Course code	ETES203				
Category	Engineering Science Course				
Course title	Programming for Problem Solving				
Scheme and Credits	L	T	P	Credits	
	3	0	0	3	

Unit 1: Introduction to Programming, Introduction to components of a computer system (disks, memory, processor, where a program is stored and executed, operating system, compilers etc.), Idea of Algorithm: steps to solve logical and numerical problems. Representation of Algorithm: Flowchart/Pseudocode with examples. From algorithms to programs; source code, variables (with data types) variables and memorylocations, Syntax and Logical Errors in compilation, object and executable code. **(8 lectures)**

Unit 2: Arithmetic expressions and precedence, Conditional Branching and Loops, Writing and evaluation of conditionals and consequent branching, Iteration and loops. **(14 lectures)**

Unit 3: *Arrays: Arrays (1-D, 2-D), Character arrays and Strings, Basic Algorithms: Searching, Basic Sorting Algorithms (Bubble, Insertion and Selection), Finding roots of equations, notion of order of complexity through example programs (no formal definition required).* ***(12 lectures)***

Unit 4: Function: Functions (including using built in libraries), Parameter passing in functions, call by value, Passing arrays to functions: idea of call by reference, Recursion: Recursion, as a different way of solving problems. Example programs, such as Finding Factorial, Fibonacci series, Ackerman function etc. Quick sort or Merge sort. (10 lectures)

Unit 5:
Structure: Structures, Defining structures and Array of Structures, Pointers: Idea of pointers, Defining pointers, Use of Pointers in self-referential structures, notion oflinked list (no implementation). File handling (only if time is available, otherwise should be done as part of the lab). **(6 lectures)**

Suggested Text Books

(i) Byron Gottfried, Schaum's Outline of Programming with C,McGraw-Hill
(ii) E. Balaguruswamy, Programming in ANSI C, TataMcGraw-Hill

Suggested Reference Books

(i) Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, Prentice Hall ofIndia

Scanned with CamScanner

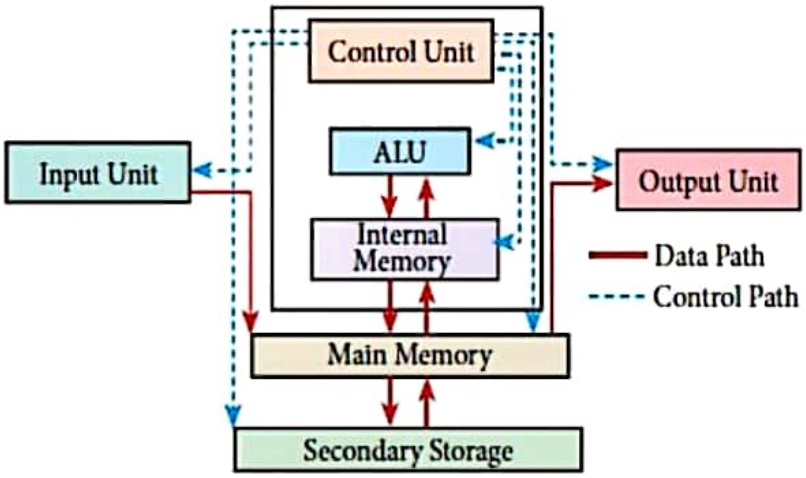
MODULE - I

COMPONENTS OF A COMPUTER

Computer is a combination of hardware and software. Hardware is the physical component of a computer like motherboard, memory devices, monitor, keyboard etc., while software is the set of programs or instructions. Both hardware and software together make the computer system.

Functional components of a computer

Every task given to a computer follows an Input- Process- Output Cycle (IPO cycle). It needs certain input, processes that input and produces the desired output. The input unit takes the input, the central processing unit does the processing of data and the output unit produces the output. The memory unit holds the data and instructions during the processing.



1. Input Unit

Input unit is used to feed any form of data to the computer, which can be stored in the memory unit for further processing. Example: Keyboard, mouse, light pen, joy stick etc.

2. Central Processing Unit

CPU is the major component which interprets and executes software instructions. It also controls the operation of all other components such as memory, input and output units. It accepts binary data as input, process the data according to the instructions and provide the result as output. The CPU has three components which are control unit, arithmetic and logic unit (ALU) and memory unit.

Scanned with CamScanner

2.1 Arithmetic and Logic Unit

The ALU is a part of the CPU where various computing functions are performed on data. The ALU performs arithmetic operations such as addition, subtraction, multiplication, division and logical operations. The result of an operation is stored in internal memory of CPU. The logical operations of ALU promote the decision-making ability of a computer.

2.2 Control Unit

The control unit controls the flow of data between the CPU, memory and I/O devices. It also controls the entire operation of a computer.

2.3. Memory Unit

The Memory Unit is of two types which are primary memory and secondary memory. The primary memory is used to temporarily store the programs and data when the instructions are ready to execute. The secondary memory is used to store the data permanently. The Primary Memory is volatile, that is, the content is lost when the power supply is switched off. The Random Access Memory (RAM) and ROM are the examples of a main memory. The Secondary memory is non volatile, that is, the content is available even after the power supply is switched off. Hard disk, CD-ROM, DVD-ROM etc are examples of secondary memory.

3. Output Unit

An Output Unit is a hardware component that conveys information to users in an understandable form. Example: Monitor, Printer, Plotter

---

FUNDAMENTALS OF C

C Programming is a general-purpose, procedural, imperative computer programming language developed in 1972 by Dennis M. Ritchie at the Bell Telephone Laboratories to develop the UNIX operating system. Dennis Ritchie is known as the founder of the c language.

Basic Structure of C Language:

The program written in C language follows a basic structure. It should have one or more sections but the sequence of sections is to be followed.

Scanned with CamScanner

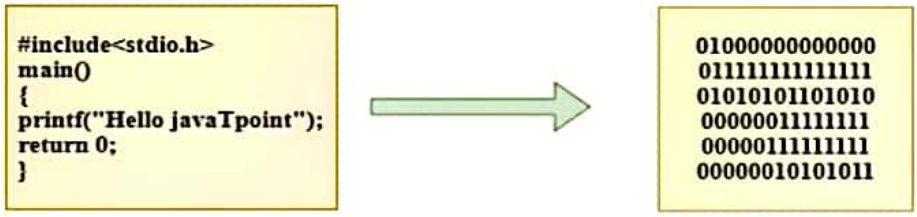
Section Name	Description
Documentation	Consists of comments, some description of the program, programmer name and any other useful points that can be referenced later. /* Program for addition*/
Link	Provides instruction to the compiler to link functions from the library file e.g. #include <stdio.h>
Definition	Consists of symbolic constants. #define PI 3.14
Global declaration	Consists of function declaration and global variables. .int i;

main( )
{
 	
}
	Every C program must have a main() function which is the starting point of the program execution.
	It has two sections
	1. Declaration section: In this the variables are declared.
	2. Executable section: This has the part of program which actually performs the task we need.

Subprograms	User defined functions.

COMPILATION PROCESS IN C

Compilation is a process of converting the source code into object code.



The compilation process in C can be divided into four steps, i.e., Pre-processing, Compiling, Assembling, and Linking.

1. Preprocessing

The source code is the code which is written using a text editor by a programmer. The source code file is saved with an extension ".c". This source code file is first passed to the preprocessor. Preprocessor removes all the comments from the source code. Then the preprocessor takes the preprocessor directive and interprets it. For example, if #include **<stdio.h>** directive is available in the program, preprocessor replace this directive with the content of the **'stdio.h'** file. Thus the code is **expanded** and is passed to the next step. The extension of the expanded file is **'.i'**

Scanned with CamScanner

2. Compiling

The code which is expanded by the preprocessor is passed to the compiler. The compiler converts this code into **assembly code** which contains mnemonics. The extension of the assembly file is '**.s**'

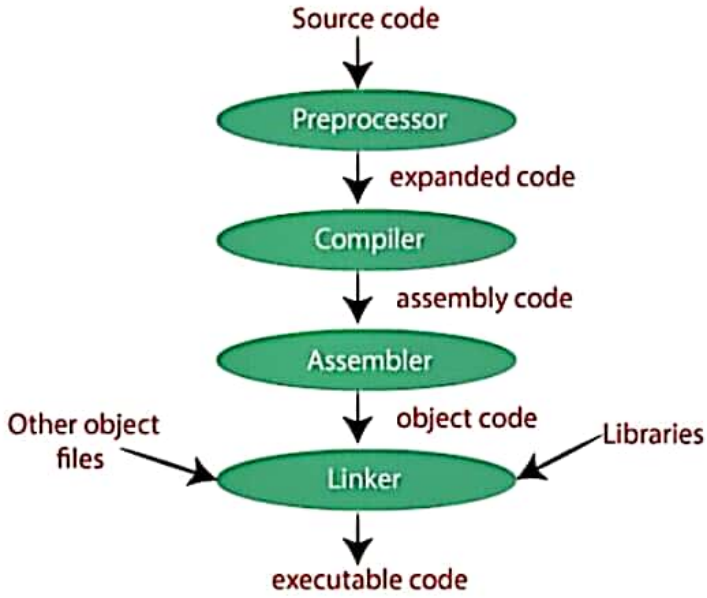
3. Assembling

The assembly code is converted into **object code** by using an assembler. Object code will contain only "0" and "1". The extension of object file is '**.obj**'

4. Linking

All programs in C use library functions. These library functions are pre-compiled, and the object code of these library files is stored with '**.lib**' extension. The main working of the linker is to combine the object code of library files with the object code of source program. For example, if we are using printf() function in a program, then the linker adds its associated code with the object code of our program. The output of the linker is **executable code**. The extension of the executable file is '.exe',

Flow Diagram



Example

Compilation process of **hello.c** source code file

Scanned with CamScanner

- Firstly, the input file, i.e., **hello.c,** is passed to the preprocessor, and the preprocessor converts the source code into expanded source code file **hello.i.**
- The expanded source code is passed to the compiler, and the compiler converts this expanded source code into assembly code file **hello.s.**
- This assembly code is then sent to the assembler, which converts the assembly code into object code which is in the form of 0 and 1. The name of the object code file would be **hello.obj.**
- After the creation of an object code, the linker creates the executable file **hello.exe.** The loader will then load the executable file for execution.

Difference between Source Code and Object Code

Source Code	Object Code
Created by the programmer	Created by the Compiler
Text rich document	Binary digits make up the Object Code
Human Readable	Machine Readable
Not system specific	System specific
Serves as input to the compiler	It is the output of the compiler
Source code is not executable	Object code is executable

TOKENS

The smallest individual units in a program are known as tokens.

Classification of tokens in C

Tokens in C language can be divided into the following categories:

- Keywords
- Identifiers
- Operators
- Constants
- Special Characters

Scanned with CamScanner

1. Keywords

Keywords in C can be defined as the **pre-defined** or the **reserved words** having its own importance, and each keyword has its own functionality. Since keywords are the pre-defined words used by the compiler, they cannot be used as the variable names. C language supports **32** keywords given below:

auto	**Double**	**int**	**struct**
break	**Else**	**long**	**switch**
case	**Enum**	**register**	**typedef**
char	**Extern**	**return**	**union**
const	**Float**	**short**	**unsigned**
continue	**For**	**signed**	**void**
default	**Goto**	**sizeof**	**volatile**
do	**If**	**static**	**while**

2. Identifiers

Identifiers in C are used for naming variables, functions, arrays, structures, etc. Identifiers in C are the user-defined words. It can be composed of uppercase letters, lowercase letters, underscore, or digits, but the starting letter should be either an underscore or an alphabet. Keywords cannot be used as identifiers. Rules for constructing identifiers in C are given below:

- The first character of an identifier should be either an alphabet or an underscore, and then it can be followed by any of the character, digit, or underscore.
- It should not begin with any numerical digit.
- In identifiers, both uppercase and lowercase letters are distinct. Therefore, we can say that identifiers are case sensitive.
- Commas or blank spaces cannot be specified within an identifier.
- Keywords cannot be represented as an identifier.
- The length of the identifiers should not be more than 31 characters.
- Identifiers should be written in such a way that it is meaningful, short, and easy to read.

Example of valid identifiers

total, sum, average, _m _, sum_1, etc.

Example of invalid identifiers

- 2sum (starts with a numerical digit)

Scanned with CamScanner

- **int** (reserved word)
- **char** (reserved word)
- m+n (special character, i.e., '+')

3. Operators

An operator in C is a special symbol used to perform the functions. The data items on which the operators are applied are known as operands. Operators are applied between the operands.

Depending on the number of operands, operators are classified as follows:

i) Unary Operator ---- Contain 1 operand

ii) Binary Operator ---- Contain 2 operands

4. Constants

A quantity that does not vary during the execution of a program is known as a constant. C supports three types of constants

1. Numeric constants - Eg. 5.2, 8 ...
2. Character constants. - Eg. 'a', 'b'...
3. String constant - Eg. "apple",

Strings in C

Strings in C are always represented as an array of characters having null character '\0' at the end of the string. This null character denotes the end of the string. Strings in C are enclosed within double quotes, while characters are enclosed within single characters. The size of a string is a number of characters that the string contains.

Eg. char a[10] = "hello"; // The compiler allocates the 6 bytes to the 'a' array.

There are two ways of declaring constant:

1. Using const keyword
2. Using #define pre-processor

1) const keyword

The const keyword is used to define constant in C programming.

```
const float PI=3.14;
```

The value of PI variable can't be changed.

If we try to change the value of PI, it will render compile time error.

Scanned with CamScanner

```
#include<stdio.h>
int main()
{
const float PI=3.14;
PI=4.5;
printf("The value of PI is: %f",PI);
}
```

Output:

Compile Time Error: Cannot modify a const object

2) C #define preprocessor

Syntax:

```
#define value
```

Example

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

Output:

3.140000

5. Special characters

Some characters used in C have a special meaning which cannot be used for another purpose.

- **Square brackets [ ]:** The opening and closing brackets represent the single and multidimensional subscripts.
- **Simple brackets ( ):** It is used in function declaration and function calling. For example, printf() is a pre-defined function.
- **Curly braces { }:** It is used in the opening and closing of blocks of code.

Scanned with CamScanner

- **Comma (,):** It is used for separating variables, separating function parameters in a function call.
- **Hash/pre-processor (#):** It is used to specify pre-processor directives.
- **Asterisk (*):** This symbol is used to represent pointers and also used as an operator for multiplication.
- **Tilde (~):** It is used as a bitwise operator and as a destructor to free memory.
- **Period (.):** It is used to access a member of a structure or a union.

VARIABLES

Variable is defined as named memory location. It is used to store data. Its value will be changed during execution. It must be declared first to reserve memory location for storing the value of the variable.

Variable Declaration

```
data type variable_list;
```

Example:

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

Here, a, b, c is variables. The int, float and char are the data types.

We can also provide values while declaring the variables. This is called as variable initialization.

```
int a=10,b=20;   //declaring 2 variable of integer type
char c='A';
```

Rules for defining variables

- A variable can have alphabets, digits, and underscore.
- A variable name can start with the alphabet and underscore only. It can't start with a digit.
- No whitespace is allowed within the variable name.
- A variable name must not be any reserved word or keyword, e.g. int, float, etc.

Valid variable names:

```
int a;
int _ab;
int a30;
```


Scanned with CamScanner

Invalid variable names:

```
int 2;
int a b;
int long;
```

Types of Variables

There are two ways to categorize variables

1. Based on datatype : Depending on the type of data it holds, variable is classified into integer variable, floating point variable, character variable and string variable
2. Based on storage class associated with a variable such as automatic, external, static , register

1. Automatic Variable

All variables in C that are declared inside the block are automatic variables by default. We can explicitly declare an automatic variable using **auto keyword**. This is also called as **local variable**. These variables are confined to a single function. It does not retain its value once control is transferred out of the defined function.

