CS3251 PROGRAMMING IN C L T P C
3 0 0 3

COURSE OBJECTIVES:

i To understand the constructs of C Language.
i To develop C Programs using basic programming constructs
i To develop C programs using arrays and strings
i To develop modular applications in C using functions
i To develop applications in C using pointers and structures
i To do input/output and file handling in C

UNIT I BASICS OF C PROGRAMMING 9

Introduction to programming paradigms – Applications of C Language - Structure of C program – C programming: Data Types - Constants – Enumeration Constants - Keywords – Operators: Precedence and Associativity - Expressions - Input/Output statements, Assignment statements – Decision making statements - Switch statement - Looping statements – Preprocessor directives - Compilation process

UNIT II ARRAYS AND STRINGS 9

Introduction to Arrays: Declaration, Initialization – One dimensional array – Two dimensional arrays - String operations: length, compare, concatenate, copy – Selection sort, linear and binary search.

UNIT III FUNCTIONS AND POINTERS 9

Modular programming - Function prototype, function definition, function call, Built-in functions (string functions, math functions) – Recursion, Binary Search using recursive functions – Pointers – Pointer operators – Pointer arithmetic – Arrays and pointers – Array of pointers – Parameter passing: Pass by value, Pass by reference.

UNIT IV STRUCTURES AND UNION 9

Structure - Nested structures – Pointer and Structures – Array of structures – Self referential structures – Dynamic memory allocation - Singly linked list – typedef – Union - Storage classes and Visibility.

UNIT V FILE PROCESSING 9

Files – Types of file processing: Sequential access, Random access – Sequential access file - Random access file - Command line arguments.

COURSE OUTCOMES:

Upon completion of the course, the students will be able to
CO1: Demonstrate knowledge on C Programming constructs
CO2: Develop simple applications in C using basic constructs
CO3: Design and implement applications using arrays and strings
CO4: Develop and implement modular applications in C using functions.
CO5: Develop applications in C using structures and pointers.
CO6: Design applications using sequential and random access file processing.

TOTAL : 45 PERIODS

TEXT BOOKS:

1. ReemaThareja, "Programming in C", Oxford University Press, Second Edition, 2016.
2. Kernighan, B.W and Ritchie,D.M, "The C Programming language", Second Edition, Pearson Education, 2015.

REFERENCES:

1. Paul Deitel and Harvey Deitel, "C How to Program with an Introduction to C++", Eighth edition, Pearson Education, 2018.
2. Yashwant Kanetkar, Let us C, 17 th Edition, BPB Publications, 2020.
3. Byron S. Gottfried, "Schaum's Outline of Theory and Problems of Programming with C", McGraw-Hill Education, 1996.
4. Pradip Dey, Manas Ghosh, "Computer Fundamentals and Programming in C", Second Edition, Oxford University Press, 2013.
5. Anita Goel and Ajay Mittal, "Computer Fundamentals and Programming in C", 1 st Edition, Pearson Education, 2013.

CS 3251 PROGRAMMING IN C

UNIT - I

BASICS OF C PROGRAMMING

Introduction to programming paradigms - Applications of C Language - Structure of C Program - C Programming: Data Types - Constants - Enumeration constants - Keywords - Operators: Precedence and Associativity - Expressions - Input/output statements, Assignment statements - Decision making statements - Switch statement - Looping statements - Preprocessor directives - Compilation Process.

Anita Goel

1.1. INTRODUCTION TO PROGRAMMING PARADIGMS

Programming Paradigm

* Different patterns and models for writing a program.

classification:

1. Structured Programming
2. Object-oriented Programming (OOP) and
3. Aspect-oriented Programming (AOP)

* **Unstructured Programming:**
  → All actions of a small and simple program were defined within a single program only.

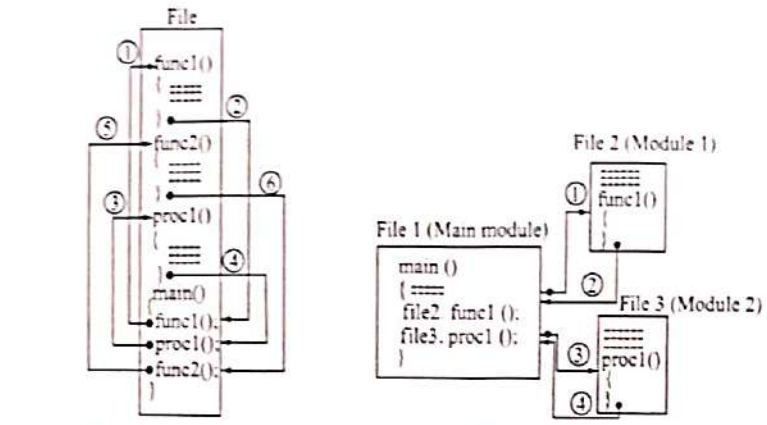
1.1.1. Structured Programming:

* Involves building of programs using small modules. The modules are easy to read and write
2. The problem to be solved is broken down into small tasks that can be written independently. Once written, the small tasks are combined together to form the complete task.
3. Performed in two ways:
   i) Procedural Programming
   ii) Modular Programming.

Fig.

4. **Procedural Programming:**
   * A given task to be divided into small procedures, functions or subroutines.

Procedural Program:
→ A single file consisting of many procedures and functions. and a function named main()



(i) Procedural programming (ii) Modular programming

* A procedure or function performs a specific task.

→ The function main()
- integrates the procedures and functions by making calls to them

* When a procedure or function is called, the execution control jumps to the called procedure or function, the procedure or function is executed, and after execution the control comes back to the calling procedure or function.

5. Modular Programming:

* Breaking down of a program into a group of files
- where each file consists of a program that can be executed independently.

→ The problem is divided into different independent but related tasks.

* For each identified task,
  - a separate program (module) is written, which is a program file that can be executed independently.
* The different files of the program are integrated using a main program file.
* The main program file invokes the other files in an order that fulfills the functionality of the program.

6. Structured Programming:
   * The approach to develop the software is process-centric or procedural.
   → The procedures and modules become tightly interwoven and interdependent
   - They are not re-usable

7. Examples: C, COBOL, Pascal.

1.1.2. Object-Oriented Programming (OOP):

* The software is broken into components not based on their functionality, but based on the components or parts of the software
→ Each component consists of data and the methods that operate on the data.
* The components are complete by themselves and are re-usable

Terms:

1. class:
   * Basic building block of the object-oriented programming.

* A class consists of data attributes and methods that operate on the data defined in the class.

2. **Object:**
   * a runtime instance of the class.
   * An object has a state, defined behavior and a unique identity.

→ **state**: represent by the data defined in the class.

→ The methods defined in the class represent object behavior.

- A class is a template for a set of objects that share common data attributes and common behavior.

3. **Abstraction:**
   * Allows picking out the relevant details of the object, and ignoring the non-essential details.
   — Encapsulation is a way of implementing abstraction

4. **Encapsulation:**
   * Information hiding
     → hides the data defined in the class.
   * Encapsulation separates implementation of the class from its interface.

5. **Inheritance:**
   * Allows a new class called the derived class, to be derived from an already existing class known as the base class.

→ The derived class (subclass) inherits all data and methods of the base class (superclass).

6. **Polymorphism:**
   * Means many forms.
   * It allows different objects to respond to the same message in different ways.

* Increases the flexibility of the program.

- C++ and Java are object-oriented Programming languages.

1.1.3 Aspect-Oriented Programming (AOP)

* A new programming paradigm that handles the crosscutting concerns of the software.

→ **crosscutting concerns:**
- Global concerns like logging, authentication, security, performance etc. that do not fit into a single module or related modules.

* Focuses on the issue of handling crosscutting concerns at the programming language level.
* Helps the programmer in clearly separating the core concerns and the crosscutting concerns of the software.
* AOP introduces a new modular unit called "aspect"
  → that encapsulates the functionality of the crosscutting concerns.
  → Aspects of a system are independent elements that can be changed, inserted or removed at compile time, and even reused without affecting the rest of system.
* Aspects are similar to the classes of OOP.
* At compile time, the classes of OOP and the aspects are combined into a final executable form using an "aspect weaver".
* AspectJ and AspectC are examples of aspect-oriented Programming languages.

