;
a+=1;
```

output:

6

Eg:
```
#include <stdio.h>
int main()
{
    int a=5, b=2;
    a=a-b;
    printf("%d", a);
}
```

output:

3

Eg:
```
int a=5, b=2;
a-=b;
```

output:

3

6) Conditional operators:

* A ternary operator pair ?: is available in C to construct conditional expressions of the form:

exp1 ? exp2 : exp3

* Here, exp1 is evaluated first.
* If it is true, then exp2 is evaluated and becomes the value of the expression.
* If it is false, then exp3 is evaluated and becomes the value of the expression.

Eg: X = (a > b) ? a : b

```
#include <stdio.h>
int main()
{
  int a=5, b=2;
  int c;
  c= (a>b) ? a:b;
  printf("%d", c);
}
```

output:

5

Eg: int a=5, b=12;

output:

12

Eg:
```
int a=5, b=12;
int c;
c= (a==b) ? a:b;
```

output: 12

Input/output Functions:

* Reading, processing and writing of data are the three essential functions of a computer program.
* C does not have any built-in input/output statements as part of its syntax.
* All input/output operations are carried out through function calls such as printf and scanf.
* These functions are collectively known as the Standard I/o library.
* Each program that uses a Standard input/output function must contain the statement - #include <stdio.h>
* I/o functions are grouped into two categories.

i) **Formatted I/o functions:** allow programmers to specify the type of data and the way it should be read in or written out.

ii) **Unformatted I/o functions:** do not allow programmers to specify the type of data and the way it should be read in or written out.

Function	Formatted	Unformatted
Input	Scanf()	getchar(), gets()
Output	printf()	putchar(), puts()

1) printf():

The printf function is used to display the data.

* This function can be used to display any combination of numerical values, single characters and strings.

```
printf("My Roll number is %d\n", rollno);
```

- "My Roll number is %d\n" — Control Statements
- rollno — printlist (or) variable
- Function arguments

* Here, % sign indicates that a conversion specification follows.
* d is known as data type character.

placeholder	Type of data item
%c	Single character
%d	Signed decimal integer
%e	Floating point number with an exponent
%f	Floating point number without an exponent
%g	Floating point number either with or without exponent value
%o	Octal number
%s	String
%u	Unsigned decimal integer
%x	Hexadecimal number

Eg:

```
#include<stdio.h>
int main()
{
    int a=5;
    char b='H';
    char c[3]="Hi";
    float d=1234.1234123;
    printf("The number is %d\n",a);
    printf("The character is %c\n",b);
    printf("The string is %s\n",c);
    printf("The float is %f\n",d);
    printf("The float is %.2f\n",d);
}
```


output:

```
The number is 5
The character is H
The string is Hi
The float is 1234.123413
The float is 1234.12
```

round off last digit

2) Scanf():

* C provides the Scanf function to read data from the standard input device

```
Scanf("Control String", &argument);
```

Eg:

```
Scanf("%d", &a);
```

* Like printf function, the Scanf function uses format specifiers preceded by a % sign to tell Scanf function what type of data is to be read.

Eg:-

```
#include<stdio.h>
int main()
{
    int a;
    char b;
    char c[3];
    printf("Enter the input\n");
    scanf("%d gap %c gap %s", &a, &b, c);
    printf("The number is %d \n", a);
    printf("The character is %c\n", b);
    printf("The String is %s\n", c);
    printf("The values are %d gap %c gap %s", a, b, c);
}
```

output:

```
Enter the input 5
H
Hi
The number is 5
The character is H
The String is Hi
The values are 5 H Hi
```

3) putchar() and puts():

* The putchar function displays one character on the display monitor,
* The character to be displayed is of type char

```
putchar(char_variable);
```

* The puts function displays a string of characters on the display monitor

```
puts(String_Variable);
```

Eg:
```
#include<stdio.h>
int main()
{
  char b;
  printf("Enter input\n");
  b=getchar();
  putchar(b);
}
```

output:

Enter input H
H

4) getchar() and gets():

* The getchar function accepts a single character from the keyboard.
* The function does not require any arguments.

```
ch_variable = getchar();
```

* The gets function reads a string of characters from the keyboard.

```
gets(string);
```

Eg:
```
#include<stdio.h>
int main()
{
  char b[3];
  printf("Enter input\n");
  gets(b);
  puts(b);
}
```

output:

Enter input Hi
Hi

Built-in Functions:

* A function is a named block of code that performs a specific task.
* Built-in functions are functions that are already provided as part of the C Standard library.
* These functions offer pre-built functionality for common tasks, simplifying programming and often providing optimized code for frequently used operations.
* To use these functions, need to include the appropriate header file using the #include directive

```
#include<libraryname.h>
```


Common built-in Functions:

Function	Header File
Scanf(), printf()	Stdio.h
getchar(), putchar()	Stdio.h
gets(), puts()	Stdio.h
Sqrt(), pow()	math.h
Sin(), Cos(), tan()	math.h
abs()	math.h
ceil, floor()	math.h
tolower(), toupper()	Ctype.h
Strlen(), Strcpy(), Strcat()	String.h
malloc(), free()	Stdlib.h

Eg:

```
#include <stdio.h>
#include <math.h>
int main()
{
    float a;
    printf("Enter the input\n");
    scanf("%f", &a);
    printf("The number is %.2f\n", a);
    printf("The sqrt is %.2f\n", sqrt(a));
}
```

output:

```
Enter the input 16.0
The number is 16.00
The sqrt is 4.00
```


Eg:

```
#include <stdio.h>
#include <math.h>
int main()
{
    float b = -3.0;
    printf("The abs is %.2f\n", abs(b));
}
```

abs change -ve to +ve

output:

```
The abs is 3.00
```

Eg:

```
#include <stdio.h>
#include <math.h>
int main()
{
    float b = 4.4;
```

```
printf("The ceil is %.2f \n", ceil(b));
printf("The floor is %.2f\n", floor(b));
}
```

output:

The ceil is 5.00 → Ceil means up. Neatrandoff num

The floor is 4.00 → floor means down.

Eg:

```
#include <stdio.h>
#include <ctype.h>
int main()
{
    char a='a', b='D';
    printf("The upper is %c\n", toupper(a));
    printf("The lower is %c \n", tolower(b));
}
```

output:

The upper is A

The lower is d

Eg:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char a[10]="computer", b[10]="science";
    printf("The length is %d \n", strlen(a));
}
```

output:

The length is 8

Eg:

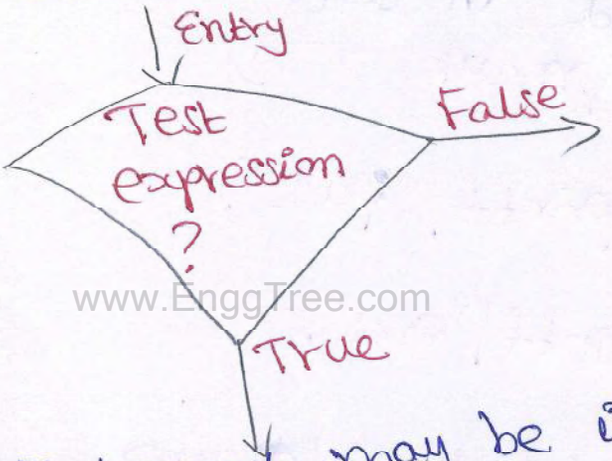
```
#include<stdio.h>
#include<string.h>
int main()
{
   char a[10]="Computer ", b[10]="Science";
   printf("The Concat is %s \n", strcat(a,b));
}
```

output:

The Concat is Computer Science

if:

* The if statement is a powerful decision making statement and is used to control the flow of execution of statements.
* It allows the computer to evaluate the expression first and then depending on whether the value of the expression is true or false, it transfers the control to a particular statement.



* The if statement may be implemented in different forms depending on the complexity of conditions to be tested.

The different forms are,

Simple if statement:

The general form of a simple if statement is:

```
if (test expression)
{
    statement-block;
}
statement-x;
```

* The Statement block may be a single or a group of statements.
* If the test expression is true, the Statement-block will be executed; otherwise the Statement-block will be skipped and execution will jump to Statement-X.
* If the condition is true, both the Statement-block and Statement-X are executed in sequence.

Eg:
```
if(marks > 70)
{
  printf("pass");
}
printf("End");
```


Eg:
```
#include<stdio.h>
int main()
{
  int marks = 80;
  if(marks > 50)
  {
    printf("you are pass\n");
  }
  printf("The above is your result");
}
```

output:

```
you are pass
The above is your result
```

Eg:
```
#include<stdio.h>
int main()
{
   int marks=40;
   if(marks>50)
   {
      printf("you are pass\n");
   }
   printf("The above is your result");
}
```

output:

The above is your result

---

if-else:

* The if-else statement is an extension of the simple if statement.
* The general form is:

```
if(test expression)
{
  True-block statements;
}
else
{
   False-block statements;
}
Statement-X;
```

* If the test expression is true, then the true-block statements immediately following the if statement are executed; otherwise the false-block statements are executed.

* In either case, either true-block or false-block will be executed, not both.