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

2. External Variable

These variables are not confined to a single function. We can share this variable in multiple C source files. This is also called as **global variable**. We can declare an external variable using **extern keyword**.

```
extern int a;
```

3. Static Variable

A variable that is declared with the static keyword is called static variable. It retains its value between multiple function calls. We can declare static variable using **static keyword**.

```
void function1()
{
int x=10;     //local variable
```


Scanned with CamScanner

```
static int y=10;  //static variable
x=x+1;
y=y+1;
printf("%d,%d",x,y);
}
```

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

4. Register variable

Values of register variables are stored in registers found in CPU rather than in memory.

We can declare a register variable using register **keyword**.

```
register int a;
```

DATA TYPES

C supports several different types of data. The memory requirements for each data type vary. A data type is essential to identify the storage representation and the type of operations that can be performed on that data. C supports four different classes of data types.

Types	Data Types
Basic Data Type	int, char, float, double
Derived Data Type	array, pointer, structure, union
User Defined Data Type	enum, type def
Void Data Type	void

1. Basic Data Types

The basic data types are integer-based and floating-point based. C language supports signed, unsigned, long, short modifiers along with integer data types.

The memory size of the basic data types may change according to 32 or 64-bit operating system.

In the table below size is given **according to 32-bit architecture**.

Scanned with CamScanner

Data Types	Memory Size	Range
Char	1 byte	−128 to 127
signed char	1 byte	−128 to 127
unsigned char	1 byte	0 to 255
Int	2 byte	−32,768 to 32,767
signed int	2 byte	−32,768 to 32,767
unsigned int	2 byte	0 to 65,535
short int	2 byte	−32,768 to 32,767
signed short int	2 byte	−32,768 to 32,767
unsigned short int	2 byte	0 to 65,535
long int	4 byte	-2,147,483,648 to 2,147,483,647
signed long int	4 byte	-2,147,483,648 to 2,147,483,647
unsigned long int	4 byte	0 to 4,294,967,295
Float	4 byte	
Double	8 byte	
long double	10 byte	

All arithmetic operations such as addition, subtraction etc is possible on basic data types.

2. Derived Data Types

Data types that are derived from fundamental data types are derived types.

Example: **arrays, pointers, structures, unions** etc.

3. User Defined Data types

Users can define new data types. This new data type can then be used to declare variables. The main advantage of user defined data type is that it increases the program's readability.

There are two methods

1. By using type def

Example

```
type def int  numbers;
numbers  num1,num2;
```

In this example, num1 and num2 are declared as int variables.

2. By using enum

It is used to assign names to constants which make a program easy to read and maintain. The keyword "enum" is used to declare an enumeration.

Scanned with CamScanner

Syntax:

enum identifier {const1,const2, const3,... }

Example:

enum week{sunday, monday, tuesday, wednesday, thursday, friday, saturday};

enum week day;

Compiler automatically assigns integer digits beginning from 0 to all the enumeration constants. For example, "sunday " will have value 0, "monday" value 1 and so on.

4. void data type

void is an incomplete type. It means "nothing" or "no type". For example, if a function is not returning anything, its return type should be void. Variables of void datatype cannot be created.

ALGORITHM

Definition:

Algorithm is a set of sequential well defined steps to solve a given problem. It should be precise, complete, unambiguous and contain finite number of logical steps for solving a problem.

Algorithm writing is the first step in problem solving.

Steps in algorithm development:

1. Identification of input
2. Identification of data processing operations
3. Identification of output

Example

Algorithm to find the area of the square

Step 1 : Start
Step 2: Read the side of the square a.
Step 3: Area = a*a
Step 4: Output the Area
Step 5: Stop.

FLOWCHART

Definition:

Pictorial representation of an algorithm is called as flowchart.

Description:

Flowchart shows the process involved in solving a problem and the flow of control in a visual manner. There are three types of control flow.

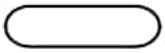
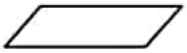
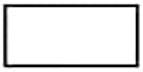
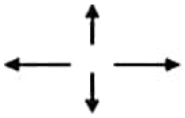
Scanned with CamScanner

1. Sequential - Statements are executed one after another in the same order as they are in the program.
2. Branching - Skipping execution of statements based on condition.
3. Looping - Repeated execution of statements until a condition is false.

A flowchart cannot be directly entered in a computer. It must be converted into a program using any high level language such as c, c++, java and etc.,

Symbols used in Flowcharts

Flow charts are drawn using certain special symbols such as Rectangles, Diamonds, Ovals and small circles. These symbols are connected by arrows called flow lines.

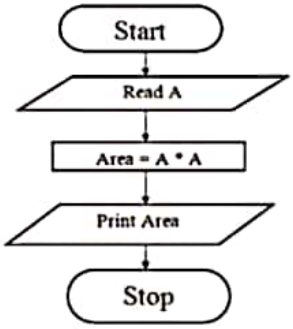
Symbol Name	Symbol	function
Oval Rounded Rectangle		Used to represent start and end of flowchart
Parallelogram		Used for input and output operation
Rectangle		Processing: Used for arithmetic operations and data-manipulations
Diamond		Decision making. Used to represent the operation in which there are two/three alternatives, true and false etc
Arrows		Flow line Used to indicate the flow of logic by connecting symbols
Circle		Page Connector
Pentagon		Off Page Connector

Elongated hexagon

Definite Loop

Scanned with CamScanner

Flowchart to find the area of the square



Uses of Flowchart

- Easier to understand, at a glance, than a narrative description.
- We can review and debug programs easily with the help of flowcharts.
- They provide effective program documentation.
- With a flowchart drawn, it is easier to explain a program or discuss a solution.
- Easy and efficient to analyze problem.
- Easy to convert the flowchart into any programming language code.

PSEUDO CODE

Pseudo code is a rough code. It is defined as an informal high level description of an algorithm in natural **language** rather than in a **programming language**.

It doesn't follow the syntax rules of any programming language. But it follows the structural conventions of a normal programming language.

It is intended for human reading rather than machine reading. It omits the details that are essential for machine understanding such as variable declaration, header file inclusion etc.

It is easy to write programs from pseudo code rather than flowchart. Pseudo Code is more commonly used by experienced programmers while Flowchart is by beginners.

We can write pseudo code freely as long as it is easy to understand for other persons. But it is suggested to use commonly used keywords from programs (i.e. if, then, else, while, do, repeat, for and etc.) and follow certain programming style (i.e. c, Pascal, C++, etc.).

Scanned with CamScanner

Common pseudo code verbs

- Input : Read, Obtain, get, Input
- Output: Print, Display, Show, Write
- Processing: Compute, Calculate, Determine
- Initialize: Set, Init
- Add one: Increment

Branching statements are written as

```
if ( conditional statement)
        statement block
else
        statement block
```

Looping statements are written as

1. while (Condition)
   statement block
2. Repeat
   statement block
   Until (Condition)

Example

```
Read n
Set sum to 0
Set i to 1
While (  i< n)
 Compute sum = sum +i
Display sum
```

INPUT AND OUTPUT STATEMENTS

The **getchar** is a simple function to read a single character from the input device.

```
varname=getchar();
```


Scanned with CamScanner

The **putchar** is a simple function to output a single character on the output device.

putchar(varname);

The getchar() and putchar() is used only for one input and is not formatted. For formatted input and output **scanf and printf statements** are used. . Both functions are library functions, defined in stdio.h (header file).

Scanf statement

Syntax

scanf("format specifier", &v1, &v2,...&vn);

Format specifier specifies the format in which data is to be entered.

Where v1,v2 are the variables

Example

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

%d used for integers

%f used for floats

%l used for long

%c used for character

%s used for string

printf statement

Syntax

printf("format specifier ", v1, v2,...vn);

Example

printf("%d",a);

SYNTAX AND LOGICAL ERRORS IN COMPILATION

Syntax Error

Each programming language has its own set of rules or syntax to write the program. Programmer should write the program according to the correct syntax. If not, it will cause an error. This error type is known as a syntax error. This error occurs at the time of compilation.

It is easy to identify and remove syntax errors because the compiler displays the location and type of error. Most frequent syntax errors are:

Scanned with CamScanner

- missing semicolons
- missing curly braces
- undeclared variables
- Mis-spelled keywords or identifiers.

The program will not get compiled until the syntax error is fixed.

Logic Error

Errors which provide incorrect output but appear to be error free are called logical errors. These errors occur due to faults in algorithm. A program with logical error will not cause the program to terminate the execution but the generated output is wrong. When a syntax error occurred, it is easy to detect the error because the compile specifies about error type and the line that the error occurs. But identifying a logical error is hard because there is no compiler message. Therefore, the programmer should read each statement and identify the error on his own. One example of logical error is the wrong use of operators. If the programmer used division (/) operator instead of multiplication (*), then it is a logical error.

Comparison of Syntax and Logic Error

Syntax Error	Logic Error
A syntax error occurs due to violation of rules of a programming language.	A logical error occurs due to a fault in the algorithm.
Compiler indicates the syntax error with the location and what the error is.	The programmer has to detect the error by himself.
It is easier to identify a syntax error.	It is comparatively difficult to identify a logical error.

OPERATING SYSTEM

An Operating System (OS) is a system software which is used to control and co-ordinate the activities of computer.

Examples:

- Windows, Unix, Linux, MS-DOS etc

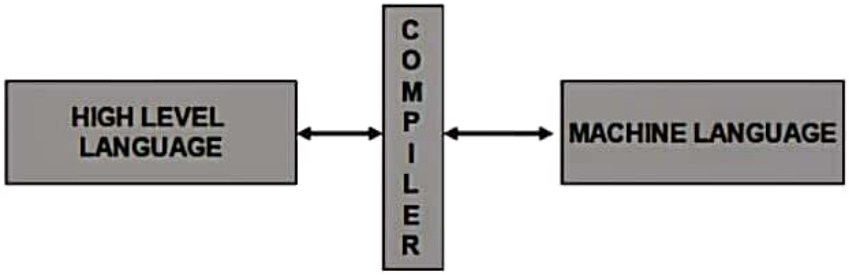
Functions of OS

- It provides an interface between the hardware and the user.
- It controls and co-ordinate the entire computer system.
- It controls the allocation and use of various resources.
- It controls various application programs.

Scanned with CamScanner

COMPILER

A compiler is a system software that transforms the high level language into machine level language. The program written in high level language is known as source program and the corresponding machine level language program is called as object program. Compiler read the program at-a-time and searches the error and lists them. If the program is error free then it is converted into object program.



DIFFERENCE BETWEEN SYSTEM SOFTWARE AND APPLICATION SOFTWARE

S. No.	System Software	Application Software
1.	System software is a general purpose software which is used for operating computer hardware.	Application software is a specific purpose software which is used by user to perform specific task.
2.	System softwares are installed on the computer when operating system is installed.	Application softwares are installed according to user's requirements.
3.	In general, the user does not interact with system software because it works in the background.	In general, the user interacts with application sofwares.
4.	System software can run independently. It provides platform for running application softwares.	Application software can't run independently. They can't run without the presence of system software.
5.	Some examples of system softwares are Operating System, compiler, assembler, debugger, driver, etc.	Some examples of application softwares are word processor, web browser, media player, etc.

Scanned with CamScanner

MODULE II

OPERATORS

An operator is a symbol used to perform mathematical, logical and relational operations.

C operators can be classified as

1. Unary Operators
2. Arithmetic operators
3. Relational operators
4. Logical operators
5. Assignment operator
6. Equality operator
7. Conditional operator
8. Bitwise operators
9. Special operators

1. UNARY OPERATORS:

The following table shows all the unary operators supported by the C language. Assume variable **A** holds 10 and variable **B** holds 20

Operator	Description	Example
–	Negative of the operand	– B = -20
++	Increases the integer value by one.	y = ++A; y = 11
--	Decreases the integer value by one.	y = --A; y = 9

Note: If Y = A++ ; y =10

Y = A-- ; y =10

2. ARITHMETIC OPERATORS:

The following table shows all the arithmetic operators supported by the C language. Assume variable **A** holds 10 and variable **B** holds 20

Operator	Description	Example
+	Adds two operands.	A + B = 30
–	Subtracts second operand from the first.	A – B = -10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by denominator.	B / A = 2
%	Modulus Operator and remainder of after an integer division.	B % A = 0

Scanned with CamScanner

3. RELATIONAL OPERATORS:

The following table shows all the relational operators supported by C. Assume variable **A** holds 10 and variable **B** holds 20.

Operator	Description	Example
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true.

4. LOGICAL OPERATORS :

This combines two or more relational expressions. Following table shows all the logical operators supported by C language. Assume variable **A** holds 1 and variable **B** holds 0

Operator	Description	Example
&&	Called as Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called as Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A || B) is true.
!	Called as Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

5. EQUALITY OPERATORS:

Assume variable **A** holds 10 and variable **B** holds 20.

Operator	Description	Example
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	(A != B) is true.

6. ASSIGNMENT OPERATORS:

They are used to assign the result of an expression to a variable.

Scanned with CamScanner

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	C = A + B will assign the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assigns the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A

7. BITWISE OPERATORS:

The following table lists the bitwise operators supported by C for manipulation of data at bit level. They are not applied to float or double.

Assume variable 'A' holds 60 and variable 'B' holds 13

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = ~(60), i.e,.11000011
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

Scanned with CamScanner

8. SPECIAL OPERATORS:

These are the operators which do not fit in any of the above classification.

Operator	Description	Example
sizeof()	Returns the size of a variable.	sizeof(a), if a is integer, will return 2.
&	Returns the address of a variable.	&a; returns the actual address of the variable.
*	Pointer to a variable.	*a;
? :	Conditional Operator to construct conditional expressions. Also called as ternary operator. It is an alternative for if else statement. Syntax: expression1 ? expression2 : expression3	Y = (A>B) ? A : B If A>B is true then Y = A otherwise Y=B

,	comma (,) works as a separator and an operator too. Sometimes we assign multiple values to a variable using comma, in that case comma is known as operator.	a = 10,20,30;
		b = (10,20,30);
		In the first statement, value of a will be 10.
		
		In the second statement, value of b will be 30, because when multiple values are given with comma operator within the braces, then right most value is considered as result of the expression.

OPERATOR PRECEDENCE

Operator Precedence determines the order in which different operations are carried out in an expression with more than one operators

For example 10 + 20 * 30 is calculated as 10+ (20 * 30) and not as (10 + 20) * 30.

Associativity specifies the direction of evaluating an expression with more than one operator of same priority. It may be left to right or right to left.

Scanned with CamScanner

Precedence and associativity of C operators

Category	Operator	Associativity
Unary	- ! ~ ++ - -	Right to left
Arithmetic	* / %	Left to right
Arithmetic	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=	Right to left
Comma	,	Left to right

CONDITIONAL BRANCHING STATEMENTS

The order of execution of statements is not sequential in branching statements and it is based on some conditions. The different branching statements are

1. if statement
2. switch ... case statement

1. if Statement

There are different forms of if statements. They are

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

i. Simple if statement

This statement is used to check some given condition and perform some operations depending upon the correctness of that condition.

Syntax

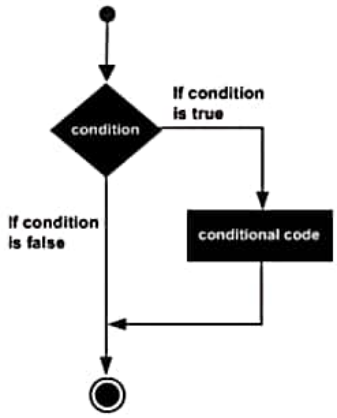
```
if (expression)
{
  //Statement block1 ;
}
  Statement2;
```


Scanned with CamScanner

Description

If the expression returns true, then statement block1 will be executed, otherwise these statements are skipped.