* After choosing the suitable programming paradigm, the coding of the logic of a program has to be done in a computer programming language.

characteristics of a Good Program.

1. The program should be well-written so that it is easily readable and structured.
2. The program should not have hard-coded input values.
   - must be a general program that accepts input from the user.
3. The programmer should also be well-documented so that later the author or any other programmer can understand the program.
4. A program must be designed to be portable.
   - minimum dependence on a particular OS.

1.2. APPLICATIONS OF C LANGUAGE

1) System Programming:

→ To implement i) operating systems

ii) Embedded systems applications

* portability
* Efficiency
* the ability to access specific hardware addresses
* low runtime demand on system resources

2) Compilers, libraries and interpreters of other languages are implemented in C.

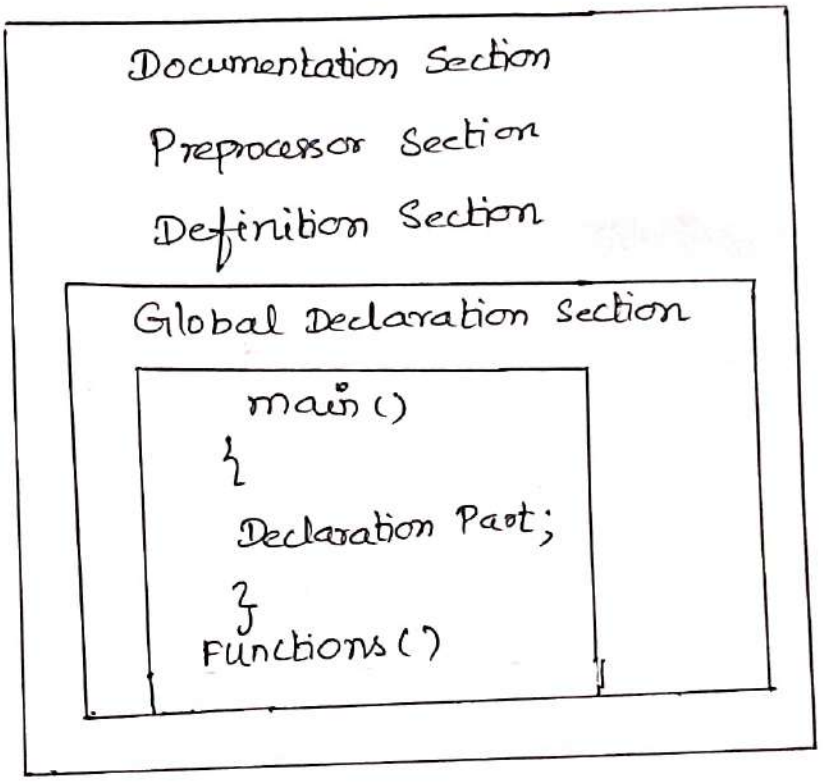
3) Used as intermediate language.

EX: (other languages)

→ BitC, Gambit, the Glasgow Haskell compiler, Squeak, Vala

4) Used to implement end-user applications.

1.3 STRUCTURE OF C PROGRAM



* C program composed of 3 important sections:
  - i) Preprocessor directives } → optional. (i.e) may or may not present in C pgm)
  - ii) Global Declarations }
  - iii) Functions → mandatory.

First C Program:

```
#include <stdio.h>
main()
{
 printf(" Welcome to the world of C.");

}
```

* A C program is made up of functions

1. Documentation Section:

* It consists of set of comment lines
  - to specify the name of the programmer
  - helps the reader to understand the code clearly.
* Non-executable statements. → they are not processed by the compiler.

Two types of comments:

i) Single-line comment
ii) Multi-line comment.

i) Single-line comment:

* // is used to comment a single line
  - automatically terminated with end of line.

Eg: // Adding 2 numbers

ii) Multi-line comment:

→ starts with /* and terminates with */
→ used when multiple lines of text are to be commented.
* All statements that lie within these characters are commented.

EX: /*
. . . .
. . . . */

2. Preprocessor Section:

* Used to link system library files for defining the macros and for defining the conditional inclusion.

→ start with a pound symbol #
→ # should be the first non-white space character in a line
→ terminated with a new line character, not with a semicolon ;

* Preprocessor directives are executed before the compiler compiles the source code.
  → change the source code.
→ To add the code (include directive) will be required.

Ex:

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

→ Preprocessor directive statement
→ includes standard input/output header (.h) file.
* The file is to be included if standard input/output functions like

```
printf
scanf
```

are to be used in a program.

3. Global Declaration Section:
   * It is optional
   → The variables that are used in many functions are declared as global variables in this section.
   → declares the variables in outside of the main fn.

4. Functions section:
   * Mandatory & must be present in a C program.
   * Can have one or more functions.
   → A function named main is always required

   Two parts:
   i) Header of the function
   ii) Body of the function

   i) Header:

```
main()
{
```

General form:

```
[return type]   function_name ([arg_list])
{

}
```

Body of the Function:

* Set of statements enclosed within curly brackets known as braces

Types of statements

a) Non-executable statements → declaration stmt
b) Executable statements.

→ first non-executable statements are present, then executable statements are written.

EX:

```
// C Program - Addition of two numbers
#include <stdio.h>
#include <conio.h>
main()
{
    int a,b,c;                          // Declaration
    printf(" Enter a number :");
    scanf(" %d", &a);
    printf("\nEnter b number :");
    scanf(" %d", &b);
    c = a+b;
    printf("\nThe sum is : %d", c);
    getch();
}
```

output:

```
Enter a number : 5
Enter b number : 10
The sum is : 15
```

1.4 C PROGRAMMING

1.4.1 DATA TYPES

* The type of the data, that are going to access within the program.
* Data type is one of the most important attributes of an identifier
- determines the possible values that an identifier can have and the valid operations that can be applied on it.
- Each data type may have predefined memory requirement and storage representation.

* Data types are broadly classified as:

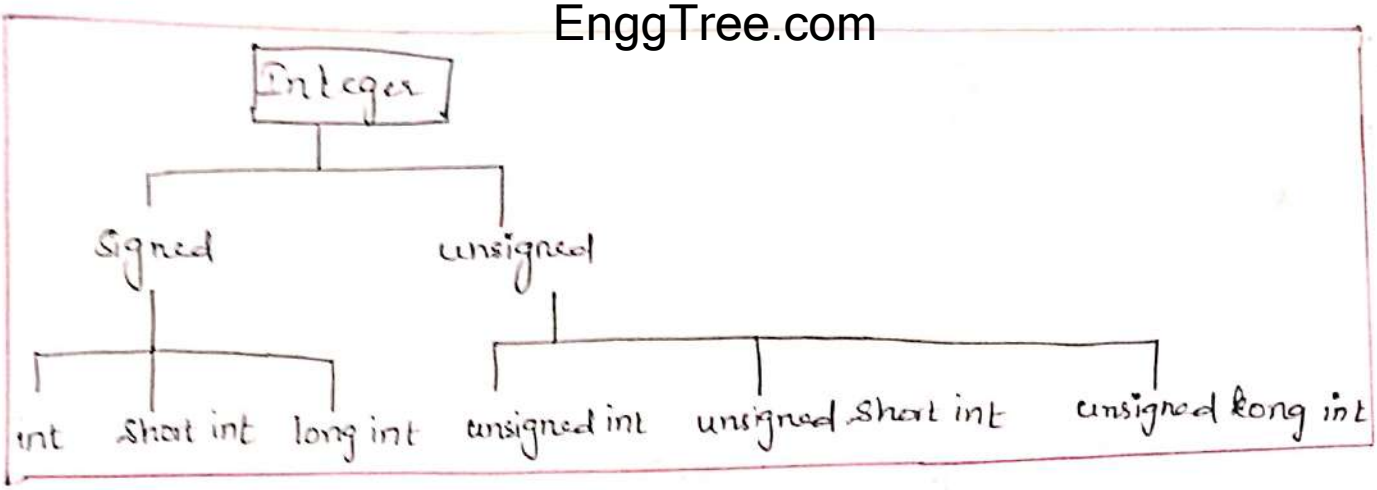
1. Basic data types (primitive data types)
2. Derived data types
3. User-defined data types