Eg:
```
if(marks > 70)
{
  printf("pass");
}
else
{
  printf("Fail");
}
```

Eg:
```
#include <stdio.h>
int main()
{
  int marks = 60;
  if (marks > 50)
  {
    printf("You are pass\n");
  }
  else
  {
    printf("You are fail\n");
  }
  printf("The above is your result");
}
```

output:

You are pass
The above is your result

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

```
{
    int marks = 40;
    if (marks > 50)
    {
      printf("You are pass\n");
    }
    else
    {
      printf(" You are fail\n");
    }
    printf("The above is your result");
}
```

output:

You are fail
The above is your result

Nested if:

* When a series of decisions are involved, we use more than one if-else statement in nested form.

```
if (test Condition-1)
{
  if (test Condition-2)
  {
    Statement-1;
  }
  else
  {
    Statement-2;
  }
}
else
  {
    Statement-3;
  }
```

```
#include <stdio.h>
int main()
{
    int marks = 60;
    if (marks > 50)
    {
        if (marks>70)
        {
            printf(" you have scored distinction\n");
        }
        else
        {
            printf(" you have scored just pass\n");
        }
    }
    else
    {
        printf(" you are fail\n");
    }
    printf("The above is your result");
}
```


Output:

```
you have scored just pass
The above is your result
```

Eg:

```
#include <stdio.h>
int main()
{
    int marks=90;
    if (marks > 50)
    {
        if(marks > 70)
        {
            printf("you have scored distinction\n");
        }
        else
        {
            printf(" you have scored just pass\n");
        }
    }
```

```
else
{
    printf("you are fail\n");
}
printf("The above is your result");
}
```

output:

you have scored distinction
The above is your result

Switch-Case:

* It can use an if statement to control the selection.
* However, the complexity of such a program increases dramatically when the number of alternatives increases.
* C has a built-in multiway decision statement known as a switch.
* The switch statement tests the value of a given variable (or expression) against a list of case values and when a match is found, a block of statements associated with that case is executed.

```
switch(expression)
{
    case value-1:
            block-1;
            break;
    case value-2:
            block-2;
            break;
    ...
    default: default-block;
             break;
```

}
Statement-X;

* The expression is an integer expression or characters.
* Each of the values (value-1, value-2, etc.,) should be unique within switch statement.
* When the switch statement is executed, the value of the expression is successfully compared against the values (value-1, value-2)
* If a case is found whose value matches with the value of the expression, then the block of statements that follows the case are executed.
* The break statement at the end of each block signals the end of a particular case and causes an exit from the switch statement, transferring the control to the statement-X.
* The default is an optional case. When present, it will be executed if the value of the expression doesnot match with any of the case values.
* If not present, no action takes place if all matches fail and the control goes to the Statement-X.

Eg:
```
#include<stdio.h>
int main()
{
    int marks=98;
    int index= marks/10;
    switch(index)
    {
    case 10:
        printf("O Grade");
        break;
    case 9:
        printf("A+ Grade");
        break;
    case 8:
        printf("A Grade");
        break;
    case 7:
        printf("B+ Grade");
        break;
    case 6:
        printf("B Grade");
        break;
    case 5:
        printf("C Grade");
        break;
    default:
        printf("Fail");
        break;
    }
}
```

output:

A+ Grade

Eg:
```
#include <stdio.h>
int main()
{
    int marks=65;
    int index = marks/10;
    switch (index)
    {
    case 10:
        printf("O Grade");
        break;
    case 9:
        printf("A+ Grade");
        break;
    case 8:
        printf("A Grade");
        break;
    case 7:
        printf("B+ Grade");
        break;
    case 6:
        printf("B Grade");
        break;
    case 5:
        printf("C Grade");
        break;
    default:
        printf("Fail");
        break;
    }
}
```

output:

B Grade

Eg:
```
#include <stdio.h>
int main()
{
    int marks=13;
    int index = marks/10;
    switch(index)
    {
    case 10:
        printf("O Grade");
        break;
    case 9:
        printf("A+ Grade");
        break;
    case 8:
        printf("A Grade");
        break;
    case 7:
        printf("B+ Grade");
        break;
    case 6:
        printf("B Grade");
        break;
    case 5:
        printf("C Grade");
        break;
    default:
        printf("Fail");
        break;
    }
printf("\n The above is your result");
}
```

output:

Fail
The above is your result.

While:

* The simplest of all the looping structures in C is the while statement.
* The basic format of the while statement is:

```
while( test condition)
{
  body of the loop
}
```

* The while is an entry-controlled loop statement.
* The test condition is evaluated and if the condition is true, then the body of the loop is executed.
* After execution of the body, the test-condition is once again evaluated and if it is true, the body is executed once again.
* This process of repeated execution of the body continues until the test-condition finally becomes false and the control is transferred out of the loop.
* On exit, the program continues with the statement immediately after the body of the loop.
* The body of the loop may have one or more statements.

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

output:
```
1
2
3
4
5
```


Eg:
```
#include<stdio.h>
int main()
{
    int i=3;
    while(i<=5)
    {
        printf("%d\n",i);
        i=i+1;
    }
}
```

output:
```
3
4
5
```

do-While:

* The while loop construct makes a test of condition before the loop is executed.
* Therefore, the body of the loop may not be executed at all if the condition is not satisfied at the very first attempt.
* On some occasions, it might be necessary to execute the body of the loop before the test is performed.
* Such situations can be handled with the help of the do Statement.

```
do
{
    body of the loop
}
while (test-condition);
```


* On reaching do Statement, the program proceeds to evaluate the body of the loop first.
* At the end of the loop, the test-condition in the while Statement is evaluated.
* If the condition is true, the program continues to evaluate the body of the loop once again.
* This process continues as long as the condition is true.
* When the condition become false,

the loop will be terminated and the control goes to the statement that appears immediately after the while statement.

Eg:

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


output:

```
1
2
3
4
5
```

Eg:

```
#include <stdio.h>
int main()
{
    int i=7;
    do
    {
        printf("I'm inside do while loop\n");
        printf("%d\n", i);
        i=i+1;
    }
```

```
while(i<=5);
printf("I'm out of do while loop");
}
```

output:

```
I'm inside do while loop
7
I'm out of do while loop
```

Eg:

```
#include<stdio.h>
int main()
{
  int i=7;
  while(i<=5)
  {
    printf("I'm inside while loop\n");
    printf("%d\n",i);
    i=i+1;
  }
  printf("I'm out of while loop\n");
  do
  {
    printf("I'm inside do while loop\n");
    printf("%d\n",i);
    i=i+1;
  }
  while(i<=5);
  printf("I'm out of do while loop\n");
}
```

output:

```
I'm out of while loop
I'm inside do while loop
7
I'm out of do while loop
```

for:

* The for-loop is another entry-controlled loop that provides a more concise loop control structure.

```
for(initialization; test-condition; increment)
{
   body of the loop
}
```

* Initialization of the control variables is done first, using assignments such as i=1 and count=0. The variables i and count are known as loop-control variables.
* The value of the control variable is tested using the test-condition.
* The test condition is a relational expression such as i<10 that determines when the loop will exit.
* If the condition is true, the body of the loop is executed; otherwise the loop is terminated.
* When the body of the loop is executed, the control is transferred back to the for statement after evaluating the last statement in the loop.
* Now, the control variable is incremented using the assignment statement such as i=i+1 and the new value is again tested.

* If the condition is satisfied, the body of the loop is again executed.
* This process continues till the value of the control variable fails to satisfy the test-condition.

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


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

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

Output:

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

Nested loops:

It is one for statement within another for statement is allowed in C. The syntax is,

```
for(initialization; test-condition; increment)
{
  body of the outer loop
  for(initialization; test-condition; increment)
  {
    body of the inner loop
  }
}
```

Eg:

```
#include<stdio.h>
int main()
{
  int i,j;
  for(i=1 ; i<=3; i++)
  {
    for(j=1; j<=3; j++)
    {
      printf("%d\t", i*j);
    }
    printf("\n");
  }
return 0;
```

Output:

```
1 2 3
2 4 6
3 6 9
```

Eg:

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


Output:

```
1  2  3  4  5  6  7  8  9  10
2  4  6  8  10 12 14 16 18 20
3  6  9  12 15 18 21 24 27 30
4  8  12 16 20 24 28 32 36 40
5  10 15 20 25 30 35 40 45 50
6  12 18 24 30 36 42 48 54 60
7  14 21 28 35 42 49 56 63 70
8  16 24 32 40 48 56 64 72 80
9  18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

Jump Statements:

* Loops perform a set of operations repeatedly until the control variables fails to satisfy the test-condition.
* The number of times a loop is repeated is decided in advance and the test condition is written to achieve this.
* Sometimes, when executing a loop it becomes desirable to skip a part of the loop or to leave the loop as soon as a certain condition occurs.
* C permits a jump from one statement to another within a loop as well as jump out of a loop.
* There are 3 types of jump statements in C.
  1) Break
  2) Continue
  3) Goto
* These statements can be used within the loops for exiting the current iteration/ entire loop as per the requirement.

1) Break:

* The break statement is used to immediately terminate the execution of the loop. (for, while, do-while) or a switch case block.

* Control moves to the statement following the loop or switch.

```
while(...)
{
...
if(condition)
  break;
}
```

Eg:

```
#include<stdio.h>
int main()
{
  int i;
  for(i=1; i<=5; i++)
  {
    if(i==3)
    {
      break;
    }
    printf("%d\n", i);
  }
  printf("Bye\n");
}
```

output:

```
1
2
Bye
```

2) Continue:

The continue statement skips the

remaining code in the current iteration of a loop and proceeds with the next iteration of the loop.

```
while (...)
{
  ...
  if (condition)
      continue;
}
```

Eg:

```
#include <stdio.h>
int main()
{
   int i;
   for (i=1; i<=5; i++)
   {
     if (i==3)
     {
       continue;
     }
     printf("%d\n", i);
   }
   printf("Bye\n");
}
```

output:

```
1
2
4
5
Bye
```

3) Goto:

* The goto statement transfers the control to the labeled statement within the same function.
* It allows jumping to another part of the code, but its use is discouraged due to potential readability issues.

```
while (...)
{
  ....
  if (condition)
    goto abc;
  abc;
}
```


Eg:

```
#include <stdio.h>
int main()
{
  int i;
  for(i=1; i<=5; i++)
  {
    if (i==3)
    {
      goto Hi;
    }
    printf("%d\n", i);
  }
  printf("Bye\n");
Hi:
  printf("Hi\n");
}
```

output:

```
1
2
Hi
```

Functions

Function Declaration, Definition and Calling:

Function:

* A function is a named block of code that performs a specific task.
* It allows to write a piece of code logic once and reuse it wherever needed in program.
* C functions can be classified into two categories namely library functions and user-defined functions.
* Inorder to make use of a user-defined function, need to establish three elements that are related to functions.
  1) Function Definition
  2) Function Call
  3) Function Declaration

Function Definition:

* The function definition is an independent program module that is specifically written to implement the requirements of the function.
* A function definition also known as function implementation shall include the following elements:
  1) Function Name
  2) Function Type
  3) List of parameters
  4) Variable Declarations

5) Function Statements
6) A return Statement

Function Definition:

All the six elements are grouped into two parts namely,
1) Function Header
2) Function Body

```
function_type function_name (parameter list)
{
    local variable declaration;
    executable Statement 1;
    executable Statement 2;
    - - - - - - - - - - - -
    - - - - - - - - - - - -
    return Statement;
}
```


Function Declaration:

* All functions in a C program must be declared, before they are invoked.
* A function declaration also known as function prototype consists of four parts:
1) Function Type
2) Function Name
3) parameter list
4) Terminating Semicolon

Syntax:

function-type function-name (parameter list);

* The parameter list must be separated by commas.
* The parameter names do not need to be the same in the function declaration and the function definition.
* The types must match the types of parameters in the function definition, in number and order.
* The return type is optional, when the function returns int type data.
* When the declared types do not match with the types in the function definition, Compiler will produce an error.

Function Calling:

* A function can be called by simply using the function name followed by a list of actual parameters, if any enclosed in parenthesis.

function-name(parameter1, parameter2...);

* When the compiler encounters a function call, the control is transferred to the function.
* The function is then executed line by line as described and a value is returned when a return statement is encountered.

Eg:

```
#include<stdio.h>
// function Declaration
int add(int a, int b);
int main()
{
    int number 1=5, number 2=6, result;
    // function Call
    result = add(number 1, number 2);
    printf("Sum = %d\n", result);
    return 0;
}
// Function Definition
int add(int a, int b)
{
    int c;
    c = a+b;
    return c;
}
```


Output:

Sum = 11

Function parameters and Return Types:

Based on the data flow between the calling function and called function, the functions are classified. A function depending on

Whether arguments are present or not and whether a value is returned or not, may belong to one of the following categories.

1) Function with No Arguments and No Return Value:

In this type of functions there is no data transfer between calling function and called function. The execution control jumps from calling function to called function and executes called function and finally comes back to the calling function.