Flow Chart



Example:

```
#include <stdio.h>
void main( )
{
   int x, y;
   x = 15;
   y = 13;
   if (x > y )
   {
      printf("x is greater than y");
   }
}
```

Output

x is greater than y

ii. if...else statement

The if-else statement is used to perform two operations for a single condition. One is for the correctness of that condition, and the other is for the incorrectness of the condition. Here, we must notice that if and else block cannot be executed simultaneously.

Syntax

```
if (expression)
{
   //statement block1;
}
else
{
```

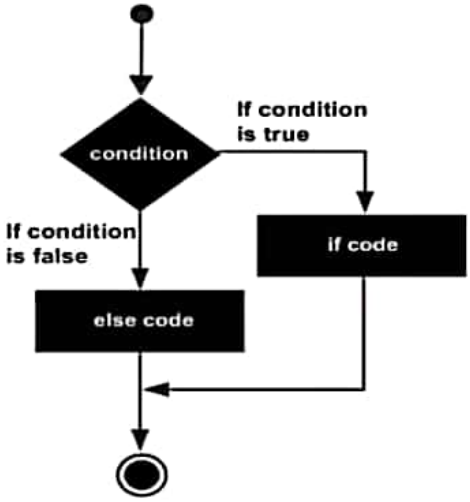

Scanned with CamScanner

```
  // statement block2;
}
```

Description

If the expression is true, the statement-block1 is executed, else statement-block2 is executed.

Flow Chart



Example:

```
#include <stdio.h>
void main( )
{
   int x, y;
   x = 15;
   y = 18;
   if (x > y )
   {
      printf("x is greater than y");
   }
   else
   {
      printf("y is greater than x");
   }
}
```

Output

y is greater than x

Scanned with CamScanner

iii. Nested if....else statement

There are many different forms. The most general form of two layer nesting is

Syntax

```
if ( expression1 )
{
   if( expression2)
   {
     // statement block1;
   }
   else
   {
      //statement block2;
   }
}
else
{
   //statement block3;
}
```

Description

If expression1 is false then statement-block3 will be executed, otherwise the execution continues and enters inside the first if, and evaluates expression2. If true statement-block1 is executed otherwise statement-block2 is executed.

Example:

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


Scanned with CamScanner

```
    else
    {
      if (b > c)
      {
        printf("b is the greatest");
      }
      else
      {
        printf("c is the greatest");
      }
    }
}
```

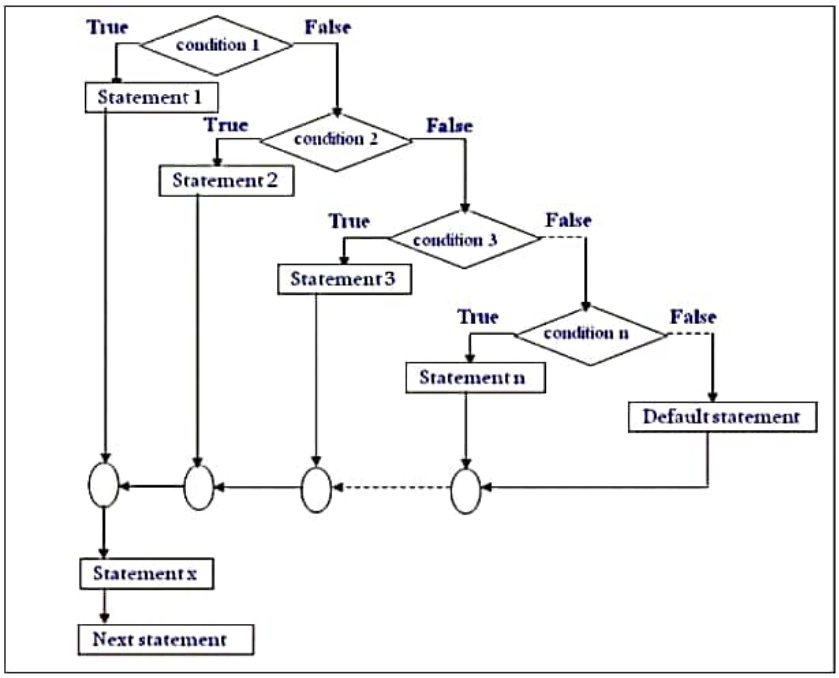
iv. if-else-if ladder statement

The if-else-if ladder statement is an extension to the if-else statement. It is used in the scenario where there are multiple conditions. In if-else-if ladder statement, if a condition is true then the statements defined in the if block will be executed, otherwise if some other condition is true then the statements defined in the corresponding else-if block will be executed. At the last if none of the condition is true, then the statements defined in the else block will be executed. It is similar to the switch case statement where the default is executed instead of else block if none of the cases is matched.

Syntax:

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


Scanned with CamScanner



2. Switch ... Case Statement

The switch statement is an alternate to if-else-if ladder statement which allows us to execute multiple operations for the different possible values of a single variable called switch variable.

Syntax

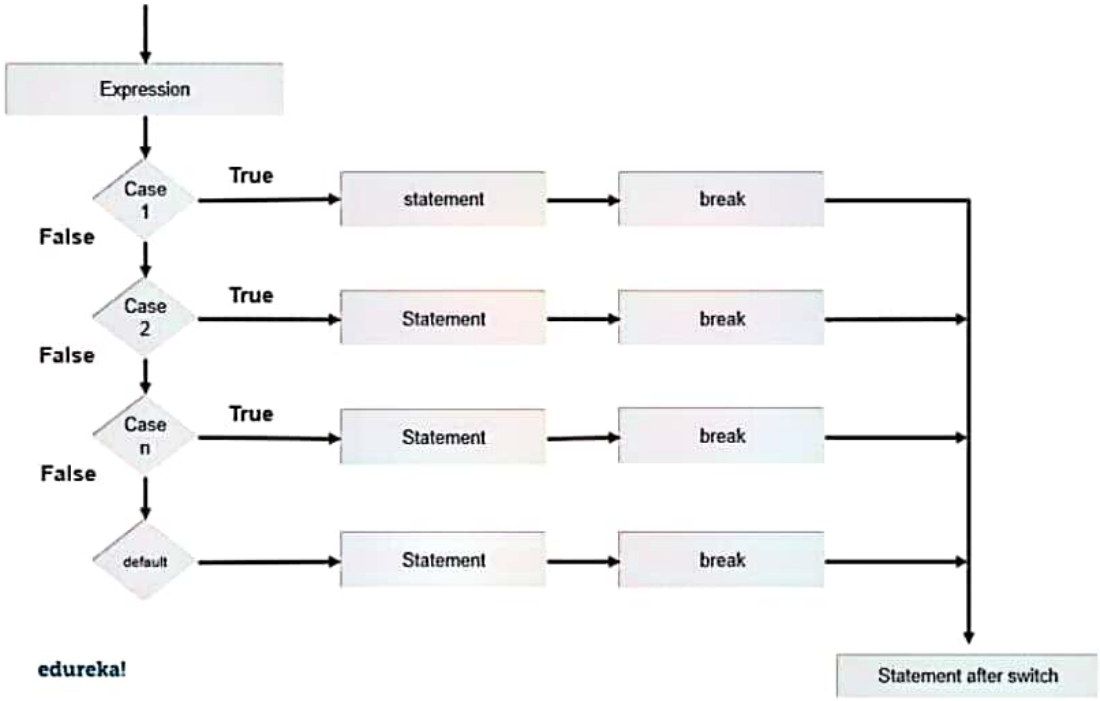
```
switch (expression)
    {
      case value-1:
            statement block-1;
            break;
      case value-2:
            statement block-2;
            break;
      case value-3:
            statement block-3;
            break;
      case value-4:
            statement block-4;
           break;
      default:
             default-block;
            break;
    }
```


Scanned with CamScanner

Description

The expression in switch is evaluated and then compared to the values present in different cases. It executes the block of code which matches the case value. If there is no match, then default block is executed(if present).

Flowchart



Example:

```
#include<stdio.h>

void main()
{
   int num;
   printf("\n\nEnter a number between 0 - 3 ");
   scanf("%d",&num);
   switch(num)
     {
      case 0:
           printf("\nEntered number is Zero \n");
           break;
      case 1:
            printf("\nEntered number is One \n");
            break;
```


Scanned with CamScanner

```
    case 2:
        printf("\nEntered number is Two \n");
        break;
    case 3:
        printf("\nEntered number is three \n");
        break;
    default:
        printf("\n Enter the number between is 0 - 3 \n"); break;
  }
}
```

Output:

Enter a number between 0 - 3: 0

Entered number is Zero

Points to remember

1. The expression (after switch keyword) must yield an integer value not a float value
2. The case values must be unique and must end with a colon (:)
3. break statement is used to exit the switch block. If it is not used, then all the consecutive blocks of code will get executed after the matching block.
4. default case is executed when none of the case values matches the value of switch expression.

Difference between switch and if

- if statements can evaluate float conditions but switch statements cannot evaluate float conditions.
- if statement can evaluate relational operators. switch statement cannot evaluate relational operators.

[Note: No curly braces are required if there is a single statement inside if part and else part]

Iteration and loops

Repeated execution of a single or block of statements for a specified number of times or until the specified condition is satisfied is called as looping or iteration. The different looping statements are:

1. for statement

Scanned with CamScanner

2. while statement
3. do while statement

We have two types of looping structures.

- Condition is checked before entering the statement block called **entry control**.
- Condition is checked after the statement block called **exit control**.

1. for statement

The for statement is the most commonly used looping statement in C.

Syntax

```
for( expression 1; expression 2; expression 3)
{
    statement block;
}
```

Expression 1 is used to initialize a index/ control/counter variable. This is an assignment expression.

Expression 2 represents a condition. It must be true for the loop to continue execution. This is a logical expression

Expression 3 is used to alter the value of the index variable. This is a unary expression or an assignment expression.

Description

The process of execution involves the following steps

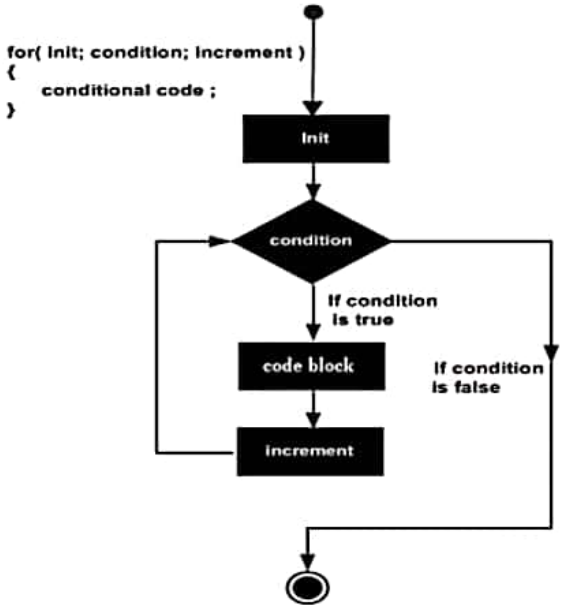
Step1: First the index variable gets initialized.

Step 2: The condition is checked, where the index variable is tested for the given condition. If the condition returns true then the C statements inside the body of **for loop** gets executed. If the condition returns false then the for loop gets terminated and the control comes out of the loop.

Step 3: After successful execution of statements inside the body of loop, the index variable is altered depending on the operation (++ or –).

Scanned with CamScanner

Flow Chart



Example

```
/* printing n numbers */
#include<stdio.h>
void main()
{
int n,i = 1;
for(i=1; i<=5; i++)
printf("%d",i)
}
```

Output

1
2
3
4
5

2. while statement

It is an entry controlled loop. The condition is evaluated and if it is true then body of loop is executed. After execution of body the condition is once again evaluated and if it is true body is executed once again. This goes on until test condition becomes false.

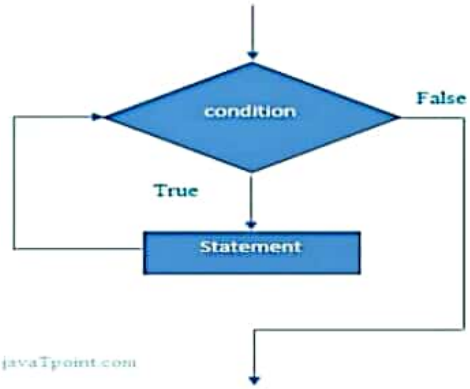
Syntax

```
while (condition)
{
```


Scanned with CamScanner

```
    // body of the loop
}
```

Flow chart



Example

```
/* printing n numbers */
#include<stdio.h>
void main()
{
 int n,i = 1;
 while(count<=5)
 {
  printf("%d",i);
  ++i;
 }
}
```

Output

1
2
3
4
5

3. do while statement

The do while is an exit controlled loop and its body is executed at least once.

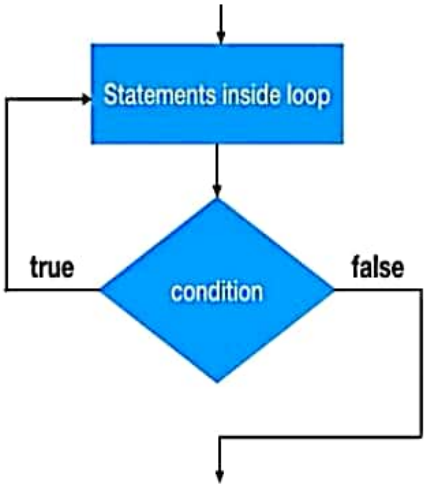
Syntax

```
do
{
 //body of the loop
```


Scanned with CamScanner

```
}
while(condition);
```

Flowchart



Example

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

Output

```
1
2
3
4
5
```

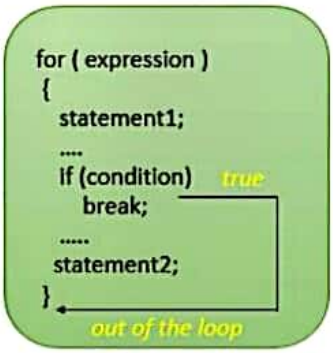
Scanned with CamScanner

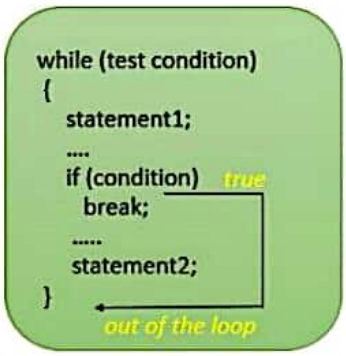
UNCONDITIONAL STATEMENTS

Unconditional statement allows transferring the flow of control to another part of program without evaluating conditions. These are also called as jumping statements. There are four jumping statements in C. They are break, continue, goto and return.

1. Break Statement:

Break statement is used to terminate any type of loop e.g, while loop, do while loop or for loop. The break statement terminates the loop body (jump out of the loop skipping the code below it) immediately and passes control to the next statement after the loop. In case of inner loops, it terminates the control of inner loop only.





Example

```
#include<stdio.h>
void main ()
{
  int i;
  for(i = 1; i<=10; i++)
  {
    printf("%d ",i);
    if(i == 5)
    break;
  }
 printf("/ncame outside of loop i = %d",i);
}
```


Scanned with CamScanner

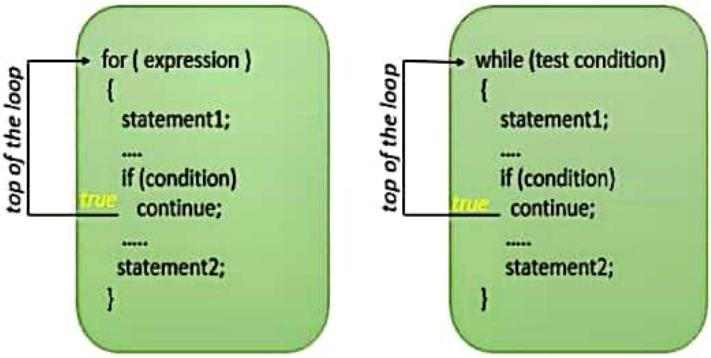
Output

1 2 3 4 5

came outside of loop when i = 5

2. Continue Statement:

Continue statement is used to bring the program control to the beginning of the loop. The continue statement skips some lines of code inside the loop and continues with the next iteration. It is mainly used for a condition so that we can skip some code for a particular condition.