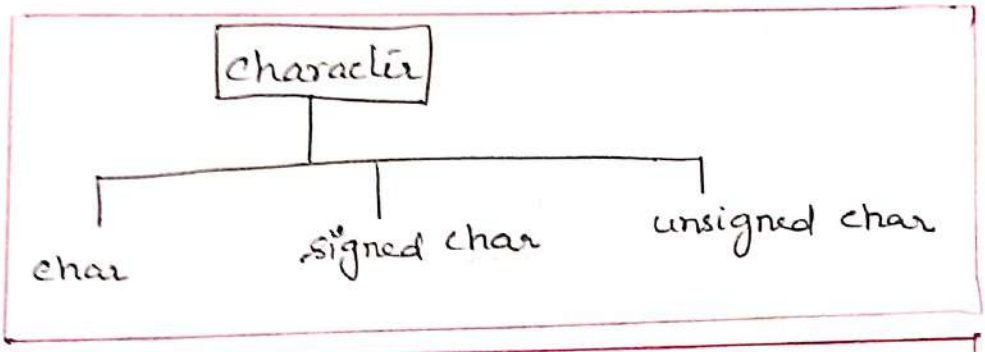
1. Basic data types:

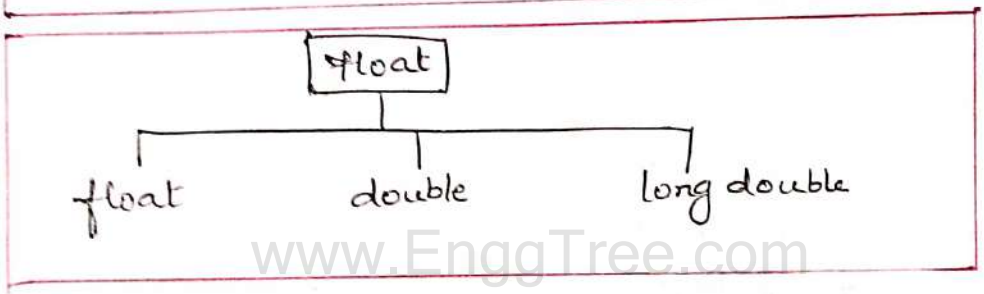
S.No	data type	Keyword	
1.	character	char	→ single character
2.	Integer	int	→ numbers
3.	Single Precision floating point (32 bits)	float	
4.	Double-precision floating point (64 bits)	double	
5.	No value available	void	→ null data type, has no arguments.

Bytes occupied:

Data type	Memory bytes	Range	Control string	Example
int	2 bytes	-32,768 to +32,768	%.d or %.i	int a=39;
char	1 byte	-128 to +128	%.C	char a='n';
float	4 bytes	3.4E-38 to 3.4E+38	%.f or %.g	float f=29.74;
double	8 bytes	1.7E-308 to 1.7E+308	%.lf	double d=29771177076







2. Derived data types:

- derived from basic data types.

1. Array type

eg: char[], int[], etc.

2. Pointer type:

eg: char*, int*, etc.

3. Function type:

eg: int (int, int), float(int), etc.

3. User-defined Data types:

*provides flexibility to the user to create new data types.

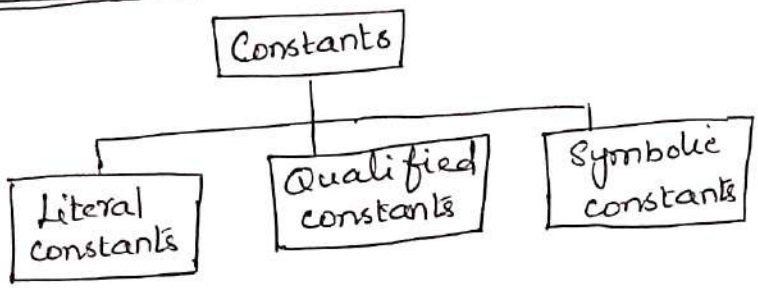
- newly created data types are called user-defined data types. → created by using
  1. Structure
  2. Union
  3. Enumeration.

1.4.2 CONSTANTS

CONSTANTS:

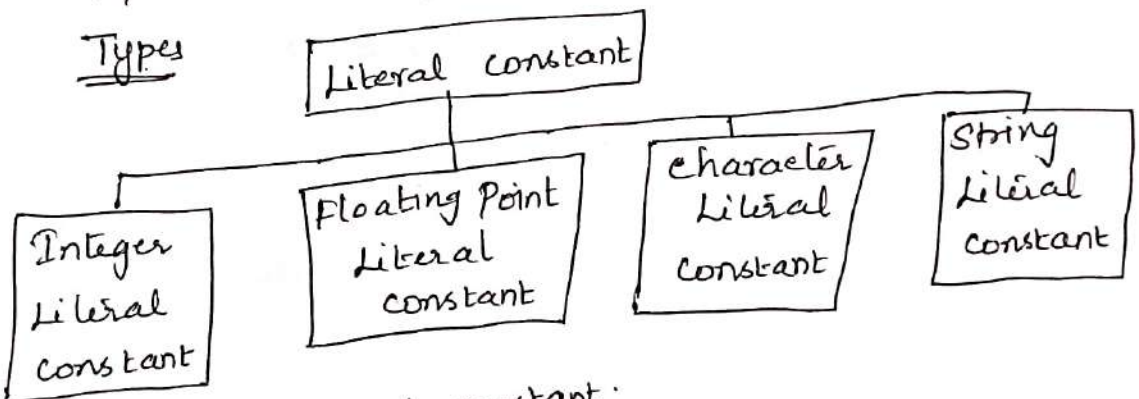
* A constant is an entity whose value remains the same throughout the execution of a program.
  - can be placed on the right side of the assignment operator.

classification:



i) Literal constant:-
   → denotes a fixed value
   → may be an integer, floating point number, character or a string.
   → determined by its value.

Types



a) Integer Literal constant:

- Integer values like -1, 2, 8 etc.

i) Decimal
eg: +89, -245

ii) Octal
eg: 025, 0173
- first digit - 0

iii) Hexadecimal:
begin with 0x
eg: 12 → 0xC

Rules

i) An integer literal constant must have at least one digit
ii) It should not have any decimal point.
iii) It can be either positive or negative.
iv) No special characters and blank spaces are allowed within an integer literal constant

b) Floating point Literal constant:

- values like -23.1, 12.8 etc.
- can be written in a fractional form or in an exponential form.

Rules i) fractional form

- must have atleast one digit
- should have a decimal point
- can be either positive or negative
- No special characters and blank spaces are allowed within a constant.
- type double

eg: 23.45

ii) exponential form.

mantissa part / exponential part.

eg. 2e10 (or) 2×10^{10}

-2.5E12

-2.5e-12

c) Character Literal constant:

* can have one or at most two characters enclosed within single quotes

eg 'A', 'a', '\n'

classification:

i) Printable character Literal constant

ii) Non printable character Literal constant.

i) Printable character Literal constant:

→ All characters of source character set except ?, \ and \n

Nonprinta

eg: '6', 'A', '#'

ii) Non-printable character Literal constant

→ represented with the help of escape sequences

Escape sequence

\', \", \?, \\

\a, \b, \f, \n, \r, \t, \v, \0

- Consists of \ (backslash) followed by a character and both enclosed within a single quotes.
- treated as a single character

d) String Literal Constant:

* consists of a sequence of characters enclosed within double quotes.
* terminated by a null character ('\0')

eg: "ABC"

number of bytes occupied → number of characters + 1

(double quotes) " " → occupies one byte

length of the string → number of characters.

ii) Qualified constants:

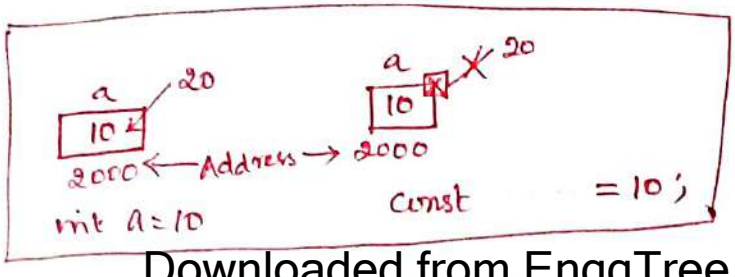
- created by using const qualifier

eg: const char a = 'A';

qualified constant

int a = 10 → allocates 2 bytes to a & initializes 10.
- possible to modify the value of a.

const int a = 10 → The qualifier places a lock on the box after placing the value in it
- Possible to see, but not possible to modify



(iii) Symbolic constants:

- created with the help of the define preprocessor directive

eg: # define PI 3.14124

PI → Symbolic constant.