```
#include<stdio.h>
Void main()
{
Void addition();
 addition();
}
Void addition()
{
int num1, num2;
printf("Enter any two integer numbers:");
Scanf("%d%d", &num1, &num2);
printf("Sum = %d", num1 + num2);
}
```

Output:

Enter any two integer numbers: 20 30

Sum = 50

2) Functions with no arguments and With Return Value:

* In this type of functions there is no data transfer from calling function to called function (parameters) but there is data transfer from called function to calling function (return value).
* The execution control jumps from calling function to called function and executes called function and finally comes back to the calling function along with a return value.

```
#include<stdio.h>
Void main()
{
int result;
int addition();
result = addition();
printf("Sum = %d", result);
}
int addition()
{
```

```
int num1, num2;
printf("Enter any two integer numbers:");
scanf("%d %d", &num1, &num2);
return(num1 + num2);
}
```

output:

Enter any two integer numbers: 20 30

Sum = 50

3) Functions with Arguments and no return value:

* In this type of functions there is data transfer from calling function to called function (parameters) but there is no data transfer from called function to calling function (return value)
* The execution control jumps from calling function to called function along with the parameters and executes called function and finally come back to the calling function.

```
#include <stdio.h>
Void main()
{
int num1, num2;
Void addition(int, int);
printf("Enter any two integer numbers:");
```

```
scanf("%d%d", &num1, &num2);
addition(num1, num2);
}
void addition(int a, int b)
{
printf("sum = %d", a+b);
}
```

output:

Enter any two integer numbers: 20 30

sum = 50

4) Functions with arguments and Return value:

* In this type of functions there is data transfer from calling function to called function (parameters) and also from called function to calling function (return value).
* The execution control jumps from calling function to called function along with parameters and execution called function and finally comes back to the calling function along with a return value.

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

```
{
int num1, num2, result;
int addition(int, int);
printf("Enter any two integer numbers:");
Scanf("%d%d", &num1, &num2);
result = addition(num1, num2);
printf("Sum = %d", result);
}
int addition(int a, int b)
{
return (a+b);
}
```

output:

Enter any two integer numbers: 20 30

Sum = 50

Call by Value and Call by Reference:

Parameter passing:

The technique used to pass data from one function to another is known as parameter passing. It can be done in two ways:

1) Call by Value (pass by Value):

* In Call by Value, values of actual parameters are copied to the variables in the parameter list

of the called function.

* The called function works on the copy and not on the original values of the actual parameters.
* This ensures that original data in the calling function cannot be changed accidentally.

Void modify (int x)

Eg:

```
#include <stdio.h>
Void modify (int x);
int main()
{
    int a = 5;
    printf("The value of a before the function
                is %d\n", a);

    modify (a);
    printf("The value of a after the function
                is %d\n", a);

    return 0;
}
Void modify (int x)
{
    x = x+10;
    printf("The value of a inside the function
                is %d\n", x);
}
```

output:-

The Value of a before the function is 5
The value of a inside the function is 15
The value of a after the function is 5

2) Call by Reference (pass by Reference or pointer)

* In call by reference, the memory addresses of the variables rather than the copies of the values are sent to the called function.
* In this case, the called function directly works on the data in the calling function and the changed values are available in the calling function for its use.

Void modify (int *x)

Eg:

```
#include <stdio.h>
Void modify (int *x);
int main()
{
   int a=5;
   printf("The Value of a before the
          function is %d\n", a);
```

```
modify(&a);
printf("The value of a after the function
                is %d\n", a);
return 0;
}
void modify(int *x)
{
*x = *x + 10;
printf("The value of a inside the
        function is %d\n", x);
}
```


output:

The value of a before the function is 5
The value of a inside the function is 15
The value of a after the function is 15

Recursive Functions:

* When a called function in turn calls another function, a process of chaining occurs.
* It is a special case of this process, where a function calls itself.
* A very simple example is presented below:

```
main()
{
    printf("Hi Hello\n");
    main();
}
```

Eg: Factorial of a number

The factorial of a number n is expressed as a series of repetitive multiplications as shown below:

$n! = n(n-1)(n-2)\ldots 1$

Eg: $4! = 4*3*2*1$

$= 24$

Eg:

```
#include<stdio.h>
int factorial(int n);
int main()
{
int a=4, result;
result = factorial(a);
printf("Factorial = %d\n", result);
return 0;
}
int factorial(int n)
{
  int fact;
  if(n==1)
    return(1);
```

```
    fact = n * factorial (n-1);
    return (fact);
}
```

output:

Factorial = 24

Scope and Lifetime of Variables:

* The scope of the variable determines over what region of the program a variable is actually available for use.
* The variables may be categorized depending on the place of their declaration as local and global variables.
* In C, not only do all variables have a data type, also have a storage class.

1) Automatic Variables:
2) External Variables
3) Static Variables
4) Register Variables

1) Automatic Variables:

* It is declared inside a function in which they are to be utilized.
* They are created when the function is called and destroyed automatically when the function is exited, hence the name automatic.
* It is therefore local to the function in which they are declared.
* Because of this property, it is also referred as local variable.
* It also use the keyword auto to declare automatic variables explicitly.

Eg:

```
#include <stdio.h>
Void demo();
int main()
{
   demo();
   return 0;
}
void demo()
{
   int x=5;
   printf("x = %d\n", x);
}
```

output:

x = 5

2) External Variables:

* Variables that are both alive and active throughout the entire program are known as external variables. It is also known as global variables.
* Unlike local variables, global variables can be accessed by any function in the program.
* It is declared outside the function.
* It also use the keyword extern to show the program that it is a global variable.

eg:

```
#include<stdio.h>
int x=5;
void demo();
int main()
{
  demo();
  printf("outside x=%d\n",x);
  return 0;
}
void demo
{
  printf("Inside x=%d\n",x);
}
```

3) Static Variables:

* As the name suggests, the value of Static Variables persists until the end of the program.
* A variable can be declared Static using the keyword Static like

```
Static int X;
```

* A Static Variable may be either an internal type or an external type depending on the place of declaration.

Eg:

```
#include <stdio.h>
void demo();
int main()
{
   demo();
   demo();
   return 0;
}
void demo()
{
   int X=5;
   printf("Inside X=%d\n",X);
   X=X+1;
}
```

output:

```
Inside X=5
```

Eg:

```
#include<stdio.h>
Void demo();
int main()
{
   demo();
   demo();
   return 0;
}
Void demo()
{
   Static int X=5;
   printf("Inside X=%d\n",X);
    X=X+1;
}
```


output:

```
Inside X=5
Inside X=6
```

Eg:

```
#include<stdio.h>
 Void demo();
 int main()
{
   demo();
   demo();
   demo();
    return 0;
}
Void demo()
{
  int x=1;
```

```
    printf("Inside X=%d\n", X);
    X = X+1;
}
```

output:

```
Inside X=1
Inside X=1
Inside X=1
```

Eg:

```
#include<stdio.h>
void demo();
int main()
{
  demo();
  demo();
  demo();
  return 0;
}
void demo()
{
  static int X=1;
  printf("Inside X=%d\n", X);
  X = X+1;
}
```

output:

```
Inside X=1
Inside X=2
Inside X=3
```

4) Register Variables:

* It tell the compiler that a variable should be kept in one of the machine's register instead of keeping in the memory.
* Since a register access is much faster than a memory access, keeping the frequently accessed variables in the register will lead to faster execution of programs.

```
register int count;
```

Header Files and Modular programming:

Header Files:

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

Header File	Purpose	Example
stdio.h	provides functions to perform standard I/o operations	printf(), scanf()
conio.h	provides functions to perform console I/o operations	clrscr(), getch()
string.h	provides functions to	strlen()

Stdlib.h	provides functions to perform general functions	Calloc, malloc
math.h	provides functions to perform mathematical operations	sqrt(), pow()
time.h	provides functions to perform operations on time and date	time() localtime()
graphics.h	provides functions to draw graphics	circle() rectangle()

Modular programming:

* In real-life programming environment, we more than one source file which may be compiled separately and linked later to form an executable object code.
* It involves organizing a program into independent, reusable units called modules.
* This approach enhances code reusability, maintainability and reusability particularly for larger projects.
* It consists of functions, header files that act as modules and source C files.

Eg:

```
#include<stdio.h>
int add(int x, int y);
int sub(int x, int y);
int main()
{
   int a=10, b=5;
   printf("Sum = %d\n", add(a,b));
   printf("Difference = %d\n", sub(a,b));
   return 0;
}
int add(int x, int y)
{
   return x+y;
}
int sub(int x, int y)
{
   return x-y;
}
```

output:

```
Sum = 15
Difference = 5
```

main.c

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

```
    printf("Sum = %d\n", add(a,b));
    printf("Difference = %d\n", sub(a,b));
    return 0;
}
```

module.h