Example

```
#include<stdio.h>
void main ()
{
   int i;
   for(i = 1; i<=10; i++)
  {
       if (i == 5)
           continue;
      printf("%d ",i);
   }
 }
```

Output

1 2 3 4 6 7 8 9 10

Scanned with CamScanner

3. Goto And Labels:

Goto is used to transfer the program control to a predefined label. A label is an identifier followed by a colon. It can be used to break the multiple loops which can't be done by using a single break statement.

Syntax:

```
label :
//some part of the code;
goto label;
```

Example

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

Output

1 2 3 4 5

4. return statement

The return statement terminates the execution of a function and returns control to the calling function. Execution resumes in the calling function at the point immediately following the calling statement.

Type Casting in C

Typecasting allows us to convert one data type into other. In C language, we use cast operator for typecasting which is denoted by (type).

Syntax:

```
(type)value;
```


Scanned with CamScanner

Without Type Casting:

```
int f= 9/4;
printf("f : %d\n", f );
```

Output: 2

With Type Casting:

```
float f=(float) 9/4;
printf("f : %f\n", f );
```

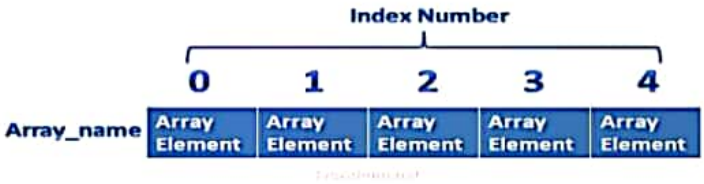
Output: 2.250000

Scanned with CamScanner

UNIT III

What is an Array?

- An array is a collection of similar data-type elements stored sequentially in memory.
- Array size is defined at the time of declaration and can't be altered that after.
- Elements of array can be referenced with an index number.
- This index number starts with 0.



A simple array takes continuous locations in memory as shown below.

int Arr[5] {Array of 5 Elements }

	Arr[0]	Arr[1]	Arr[2]	Arr[3]	Arr[4]			

Note:

If there is no free contiguous memory locations as size of array, the declaration of array will be failed.

Why we need Array:

Let us consider to find out the average of 100 integer numbers entered by user. In C, you have two ways to do this:

1) Define 100 variables with int data type and then perform 100 scanf() operations to store the entered values in the variables and then at last calculate the average of them.

2) **Have a single integer array to store all the values, loop the array to store all the entered values in array and later calculate the average.**

From the second solution, it is convenient to store same data types in one single variable and later access them using array index

Array Declaration:

Scanned with CamScanner

While declaring an array, we must have 3 things

a. **Array datatype**,

b. **Array name** and

c. **Array size**.

Types of C Arrays:

There are 2 types of C arrays. They are,

1. One dimensional array
   Only one index should be used to access the elements of the array.
2. Multi dimensional array
   More than one index should be used to access the elements of the array. This includes
   i. Two dimensional array
   ii. Three dimensional array
   iii. Four dimensional array etc.

ONE DIMENSIONAL ARRAY(1-D Array)

Declaration of one dimensional array.

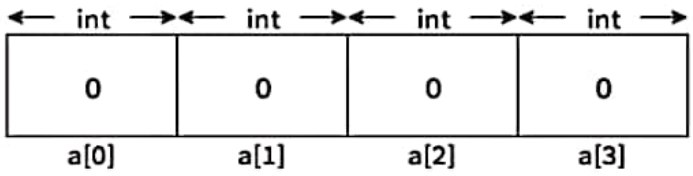
```
data_type  array_name[size of array]
```

Examples:

- if you want to declare an integer array with four elements,

```
int a[4];
```

This statement allocates a contiguous block of memory for four integers and initializes all the values to 0. This is how it is laid out in memory:



Note:

Array indexes start from zero and end with (array size – 1). So for the above array, you can use the first element with a[0], second element with a[1], third element with a[2] and fourth (last) element with a[3].

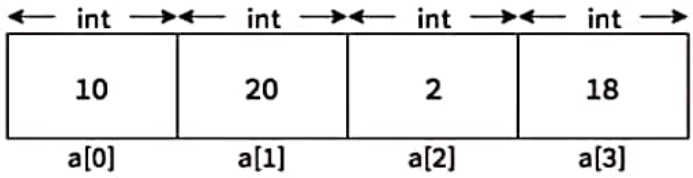
- You can use the indexes to set or get specific values from the array.

```
a[0] = 10;
a[1] = 20;
```


Scanned with CamScanner

```
a[2] = a[1] / a[0]; // a[2] will be set to 20/10 = 2
a[3] = a[1] - 2;   // a[3] will be set to 20-2 = 18
```

After these changes, here is how the array will look like in memory:



You can print them by using:

```
printf("%d %d %d %d\n", a[0], a[1], a[2], a[3]);
```

Note:

C does not enforce any array bounds checks, and accessing elements outside the maximum index will lead to "undefined behaviour". **if you try to access a[5], the element is not available. This may cause unexpected output (your program to crash or behave abnormally).**

- If you want to save a-z characters in an array, define it as following

```
char arr[26];
```

- Similarly an array can be of any data type such as double, float, short etc.

Array Initialization:

There are two ways of array initialization:

1. Initialize array at time of declaration – means save all the values in array columns during declaration like below.

```
Eg1.: int num[6] = {1, 3, 5, 7, 9, 11};

Eg2.: char letters[5] = {'a', 'b', 'c', 'd', 'e'};

Eg3.: float numbers[3] = {13.25, 12.09, 8.1};
```

You can also initialize an array without arraysize.

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

Here, we haven't specified the size. Compiler knows its size is 5 as we are initializing it with 5 elements. **However, you cannot skip both the size and the initializer list, and write as int mark[]. If you skip both of them, C cannot create the array, and this will lead to a compile-time error**

Scanned with CamScanner

mark[0]	mark[1]	mark[2]	mark[3]	mark[4]
19	10	8	17	9

2. Initialize array during program execution – means all array elements will be fill at time of execution programs. It has a benefit that we can save elements from user input.

```
int arr[5];
int i;
for(i=0;i<5;i++)
{
 printf("Enter a number: ");
 scanf("%d", &num);
 arr[i] = num;
}
```

Note:

1. You can also make an array that is bigger than the initializer list, like

```
int a[6] = {10, 20, 30, 40};
printf("%d %d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4], a[5]);
```

In this case, the rest of the elements are initialized with zero. In our above example, elements from a[0] to a[3] will be initialized, whereas a[4] and a[5] will be set to zero.

(ie) a[0]=10, a[1]= 20,a[2]= 30, a[3]= 40, a[4]= 0 and a[5]= 0.

Accessing Array Elements:

In array we can access any element by specifying their index number. For example if we want to access the element stored on index 2 in array named arr. Use following

```
int value;
value = arr[2];
```

Or we can fetch and print entire array elements using for or while loop

```
int i;
for(i=0;i<5; i++)
{
 printf("%dn", arr[i] );
}
```

Example:

1. **To find out the average of 4 integers**

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


Scanned with CamScanner

```
  int avg = 0;
  int sum =0;
  int x=0;

  /* Array- declaration – length 4*/
  int num[4];

  /* We are using a for loop to traverse through the array
     while storing the entered values in the array */

 for (x=0; x<4;x++)
  {
     printf("Enter number %d \n", (x+1));
     scanf("%d", &num[x]);
  }

 for (x=0; x<4;x++)
  {
     sum = sum+num[x];
  }

  avg = sum/4;
  printf("Average of entered number is: %d", avg);
  return 0;
}
```

Output:

```
Enter number 1
10
Enter number 2
10
Enter number 3
20
Enter number 4
40
Average of entered number is: 20
```

TWO DIMENSIONAL ARRAY (2 -D Array)

Two Dimensional Array in C is the simplest form of Multi-Dimensional Array. It is defined as an array of arrays. 2D array can be seen as a table or matrix which can have any number of rows and columns. We can access the elements using 2 indexes, row index and column index. It is represented as below.

Scanned with CamScanner

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

x[1][2] refers to the element stored in the 2nd row and 3rd column, because index value always ranges from 0 to maximum size-1.

The elements of the 2D array are stored in contiguous memory locations in a row-wise manner, starting from first row and ending with last row.

Declaration:

data_type array_name[row size][column size];

For example,

float x[3][4];

Here, x is a two-dimensional (2D) array with 3 rows and each row has 4 columns. This array can hold 12 elements (3 * 4).

Initialization of a 2D array

There are Different ways to initialize two-dimensional array

Eg1.: int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};
Eg2.: int c[][3] = {{1, 3, 0}, {-1, 5, 9}};
Eg3.: int c[2][3] = {1, 3, 0, -1, 5, 9};

Eg4.:// Declare a two-dimensional array with 3 rows and 2 columns

```
int table[3][2]:
// create and initialize  an array
int table[3][2] = { {10, 22}, {33, 44}, {45, 78} };
                    or
int table[3][2] = {10, 22, 33, 44, 45, 78 };
                    or
int table[3][2] = {
                  {10, 22},  /*  initializers for row indexed by 0 */
                  {33, 44 },/*  initializers for row indexed by 1 */
                  {45, 78 } /*  initializers for row indexed by 2 */
                   };
```


Scanned with CamScanner

Printing a Two-dimensional array

To print all the elements of a two dimensional array we use a doubly-nested for-loop. In the following example, there are 3 rows and 2 columns.

```
#include <stdio.h>
main()
{
 int row,col;
 int table[3][2] = { {10, 22}, {33, 44}, {45, 78} };
 for (row = 0; row < 3; row++)
 {
 for (col = 0; col < 2; col++)
 {
 printf("%d\t",table[row][col]);
 }
 printf("\n");
 }
}
```

CHARACTER ARRAYS AND STRINGS

Strings are one-dimensional array of characters terminated by a **null** character '\0'. character arrays are used to manipulate text such as word or sentences. Each character in the array occupies one byte of memory, and the last character must always be 0.

Declaring a string

Syntax : char string_name [size];

Example : char name[10];

Initializing a string

There are two ways to initialize a string.

1. By char array

   To hold the null character at the end of the array, the size of the character array must be one more than the number of characters.

   ```
   char ch[6]={'H', 'E', 'L', 'L', 'O', '\0'};
   ```

2. By string literal

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

   You do not place the *null* character at the end of a string constant. The C compiler will automatically place '\0' at the end of the string when it initializes the array.

Scanned with CamScanner

Reading Strings

Strings can be read from the terminal using scanf() or gets().

1. scanf("%s", string1);
   scanf() reads a sequence of characters and terminates when the first white space is encountered or a new line character ('\n') is encountered.
2. gets(string1);
   gets() terminates only when new line character ('\n') is encountered.

Displaying Strings

Strings can be displayed using printf() or puts().

printf("%s, string1);

puts(string2);

Example:

```
printf(%s", name);

puts(address);
```

String Operations

To perform string operations many important library functions are defined in "string.h" header file.

No.	Function	Description
1)	strlen(string)	Returns the total numbers of characters in string.
2)	strcpy(destination, source)	copies the contents of source string to destination string.
3)	strcat(first_string, second_string)	Joins first string with second string. The result of the string is stored in first string.
4)	strcmp(string1, string2)	compares string1 with string2. If both strings are same, it returns 0. if String1 is Greater than String2, it returns positive integer if String1 is lesser than String2, it returns negative integer
5)	strrev(string)	returns reverse of string.

Scanned with CamScanner

6)	strlwr(string)	returns string characters in lowercase.
7)	strupr(string)	returns string characters in uppercase.

Two Dimensional Character Array.

The first index of the array is used for defining total numbers of strings and the second index is used for defining length of the string.

Example:

```
char name[5][10]
```

It declares 5 names and length of each name is up to 10.

SEARCHING ALGORITHMS

Searching is the process of finding the position of given value in a list or an array. To search an element in a given array, there are two popular algorithms available:

1. Linear
2. Binary

1. **Linear Search**

Linear search is a very basic and simple search algorithm. It is used with unsorted or unordered lists.

Algorithm

Step 1: Iterate over every element of the array to check if it matches with the number we're looking for.
Step 2: when the element is matched , return the index of the element in the array.
Step 3: else return -1.

Program:

```
#include <stdio.h>

int main() {
  // declare an array, a loop variable, and the number to search
  int a[5], i, search;

  // declare another variable to keep track of the index where
  // the number was found.
  int pos = -1;

  printf("Enter five numbers:\n");
```


Scanned with CamScanner

```
    // read each number from the user
    for (i = 0; i < 5; i++) {
       scanf("%d", &a[i]);
    }

    printf("Enter the number to search for:\n");
    scanf("%d", &search);

    // iterate over the array to find the element
    for (i = 0; i < 5; i++) {
       // is the current element equal to the number?
       if (a[i] == search) {
          // note down the new position
          pos = i;
          // break out of the loop.
          break;
       }
    }

    if (pos == -1) {
       printf("%d was not found\n", search);
    } else {
       printf("%d was found at position %d\n", search, pos);
    }

    return 0;
}
```

Output:

```
Enter five numbers:
10
25
30
26
40
30
Enter the number to search for:
30 was found at position 2
```

2. Binary Search

Binary Search is used with sorted array or list. Binary search follows divide and conquer approach in which, the list is divided into two halves and the item is compared with the middle element of the list. If the match is found then, the location of middle element is returned otherwise, we search into either of the halves depending upon the result produced through the match.

Algorithm:

Step 1: Compare the element to be searched with the element in the middle of the sorted

Scanned with CamScanner

list.

Step 2: If matched, return the index of the middle element

Step 3: If not matched, check whether the element to be searched is less or greater than the middle element.

Step 4 : If the element to be searched is lesser than the middle number, then do binary search in left half of the array.

Step 5: Else do binary search in right half of the array.

Step 6: If not matched with any elements, return -1.

SORTING ALGORITHMS

Sorting is a process of arranging elements of an array in ascending or descending order.

Consider an array

int A[10] = { 5, 4, 10, 2, 30, 45, 34, 14, 18, 9);

The Array sorted in ascending order will be given as

A[] = { 2, 4, 5, 9, 10, 14, 18, 30, 34, 45 }

There are many techniques by using which, sorting can be performed.

SN	Sorting Algorithms	Description
1	Bubble Sort	It is the simplest sort method which performs sorting by repeatedly moving the largest element to the highest index of the array. It compares each element with its adjacent element and swap them if it is not in correct order.
2	Insertion Sort	As the name suggests, insertion sort inserts each element of the array to its proper place. It is a very simple sort method .
3	Selection Sort	It finds the smallest element in the array and place it on the first place on the list, then it finds the second smallest element in the array and place it on the second place. This process continues until all the elements are placed in their correct position.

1. Bubble Sort

Scanned with CamScanner

In Bubble sort, Each element of the array is compared with its adjacent element. The algorithm processes the list in passes. A list with n elements requires n-1 passes for sorting.

Algorithm

1. Compare first element with the second element. If the first element is greater than the second element, swap them.
2. Repeat the above process with the next two elements until the last element. Now the largest element is placed in the highest index of the array.
3. Do step1 and step 2 to place the next largest element in the next highest index. Repeat the process until the list is sorted

Example

Take an array of numbers " 5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in **bold** are being compared. Three passes will be required;

First Pass

(**5 1** 4 2 8) → (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.