→ replaced by its actual value during the preprocessing stage.

1.4.3 ENUMERATION CONSTANT

* An enumeration is a list of constant integer value

```
enum bookean (NO, YES);
```

→ first name in an enum has value 0, the next 1 and so on, unless explicit values are specified.

```
enum escapes {BELL = '\a', BACKSPACE = '\b',
              TAB = '\t', NEWLINE = '\n',
              VTAB = '\v', RETURN = '\r'};

enum months {JAN = 1, FEB, MAR, APR, MAY,
             JUN, JUL, AUG, SEP, OCT,
             NOV, DEC};   /* FEB is 2, MAR is 3, etc */
```


Defined in two ways

(i)
```
enum week{Mon, Tue, Wed};
enum week day;
```

(ii)
```
enum week{Mon, Tue, Wed} day;
```

* user defined data type
* used to assign names to integral constants

EX:

```
enum week {Mon, Tue, Wed, Thur, Fri, Sat, Sun};
int main()
{
  enum week day;
  day = wed
  printf("%d", day);
  return 0;
}
```

O/P

2

1.4.4 KEYWORDS

* C has a set of 32 reserved words known as Keywords
* having fixed meaning and cannot be used as an identifier
* All Keywords must be written in lowercase letters

List of Keywords

1. auto	11. else	21. return	31. void
2. break	12. enum	22. short	32. volatile
3. case	13. extern	23. signed	
4. char	14. float	24. sizeof	
5. const	15. for	25. static	
6. continue	16. goto	26. struct	
7. default	17. if	27. switch	
8. do	18. int	28. typedef	
9. while	19. long	29. union	
10. double	20. register	30. unsigned	

1.5 OPERATORS:

1.5.1 PRECEDENCE AND ASSOCIATIVITY

* **OPERATORS:**

* C language supports a lot of operators to be used in expressions.

→ An operator specifies the operation to be applied to its constants

EX:

```
a = printf("Hello") + 2
```

3 operators:

* function call operator ()
* arithmetic operator +
* assignment operator =

classification of operators:

*) Based on

i) number of operands

ii) role of an operator.

i) Based on number of operands:

Three types:

a) Unary operator

b) Binary operator

c) Ternary operator.

a) Unary operator:

→ operates on only one operand

EX: −3 unary minus operator

b) Binary operators:

→ operates on two operands.

Ex:

2-3

binary minus operator

<<, ==, ++, &

c) Ternary operator:

→ operates on three operands

Ex:

?:

conditional operator

ii) Based on Role of operators:

1. Arithmetic operators.
2. Relational operators.
3. Equality operators.
4. Logical operators.
5. Unary operators.
6. Conditional operators.
7. Bitwise operators.
8. Assignment operators.
9. Comma Operators.
10. Sizeof operator.
11. Operator Precedence chart.

① Arithmetic Operators:

* Arithmetic operations like addition, subtraction, multiplication, division etc can be performed by using arithmetic operators.

Operator	Name
+	unary plus, Addition
-	unary minus, subtraction.
++	Increment
--	Decrement
*	Multiplication
/	Division
%	Modulus

Three modes: — Binary arithmetic

a) Integer mode → both operands are of integer type

eg: 4/3

b) Floating point mode → both operands are of floating point type

eg: 4.0/3.0

c) Mixed mode → one is integer type and another is floating point type

eg:

4/3.0

* unary plus – appear only towards the left side of its operand
* unary minus – appear only towards the left side of its operand

* Increment – i) pre-increment : ++a; → the value of the operand is incremented first and it is used for evaluation

ii) post-increment : a++; → the value of the operand is used first and then it is incremented.

* Decrement – i) pre-decrement : --a; → value is decremented first and then used for evaluation

ii) post-decrement : a--; → value is used for evaluation and then decremented.

* Division operator: (/)
  - to find the quotient
  - sign of the result is based on numerator and denominator.

* Modulus operator (%):
  - to find the remainder.
  - operands must be of integer type.

ex:

```
a = 3%2;
```

- sign depends upon the numerator.

2. Relational Operators:
   - used to compare two quantities. (operands)
   - six relational operators.

Operator	Name
<	Less than
>	greater than
<=	Less than or equal to
>=	greater than or equal to
==	Equal to
!=	Not equal to.

- No white-space character between two symbols.

* result - boolean constant ie) 0 or 1.
  0 - false
  1 - true.

* An expression that involves a relational operator forms a condition.

ex:

```
a<b
```

3 Logical Operators

→ used to logically relate the sub-expressions

operator	Name
!	Logical NOT
&&	Logical AND
||	Logical OR

AND operation

operand1	opnd2	Result
F	F	F
F	T	F
T	F	F
T	T	T

OR operation

opnd1	opnd2	Result
F	F	F
F	T	T
T	F	T
T	T	T

NOT operation

operand	Result
F	T
T	F

* Expressions are evaluated left to right

eg: E1 && E2
E1 || E2

i = i && j++

4. Bitwise Operators:

- Six operators for bit manipulation

operator	Name
~	Bitwise NOT
<<	Left shift
>>	Right shift
&	Bitwise AND
^	Bitwise X-OR
|	Bitwise OR

* operate on the individual bits of the operands
* can be applied on operands of type char, short, int, long

Bits

	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	
2 →	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	
3 →	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	
2 & 3 →	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	= 2
2 | 3 →	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	= 3

X-OR operation

opnd1	opnd2	Result
F	F	F
F	T	T
T	F	T
T	T	F

NOT - 1st complement.

Left shift - is equivalent to multiplication by 2.
right shift - is equivalent to integer division by 2.

Bits

	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	
4 →	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	
4<<1 →	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	=8
4>>1 →	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	=2

5. Assignment operator:

* A variable can be assigned a value by using an assignment operator.

operator	Name
=	simple assignment
*=	Assign product
/=	Assign quotient
%=	Assign modulus
+=	Assign sum

-=	Assign difference
&=	Assign bitwise AND
|=	Assign bitwise OR
^=	Assign bitwise XOR
<<=	Assign left shift
>>=	Assign right shift.

General form

operand1 operator = operand2

op1 = op1 op op2

ex: a/=2 ⟹ a = a/2

x = a+b
x = 10

* No white space between the symbols.

6. Miscellaneous operators:

→ other operators:

1. function call operator — ()
2. Array subscript operator — []
3. Member select operator
   a) Direct member access operator → . (dot)
   b) Indirect member access operator → → (arrow)
4. Indirection operator. *
5. Conditional operator.
6. Comma operator.
7. Sizeof operator.
8. Address-of operator.

① Conditional operator:

- ternary operator.

operator	Name
? :	conditional operator.

General form:

E1 ? E2 : E3

sub-expns

* E_1 is evaluated first
  - E_1 is true, E_2 is evaluated and E3 is ignored.
  - E_1 is false, E_3 is evaluated and E2 is ignored.

ex: a>b ? a : b

② Comma operator:
→ used to join multiple expressions together.

operator	Name
,	Comma operator.

form: E1, E2, E3, .., En

③ sizeof operator:
→ used to determine the size in bytes, which a value or a data object will take in memory.

operator	Name
sizeof	Size-of operator.