```
int add(int x, int y);
int sub(int x, int y);
```

module.c

```
#include "module.h"
int add(int x, int y)
{
    return x+y;
}
int sub(int x, int y)
{
    return x-y;
}
```

output:

Sum = 15

Difference = 5

Strings and pointers:

One-dimensional and Multi-dimensional Arrays:

Arrays:

* An array is a sequenced collection of related data items that share a common name.
* It is simply a grouping of like-type data.
* It can use arrays to represent not only simple lists of values but also tables of data in two or more dimensions
* There are following types of arrays
  i) one-dimensional arrays
  ii) Two-dimensional arrays
  iii) Multi-dimensional arrays

i) one-dimensional Arrays:

* A list of items can be given one variable name using only one subscript and such a variable is called Single-Subscripted Variable or one-dimensional array.
* Like any other variable, arrays must be declared before they are used.

* The general form of array declaration is,

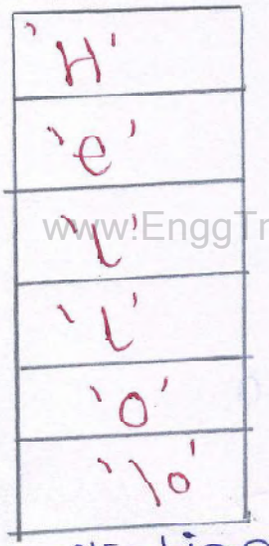
type variable_name[size];

Eg: int group[10];
float height[50];

* The C language treats character strings, simply as array of characters.

char name[6];

* If the value stored is "Hello" then,



* It can initialize the elements of arrays in the same way as the ordinary variables when they are declared.

type array_name[size]={list of values};

* The values in the list are separated by commas.

Eg: int number[5]={1,2,3};

declare the variable number as

an array of size 3 and will size the values 1,2,3 to each element.

Eg:

```
#include<stdio.h>
int main()
{
    int number[5]={10,20,30,40,50};
    printf("1D Array:\n");
    for(int i=0;i<5;i++)
    {
      printf("%d\n",number[i]);
    }
    return 0;
}
```


output:

```
1D Array:
10 20 30 40 50
```

Eg:

```
#include<stdio.h>
int main()
{
  int number[5]={10,20,30,40,50};
  printf("1D Array:\n");
  for(int i=0;i<5;i++)
  {
    printf("number[%d]=%d\n",i,number[i]);
  }
  return 0;
}
```

Output:-

1D Array:
number [0] = 10
number [1] = 20
number [2] = 30
number [3] = 40
number [4] = 50

2) Multi-Dimensional Arrays:-

* C allows arrays of three or more dimensions.
* The general form of a multi-dimensional array is,

type array_name [s1][s2][s3]... [sn];

Eg:
int survey[3][5][12];

* Here, survey is a three-dimensional array declared to contain 180 integer type elements.

Eg:

```
#include<stdio.h>
int main()
{
  int number[2][3]=
  {
    {1,2,3},
    {4,5,6}
```

```
printf("2D Array: \n");
for(int i = 0; i<2; i++)
{
  for(int j=0; j<3; j++)
  {
    printf("%d", number[i][j]);
  }
  printf("\n");
}
}
```

output:

```
2D Array:
1 2 3
4 5 6
```


Eg:

```
#include<stdio.h>
int main()
{
  int number[2][3]=
  {
    {1,2,3},
    {4,5,6}
  };
  printf("2D Array: \n");
  for(int i=0; i<2; i++)
  {
    for(int j=0; j<3; j++)
    {
```

```
        printf("number[%d][%d]=%d\n", i, j, number[i][j]);
    }
}
```

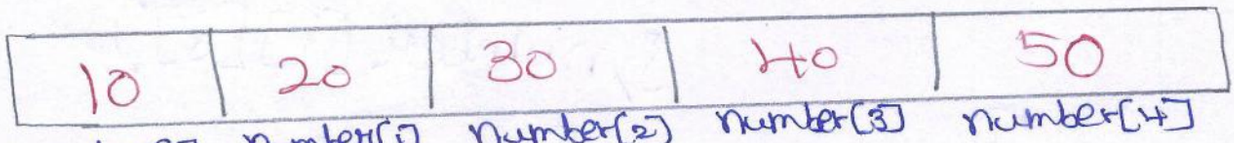
output:

```
2D Array:
number[0][0] = 1
number[0][1] = 2
number[0][2] = 3
number[1][0] = 4
number[1][1] = 5
number[1][2] = 6
```


Array operations and Traversals:

1) Finding length of an array:
2) Accessing Specific Elements
3) Modifying elements
4) Copying Arrays
5) Searching for an element
6) Linear traversals
7) Reverse traversals

```
int number[] = {10,20,30,40,50};
```



1) Finding length of an array:

```
#include<stdio.h>
int main()
{
    int numbers[] = {10,20,30,40,50};
    int size = sizeof(numbers[0]);
    int length = sizeof(numbers)/sizeof(numbers[0]);
    printf("size = %d\n", size);
    printf("length = %d\n", length);
    return 0;
}
```


Output:

```
size = 4
length = 5
```

Eg:

```
#include<stdio.h>
int main()
{
    int numbers[]={10,20,30,40,50};
    int size = sizeof(numbers);
    int length = sizeof(numbers)/sizeof(numbers[0]);
    printf("size = %d\n", size);
    printf("length= %d\n", length);
    return 0;
}
```

2) Accessing Specific Elements:

```
#include <stdio.h>
int main()
{
 int numbers[] = {10, 20, 30, 40, 50};
 printf(" 1st element = %d\n", numbers[0]);
 Printf("Last element = %d\n", numbers[4]);
 Printf(" mid element = %d\n", numbers[2]);
 return 0;
}
```

output:

1st element = 10
Last element = 50
Mid element = 30

3) Modifying Elements:

```
#include <stdio.h>
int main()
{
 int numbers[] = {10, 20, 30, 40, 50};
 numbers[1] = 90;
 printf("%d\n", numbers[1]);
 return 0;
}
```

output:

90

Eg:

```
#include<stdio.h>
int main()
{
  int numbers[]={10,20,30,40,50};
  numbers[1]=90;
  for(int i=0; i<5; i++)
  {
    printf("%d", numbers[i]);
  }
  return 0;
}
```

output:

10 90 30 40 50

4) Copying Arrays:

```
#include<stdio.h>
int main()
{
  int numbers[]={10,20,30,40,50};
  int arrays[5];
  for(int i=0; i<5; i++)
  {
    arrays[i] = numbers[i];
  }
  printf("Numbers Values\n");
  for(int i=0; i<5; i++)
  {
    printf("%d", numbers[i]);
```

```
printf("\n Arrays Values\n");
for(int i=0; i<5; i++)
{
  printf("%d", arrays[i]);
}
return 0;
}
```

output:

Number Values 10 20 30 40 50
Array Values 10 20 30 40 50

5) Searching for an element:

```
#include<stdio.h>
int main()
{
  int numbers[] = {10,20,30,40,50};
  int key = 50;
  for(int i = 0; i<5; i++)
  {
    if(numbers[i] == key)
    {
      printf("Element %d found at index %d",
                                key, i);
      break;
    }
  }
  return 0;
}
```

output:

Element 50 found at index 4

6) Linear Traversals:

```
#include <stdio.h>
int main()
{
  int numbers[] = {10, 20, 30, 40, 50};
  for (int i = 0; i < 5; i++)
  {
    printf("%d", numbers[i]);
  }
  return 0;
}
```

output:

10 20 30 40 50


```
#include <stdio.h>
int main()
{
  int numbers[] = {10, 20, 30, 40, 50};
  int length = sizeof(numbers)/sizeof(numbers[0]);
  for (int i = 0; i < length; i++)
  {
    printf("%d", numbers[i]);
  }
  return 0;
}
```

output:

10 20 30 40 50

```
#include<stdio.h>
int main()
{
 int numbers[]={10,20,30,40,50,60,70,80,90,100};
 int length=sizeof(numbers)/sizeof(numbers[0]);
 for(int i=0;i<5;i++)
 {
  printf("%d",numbers[i]);
 }
 return 0;
}
```

output:

10 20 30 40 50

7) Reverse Traversals:


```
#include<stdio.h>
int main()
{
 int numbers[]={10,20,30,40,50};
 int length=sizeof(numbers)/sizeof(numbers[0]);
 for(int i=length-1;i>=0;i--)
 {
  printf("%d",numbers[i]);
 }
 return 0;
}
```

output:

50 40 30 20 10

String Handling : String Declaration, Input/ Output:

String Handling:

* A String is a sequence of characters that is treated as a single data item.
* Any group of characters defined between double quotation marks is a string constant.

"Hi, How are You?"

* Character Strings are often used to build meaningful and readable programs.

i) String declaration
ii) Input/output
iii) String library functions

String Declaration:

* C doesnot support strings as a data type.
* However, it allows us to represent strings as character arrays.
* In C, therefore a String variable is

always declared as an array of characters terminated by a null character (\0).

```
char stringname[size];
```

Eg:

```
char city[10];
char city[10]="New York";
char city[]="New York";
```

Eg:

```
#include<stdio.h>
int main()
{
  char city[50]="New York";
  printf("%s", city);
}
```

output:

New York

String input and output functions:

The String input can be given using the following functions:

i) Scanf
ii) getchar
iii) gets

The String output can be displayed using the following functions:

i) printf
ii) putchar
iii) puts

Scanf function:

* The input function Scanf can be used with %s format specification to read in a String of characters.

Eg: char address[10];
Scanf("%s", address);

* The problem with the Scanf function is that it terminates its input on the first white space it finds.
* Therefore if "NEW YORK" is typed, then only the String "NEW" will be read into the array address.

Eg:
```
#include<stdio.h>
int main()
{
    char city[50];
    printf("Enter the city\n");
    Scanf("%s", city);
    printf("\n%s", city);
}
```

output:

Enter the city
New York

New

printf function:

The format %s can be used to display an array of characters that is terminated by the null character.

Eg: printf("%s", name);

can be used to display the entire contents of the array name.

gets function:

Another and more convenient method of reading a string of text containing whitespaces is to use the library function gets available in the <stdio.h> header file.

It is a simple function with one string parameter and called as under:

gets(str);

Eg:
```
char line[80];
gets(line);
```

puts function:

* Another and more convenient method of printing string values is to use the library function puts available in the <stdio.h> header file.
* It can be invoked under,

```
puts (Str);
```

Eg:

```
char line[50];
gets (line);
puts (line);
```

Eg:

```
#include <stdio.h>
int main()
{
    char city[50];
    printf("Enter the city\n");
    gets (city);
    puts (city);
}
```

output:

```
Enter the city
New York
New York
```

getchar function:

* It can use getchar to read a single character from the terminal.
* It can also use this function repeatedly to read successive single characters from the input and place them into a character array.
* Thus, an entire line of text can be read and stored in an array.
* The reading is terminated when a newline character (\n) is entered and the null character is then inserted at the end of the string.