(1 **5 4** 2 8) → (1 **4 5** 2 8), Swap since 5 > 4

(1 4 **5 2** 8) → (1 4 **2 5** 8), Swap since 5 > 2

(1 4 2 **5 8**) → (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass

(**1 4** 2 5 8) → (**1 4** 2 5 8)

(1 **4 2** 5 8) → (1 **2 4** 5 8), Swap since 4 > 2

(1 2 **4 5** 8) → (1 2 **4 5** 8)

(1 2 4 **5 8**) → (1 2 4 **5 8**)

Now, the array is already sorted, but the algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass

(**1 2** 4 5 8) → (**1 2** 4 5 8)

(1 **2 4** 5 8) → (1 **2 4** 5 8)

(1 2 **4 5** 8) → (1 2 **4 5** 8)

(1 2 4 **5 8**) → (1 2 4 **5 8**)

2. Insertion Sort

Scanned with CamScanner

Insertion sort works similarly as we sort cards in our hand in a card game. This sort inserts each element of the array to its proper place. It is a very simple sort method .

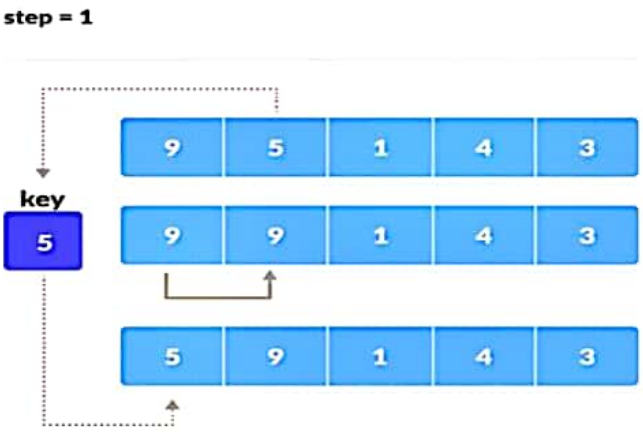
Algorithm:

1. Assume the first element in the array is sorted. Take the second element and store it as key
2. Compare key with the first element. If the first element is greater than key, then key is placed in front of the first element.
3. Take the next element as key and compare it with the elements on the left of it. Place it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.
4. Repeat step 3 for every unsorted element.

Example1:

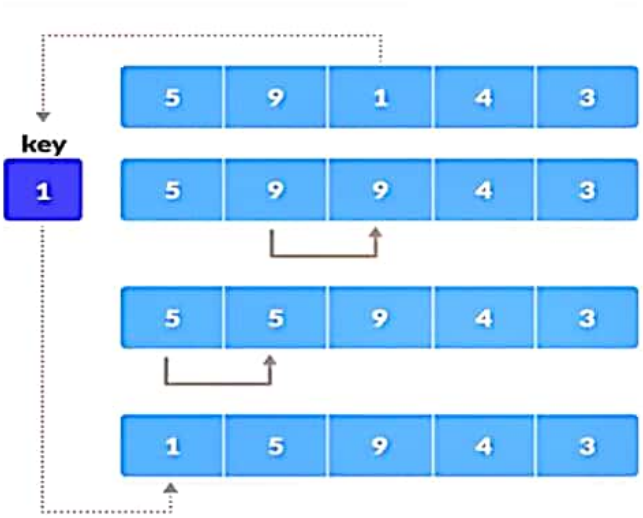
Sort the following array

9,5,1,4,3

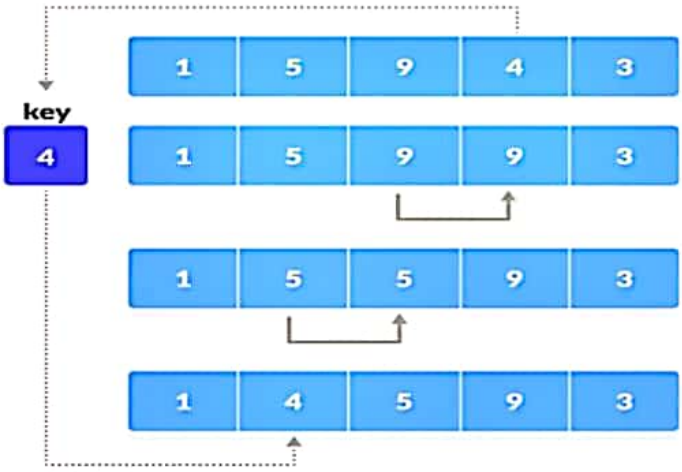


Scanned with CamScanner

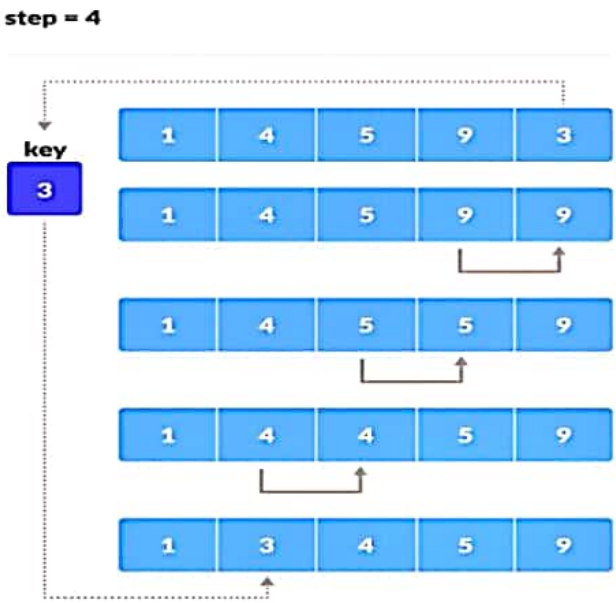
step = 2



step = 3



Scanned with CamScanner



3. Selection Sort

It finds the smallest element in the array and places it on the first place on the list. Then it finds the second smallest element in the array and places it on the second place. This process continues until all the elements are placed in their correct position.

Algorithm

1. Set the first element as minimum
2. Compare minimum with the next element. If that element is smaller, then assign it (ie next)as minimum.
3. Repeat this until the last element.
4. Swap the first element with minimum.
5. Repeat steps 1 to 4 from the first unsorted element until all the elements are placed at their correct positions

Example1:

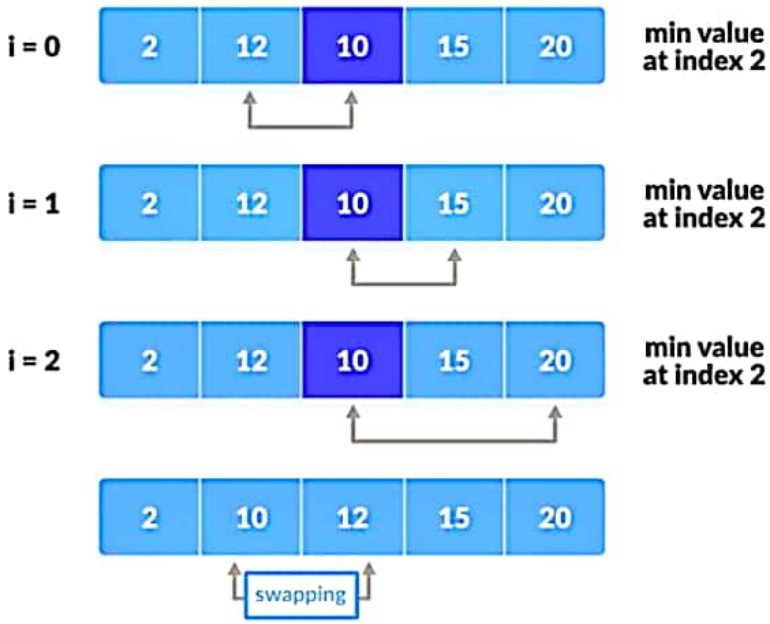
Sort the following array

2, 12, 10, 15, 20.

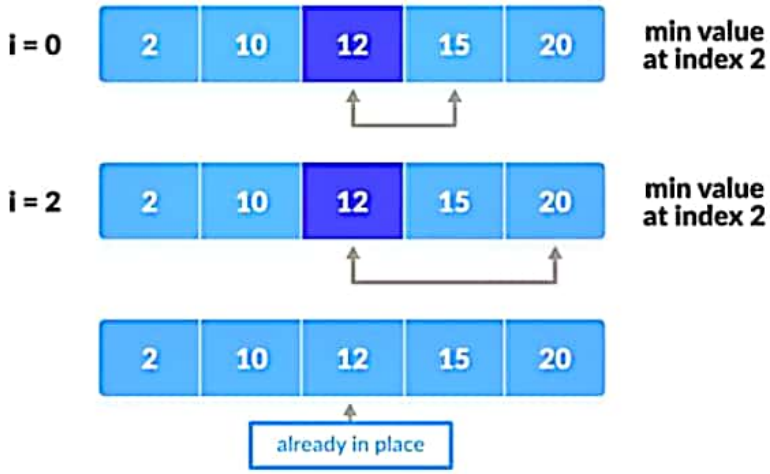
Scanned with CamScanner

1.

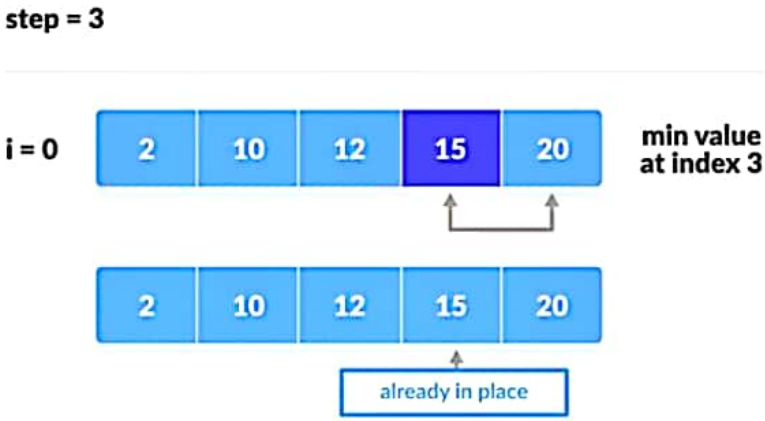
step = 1



step = 2



Scanned with CamScanner



COMPLEXITY ANALYSIS

Analysis of algorithms focuses on the computation of space and time complexity.

Space Complexity

Space Complexity of an algorithm denotes the total space used or needed by the algorithm for its working, for various input sizes.

```
for( i = 0; i < n-1; i++)
  scanf("%d",&a[i]);
```

In the above example, we are creating a array of size n. So the space complexity of the above code is in the order of "n" i.e. if n will increase, the space requirement will also increase accordingly.

Time Complexity

Time Complexity of algorithm is not equal to the actual time required to execute a particular code. It is the number of operations an algorithm performs to complete its task with respect to input size (considering that each operation takes the same amount of time). The algorithm that performs the task in the smallest number of operations is considered the most efficient one.

Time Complexity of algorithm is not equal to the actual time required to execute a particular code but the number of times a statement executes. There are three types of time complexities which can be analyzed for the algorithm:

- Best case time complexity [**Ω Notation**] : It is defined as the minimum number of steps required for an input of size **n**.
- Worst case time Complexity [Big-O Notation]: : It is defined as the maximum number of steps required for an input of size **n**.
- Average Time complexity Algorithm[**Θ Notation**]: : It is defined as the average number of steps required for an input of size **n**.

Scanned with CamScanner

Example: Consider linear search algorithm.

We have one array named "arr" and an integer "k". We need to find if that integer "k" is present in the array "arr" or not? If the integer is there, then return 1 or return 0.

Now, one possible solution for the above problem is traverse each and every element of the array and compare that element with "k". If it is equal to "k" then return 1, otherwise, keep on comparing for more elements in the array and if you reach at the end of the array and you did not find any element, then return 0.
The section of code for the above task is

```
for (int i = 0; i < n; ++i)
{
    if (arr[i] == k)
     return 1;
}
return 0;
```

Time Complexity Analysis

In the above code

* i = 0 ----------> will be executed once
* i < n ----------> will be executed n+1 times
* i++ ----------> will be executed n times
* if(arr[i] == k) --> will be executed n times
* return 1 ---------> will be executed once(if "k" is there in the array)
* return 0 ---------> will be executed once (if "k" is not there in the array)

Each statement in code takes constant time, "C". So, if a statement is executed "N" times, then it will take C*N amount of time. Here we assume that each statement is taking 1sec of time to execute.

Now, consider the following inputs to the above algorithm

- If the input array is [1, 2, 3, 4, 5] and you want to find if "1" is present in the array or not, then the if-condition of the code will be executed 1 time and it will find that the element 1 is there in the array. So, the if-condition will take 1 second here.
- If the input array is [1, 2, 3, 4, 5] and you want to find if "3" is present in the array or not, then the if-condition of the code will be executed 3 times and it will find that the element 3 is there in the array. So, the if-condition will take 3 seconds here.
- If the input array is [1, 2, 3, 4, 5] and you want to find if "6" is present in the array or not, then the if-condition of the code will be executed 5 times and it will find that the element 6 is not there in the array and the algorithm will return 0 in this case. So, the if-condition will take 5 seconds here.

As you can see that for the same input array, we have different time for different values of "k". So, this can be divided into three cases:

Scanned with CamScanner

- **Best case:** This is the lower bound on running time of an algorithm. We must know the case that causes the minimum number of operations to be executed. In the above example, our array was [1, 2, 3, 4, 5] and we are finding if "1" is present in the array or not. So here, after only one comparison, you will get that your element is present in the array. So, this is the best case of your algorithm.
- **Average case:** We calculate the running time for all possible inputs, sum all the calculated values and divide the sum by the total number of inputs.
- **Worst case:** This is the upper bound on running time of an algorithm. We must know the case that causes the maximum number of operations to be executed. In our example, the worst case can be if the given array is [1, 2, 3, 4, 5] and we try to find if element "6" is present in the array or not. Here, the if-condition of our loop will be executed 5 times and then the algorithm will give "0" as output.

Scanned with CamScanner

UNIT IV: FUNCTIONS

Introduction to Functions

A number of statements grouped into a single logical unit are called a function. The use of function makes programming easier since repeated statements can be grouped into functions. Splitting the program into separate function make the program more readable and maintainable. A function definition has two principal components:

(i) the function header

(ii) body of the function.

The function header is the data type of return value followed by function name and a set of arguments. Associated type to which function accepts precedes each argument. The function header statement can be written as

return_type function_name (type1 arg1,type2 arg2,..,typen argn)

where return_type represents the data type of the item that is returned by the function, function_name represents the name of the function, and type1,type2,...,typen represents the data type of the arguments arg1,arg2,..,argn.

Example: function returns the sum of two integers.

```
int add(int p, int q)
{
    return p+q;//Body of the function
}
```

Here p and q are arguments. The arguments are called formal arguments or formal parameters, because they represent the name of the data item that is transferred into the function from the calling program. The corresponding arguments in the function call are called actual arguments or actual parameters, since they define the data items that are actually transferred. A function can be invoked whenever it is needed. It can be accessed by specifying its name followed by a list of arguments enclosed in parenthesis and separated by commas.

The following condition must be satisfied for function call.

•The number of arguments in the function calls and function declaration must be same.

Scanned with CamScanner

•The prototype of each of the argument in the function call should be same as the corresponding parameter in the function declaration statement.

For example the code shown below illustrate how function can be used in programming

```
//C Program for Addition of Two Number's using User Define Function
#include<stdio.h>
#include<conio.h>
float add(float,float); // function declaration
void main()
{       float a,b,c;
        clrscr();
        printf("Enter the value for a & b\n\n");
        scanf("%f%f",&a,&b);
        c=add(a,b);
        printf("\nc=%f",c);
        getch();
}

Here is the function definitation
float add(float x,float y)
{
        float z;
        z=x+y;
        return(z);
}
```

There are two types of functions in C on basis of whether it is defined by user or not.

- Library function
- User defined function

Library functions are the in-built function in C programming system.

For example:

main(): The execution of every C program starts from this main.

printf(): It is used for displaying output in C.