General form:

a) sizeof expression or sizeof (expression

ex:
sizeof 2
sizeof(a)
sizeof(2+3)

b) sizeof (type-name)

ex:
sizeof(int)
sizeof(int*)
sizeof(char)

* The type of result of evaluation of the sizeof operator is int.
* The operand of sizeof operator is not evaluated.
* cannot be applied on operands of incomplete type or function call.

④ Address-of Operator:

→ used to find the address.

operator	Name
&	Address-of operator.

Syntax:

& operand

↓

Variable or function.

* cannot be applied to constants, expressions and variables with register storage class.

Precedence of all operators.

	operator
1.	() [] → .
2.	! ~ + - ++ -- & * sizeof
3.	* / %
4.	+ -
5.	<< >>
6.	< > <= >=

7	== !=
8.	&
9	^
10.	|
11.	&&
12.	||
13.	?:
14	= += /= %= += -= &= |= ^= <<= >>=
15.	,

* Associativity:

* if operators of the same precedence appear together, then the operators are evaluated according to their associability.

→ left-to-right

→ right-to left. a = b = 3

ex: 2 * 3/5

2 * 3 = 6

6/5

a * (b+c) + c * (d+e)

y = a * x * x + b * x + c

(6) (1) (2) (4) (3) (5)

z = p * r % q + w / x - y

(6) (1) (2) (4) (3) (5)

i = 2 * 3/4 + 4/4 + 8 - 2 + 5/8

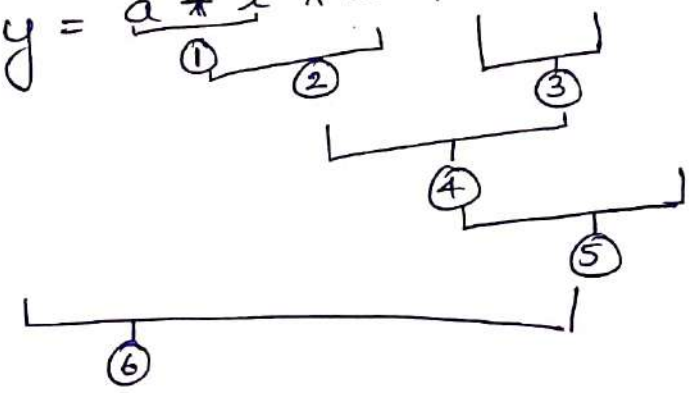
* Associativity of operators determines the order in which operators of equal precedence are evaluated when they occur in the same expression.

→ defines the direction left-to-right or right-to-left in which the operator acts upon its operands

L to R	R to L
( ) [ ] . ++ --	++ , --,
* / % + -	! ~, +, -, &, *
<< >>	?:
< <= > >=	= += -= *= /= %=
== !=	>>= <<= &= ^= |=
&	
^	
|	
&&	
||	
,	

Ex:

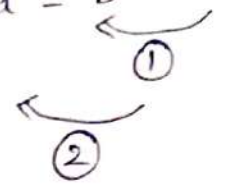
① $y = a * x * x + b * x + c$



② $z = p * r \% q + w / x - y$

(6) (1) (2) (4) (3) (5)

③ $a = b = 3$



1.5.2 EXPRESSIONS

* An expression is made up of one or more operands and operators that specify the operations to be performed on operands

→ An expression is a sequence of operands and operators that specifies the computation of a value.

Ex:

$a = 2+3$

operands = 3 ⇒ a, 2, 3

operators = 2 ⇒ =, +

= : assignment operator

+ : Arithmetic operator.

Operands:

* An operand specifies an entity on which an operation is to be performed.

→ An operand can be a variable name, a constant, a function call or a macro name.

Ex:

$a = printf("Hello") + 2$

- variable name
- function call
- constant ⇒ 3 operands
- Assignment operator
- Arithmetic operator

Operators:

* An operator specifies the operation to be applied to its constants.

Ex:

```
a = printf("Hello")+2
```

Classification of Expressions:

i) Simple Expressions
ii) Compound Expressions.

i) Simple Expressions:

* An expression that has only one operator.

Ex:

```
a+2
```

ii) Compound Expressions:

* An expression that involves more than one operator

Ex:

```
b = 2+3*5
```


→ Determine the order in which operators will operate.
→ depends on the precedence and associativity of operators.

```
b = 2+ 3*5
      ⇓
      15
```

* → highest prec.
+
=

```
b = 2+15
    ⇓
b = 17
```

x —— x

1. Arithmetic operators:

Sample program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int b,c,d;
int sum, mul, sub, rem;
float div;
clrscr();
printf("Enter values of b,c,d:");
scanf("%d%d%d\n\n\n", &b,&c,&d);
sum=b+c;
sub=b-c;
mul=b*c;
div=b/c;
rem=b%d;
printf("\n sum=%d,\n sub=%d,\n mul=%d,\n div=
%f",sum,sub,mul,div);
printf("\n Remainder of division of b&d is
%d",rem);
getch();
}
```

Output:

```
Enter the values of b,c,d:
2
3
4
Sum=5;
Sub=-1
Mul=6
Div=0.6666666
Reminder of division of b&d is 0
```

2. Relational Operators:

Sample Program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
clrscr();
printf("\n condition: Return value \n");
printf("\n 5!=5: %5d", 5!=5);
printf("\n 5==5:%d",5==5);
printf("\n 5>=50: %d",5>=50);
printf("\n 5<=50: %d",5<=50);
printf("\n 5!=3: %5d", 5!=3);
getch();
}
```

Output:

```
Condition : Return value
5!=5 : 0
5==5 : 1
5>=50 : 0
5<=50 : 1
5!=3 : 1
```

3. Logical Operators:

Sample program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int c1,c2,c3;
clrscr();
printf("Enter the values c1,c2,c3");
scanf("%d%d%d", &c1,&c2&c3);
if((c1<c2)&&(c1<c3))
printf("c1 is less than c2 and c3");
if(!(c1<c2))
printf("\n c1 is greater than c2");
if((c1<c2)||(c1<c3))
printf("c1 is less than c2 or c3 both");
getch();
}
```


Output:

```
Enter the values c1,c2,c3
9
6
3
C1 is greater than c2
```

4. Bitwise operator:

Sample Program:

```
#include<stdio.h>
#include<conio.h>
Void main()
{
int a,b,c;
clrscr();
a=10;
b=20;
c=a&b;
printf("Bitwise AND=%d",c);
c=a/b;
printf("Bitwise OR=%d",c);
c=a+b;
printf("Bitwise XOR=%d",c);
c=~a;
```

```
printf("one's complement=%d",c);
getch();
}
```

Output:
AND=2
OR=10
XOR=8
One's complement= 5

5. Assignment Operator:
Sample Program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j,k;
clrscr();
k=(i=4,j=5);
printf("k=%d",k);
getch();
}
```

Output:
k=5

6. Increment / Decrement Operator:
Sample Program :

```
#include<stdio.h>
#include<conio.h>
Void main()
{
int a=10;
clrscr();
printf("a++=%d\n",a++);
printf("++a=%d\n",++a);
printf("-- a=%d\n",--a);
printf("a--=%d\n",a--);
getch(); }
```