```
char ch;
ch = getchar();
```

Eg:

```
#include <stdio.h>
int main()
{
  char ch;
  printf("Enter the character\n");
  ch = getchar();
  putchar(ch);
}
```

Enter the character
A
A

Eg:

```
#include<stdio.h>
int main()
{
    char line[100];
    int i=0;
    char ch;
    printf("Enter the line\n");
    while((ch=getchar())!='\n')
    {
        line[i]=ch;
        i++;
    }
    line[i]='\0';
    printf("you entered: %s", line);
}
```

output:

Enter the line
Hi, how are you?
you entered: Hi, how are you?

putchar function:

* Like getchar, C supports putchar to output the values of character variables.

```
char ch = 'A';
putchar(ch);
```

* It can also use this function repeatedly to output a string of characters stored in an array using a loop.

Eg:

```
#include<stdio.h>
int main()
{
  char Str[] = "Hello, World!";
  int i=0;
  while(Str[i]!='\0')
  {
    putchar(Str[i]);
    i++;
  }
  return 0;
}
```

output:

```
Hello, World!
```

String Library Functions:

* The C library supports a large number of string-handling functions that can be used to carry out many of the string manipulations.
* Following are the most commonly used string-handling functions:
  i) Strcat()
  ii) Strcmp()
  iii) Strcpy()
  iv) Strlen()
  v) Strncpy()
  vi) Strncmp()
  vii) Strncat()

i) Strcat() Function: String Concatenation

* The strcat function joins two strings together.
* It takes the following form:

```
strcat(string1, string2);
```

* When this function is executed, string2 is appended to string1.
* It does so by removing null characters at the end of string1 and placing

String2 from there.
* The String at String2 remains unchanged.

Eg:

```
#include<stdio.h>
#include<string.h>
int main()
{
    char Str1[] = "apple";
    char Str2[] = "banana";
    Strcat(Str1, Str2);
    printf("Result:%s", Str1);
    printf("\nResult:%s", Str2);
}
```


output:

Result: applebanana
Result: banana

2) Strncat() Function:

It is another concatenation function that takes three parameters as,

Strncat(S1, S2, n);

Eg:

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

```
{
char Str1[] = "apple";
char Str2[] = "banana";
Strncat(Str1, Str2, 2);
printf("Result: %s", Str1);
}
```

output:

Result: appleba

3) Strcmp() Function: String Comparison

* The Strcmp function compares two strings identified by the arguments and has a value 0 if they are equal.
* If they are not, it has the numeric difference between the first nonmatching characters in the strings.

Strcmp(String1, String2);

* It compares the left-most n characters of S1 to S2 and returns,
  0, if they are equal
  negative number, if S1 < S2
  positive number, Otherwise

Eg:

```
#include <stdio.h>
#include <String.h>
int main()
{
    char str1 [] = "apple";
    char str2 [] = "apple";
    int result = strcmp (str1, str2);
    printf ("Result: %d", result);
}
```

output:

Result: 0

Eg:

```
#include <stdio.h>
#include <String.h>
int main()
{
    char str1 [] = "apple";
    char str2 [] = "bpple";
    int result = strcmp (str1, str2);
    printf ("Result = %d", result);
}
```

output:

Result: -1

Eg:

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

```
int main()
{
  char Str1[]="bpple";
  char Str2[] = "apple";
  int result = Strcmp(Str1, Str2);
  printf("Result : %d", result);
}
```

output:

Result : 1

4) Strncmp() Function:: String Comparison

* A variation of the function Strcmp is the function Strncmp.
* It has three parameters as illustrated in the function call below:

```
Strncmp(S1, S2, n);
```

Eg:

```
#include<stdio.h>
#include<string.h>
int main()
{
  char Str1[]="applf";
  char Str2[] = "apple";
  int result = Strncmp(Str1, Str2, 3);
  printf("result : %d", result);
}
```

output:

Result : 0

Eg:

```
#include<stdio.h>
#include<string.h>
int main()
{
  char Str1[]="apple";
  char Str2[]="apple";
  int result=Strncmp(Str1,Str2,3);
  printf("Result:%d",result);
}
```

output:

Result: 1

5) Strcpy() Function: String Copy

* The Strcpy function works almost like a string- assignment operator.
* It takes the form:

  Strcpy(string1, string2);

* In addition to the function Strcpy that copies one string to another, have another function Strncpy that copies only the left-most n characters of the source string to the target string variable.
* It is a three-parameter function and is invoked as follows:

Strncpy(s1, s2, n);

* Since the first n characters may not include the terminating null character, have to place it explicitly in the n+1th position of s_2 as shown below:

s1[n+1] = '\0';

Eg:

```
#include<stdio.h>
#include<string.h>
int main()
{
  char str1[10]="apple";
  char str2[10];
  strcpy(str2, str1);
  printf("Result: %s", str2);
}
```

Output:

Result: apple

Eg:

```
#include<stdio.h>
#include<string.h>
int main()
{
  char str1[10]="apple";
  char str2[10];
```

```
Strncpy (Str2, Str1, 3);
printf(" Result: %s", Str2);
}
```

output:

Result: app

Eg:

```
#include <stdio.h>
#include <String.h>
int main()
{
    char Str1[10]="apple";
    char Str2[10];
    Strncpy (Str2, Str1, 3);
    Str2[3]='\0';
    printf(" Result: %s", Str2);
}
```


output:

Result: app

6) Strlen() Function: String length

* The Strlen function counts and returns the number of characters in a String.
* It takes the form
  n = Strlen (String).

Eg:

```
#include <stdio.h>
#include <string.h>
int main()
{
   char str1[10]="apple";
   int result=strlen(str1);
   printf("Result:%d", result);
}
```

output:

Result:5

Eg:

```
#include <stdio.h>
#include <string.h>
int main()
{
   char str1[]="applebanana";
   int result=strlen(str1);
   printf("Result:%d", result);
}
```

output:

Result: 11

Pointer Arithmetic:

* A pointer is a derived data type in C.
* pointers contain memory addresses as their values.
* Since these memory addresses are the locations in the computer memory where program instructions and data are stored, pointers can be used to access and manipulate data stored in the memory.
* pointers are more efficient in handling arrays and data tables.
* pointers allow C to support dynamic memory management.
* Whenever declare a variable the system allocates, somewhere in the memory, an appropriate location to hold the value of the variable.

```
int quantity = 179;
```

* This statement instructs the system to find a location for the integer variable quantity and puts the value 179 in that location.
* Assume, the system has chosen the

address location 5000:

Quantity ⟵ Variable

179 ⟵ Value

5000 ⟵ Address

* During execution of the program, the system always associates the name quantity with the address 5000.
* It may have access to the value 179 by using either the name quantity or the address 5000.
* Since memory addresses are simply numbers, they can be assigned to some variables.
* Such variables that hold memory addresses are called pointer variables.

Declaring pointer Variables:

* In C, every variable must be declared for its type.
* Since pointer variables contain addresses that belong to a separate data type, they must be declared as pointers before use them.

```
data_type *pointer_name;
```

* It tells the compiler three things about the variable pointer_name:
i) The asterisk (*) tells that the variable is a pointer variable.
ii) pointer_name needs a memory location.
iii) pointer_name points to a variable of type data_type.

Initialization of pointer variables:

* The process of assigning the address of a variable to a pointer variable is known as initialization.
* Once a pointer variable has been declared, can use the assignment operator to initialize the variable.

```
int quantity;
int *p;
p = &quantity;
```

* It can also combine the initialization with the declaration.

```
int *p = &quantity;
```

Accessing a variable through its pointer:

* Once a pointer has been assigned the address of a variable, the question remains as to how to access the value of the variable using the pointer.

* This is done by using another unary operator * (asterisk) usually known as the indirection operator.
* Another name for the indirection operator is the dereferencing operator.

```
int quantity, *p,n;
quantity = 179;
p = &quantity;
n = *p;
```

Eg:

```
#include<stdio.h>
int main()
{
int quantity = 179;
int *p = &quantity;
printf("Value of quantity : %d\n", quantity);
printf("Value of quantity in p : %d\n", *p);
printf("Address of quantity: %d\n", &quantity);
printf("Address of quantity in p: %d\n", p);
*p = 80;
printf("Value of quantity : %d\n", quantity);
printf("Value of quantity in p: %d\n", *p);
printf("Address of quantity: %d\n", &quantity);
printf("Address of quantity in p: %d\n", p);
return 0;
}
```

Output:

Value of quantity : 179
Value of quantity in p : 179
Address of quantity : 14679576
Address of quantity in p : 14679576
Value of quantity : 80
Value of quantity in p : 80
Address of quantity : 14679576
Address of quantity in p : 14679576

Pointer Expressions:

* Like other variables, pointers can be used in expressions.
* For example, if p1 and p2 are properly declared and initialized pointers, then the following statements are valid:

```
y = *p1 **p2;
sum = sum + *p1;
*p2 = *p2 + 10;
```

Eg:

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

```
int quantity1=179, quantity2=80;
int *p = &quantity1, *q=&quantity2;
printf("Addition :%d\n", quantity1+quantity2);
printf("Addition: %d\n", *p+*q );
printf("Addition: %d\n",*p+quantity2);
}
```

output:

Addition :259
Addition : 259
Addition: 259

Eg:

```
#include<stdio.h>
int main()
{
int quantity1=179, quantity2=80;
int *p = &quantity1, *q=&quantity2;
printf("Subtraction: %d\n", quantity1-quantity2);
printf("Subtraction: %d\n", *p-*q);
printf("Subtraction: %d\n", *p-quantity2);
}
```

output:

Subtraction: 99
Subtraction: 99
Subtraction: 99

Pointers and Arrays:

* When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations.
* The base address is the location of the first element (index 0) of the array.
* The compiler also defines the array name as a constant pointer to the first element.

int X[5] = {1, 2, 3, 4, 5};

* Suppose the base address of X is 1000 and assuming that each integer requried two bytes, the five elements will be stored as follows:

Elements →	X[0]	X[1]	X[2]	X[3]	X[4]
Value →	1	2	3	4	5
Address →	1000	1002	1004	1006	1008

* The name X is defined as a constant pointer pointing to the first element, X[0] and therefore the value of X is 1000, the location

where X[0] is stored.

* That is,

  X = X[0] = 1000

* If declare p as an integer pointer, then can make the pointer p to point to the array X by the following assignment:

  p = X; (equivalent to) p = &X[0];

* Now, can access every value of X using p++ to move from one element to another.

  p = &X[0] = 1000

  p+1 = &X[1] = 1002

  p+2 = &X[2] = 1004

Eg:

```
#include<stdio.h>
int main()
{
  int numbers[5]={1,2,3,4,5};
  int *p = numbers;
  printf("Array elements\n");
  for(int i=0; i<5; i++)
```

```
{
  printf("Element %d : %d\n", i, numbers[i]);
}
printf("Array elements using pointers\n");
for(int i=0; i<5; i++)
{
  printf("Element %d : %d\n", i, *(p+i));
}
*(p+1) = 20;
printf("Array elements using pointers after
                change\n");
for(int i=0; i<5; i++)
{
  printf("Element %d : %d\n", i, *(p+i));
}
}
```


output:

```
Array elements
Element 0 : 1
Element 1 : 2
Element 2 : 3
Element 3 : 4
Element 4 : 5
Array elements using pointers
```

```
Element 0: 1
Element 1: 2
Element 2: 3
Element 3: 4
Element 4: 5
Array elements using pointers after change
Element 0: 1
Element 1: 20
Element 2: 3
Element 3: 4
Element 4: 5
```


Pointers to Function:

* A function like a variable has a type and an address location in the memory.
* It is therefore, possible to declare a pointer to a function which can then be used as an argument in another function.
* A pointer to a function is declared as follows:

```
type (*fptr) ();
```

* This tells the compiler that fptr is a pointer to a function which returns type value.
* The parenthesis around *fptr are necessary.
* It can make a function pointer to point to a specific function by simply assigning the name of the function to the pointer.

Eg:

```
int mul (int, int);
int (*p) ();
p = mul;
```

* It will declare p as a pointer and mul as a function and then make p to point to the function mul.
* To call the function mul, use the pointer p with the list of parameters.

```
(*p) (x, y);
```

This is equivalent to,

```
mul (x, y);
```

Note:
The parenthesis around *p

Eg:

```
#include<stdio.h>
int add(int a, int b);
int main()
{
   int num1=5, num2=3;
   int (*p)(int,int) = add;
   printf("Addition using function:%d\n",
                        add(num1,num2));
   printf("Addition using pointer function:
                %d\n", p(num1,num2));
}
int add(int a,int b)
{
   return a+b;
}
```


Output:

```
Addition using function:8
Addition using pointer function:8
```

Eg:

```
#include<stdio.h>
int mul(int a, int b);
int main()
{
```

```
int num1=5, num2=3;
int (*p)(int,int)= mul;
printf("Multiply using function:%d\n",
                    mul(num1,num2));
printf("multiply using pointer function:
                %d\n", p(num1,num2));
}
int mul(int a, int b)
{
  return a*b;
}
```

Output:

```
Multiply using function:15
Multiply using pointer function:15
```


Dynamic Memory Allocation:

* C language requires the number of elements in an array to be specified at compile time. But may not be able to do so always.
* Our initial judgement of size, if it is wrong, may cause failure of the program or wastage of memory space.
* The process of allocating memory at runtime is known as dynamic memory

allocation.

* Although C does not have this facility, there are H library routines known as memory management functions that can be used for allocating and freeing memory during program execution.

Memory Allocation:

Function	Task
malloc	Allocates request size of bytes and returns a pointer to the first byte of the allocated space
calloc	Allocates space for an array of elements, initialized them to zero and then returns a pointer to the memory.
free	Free previously allocated space
realloc	modifies the size of previously allocated space.

Eg: malloc Function

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

```
int main()
{
  int *arr;
  int n;
  printf("Enter number of elements:\n");
  scanf("%d", &n);
  arr = (int *)malloc(n*sizeof(int));
  printf("Enter %d integers:\n", n);
  for(int i=0; i<n; i++)
  {
    scanf("%d", &arr[i]);
  }
  printf("\n You entered :\n");
  for(int i=0; i<n; i++)
  {
    printf("%d ", arr[i]);
  }
  return 0;
}
```


output:

```
Enter number of elements: 4
Enter 4 integers:
1
2
3
4
```

You entered: 1 2 3 4

Eg: Calloc Function

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *arr;
    int n;
    printf("Enter number of elements:\n");
    scanf("%d", &n);
    arr = (int *) calloc(n, sizeof(int));
    printf("Enter %d integers:\n", n);
    for(int i=0; i<n; i++)
    {
        scanf("%d", &arr[i]);
    }
    printf("\n You entered:\n");
    for(int i=0; i<n; i++)
    {
        printf("%d", arr[i]);
    }
    return 0;
}
```

output:

```
Enter number of elements:3
Enter 3 integers:
```

```
1
2
3
You entered: 1 2 3
```

Eg: Free Function

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
  int *arr;
  int n;
  printf("Enter number of elements:\n");
  scanf("%d", &n);
  arr = (int *) calloc(n, sizeof(int));
  printf("Enter %d integers:\n", n);
  for(int i=0; i<n; i++)
  {
    scanf("%d", &arr[i]);
  }
  printf("\n You Entered:\n");
  for(int i = 0; i<n; i++)
  {
    printf("%d", arr[i]);
  }
  free(arr);
  return 0;
}
```

Output:

```
Enter number of elements: 3
Enter 3 integers:
1
2
3
You entered: 1 2 3
```

Eg: Realloc Function

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
  int *arr;
  int n, new_n;
  printf("Enter number of elements:\n");
  scanf("%d", &n);
  arr = (int *) malloc(n*sizeof(int));
  printf("Enter %d integers:\n", n);
  for(int i=0; i<n; i++)
  {
    scanf("%d", &arr[i]);
  }
  printf("\n You entered:\n");
  for(int i=0; i<n; i++)
  {
    printf("%d", arr[i]);
  }
  printf("\nEnter new size:\n");
```

```
scanf("%d", &new_n);
arr = (int *) realloc(arr, new_n * sizeof(int));
if(new_n > n)
{
  printf("Enter %d more elements:\n", new_n - n);
  for(int i = n; i < new_n; i++)
  {
    scanf("%d", &arr[i]);
  }
}
printf("\n Final array:\n");
for(int i = 0; i < new_n; i++)
{
  printf("%d ", arr[i]);
}

return 0;
}
```

output:

```
Enter number of elements: 5
Enter 5 integers:
1
2
3
4
5
You entered: 1 2 3 4 5
```

Enter new size: 7
Enter 2 more elements:
6
7
Final array: 1 2 3 4 5 6 7

Output:

Enter number of elements: 5
Enter 5 integers:
1
2
3
4
5
You Entered: 1 2 3 4 5
Enter new size: 2
Final Array: 1 2

Structures and Unions

Defining and using Structures:

Structures:

* Arrays can be used to represent a group of data items that belongs to the same type.
* However, cannot use array if want to represent a collection of data items of different types using a single name.
* C supports a constructed data type known as Structures, a mechanism for packing data of different types.
* A structure is a convenient tool for handling a group of logically related data items.
* Structures help to organize complex data in a more meaningful way.

Defining a Structure:

* Unlike arrays, structures must be defined first for their format that may be used later to declare structure variables.

* The general format of a structure definition is as follows:

```
Struct tag_name
{
   data_type member 1;
   data_type member 2;
   - - - - - - - - - - - -
   - - - - - - - - - - - -
};
```

Eg:

```
#include <stdio.h>
Struct Student
{
   int id;
   char name[50];
   float marks;
};
int main()
{
   return 0;
}
```


Declaring Structure Variables:

* After defining a structure format can declare variables of that type.
* A structure variable declaration is similar to the declaration of variables

of any other data types.

* It includes the following elements:

i) The keyword Struct
ii) The structure tag name
iii) List of variable names separated by commas
iv) A terminating semicolon

Eg:

```
#include<stdio.h>
Struct Student
{
    int id;
    char name[50];
    float marks;
};
int main()
{
    Struct Student S1;
    printf("Enter the input\n");
    Scanf("%d %s %f", &S1.id, S1.name, &S1.marks);
    printf("Student ID:%d\n", S1.id);
    printf("Student Name:%s\n", S1.name);
    printf("Student Marks:%.2f\n", S1.marks);
    return 0;
}
```

Output:

```
Enter the input
101
Jeremiah
67.35
Student ID: 101
Student Name: Jeremiah
Student Marks: 67.35
```

Eg:

```
#include <stdio.h>
#include <string.h>
Struct Student
{
    int id;
    char name [50];
    float marks;
};
int main()
{
    Struct Student s1;
    s1.id = 102;
    Strcpy (s1.name, "Jeremiah");
    s1.marks = 24.46;
    printf("Student ID: %d\n", s1.id);
    printf("Student Name: %s\n", s1.name);
    printf("Student Marks: %.2f\n", s1.marks);
```

```
    return 0;
}
```

output:

```
Student ID: 102
Student Name: Jeremiah
Student Marks: 24.46
```

Structure Initialization:

* Like any other data type, a structure variable can be initialized at compile time.
* The compile-time initialization of a structure variable must have the following elements:

i) The keyword struct
ii) The structure tag name
iii) The name of the variable to be declared
iv) The assignment operator =
v) A set of values for the members of the structure variable separated by commas and enclosed in braces.
vi) A terminating semicolon

Eg:

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

```
Struct Student
{
   int id;
   char name[50];
   float marks;
};
int main()
{
   Struct Student S1 = {105, "Jeremiah", 56.87};
   printf("Student ID: %d\n", S1.id);
   printf("Student Name: %s\n", S1.name);
   printf("Student Marks: %.2f\n", S1.marks);
   return 0;
}
```


output:

```
Student ID: 105
Student Name: Jeremiah
Student Marks: 56.87
```

Copying and Comparing Structure Variables:

* Two variables of the same structure type can be copied the same way as ordinary variables.
* If person 1 and person 2 belong to the same structure, then the

following statements are valid:

```
person1 = person2;
person2 = person1;
```

* However, the statements such as,

```
person1 == person2
person2 == person1
```

are invalid

Eg:

```
#include <stdio.h>
#include <string.h>
struct student
{
  int id;
  char name[50];
  float marks;
};
int main()
{
 struct student S1 = {105, "Jeremiah", 56.87};
 struct student S2 = {107, "karthik", 90.87};
 printf("Student 1 ID : %d\n", S1.id);
 printf("Student 1 Name : %s\n", S1.name);
 printf("Student 1 Marks : %.2f\n", S1.marks);
 printf("Student 2 ID : %d\n", S2.id);
```

```
printf("Student 2 Name: %s\n", s2.name);
printf("Student 2 Marks: %.2f\n", s2.marks);
return 0;
}
```

output:-

```
Student 1 ID: 105
Student 1 Name: Jeremiah
Student 1 Marks: 56.87
Student 2 ID: 107
Student 2 Name: karthik
Student 2 Marks: 90.87
```


Array of Structures:-

* It use Structures to describe the format of a number of related variables.
* For example, in analyzing the marks obtained by a class of students, may use a template to describe student name and marks obtained and then declare all the students as Structure Variables.
* In such cases, may declare an array of Structures, each element of an array

Representing a Structure Variable.