Scanned with CamScanner

scanf(): It is used for taking input in C.

sqrt(): to find the square root of a number

User defined function

C provides programmer to define their own function according to their requirement known as user defined functions.

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

Advantages of user defined functions

•User defined functions helps to decompose the large program into small segments.

•If repeated code occurs in a program, function can be used to include those codes and execute when needed by calling that function.

Example of user defined function

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


Scanned with CamScanner

```
int add(int a,int b) //function declaratory
{
        int add;
        add=a+b;
        return add; //return statement of function
}
```

Every function in C programming should be declared before they are used. This type of declaration are also called function prototype. Function prototype gives compiler information about function name, type of arguments to be passed and return type. Syntax of function prototype

return_type function_name(type(1) argument(1),.....,type(n) argument(n));

Control of the program cannot be transferred to user-defined function unless it is called.
Syntax of function call

function_name(argument(1),....argument(n));

Passing Argument to a Function

Arguments can be passed to a function by two methods,

- pass by value
- pass by reference.

Pass by value

When a single value is passed to a function via an actual argument, the value of the actual argument is copied into the function. Therefore, the value of the corresponding formal argument can be altered within the function, but the value of the actual argument within the calling routine will not change. This procedure for passing the value of an argument to a function is known as passing by value.

Scanned with CamScanner

```
#include <stdio.h>
void main()
{
        int x=3;
        printf("\n x=%d(from main, before calling the function"),x);
        change(x);
        printf("\n\nx=%d(from main, after calling the function)",x);
}
void change(x)
{
        int x;
        x=x+3;
        printf("\nx=%d(from the function, after being modified)",x);
        return;
}
```

The original value of x (i.e. x=3) is displayed when main begins execution. This value is then passed to the function change, where it is sum up by 3 and the new value displayed. This new value is the altered value of the formal argument that is displayed within the function. Finally, the value of x within main is again displayed, after control is transferred back to main from change.

x=3 (from main, before calling the function)

x=6 (from the function, after being modified)

x=3 (from main, after calling the function)

Passing an argument by value allows a single-valued actual argument to be written as an expression rather than being restricted to a single variable. But it prevents information from being transferred back tothe calling portion of the program via arguments. Thus, passing by value is restricted to a one-way transfer of information.

Pass by reference

Instead of passing the value of variable, address or reference is passed and the function operate on address of the variable rather than value. Here formal argument is altered to the actual argument, it means formal arguments calls the actual arguments.

Example:-

Scanned with CamScanner

```
void main()
{
        int a,b;
        change(int *,int*);
        printf("enter two values:\n");
        scanf("%d%d",&a,&b);
        change(&a,&b);  /*address of a and b are passed to the function*/
        printf("after  changing two value of a=%d and b=%d\n:"a,b);
}
change(int *a, int *b)
{
        int  k;
        k=*a;
        *a=*b;
        *b= k;
        printf("value in this function  a=%d and b=%d\n",*a,*b);
}
```

Arrays are passed differently than single-valued entities. If an array name is specified as an actual argument, the individual array elements are not copied to the function. Instead the location of the array is passed to the function. If an element of the array is accessed within the function, the access will refer to the location of that array element relative to the location of the first element. Thus, any alteration to an array element within the function will carry over to the calling routine.

```
#include <stdio.h>
#define SIZE 5
void showarray(int array[]);
int main()
{
          int n[] = { 1, 2, 3, 5, 7 };
           puts("Here's your array:");
          showarray(n);
          return(0);
}
```


Scanned with CamScanner

```
void showarray(int array[])
{
        int x;
        for(x=0;x<SIZE;x++)
        printf("%dt",array[x]);
        putchar('n');
}
```

Recursion

When function calls itself (inside function body) again and again then it is called as recursive function. In recursion calling function and called function are same. According to recursion problem is defined in term of itself. Here statement with in body of the function calls the same function and same times it is called as iterative definition. Recursion is the process of defining something in form of itself.

Recursion always consists of two main parts.

- a terminating case that indicates when the recursion will finish
- a call to itself that must make progress towards the terminating case.

Eg:

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

void rec()
{
   if(base_condition)
   {
      // terminating condition
   }
   statement 1;
   ...
   rec();
}
```

Example:

/*calculate factorial of a no.using recursion*/

Scanned with CamScanner

```
int fact(int);
void main()
{
        int num;
        printf("enter a number");
        scanf("%d",&num);
        f=fact(num);
        printf("factorial is =%d\n",f);
}
fact (int num)
{
if (num==0||num==1)
        return 1;
else
        return(num*fact(num-1));
}
```

Display Fibonacci series using Recursion

```
#include<stdio.h>
int Fibonacci(int);
int main()
{
  int n, i = 0, c;
  scanf("%d",&n);
  printf("Fibonacci series\n");
  for ( c = 1 ; c <= n ; c++ )
  {
    printf("%d\n", Fibonacci(i));
    i++;
  }
  return 0;
}
int Fibonacci(int n)
{
  if ( n == 0 )
```


Scanned with CamScanner

```
    return 0;
  else if ( n == 1 )
    return 1;
  else
    return ( Fibonacci(n-1) + Fibonacci(n-2) );
}
```

Ackerman Function implementation using Recursion

All primitive recursive functions are total and computable, but the Ackermann function illustrates that not all total computable functions are primitive recursive.It's a function with two arguments each of which can be assigned any non-negative integer. It is computed as,

$$\begin{aligned} \mathrm{A}(0,n) &= n+1 \\ \mathrm{A}(m+1,0) &= \mathrm{A}(m,1) \\ \mathrm{A}(m+1,n+1) &= \mathrm{A}(m,A(m+1,n)) \end{aligned}$$

where m and n are non negative numbers.

```
#include<stdio.h>
int ack(int m, int n);
main()
{
        int m,n;
        printf("Enter two numbers :: \n");
        scanf("%d%d",&m,&n);
        printf("\nOUTPUT :: %d\n",ack(m,n));
}

int ack(int m, int n)
{
        if(m==0)
                return n+1;
        else if(n==0)
                return ack(m-1,1);
            else
                return ack(m-1,ack(m,n-1));
}
```

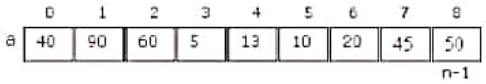

Scanned with CamScanner

Quick sort –Recursion

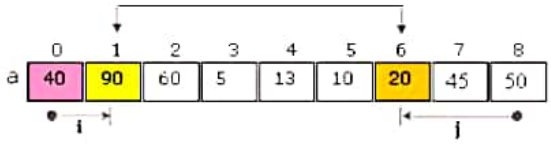
Quick sort algorithm first selects a value that is to be used as split-point (pivot element) from the list of given numbers. Elements are arranged so that, all the numbers smaller than the split-point are brought to one side of the list and the rest on the other. This operation is called splitting.

After this, the list is divided into sub lists, one sub list containing the elements less than the pivot element and the other containing the elements more than the pivot element. These two lists are again individually sorted using Quick sort algorithm that is by again finding a split-point and dividing into two parts. The process is recursively done until all the elements are arranged in order. So it is called divide and conquer algorithm.

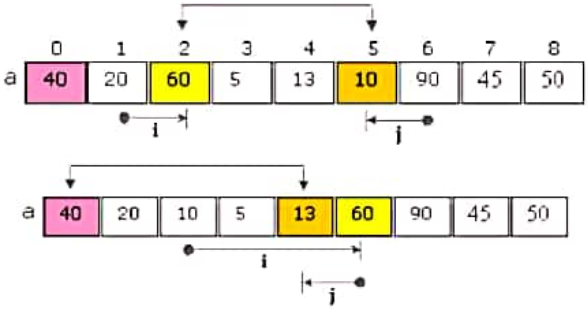
Consider the given list,



Select the first element as pivot element and place it, at its position using an algorithm.
Find the biggest element than the pivot element from first using "i" and smallest element from the last using "j" and interchange them.



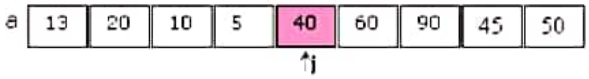
follow the same procedure as long as i<j



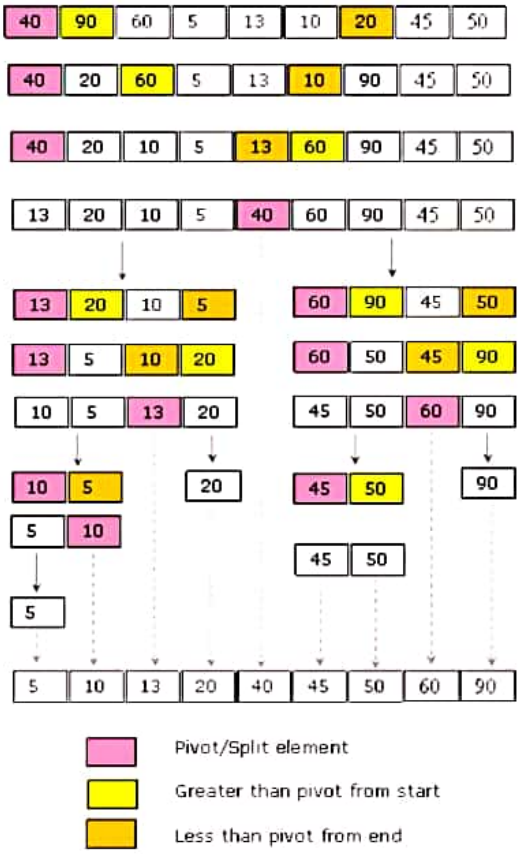
As i<j is false, the current process is stopped and a[start] and a[j] are interchanged.

Now the pivot element is at its position "j", which is the split position for next sub arrays because all elements to its left are smaller and to its right are greater.

Scanned with CamScanner



continue the same process recursively with other sub arrays. It can be done using a recursive procedure




```
#include<stdio.h>
int a[50];
void qsort(int,int);
int split(int,int);
int main()
{
        int n,i;
        printf("How many elements?");
        scanf("%d",&n);
        printf("Enter %d elements:\n",n);
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        qsort(0,n-1);
```


Scanned with CamScanner

```
        printf("The resultant array:\n");
        for(i=0;i<n;i++)
                printf("%5d",a[i]);
        return 0;
}
void qsort(int start,int end)
{
        int s;
        if(start>=end)
                return;
        s=split(start,end);
        qsort(start,s-1);
        qsort(s+1,end);
}
int split(int start,int end)
{
        int p=a[start];
        int i=start,j=end,temp;
        while(i<j)
        {
                while(a[i]<=p)
                i++;
                while(a[j]>p)
                j--;
                if(i<j)
                        temp=a[i],a[i]=a[j],a[j]=temp;
        }
        a[start]=a[j];
        a[j]=p;
        return j;
}
```

Merge Sort is based on divide and conquer algorithm.

Scanned with CamScanner

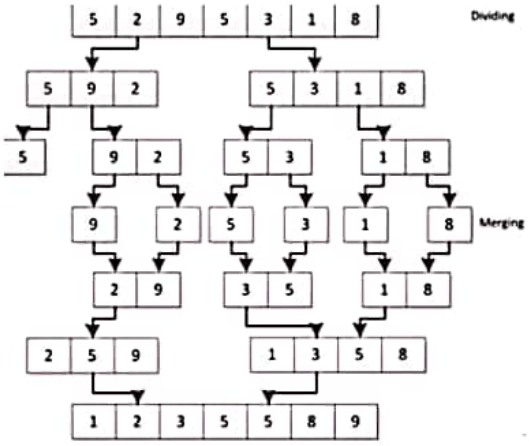
1) DIVIDING

In Merge Sort, take a middle index and break the array into two sub-arrays. These sub-array will go on breaking till the array have only one element.

2) MERGING

With the single elements left, start merging the elements in the same order in which divided them. During Merging, sort the sub-arrays, because sorting 10 arrays of 2 elements is cheaper than sorting an array of 20 elements.

In the end, an array of elements is resulted, which is sorted.




```
#include<stdio.h>
void merge(arr, low, mid, high )
{
        int temp[MAX];
        int i = low;
        int j = mid +1 ;
        int k = low ;
        while( (i <= mid) && (j <=high) )
        {
                if(arr[i] <= arr[j])
                        temp[k++] = arr[i++] ;
                else
                        temp[k++] = arr[j++] ;
```


Scanned with CamScanner

```
        }/*End of while*/
        while( i <= mid )
                temp[k++]=arr[i++];

        while( j <= high )
                temp[k++]=arr[j++];

        for(i= low; i <= high ; i++)
                arr[i]=temp[i];

}
void merge_sort( int low, int high )
{
        int mid;
        if( low != high )
        {
                mid = (low+high)/2;
                merge_sort( low , mid );
                merge_sort( mid+1, high );
                merge( low, mid, high );
        }
}
int main()
{
        int i,n;
        printf("Enter the number of elements : ");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("Enter element %d : ",i+1);
                scanf("%d",&arr[i]);
        }
        printf("Unsorted list is :\n");
        for( i = 0 ; i<n ; i++)
```


Scanned with CamScanner

```
            printf("%d ", arr[i]);
        merge_sort( 0, n-1);
        printf("\nSorted list is :\n");
        for( i = 0 ; i<n ; i++)
            printf("%d ", arr[i]);
        printf("\n");
        return 0;
}
```


Scanned with CamScanner

Unit V

Structures, pointers and files

Structures

Structure is a user-defined datatype in C language which allows us to combine data of different types together. Structure helps to construct a complex data type which is more meaningful. It is similar to an Array, but an array holds data of similar type only.

Syntax of a structure:

```
struct [structure_tag]
{
Member definition
Member definition
..
Member definition
} [one or more structure variables];
```

The structure tag is optional and each member definition is a normal variable definition or any other valid variable definition. At the end of the structure's definition, before the final semicolon, one or more structure variables can be specified but it is optional

Examples:

1.

```
struct Student
{
   char name[25];
   int age;
   char branch[10];
   // F for female and M for male
   char gender;
};
```

struct Student declares a structure to hold the details of a student which consists of 4 data fields, namely name, age, branch and gender. These fields are called structure elements or members. Student is the name of the structure and is called as the structure tag.

Each member can have different datatype: name is an array of char type and age is of int type branch and gender is of character type.

2.

```
struct address
{
char name[50];
char street[100];
```


Scanned with CamScanner

```
char city[50];
char state[20];
int pin;
};
```

A structure named 'address' with 5 members is declared. The members namely name, street, city, and state are of character type of length 50, 100,50 and 20 respectively and pin is of integer type.

3.

```
struct Books {
  char  title[50];
  char  author[50];
  char  subject[100];
  int   book_id;
} book;
```

A structure named 'Books' with 4 members is declared. The members title, author and subject are of character type of length 50, 50 100 respectively. The book_id is of integer type.

4.

```
struct Person
{
   char name[50];
   int citNo;
   float salary;
};
```

A structure named 'Person' with 3 members is declared. The members name is of character type, citNo of integer type and salary is of float type.

5.

```
struct
{
   float x, y;
} complex;
```

An anonymous structure with two members with float type, x and y is declared. The structure type has no tag and is therefore unnamed or anonymous.

Scanned with CamScanner

Declaring structure variable :

When a struct type is declared, no storage or memory is allocated. To allocate memory of a given structure type and work with it, we need to create variables.

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

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

struct keyword within main() function :

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


int main()
{
struct employee e1;
}
```

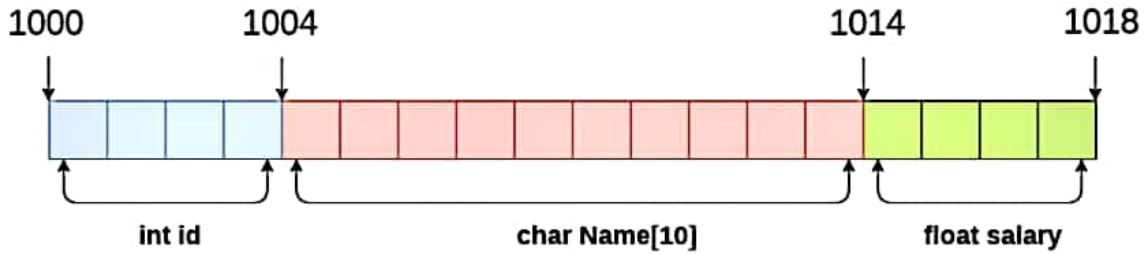
The variable e1 can be used to access the values stored in the structure.