Output:
a++=10
++a=12
--a=11
a--=11

7. Conditional Operator:
Sample program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a=5,b=2,big;
clrscr();
big=(a>b)?a:b;
printf("Largest number is %d",big);
getch();
}
```

Output:
Largest number is 5

1) To check the integer is palindrome or not
2) To find sum of 10 non-neg. nos entered by the user
3) To find the largest among 3 nos

1.6 INPUT/OUTPUT STATEMENTS

1. STREAMS
2. FORMATTING INPUT/OUTPUT
3. printf()
4. scanf()
5. Examples of printf/scanf
6. Detecting Errors During Data Input

1. Streams:

Stream:

→ A sequence of bytes of data

Types:

i) input stream
ii) output stream

i) Input stream:

* A sequence of bytes flowing into a program

ii) output stream:

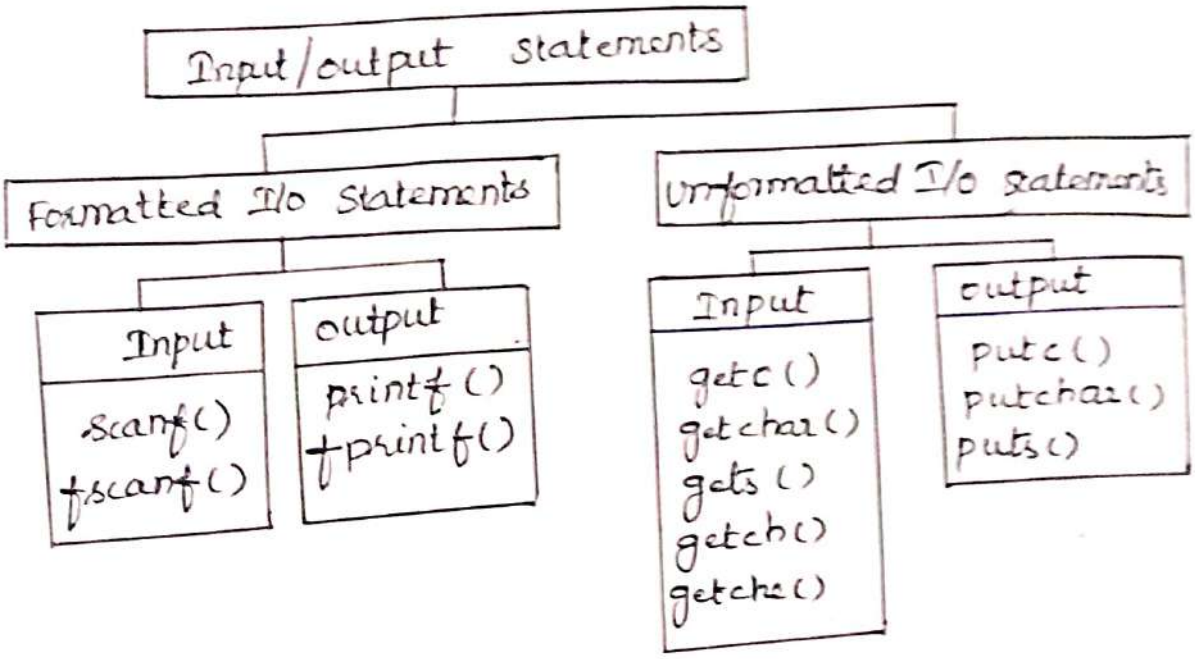
* A sequence of bytes flowing out of a program

Modes of streams:

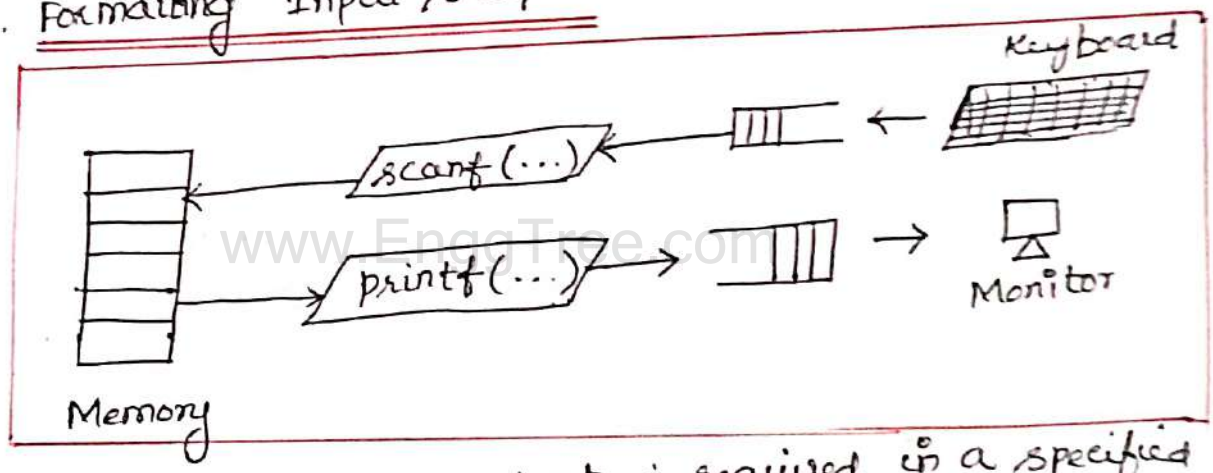
i) Text stream – only characters
ii) Binary stream – Any sort of data

Two Types of Input/output statements:

i) Formatted I/O statements
ii) Unformatted I/O statements



2. Formatting Input/output



* When input and output is required in a specified format the standard library functions are used
  - → printf()
  - → scanf()

3. printf():

* printf() function allows the user to output data of different data types on the console in a specified format.
  - translates internal values to characters

* output data can be written from the computer to standard output device using printf() function

printf ("Control String", var1, var2, . . , Var n);

Control string : → required formatting specifications enclosed within double quotes.
→ Diff. control strings are used based on data type

var1, var2, .. varn → individual data items

* printf() converts, formats and prints its arguments on the standard output

Two types of objects :

i) ordinary characters → copied to the O/P stream

ii) conversion specification
→ conversion & printing
* begins with a %

Table — Format specifiers for printf()

Conversion code	Usual variable type	Display
%c	char	single character
%d (%i)	int	signed integer
%e (%E)	float or double	exponential format
%f	float or double	signed decimal
%g (%G)	float or double	use %f or %e, whichever is shorter
%o	int	unsigned octal value
%p	pointer	address stored in pointer
%s	array of char	sequence of characters (string)
%u	int	unsigned decimal integer
%x (%X)	int	unsigned hex value
%%	none	no corresponding argument is converted, prints only a %.
%n	pointer to int	the corresponding argument is a pointer to an integer into which the number of characters displayed is placed.

Table — List of commonly used control codes

Control code	Action
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\'	Single quote
\0	Null

Table — Flag characters used in printf()

Flag	Meaning
-	Left justify the display
+	Display positive or negative sign of value
space	Display space if there is no sign
0	Pad with leading zeroes
#	Use alternate form of specifier

1. Scanf()

* scanf() reads characters from the standard input, interprets them according to the specification in format.

→ used to read input values

General form:

scanf("control string", var1_addr, var2_addr... varn_addr);

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

control string → formatting specification

Format specifiers :

Table – Format specifiers for scanf()

Conversion code	Usual variable type	Action
%c	char	Reads a single character.
%d(%i)	int	Reads a signed decimal integer.
%e(%E)	float or double	Reads signed decimal.
%f	float or double	Reads signed decimal.
%g(%G)	float or double	Reads signed decimal.
%o	int	Reads octal value.
%p	pointer	Reads in hex address stored in pointer.
%s	array of char	Reads sequence of characters (string).
%u	int	Reads unsigned decimal integer.
%x(%X)	int	Reads unsigned hex value.
%%	none	A single % character in the input stream is expected. There is no corresponding argument.
%n	pointer to int	No characters in the input stream are matched. The corresponding argument is a pointer to an integer into which the number of characters read is placed.
[...]	array of char	Reads a string of matching characters.

Ex:1

```
#include <stdio.h>
#include <conio.h>
main()
{
 char name[25];
 puts("Enter the name");
 gets(name);
 puts("\n Print the name");
 puts(name);
 getch();
}
```

Ex:2

```
#include <stdio.h>
#include <conio.h>
main()
{
  int idno;
  char name[25];
  float salary;
  printf("Enter ID no, Name, Salary : ");
  scanf("%d %s %f", &idno, name, &salary);
  printf("\n ID number : %d", idno);
  printf("\n Name : %s", name);
  printf("\n Salary : %.2f", salary);
  getch();
}
```

1.7 ASSIGNMENT STATEMENT

* Assigning the value to a variable using assignment operator is known as an assignment statement.

Syntax:

variable = constant / variable / expression

→ operator is "="

* Stores a value in the memory location which is denoted by a variable name.