Eg:

```
Struct cse student[100];
```

Eg:

```
#include<stdio.h>
#include<string.h>
Struct cse
{
   int id;
   char name[50];
   float marks;
};
int main()
{
  Struct cse S1 = {105, "Jeremiah", 56.87};
  Struct cse S2 = {106, "karthik", 90.87};
  Struct cse S3 = {107, "Sathish", 99.50};
  printf("Student 1 ID: %d\n", S1.id);
  printf("Student 1 Name: %s\n", S1.name);
  printf("Student 1 marks: %.2f\n", S1.marks);
  printf("Student 2 ID: %d\n", S2.id);
  printf("Student 2 Name: %s\n", S2.name);
  printf("Student 2 Marks: %.2f\n", S2.marks);
  printf("Student 3 ID: %d\n", S3.id);
  printf("Student 3 Name: %s\n", S3.name);
  printf("Student 3 Marks: %.2f\n", S3.marks);
  return 0;
}
```

Output:

```
Student 1 ID : 105
Student 1 name : Jeremiah
Student 1 marks: 56.87
Student 2 ID : 106
Student 2 name: karthik
Student 2 marks: 90.87
Student 3 ID : 107
Student 3 Name: Sathish
Student 3 Marks: 99.50
```

Eg:

```
#include <stdio.h>
#include <string.h>
Struct cse
{
   int id;
   char name[50];
   float marks;
};
int main()
{
   Struct cse s[3]={{105,"Jeremiah",56.87},{106,
        "karthik",90.87},{107,"Sathish",99.50}};
printf("Student 1 ID: %d\n", s1.id);
printf("Student 1 Name: %s\n",s1.name);
printf("Student 1 marks: %.2f\n",s1.marks);
```

```
printf("Student 2 ID:%d\n", s2.id);
printf("Student 2 Name:%s\n", s2.name);
printf("Student 2 Marks:%.2f\n", s2.marks);
printf("Student 3 ID:%d\n", s3.id);
printf("Student 3 Name:%s\n", s3.name);
printf("Student 3 Marks:%.2f\n", s3.marks);
return 0;
}
```

Output:

```
Student 1 ID: 105
Student 1 Name: Jeremiah
Student 1 marks: 56.87
Student 2 ID: 106
Student 2 Name: karthik
Student 2 Marks: 90.87
Student 3 ID: 107
Student 3 Name: Sathish
Student 3 Marks: 99.50
```

Eg:

```
#include <stdio.h>
#include <string.h>
Struct cse
{
  int id;
  char name[50];
  float marks;
```

```
int main()
{
 Struct cse s[3]={{105,"Jeremiah",56.87},
      {106,"karthik",90.87},{107,"Sathish",99.50}};
 for(int i=0; i<3; i++)
 {
   printf("Student %d ID: %d\n", i, s[i].id);
   printf("Student %d Name: %s\n", i, s[i].name);
   printf("Student %d Marks: %.2f\n", i, s[i].marks);
 }
 return 0;
}
```


output:

```
Student 0 ID: 105
Student 0 Name: Jeremiah
Student 0 Marks: 56.87
Student 1 ID: 106
Student 1 Name: karthik
Student 1 Marks: 90.87
Student 2 ID: 107
Student 2 Name: Sathish
Student 2 Marks: 99.50
```

Pointers to Structures:

* The name of an array stands for the address of its Zeroth element.
* The same thing is true of the names of arrays of structure variables.

```
Struct cse
{
   int id;
   char name[50];
   float marks;
};
Struct cse s[3];
Struct cse *ptr = s;
```


* This statement declares s as an array of 3 elements, each of the type struct cse and ptr as a pointer to data objects of the type struct cse.
* The assignment,

```
*ptr = s;
```

would assign the address of the Zeroth element of s to ptr.

* That is, the pointer ptr will now point to s[0].
* Its members can be accessed using the following notation:

ptr → id

ptr → name

ptr → price

Eg:

```
#include <stdio.h>
#include <String.h>
Struct cse
{
   int id;
   char name[50];
   float marks;
};
int main()
{
  Struct cse s[3]={{105,"Jeremiah",56.87},{106,
        "karthik",90.87},{107,"Sathish",99.50}};
  Struct cse *ptr = s;
  for(int i = 0; i<3; i++)
  {
    printf("Student %d ID: %d\n", i, (ptr+i)->id);
    printf("Student %d Name: %s\n", i, (ptr+i)->name);
    printf("Student %d Marks: %.2f\n", i, (ptr+i)->marks);
  }
  return 0;
}
```

Output:

```
Student 0 ID: 105
Student 0 Name: Jeremiah
Student 0 Marks: 56.87
Student 1 ID: 106
Student 1 Name: Karthik
Student 1 Marks: 90.87
Student 2 ID: 107
Student 2 Name: Sathish
Student 2 Marks: 99.50
```


Unions and their Uses:-

Unions:-

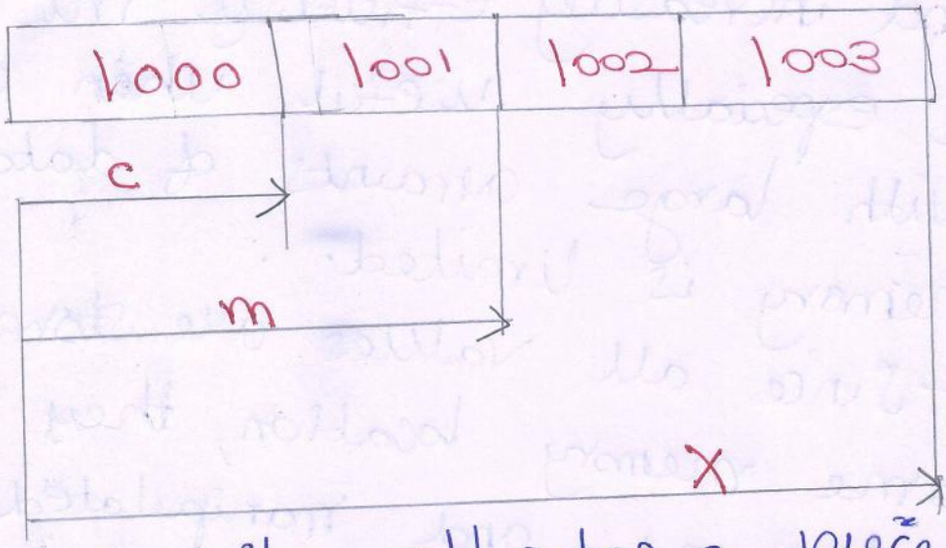
* Unions are a concept borrowed from Structures and therefore follow the same syntax as structures.
* However, there is major distinction between them interms of storage.
* In structures, each member has its own storage location, whereas all the members of the union use the same location.
* This implies that, although a union

may contain many members of different types, it can handle only one member at a time.

* Like structures, a union can be declared using the keyword union as follows:

```
union item
{
    int m;
    float x;
    char c;
} code;
```


* This declares a variable code of type union item.
* The union contain three members each with a different data type.
* However, can use only one of them at a time.
* This is due to the fact that only one location is allocated for a union variable, irrespective of its size.



* The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union.
* In the declaration above, the member X requires 4 bytes which is the largest among the members.
* To access a union member, Can use the same syntax that use for structure members.
* That is,

```
Code.m;
Code.X;
Code.c;
```

Uses:-

* Union allows data to be stored in one unit of memory, saving space

and increasing efficiency. This makes it especially useful when dealing with large amounts of data or when memory is limited.

* Since all values are stored in the same memory location, they can easily be accessed and manipulated by a single pointer. This makes it much simpler to manage complex data structures.

* By using a union, different types of variables can be stored in the same memory location. This means that different types of values can be accessed without having to create multiple variables or cast from type to type.

Eg:

```
#include<stdio.h>
union cse
{
  int id;
  float marks;
};
```

```
int main()
{
    union cse s;
    s.id = 100;
    printf("ID: %d\n", s.id);
    s.marks = 98.27;
    printf("Marks: %.2f\n", s.marks);
    return 0;
}
```

output:

```
ID: 100
Marks: 98.27
```


Eg:

```
#include<stdio.h>
union cse
{
    int id;
    float marks;
};
int main()
{
    union cse s;
    s.id = 100;
    printf("ID: %d\n", s.id);
    s.marks = 98.27;
```

```
printf("ID:%d\n", s.id);
printf("Marks:%.2f\n", s.marks);
return 0;
}
```

output:

ID: 100

ID: 1120176701

Marks: 98.27

Enumerations:

* An enum is a user-defined data type in C used to assign names to a set of integer constants.
* It improves code readability by replacing numeric values with descriptive names.
* The syntax is,

```
enum enumname
{
    Constant1;
    Constant2;
    ....
};
```

* It can also assign custom values:

```
enum enumname
{
  Constant1 = 10,
  Constant2 = 15
};
```

* The enum is used to replace magic numbers with meaningful names.
* It can be used in switch statements for better logic control.