Declaring a variable at the time of defining the structure:

```
struct employee

{  int id;

   char name[50];

   float salary;

}e1;
```

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

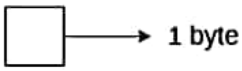
Scanned with CamScanner




```
struct Employee
{
    int id;
    char Name[10];
    float salary;
} emp;
```

sizeof (emp) = 4 + 10 + 4 = 18 bytes

where;
sizeof (int) = 4 byte
sizeof (char) = 1 byte
sizeof (float) = 4 byte



Accessing members of the structure :

There are two ways to access structure members:

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

Let's see the code to access the id member of p1 variable by. (member) operator.

```
#include<stdio.h>
#include <string.h>
struct employee
{   int id;
    char name[50];
}e1;  //declaring e1 variable for structure
int main( )
{
  //store employee information
  e1.id=101;
  strcpy(e1.name, "ABC");//copying string into char array
  //printing first employee information
  printf( "employee 1 id : %d\n", e1.id);
  printf( "employee 1 name : %s\n", e1.name);
return 0;
}
```

Output:

```
employee 1 id : 101
employee 1 name : ABC
```


Scanned with CamScanner

Designated Initialization :
Designated Initialization allows structure members to be initialized in any order.

```
#include<stdio.h>

struct Point
{
int x, y, z;
};

int main()
{
// Examples of initialization using designated initialization
struct Point p1 = {.y = 0, .z = 1, .x = 2};
struct Point p2 = {.x = 20};

printf ("x = %d, y = %d, z = %d\n", p1.x, p1.y, p1.z);
printf ("x = %d", p2.x);
return 0;
}
```

Output:
x = 2, y = 0, z = 1
x = 20

Array of structures :

```
#include<stdio.h>

struct Point
{
int x, y;
};

int main()
{
// Create an array of structures
struct Point arr[10];

// Access array members
arr[0].x = 10;
arr[0].y = 20;

printf("%d %d", arr[0].x, arr[0].y);
```


Scanned with CamScanner

```
return 0;
}
```

Output:
10 20

Structure pointer :
Like primitive types, we can have pointer to a structure. If we have a pointer to structure, members are accessed using arrow (->) operator.

```
#include<stdio.h>
struct Point
{
int x, y;
};

int main()
{
struct Point p1 = {1, 2};

// p2 is a pointer to structure p1
struct Point *p2 = &p1;

// Accessing structure members using structure pointer
printf("%d %d", p2->x, p2->y);
return 0;
}
```

Output:
1 2

Limitations of C Structures :

In C language, Structures provide a method for packing together data of different types. However, C structures have some limitations.

1. The C structure does not allow the struct data type to be treated like built-in data types:
2. We cannot use operators like +,- etc. on Structure variables.

```
struct number
{
        float x;
};
```


Scanned with CamScanner

```
int main()
{
        struct number n1,n2,n3;
        n1.x=4;
        n2.x=3;
        n3=n1+n2;
        return 0;
}

/*Output:

prog.c: In function 'main':
prog.c:10:7: error:
invalid operands to binary + (have 'struct number' and 'struct number')
n3=n1+n2;

*/
```

3. No Data Hiding: C Structures do not permit data hiding. Structure members can be accessed by any function, anywhere in the scope of the Structure.
4. Functions inside Structure: C structures do not permit functions inside Structure
5. Static Members: C Structures cannot have static members inside their body.

Pointers

A **pointer** is a variable that stores the address of another variable. Unlike other variables that hold values of a certain type, pointer holds the address of a variable. For example, an integer variable holds (or you can say stores) an integer value, however an integer pointer holds the address of a integer variable.

A simple example to understand how to access the address of a variable without pointers?

In this program, we have a variable num of int type. The value of num is 10 and this value must be stored somewhere in the memory, right? A memory space is allocated for each variable that holds the value of that variable, this memory space has an address. The value of the variable is stored in a memory address, which helps the C program to find that value when it is needed.

So let's say the address assigned to variable num is 0x7fff5694dc58, which means whatever value we would be assigning to num should be stored at the location: 0x7fff5694dc58.

Scanned with CamScanner

```
#include <stdio.h>
int main()
{
  int num = 10;
  printf("Value of variable num is: %d", num);
  /* To print the address of a variable we use %p
   * format specifier and ampersand (&) sign just
   * before the variable name like &num.
   */
  printf("\nAddress of variable num is: %p", &num);
  return 0;
}
```

Output:

Value of variable num is: 10

Address of variable num is: 0x7fff5694dc58

A Simple Example of Pointers in C

This program shows how a pointer is declared and used. There are several other things that we can do with pointers, we just need to know how to link a pointer to the address of a variable.

Important point to note is: The data type of pointer and the variable must match, an int pointer can hold the address of int variable, similarly a pointer declared with float data type can hold the address of a float variable. In the example below, the pointer and the variable both are of int type.

Scanned with CamScanner

```
#include <stdio.h>
int main()
{
  //Variable declaration
  int num = 10;
  //Pointer declaration
  int *p;
  //Assigning address of num to the pointer p
  p = #
  printf("Address of variable num is: %p", p);
  return 0;
}
```

Output:

Address of variable num is: 0x7fff5694dc58

Operators that are used with Pointers:

Lets discuss the operators & and * that are used with Pointers in C.

"Address of" (&) Operator

We have already seen in the first example that we can display the address of a variable using ampersand sign. I have used &num to access the address of variable num. The **& operator** is also known as "**Address of**" Operator.

```
printf("Address of var is: %p", &num);
```

Point to note: %p is a format specifier which is used for displaying the address in hex format.

"Value at Address"(*) Operator

The * Operator is also known as **Value at address** operator.

Scanned with CamScanner

How to declare a pointer?

```
int *p1  /*Pointer to an integer variable*/
double *p2  /*Pointer to a variable of data type double*/
char *p3   /*Pointer to a character variable*/
float *p4   /*pointer to a float variable*/
```

The above are the few examples of pointer declarations. **If you need a pointer to store the address of integer variable then the data type of the pointer should be int.** Same case is with the other data types.

By using * operator we can access the value of a variable through a pointer.
For example:

```
double a = 10;
double *p;
p = &a;
```

*p would give us the value of the variable a. The following statement would display 10 as output.

```
printf("%d", *p);
```

Similarly if we assign a value to *pointer like this:

```
*p = 200;
```

It would change the value of variable a. The statement above will change the value of a from 10 to 200.

Example of Pointer demonstrating the use of & and *

```
#include <stdio.h>
int main()
{
  /* Pointer of integer type, this can hold the
   * address of a integer type variable.
   */
  int *p;
  int var = 10;
```


Scanned with CamScanner

```
  /* Assigning the address of variable var to the pointer
   * p. The p can hold the address of var because var is
   * an integer type variable.
   */
  p= &var;
  printf("Value of variable var is: %d", var);
  printf("\nValue of variable var is: %d", *p);
  printf("\nAddress of variable var is: %p", &var);
  printf("\nAddress of variable var is: %p", p);
  printf("\nAddress of pointer p is: %p", &p);
  return 0;
}
```

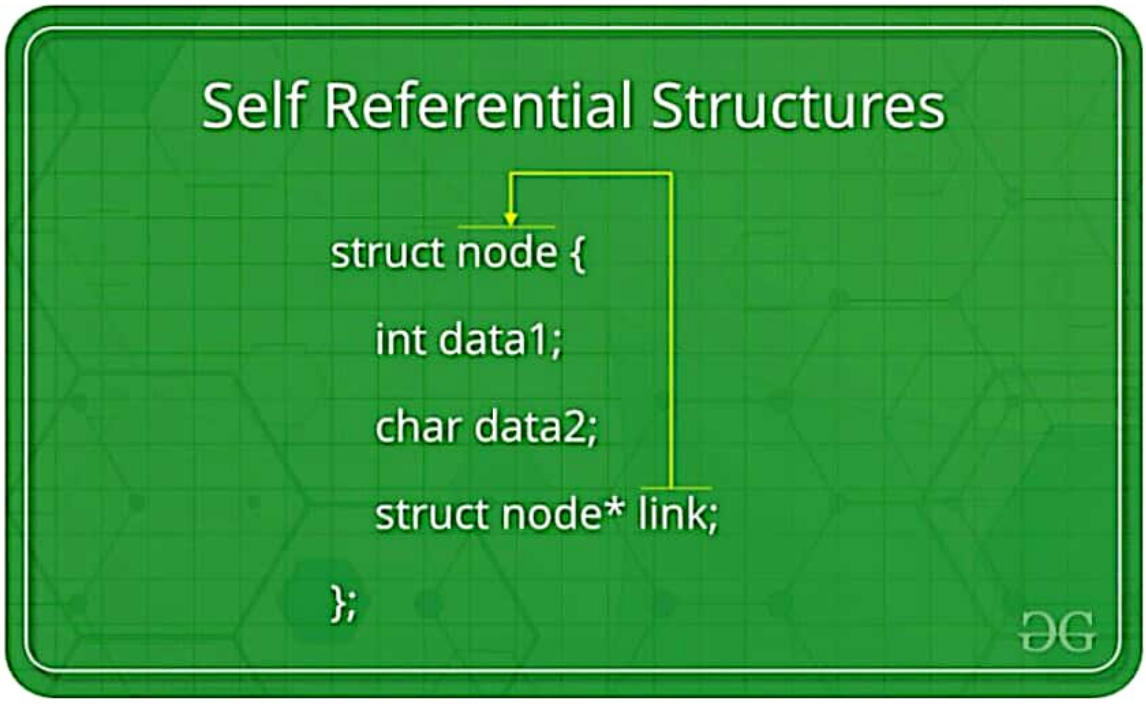
Output:

```
Value of variable var is: 10
Value of variable var is: 10
Address of variable var is: 0x7fff5ed98c4c
Address of variable var is: 0x7fff5ed98c4c
Address of pointer p is: 0x7fff5ed98c50
```

Self Referential Structures

Self Referential structures are those structures that have one or more pointers which point to the same type of structure, as their member.

Scanned with CamScanner



In other words, structures pointing to the same type of structures are self-referential in nature.

Example:

```
struct node {
    int data1;
    char data2;
    struct node* link;
};

int main()
{
    struct node ob;
    return 0;
}
```

In the above example 'link' is a pointer to a structure of type 'node'. Hence, the structure 'node' is a self-referential structure with 'link' as the referencing pointer.
An important point to consider is that the pointer should be initialized properly before accessing, as by default it contains garbage value.

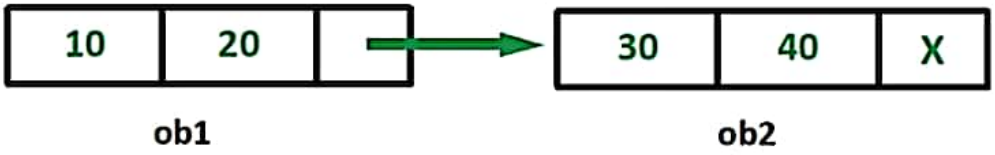
Types of Self Referential Structures

1. Self Referential Structure with Single Link
2. Self Referential Structure with Multiple Links

Self Referential Structure with Single Link: These structures can have only one self-pointer as their member. The following example will show us how to connect the objects of a self-referential structure with the single link and access the corresponding data members. The

Scanned with CamScanner

connection formed is shown in the following figure.



Example:

```
#include <stdio.h>

struct node {
    int data1;
    char data2;
    struct node* link;
};

int main()
{
    struct node ob1; // Node1

    // Initialization
    ob1.link = NULL;
    ob1.data1 = 10;
    ob1.data2 = 20;

    struct node ob2; // Node2

    // Initialization
    ob2.link = NULL;
    ob2.data1 = 30;
    ob2.data2 = 40;

    // Linking ob1 and ob2
    ob1.link = &ob2;

    // Accessing data members of  ob2 using ob1
    printf("%d", ob1.link->data1);
    printf("\n%d", ob1.link->data2);
    return 0;
}
```

Output:

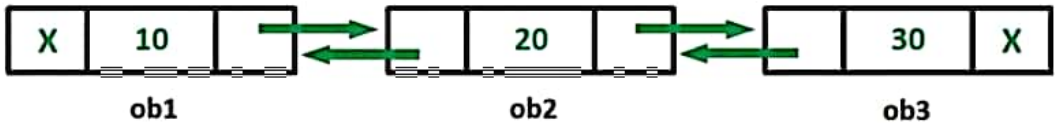
30

40

Self Referential Structure with Multiple Links: Self referential structures with multiple links can have more than one self-pointers. Many complicated data structures can be easily

Scanned with CamScanner

constructed using these structures. Such structures can easily connect to more than one nodes at a time. The following example shows one such structure with more than one links.
The connections made in the above example can be understood using the following figure.



Example:

```
#include <stdio.h>

struct node {
    int data;
    struct node* prev_link;
    struct node* next_link;
};

int main()
{
    struct node ob1; // Node1

    // Initialization
    ob1.prev_link = NULL;
    ob1.next_link = NULL;
    ob1.data = 10;

    struct node ob2; // Node2

    // Initialization
    ob2.prev_link = NULL;
    ob2.next_link = NULL;
    ob2.data = 20;

    struct node ob3; // Node3

    // Initialization
    ob3.prev_link = NULL;
    ob3.next_link = NULL;
    ob3.data = 30;

    // Forward links
    ob1.next_link = &ob2;
    ob2.next_link = &ob3;

    // Backward links
    ob2.prev_link = &ob1;
    ob3.prev_link = &ob2;

    // Accessing  data of ob1, ob2 and ob3 by ob1
```


Scanned with CamScanner

```
    printf("%d\t", ob1.data);
    printf("%d\t", ob1.next_link->data);
    printf("%d\n", ob1.next_link->next_link->data);

    // Accessing data of ob1, ob2 and ob3 by ob2
    printf("%d\t", ob2.prev_link->data);
    printf("%d\t", ob2.data);
    printf("%d\n", ob2.next_link->data);

    // Accessing data of ob1, ob2 and ob3 by ob3
    printf("%d\t", ob3.prev_link->prev_link->data);
    printf("%d\t", ob3.prev_link->data);
    printf("%d", ob3.data);
    return 0;
}
```

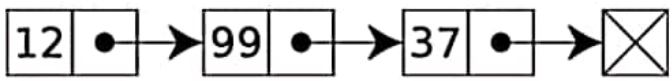
Output:

```
10  20  30

10  20  30

10  20  30
```

In the above example we can see that 'ob1', 'ob2' and 'ob3' are three objects of the self referential structure 'node'. And they are connected using their links in such a way that any of them can easily access each other's data. This is the beauty of the self referential structures. The connections can be manipulated according to the requirements of the programmer.

Notion of Linked Lists

A linked list is a dynamic data structure where each element (called a **node**) is made up of two items - the data and a reference (or pointer) which points to the next node. A linked list is a collection of nodes where each node is connected to the next node through a pointer. The first node is called a **head** and if the list is empty then the value of head is NULL.



A Simple Linked List

Scanned with CamScanner

For a real-world analogy of linked list, you can think of conga line, a special kind of dance in which people line up behind each other with hands on shoulders of the person in front. Each dancer represents a data element while their hands serve as the pointers or links to the next element.

Advantages of Linked Lists

The dynamic nature of linked list comes with a number of advantages.