Eg:

```
#include<stdio.h>
enum day{ SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY};

int main()
{
  enum day today = SATURDAY;
  printf("Day number = %d\n", today);
  return 0;
}
```


output:

```
Day number = 6
```

Eg:

```
#include<stdio.h>
enum day{ SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
          THURSDAY, FRIDAY, SATURDAY};

int main()
```

```
{
  enum day today = MONDAY;
  printf("Day number = %d\n", today);
  return 0;
}
```

output:

Day number = 1

Eg:

```
#include<stdio.h>
int main()
{
  enum day{SUNDAY=1, MONDAY, TUESDAY,
           WEDNESDAY, THURSDAY, FRIDAY, SATURDAY};
  printf("Day number = %d\n", TUESDAY);
  return 0;
}
```


output:

Day number = 3

Eg:

```
#include<stdio.h>
int main()
{
  enum day{SUNDAY=5, MONDAY, TUESDAY};
  printf("Day number = %d\n", TUESDAY);
  return 0;
}
```

Output:

Day number = 7

Eg:

```
#include<stdio.h>
int main()
{
  enum day{SUNDAY=5, MONDAY=18,
           TUESDAY=56};
  printf("Day number = %d\n", TUESDAY);
  return 0;
}
```


Output:

Day number = 56

File operations

Open, close File operations and File pointers:

File Management in C:

* A file is a place on disk where a group of related data is stored.
* C supports a number of functions that have the ability to perform basic file operations which include:
  i) Naming a file
  ii) opening a file
  iii) Reading data from a file
  iv) writing data to a file
  v) closing a file

opening a File:

* Data Structure of a file is defined as FILE in the library of Standard I/o functions definitions.
* Therefore, all files should be declared as type FILE before they are used.
* FILE is a defined data type.
* When open a file, specify what want to do with the file.

* For example, may write data to the file or read the already existing data.
* Following is the general format for declaring and opening a file:

```
FILE *fp;
fp=fopen("filename","mode");
```

* The first statement declared the variable fp as a pointer to the data type FILE.
* The second statement opens the file named filename and assigns an identifier to the FILE type pointer fp.
* This pointer which contains all the information about the file is subsequently used as a communication link between the system and the program.
* The second statement also specifies the purpose of opening this file.
* The mode does this job.
* Mode can be one of the following:

r → open the file for reading only
w → open the file for writing only

a → open the file for appending data to it

Closing a File:

* A file must be closed as soon as all operations on it have been completed.
* This ensures that all outstanding information associated with the file is flushed out from the buffers and all links to the file are broken.
* It also prevents any misuse of the file.
* The general form is:

```
fclose(file_pointer);
```


Eg:

```
#include <stdio.h>
int main()
{
  FILE *file;
  file=fopen("example.txt","r");
  if(file==NULL)
  {
    printf("Error opening file\n");
    return 1;
  }
  printf("File opened successfully\n");
```

```
    fclose(file);
    printf("File closed successfully\n");
    return 0;
}
```

output:

File opened successfully
File closed successfully

Eg:

```
#include<stdio.h>
int main()
{
  FILE *file;
  file = fopen("Hello.txt","r");
  if(file==NULL)
  {
    printf("Error opening file\n");
    return 1;
  }
  printf("File opened successfully\n");
  fclose(file);
  printf("File closed successfully\n");
  return 0;
}
```

output:

Error opening file

```
#include <stdio.h>
int main()
{
  FILE *file;
  file = fopen("Hello.txt", "w");
  if (file == NULL)
  {
    printf("Error opening file\n");
    return 1;
  }
  printf("File opened successfully\n");
  fclose(file);
  printf("File closed successfully\n");
  return 0;
}
```

Output:

File opened successfully
File closed successfully

Eg:

```
#include <stdio.h>
int main()
{
  FILE *file;
  file = fopen("Hi.txt", "a");
  if(file == NULL)
  { printf("Error opening file\n");
```

```
    return 1;
    }
    printf("File opened successfully\n");
    fclose(file);
    printf("File closed successfully\n");
    return 0;
}
```

Output:

```
File opened successfully
File closed successfully
```

Read, Write File operations and File pointers:

1) The getc and putc functions:

* The simplest I/O functions are getc and putc.
* These are analogous to getchar and putchar functions and handle one character at a time.
* The general forms are:

```
c = getc(fp2);
putc(c, fp1);
```

* Here, c is a character variable and fp1 and fp2 are file pointers.

Eg:

```
#include <stdio.h>
int main()
{
    FILE *source, *target;
    char ch;
    source = fopen("Input.txt", "r");
    target = fopen("output.txt", "w");
    if(source == NULL || target == NULL)
    {
        printf("Error opening file\n");
        return 1;
    }
    while((ch=getc(source))!=EOF)
    {
        putc(ch, target);
    }
    printf("written in target file\n");
    fclose(source);
    fclose(target);
}
```

output:

written in target file

2) The getw and putw functions:

* The getw and putw are integer-oriented functions.
* They are similar to the getc and putc functions and are used to read and write integer values.
* These functions would be useful when dealing with only integer data.
* The general forms are:

```
getw(fp);
putw(integer, fp);
```


Eg:

```
#include <stdio.h>
int main()
{
  FILE *source, *target;
  int num;
  source = fopen("Input_int.txt", "r");
  target = fopen("output_int.txt", "w");
  if(source == NULL || target == NULL)
  {
    printf("Error opening file\n");
    return 1;
  }
```

```
while( (num=getw(source)) != EOF )
{
  putw(num, target);
}
printf("written in target file\n");
fclose(source);
fclose(target);
}
```

Output:

Written in target file

3) The fprintf and fscanf functions:

* The functions fprintf and fscanf perform I/O operations that are identical to the familiar printf and scanf functions.
* The general form of fprintf and fscanf are:

```
fprintf("fp, "control string", list);
fscanf("fp, "control string", list);
```

Eg:

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

```
FILE *source, *target;
int roll;
char name[50];
float marks;
source = fopen("Input_Marks.txt", "r");
target = fopen("output_Marks.txt", "w");
if(source == NULL || target == NULL)
{
 printf("Error opening file\n");
  return 1;
}
while(fscanf(source, "%d %s %f", &roll,
                name, &marks) != EOF)
{
 fprintf(target, "%d %s %.2f\n", roll,
                    name, marks);
}
printf("written in target file\n");
 fclose(source);
 fclose(target);
}
```

output:

written in target file

Binary Vs Text Files :-

* A text file contains only textual information like alphabets, digits and special symbols.
* A binary file is merely a collection of bytes.
* This collection might be a compiled version of C program (test.exe) or music data stored in a MP4 file or a picture stored in a JPG file.
* When file is opened in binary mode, can use the functions fread() and fwrite().
* The binary file needs to be opened using the modes wb/rb for write and read modes respectively.

Eg:

```
#include <stdio.h>
Struct Student
{
  int id;
  char name[50];
  float marks;
};
```

```
int main()
{
  Struct Student S1={1, "Jeremiah", 92.5};
  FILE *fptr = fopen("Student.dat", "wb");
  if(fptr==NULL)
  {
    printf("Error opening file\n");
    return 1;
  }
  fwrite(&S1, sizeof(Struct Student), 1, fptr);
  printf("Written in file\n");
  fclose(fptr);
  return 0;
}
```

output:

Written in file

Eg:

```
#include<stdio.h>
Struct Student
{
  int id;
  char name[50];
  float marks;
};
int main()
```

```
{
  struct Student S2;
  FILE *fptr = fopen("Student.dat", "rb");
  if(fptr == NULL)
  {
    printf("Error opening File\n");
    return 1;
  }
fread(&S2, sizeof(struct Student), 1, fptr);
fclose(fptr);
printf("Id: %d\n", S2.id);
printf("Name: %s\n", S2.name);
printf("Marks: %.2f\n", S2.marks);
return 0;
}
```

output:

Id : 1

Name : Jeremiah

Marks : 92.50

Error Handling in File operations:

* It is possible that an error may occur during I/O operations in a file.
* Typical error situations include:

1) Trying to read beyond end-of-file mark.
2) Device overflow
3) Trying to use a file that has not been opened.
4) Trying to perform an operation on a file, when the file is opened for another type of operation.
5) opening a file with an invalid filename
6) Attempting to write to a write-protected file.

* If fail to check such read and write errors, a program may behave abnormally when an error occurs.
* An unchecked error may result in a premature termination of the program or incorrect output.
* C has two status-inquiry library functions feof and ferror that can

help us detect I/O errors in the files.

* The feof function can be used to test for an end of file function.
* The ferror function reports the status of the file indicated and returns a non-zero indicator if any error is detected.

Eg:

```
#include<stdio.h>
int main()
{
  FILE *source, *target;
  char ch;
  source = fopen("input.txt", "r");
  target = fopen("output.txt", "w");
  if(source == NULL || target == NULL)
  {
    printf("Error opening file\n");
    return 1;
  }
while(!feof(source))
{
  ch = getc(source);
  if(!feof(source))
  {
```

```
        putc(ch, target);
    }
}
printf("written in target file\n");
fclose(source);
fclose(target);
return 0;
}
```

output:

written in target file

Eg:

```
#include <stdio.h>
int main()
{
    FILE *source, *target;
    char ch;
    source = fopen("input.txt", "r");
    target = fopen("output.txt", "w");
    if(source == NULL || target == NULL)
    {
        printf("Error opening file\n");
        return 1;
    }
    while((ch = getc(source)) != EOF)
    {
        putc(ch target);
```

```
}
if(ferror(source))
    printf("Error reading from input file\n");
else if(ferror(target))
    printf("Error writing to output file\n");
else
    printf("written in target file\n");
fclose(source);
fclose(target);
return 0;
}
```

Output:

written in target file

Eg:

```
#include<stdio.h>
int main()
{
FILE *source, *target;
char ch;
source = fopen("input.txt", "r");
target = fopen("output.txt", "r");
if(source == NULL || target == NULL)
{
printf("error opening file\n");
```

```
    return 1;
  }
while ((ch = getc(source)) != EOF)
{
  putc(ch, target);
}
if (ferror(source))
  printf("Error reading from input file\n");
else if (ferror(target))
  printf("Error writing to output file\n");
else
  printf("written in target file\n");
fclose(source);
fclose(target);
return 0;
}
```

output:-

Error opening file

Eg:

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

```
char ch;
source = fopen("input.txt", "w");
target = fopen("output.txt", "w");
if(source == NULL || target == NULL)
{
  printf("Error opening file\n");
  return 1;
}
while((ch=getc(source)) != EOF)
{
  putc(ch, target);
}
if(ferror(source))
  printf("Error reading from input file\n");
else if(ferror(target))
  printf("Error writing to output file\n");
else
  printf("written in target file\n");
fclose(source);
fclose(target);
return 0;
}
```

output:

Error reading from input file