- In contrast to arrays, which have pre-defined or fixed length, linked lists have a dynamic length which can be increased or decreased at runtime.
- Insertion and deletion operations in the linked list are much faster in comparison to other data structure such as the queue, stack, and arrays.

Consequently, it often better to consider using a list when the exact volume and quantity is not known ahead of time and cannot be made fixed. For instance, a programmer designing a school management system cannot determine how many students will enroll in the school. Therefore, it is most efficient to choose an ordered list data structure over arrays.

Linked list as Self-referencing Structure

As self-referential structure means that at least one member of the structure is a pointer to the structure of its own type. An implementation of self-referential structure is as follows:

```
1. struct Node {
2.   int data;
3.   struct Node *next;
4. }
```

In the above code, the structure *node* contains data element *data* as well as pointer *next* which points to the structure of the same type. The (*) indicates a pointer definition and it points to the address of the next node of the linked list. As the linked list is traversed using the *next* pointer, the value of the pointer in the last node will be NULL. The self-referential structure is the reason why a linked list is called a dynamic data structure and can be expanded and pruned at runtime.

Ordered List vs Ordered Array

Suppose we have an ordered array *arr[]* having integer values.

```
1. arr[] = [50, 100, 120, 150, 200];
```


Scanned with CamScanner

If we have to insert a new value 70 into the array, then we have to move all elements after 50 to maintain the ordered array. Similarly, if we have to delete a value 100 from the array, we have to move all the values after 100. So, insert and delete operations are expensive in ordered arrays. If we maintain an ordered list, the insert and delete operations are faster and more efficient due to the use of pointers.

Following are the basic operations supported by a list.

- Insertion – Adds an element at the beginning of the **list**.
- Deletion – Deletes an element at the beginning of the **list**.
- Display – Displays the complete **list**.
- Search – Searches an element using the given key.
- Delete – Deletes an element using the given key.
-

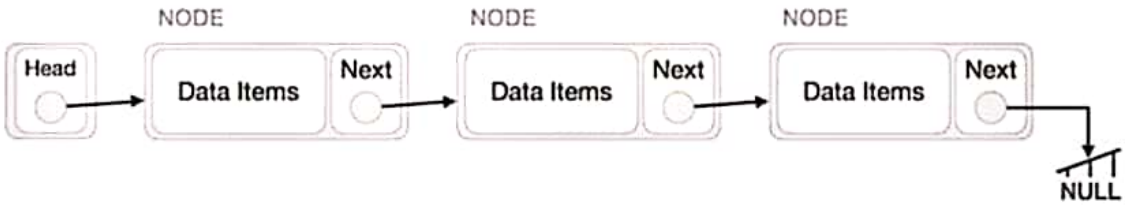
Linked List Representation

A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Link** – Each link of a linked list can store a data called an element.
- **Next** – Each link of a linked list contains a link to the next link called Next.
- **LinkedList** – A Linked List contains the connection link to the first link called First.
- Last link carries a link as null to mark the end of the list.

Linked list can be visualized as a chain of nodes, where every node points to the next node.



Types of Linked List

Following are the various types of linked list.

- **Simple Linked List** – Item navigation is forward only.
- **Doubly Linked List** – Items can be navigated forward and backward.
- **Circular Linked List** – Last item contains link of the first element as next and the first element has a link to the last element as previous.

Scanned with CamScanner

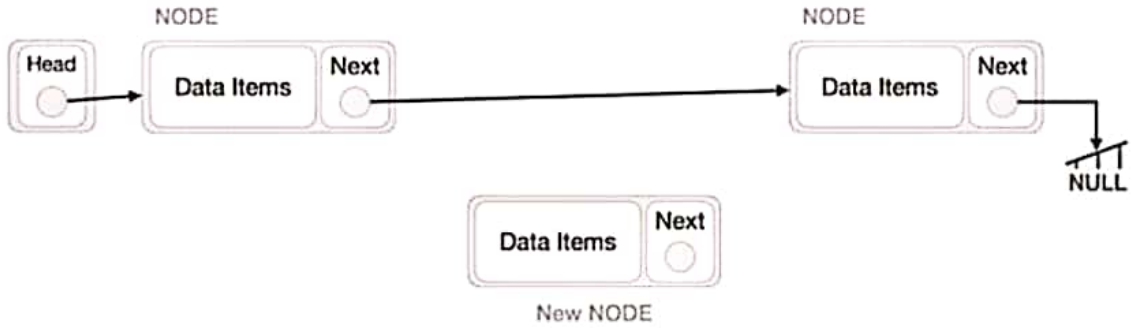
Basic Operations

Following are the basic operations supported by a list.

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Display** − Displays the complete list.
- **Search** − Searches an element using the given key.
- **Delete** − Deletes an element using the given key.

Insertion Operation

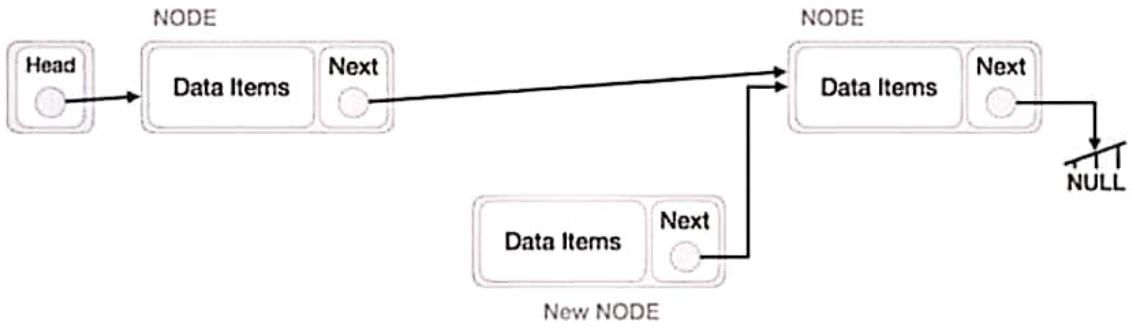
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



Imagine that we are inserting a node **B** (NewNode), between **A** (LeftNode) and **C** (RightNode). Then point B.next to C −

NewNode.next −> RightNode;

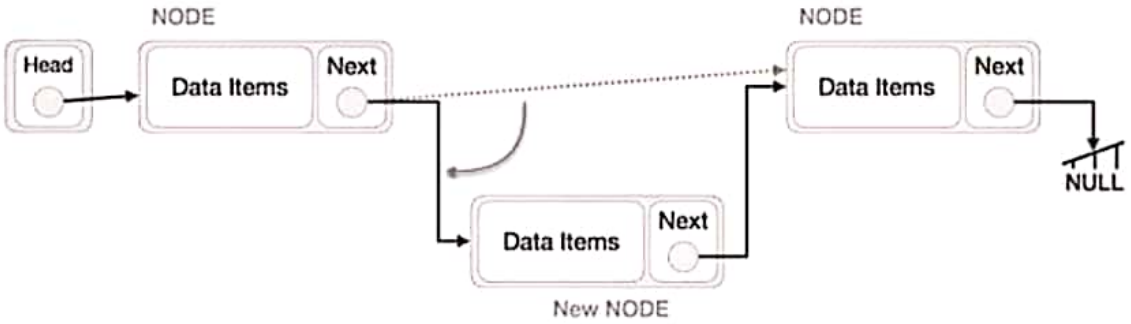
It should look like this −



Now, the next node at the left should point to the new node.

LeftNode.next −> NewNode;

Scanned with CamScanner

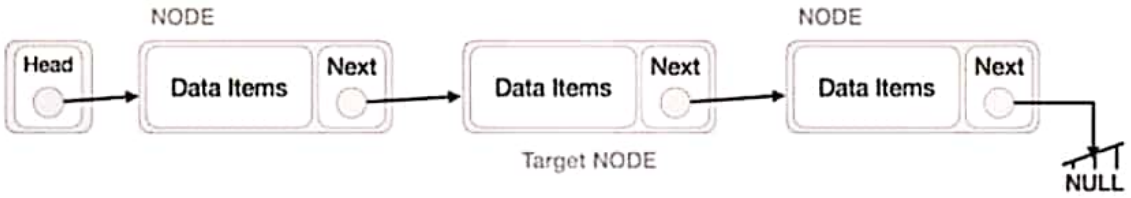


This will put the new node in the middle of the two. The new list should look like this –

Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

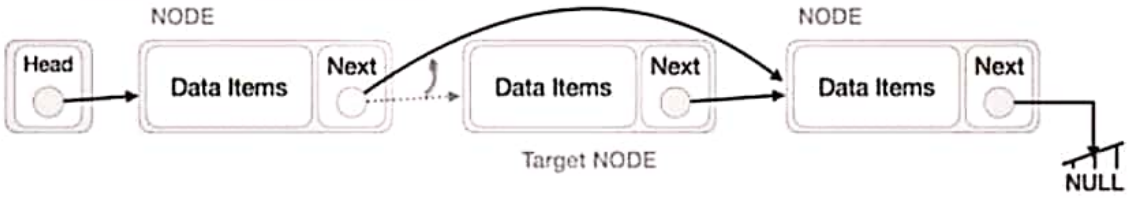
Deletion Operation

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.



The left (previous) node of the target node now should point to the next node of the target node –

LeftNode.next –> TargetNode.next;

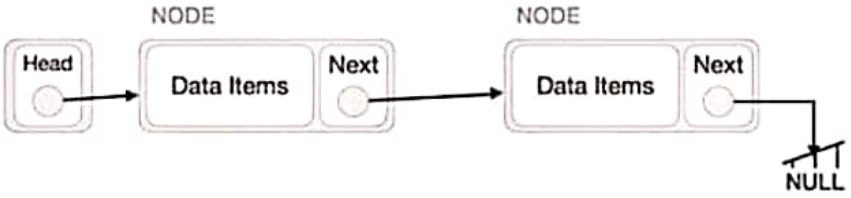


This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

TargetNode.next –> NULL;

Scanned with CamScanner

We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.



File Handling

A **file** represents a sequence of bytes on the disk where a group of related data is stored. File is created for permanent storage of data. A file is a container in computer storage devices used for storing data. The uses of files are:

- When a program is terminated, the entire data is lost. Storing in a file will preserve your data even if the program terminates.
- If you have to enter a large number of data, it will take a lot of time to enter them all. However, if you have a file containing all the data, you can easily access the contents of the file using a few commands in C.
- You can easily move your data from one computer to another without any changes.

Types of Files:

1. Text files

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

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

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

Scanned with CamScanner

2. Binary files

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

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

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

File Operations :

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

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

I/O functions:

C provides a number of functions that helps to perform basic file operations. Following are the functions:

Function Name	Operations
fopen()	create a new file or open a existing file
fclose()	closes a file
getc()	reads a character from a file
putc()	writes a character to a file
fscanf()	reads a set of data from a file
fprintf()	writes a set of data to a file
getw()	reads a integer from a file
putw()	writes a integer to a file
fseek()	set the position to desire point
ftell()	gives current position in the file
rewind()	set the position to the begining point

Working with files:

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

FILE *filepointer;

Scanned with CamScanner

Opening or creating file :-

For opening a file, fopen function is used with the required access modes. Some of the commonly used file access modes are mentioned below.

File opening modes in C:

- **"r"** – Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer which points to the first character in it. If the file cannot be opened fopen( ) returns NULL.
- **"w"** – Searches file. If the file exists, its contents are overwritten. If the file doesn't exist, a new file is created. Returns NULL, if unable to open file.
- **"a"** – Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the last character in it. If the file doesn't exist, a new file is created. Returns NULL, if unable to open file.
- **"r+"** – Searches file. If is opened successfully fopen( ) loads it into memory and sets up a pointer which points to the first character in it. Returns NULL, if unable to open the file.
- **"w+"** – Searches file. If the file exists, its contents are overwritten. If the file doesn't exist a new file is created. Returns NULL, if unable to open file.
- **"a+"** – Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer which points to the last character in it. If the file doesn't exist, a new file is created. Returns NULL, if unable to open file.

As given above, if you want to perform operations on a binary file, then you have to append 'b' at the last. For example, instead of "w", you have to use "wb", instead of "a+" you have to use "a+b". For performing the operations on the file, a special pointer called File pointer is used which is declared as

FILE *filePointer;

So, the file can be opened as

filePointer = fopen("fileName.txt", "w")

The second parameter can be changed to contain all the attributes listed in the above table.

Reading from a file :–

The file read operations can be performed using functions fscanf or fgets. Both the functions performed the same operations as that of scanf and gets but with an additional parameter, the file pointer. So, it depends on you if you want to read the file line by line or character by character.

And the code snippet for reading a file is as:

```
FILE * filePointer;
filePointer = fopen("fileName.txt", "r");
fscanf(filePointer, "%s %s %s %d", str1, str2, str3, &year);
```

Writing a file :-

The file write operations can be perfomed by the functions fprintf and fputs with similarities to read operations. The snippet for writing to a file is as :

```
FILE *filePointer ;
FilePointer = fopen("fileName.txt", "w");
 fprintf(filePointer, "%s %s %s %d", "We", "are", "in", 2012);
```


Scanned with CamScanner

Closing a file :-

After every successful fie operations, you must always close a file. For closing a file, you have to use fclose function. The snippet for closing a file is given as :

FILE *filePointer ;
filePointer= fopen("fileName.txt", "w");
---------- Some file Operations -------
fclose(filePointer)

Example 1: Program to Open a File, Write in it, And Close the File

```
// C program to Open a File,
// Write in it, And Close the File

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

int main( )
{

    // Declare the file pointer
    FILE *filePointer ;

    // Get the data to be written in file
    char dataToBeWritten[50]
        = "c is a high level language";

    // Open the existing file Test.txt using fopen()
    // in write mode using "w" attribute
    filePointer = fopen("Test.txt", "w") ;

    // Check if this filePointer is null
    // which maybe if the file does not exist
    if ( filePointer == NULL )
    {
        printf( "Test.txt file failed to open." ) ;
    }
    else
    {

        printf("The file is now opened.\n") ;

        // Write the dataToBeWritten into the file
        if ( strlen (  dataToBeWritten  ) > 0 )
        {

            // writing in the file using fputs()
            fputs(dataToBeWritten, filePointer) ;
            fputs("\n", filePointer) ;
        }
```


Scanned with CamScanner

```
        // Closing the file using fclose()
        fclose(filePointer) ;

        printf("Data successfully written in file Test.txt\n");
        printf("The file is now closed.") ;
    }
    return 0;
}
```

This program takes a character which is stored in the variable "dataToBeWritten" and stores in the file Test.txt.

After you compile and run this program, you can see a text file Test.txt created in C drive of your computer. When you open the file, you can see the integer you entered.

Example 2: Program to Open a File, Read from it, And Close the File

```
// C program to Open a File,
// Read from it, And Close the File

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

int main( )
{

    // Declare the file pointer
    FILE *filePointer ;

    // Declare the variable for the data to be read from file
    char dataToBeRead[50];

    // Open the existing file Test.txt using fopen()
    // in read mode using "r" attribute
    filePointer = fopen("Test.txt", "r") ;

    // Check if this filePointer is null
    // which maybe if the file does not exist
    if ( filePointer == NULL )
    {
        printf( "Test.txt file failed to open." ) ;
    }
    else
    {

        printf("The file is now opened.\n") ;

        // Read the dataToBeRead from the file
        // using fgets() method
```


Scanned with CamScanner

```
        while( fgets ( dataToBeRead, 50, filePointer ) != NULL )
        {

            // Print the dataToBeRead
            printf( "%s" , dataToBeRead ) ;
         }

        // Closing the file using fclose()
        fclose(filePointer) ;

        printf("Data successfully read from file Test.txt\n");
        printf("The file is now closed.") ;
    }
    return 0;
}
```

This program reads the string present in the Test.txt file and prints it onto the screen.

If you successfully created the file from Example 1, running this program will get you the string you entered.

Scanned with CamScanner