```
int total; ------> [ ? ]   total
total = (1+2) *4; --> [ 12 ]  total
```

* The expression on the right hand side of the assignment statement can be:
  i) Arithmetic expression
  ii) logical expression
  iii) Relational expression
  iv) Mixed Expression

Ex:

```
int a;
float b,c,avg,t;
avg = (b+c)/2          // Arithmetic expr.
a = b&&c;              // logical expr.
a = (b+c)&&(b+c)       // mixed expr
```

Multiple Assignment Statement:

* A single value is assigned to two or more variables.

Syntax:

```
var1 = var2 = ... = varn = Value / Expression
```

Ex:

(i) m = n = 3

(ii) a = b = (c*c + d*d)/2

Ex:

```
#include <stdio.h>

main()
{
  int a = b = 10, c, d, e = 25;
  c = a+b;
  d = (c+b) * a + e;
  printf("%d %d", c, d);
}
```

Output:

20 325

1.8 DECISION MAKING STATEMENTS

Flow of program control:

- The order in which the program statements are executed.

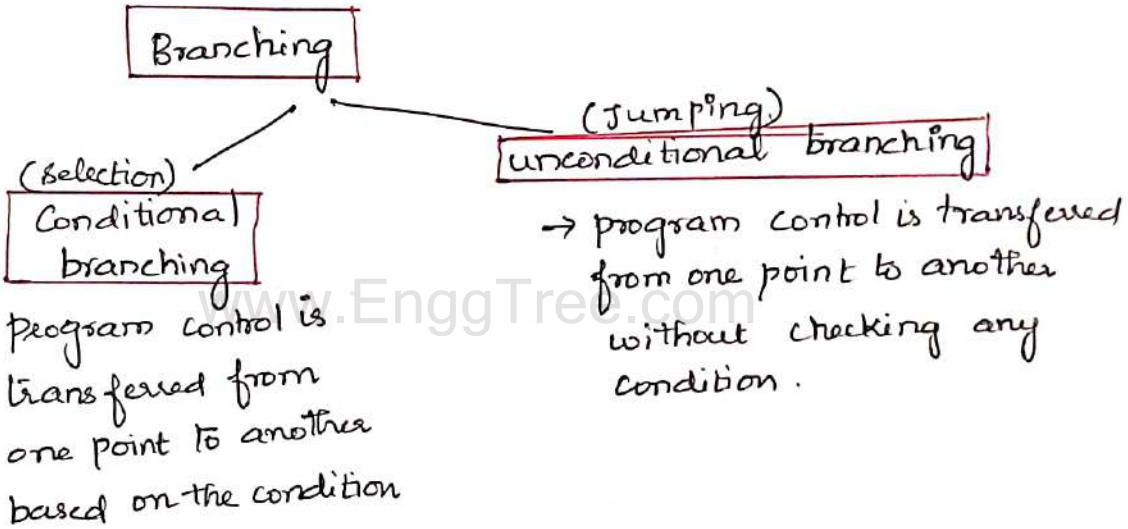
Two types:

a. Selection statements

b. Jump statements.

Branching statements:

- used to transfer the program control from one point to another.



a) selection statements:

* Based upon the outcome of a particular condition, selection statements transfer control from one point to another.

- select a statement to be executed

⇒ **statements:**

i) if statement

ii) if-else statement

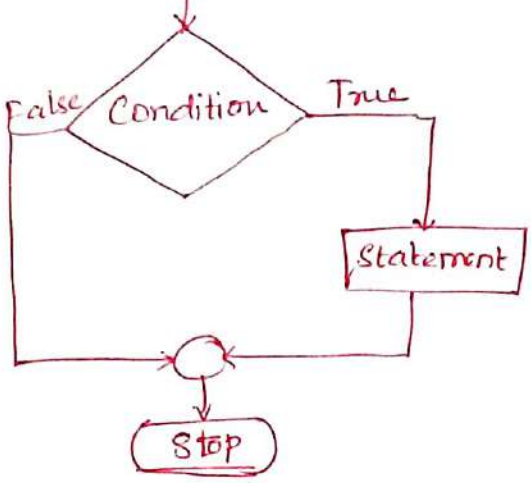
iii) if-else if statement

iv) switch statement.

i) if statement:

General form:

```
if (expression)
statement
```



Ex:

```
if (n>=18)
    printf("Eligible to vote");
```

* The condition should be specified within parenthesis
* Condition is executed first
* If the condition is true, then the statement will be executed.
* If the condition is false, the control will be directly transferred to the statements outside the braces.
* No semicolon should be placed at the end of if.

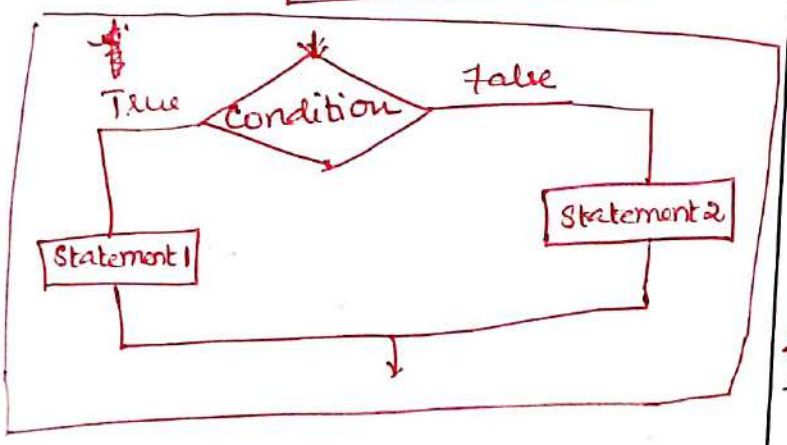
  Semicolon → error.

ii) **if-else statement:**

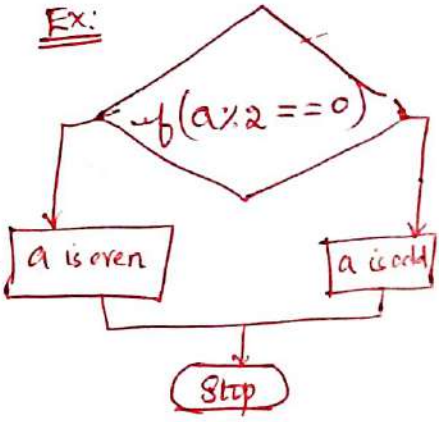
* set of actions to be performed if a particular condition is true and another set of actions to be performed if the condition is false.

General form:

```
if (expression)
    statement 1
else
    statement 2
```



Ex:



ex:-

```
if (a%2==0)
    printf("a is even");
else printf("a is odd");
```

→ Nested if statement:

* An if statement contains another if statement is called nested if statement.

General form:

```
if (expression)
  if statement
```

(or)

```
if (expression)
{
  statement
  .
  .
  if statement
  .
  .
  statement
}
```

* nesting can be done up to any level:

```
if (expression 1)
  if (expression 2)
    if (expression -n)
      statement;
```

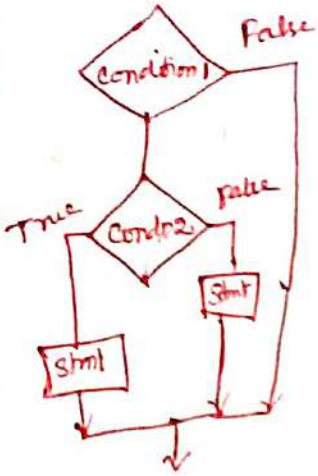

→ Nested if-else statement:

* if-else statement is, or contains another if statement or if-else statement.

dangling else problem → more number of if than else.

Syntax:

```
if (condition 1)
  {
    if (condition 2)
    {
      True statement 2;
    }
    else
    {
      False statement 2;
    }
  }
  else
  { false statement 1;
```



iii) if-else ladder (if-else-if statement)

nested if else if ladder

Syntax:

```
if (condition 1)
{
  statement 1;
}
else if (condition 2)
{
  statement 2;
}
else if (condition 3)
  { statement 3;
  }
else
  {
    default- statements;
  }
```

1.7.1

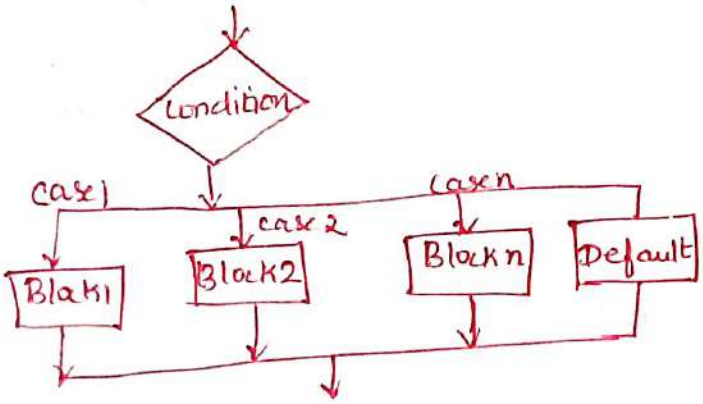
iv) Switch Statement:

- used to control complex branching operations.
- used for many conditions checking required.

→ provides an easy and organized way to select among multiple options, depending upon the outcome of a particular condition.

General form:

```
switch (expression)
statement
```



Ex:

```
switch (expression or var)
{
  case 1:
      statements;
      break;
   :
  case N:
      statements N;
      break;
  default:
      statement;
}
```

* The switch selection operation is evaluated.
* The result of evaluation is compared with the case labels until there is a match
  - if expression is matched, the execution starts from the matched case-labeled statement
  - if, it is not matched, then the default statements will be executed.

---

b) Jump statements:

* Transfers the control from one point to another without checking any condition. ie) unconditionally.

a) goto statement
b) break statement
c) continue statement
d) return statement

---

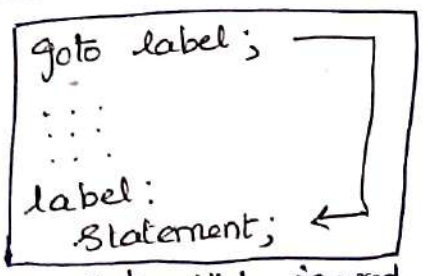
a) goto statement:-

- used to branch unconditionally from one point to another within a function.

* Transfers the program control to an identifier-labeled statement having a label name same as the label name used in the goto statement.

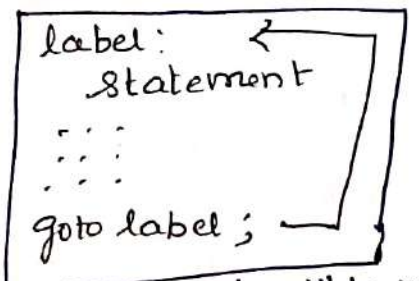
General form:-

Forward Jump



* some stmts will be ignored

Backward Jump



* some stmts will be repeatedly executed

---

b) break statement:

- used to terminate or to exit from a switch statement.

General form:

```
break;
```

Misplaced use of break

ex:

①
```
if (a==10)
{ printf("controlling expression evaluates to true");
  break;
}
```
→ compilation error

②
```
switch(a)
{
  case 1:
      printf("one");
       break;
  case 2:
      printf("two");
      break;
default:
      printf("default");
}
```

```
for(i=1; i<=10; i++)
{
  printf("%d", i);
}
```

break:

```
for(i=1; i<=10; i++)
{ if(i==3)
    break;
  printf("%d", i);
}
```

o/p 1 2

c) Continue statement:

- loop does not terminate
- used to transfer the control to the beginning of the loop.
- used within a while, do-while, for loops.

General form:

```
continue;
```

EX:

```
for(i=1; i<=12; i++)
{
  if(i==6)
  continue;
  printf("\n%d", i);
}
```

o/p

1
2
3
4
5
7
8
9
10
11
12

d) return statement:

i) * without an expression — can appear only in a function.

Syntax or General form:

```
return;
```

ii) * with an expression — should not appear in a function

General form:

```
return expression;
```

* terminates the execution of a function and returns the control to the calling function.

Ex:

① a) Multiplication Table

Backward Jump:

```
#include<stdio.h>
main()
{
  int num, i=1;
  printf("Enter the number for table:");
  scanf("%d", &num);
table:
  printf("\n %d x %d = %d \n", i, num, num*i);
  i++;
  if(i<=10)
  goto table;
}
```

b) Print n numbers

```
main()
{ int n, i=1;
  printf("Enter n value");
  scanf("%d", &n);
loop: printf("%d", i);
      i++;
      if(i<=n)
      goto loop;
}
```

② a) Forward Jump

```
#include<stdio.h>
main()
{
  printf("Hello");
  goto l1;
  printf("How are you");
l1: printf("Hi");
}
```

o/p

```
Hello
Hi
```

b)

```
main()
{
  int age;
  scanf("%d", &age);
  if(age>=18)
    goto VOTE;
  else printf("Not Eligible");
VOTE:
  printf("Eligible for vote");
}
```

i) If Statement:

Example Program :

```
#include<stdio.h>
#include <conio.h>
void main ( )
{
int a;
clrscr( );
printf("\n Enter the number:");
scanf("%d",&a);
if(a>10)
{
        printf(" \n a is greater than 10");
}
getch( );
}
```

Output :
Enter the number: 12
a is greater than 10

ii) if else statement:

Example program :

```
#include<stdio.h>
#include <conio.h>
void main ( )
{
int a;
clrscr( );
printf("\nEnter the number:");
scanf("%d",&a);
if(a>10)
{
        printf(" \n a is greater than 10");
}
else
{
        printf(" \n a is less than 10");
}
getch( );
}
```

Output :
Enter the number: 12
a is greater than 10
Enter the number: 8
a is less than 10

Nested if else statement:

Example program:

```
#include<stdio.h>
#include<conio.h>
main()
{
int n;
Printf("Entera no");
Scanf("%d",&n);
if(n==15)
{
        printf("play football");
}
        else
{
        if(n==10)
                printf("play cricket");
        else
                printf("play tennis");
}
}
```

Output:
Enter a number : 10
Playing cricket

iii) switch statement:

Example Program :

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,n;
printf("\nEnter the Number:");
scanf("%d",&n);
switch(n)
{
        case 1:
        {
        printf("\n Its in case 1");
        break;
        }
        case 2:
        {
        printf("\n Its in case 2");
        break;
        }
        default:
        {
        printf("\n Its in default");
        break;
        }
}
getch();
}
```

Output:
Enter the Number:2
Its in case 2

1.9 LOOPING STATEMENTS (Iteration statement)

* Iteration – the process of repeating the same set of statements again and again until the specified condition holds true.

* Computers execute the same set of statements again and again by putting them in a loop.

Three looping statements.

a) for statement
b) while statement
c) do-while statement.

* In general, loops are classified as:
  i) Counter-controlled loops – number of iterations is known in advance
  ii) sentinel-controlled loops – number of iterations is not known beforehand.

a) **for statement:**
   - most popular one.

General form:

```
for (initialization; condition; increment/decrement operation)
{
  statements;
}
```

- Three sections are separated by semicolon.

i) **initialization section:**
   - used to initialize the loop counter.

ii) **Condition section:**
   - tests the value of the loop counter.
   ie) determines whether the loop should continue or not.

iii) **Manipulation section / Increasing or decreasing operation**
   - manipulates the value of the loop counter so that the condition evaluates to false and the loop terminates
   ie) increasing or decreasing the value of the loop counter each time the program segment has been executed.

* The for statement is not terminated with a semicolon.
- if it is terminated with a semicolon, then the semicolon is interpreted as a null statement.

Execution:

a) Initialization section is executed only once at the start of the loop.

b) The expression present in the condition section is evaluated.

i) if it evaluates to true, the body of the loop is executed.

ii) if it evaluates to false, the loop terminates and the program control is transferred to the statement present next to the for statement.

c) After the execution of the body of the loop, the manipulation expression is evaluated.

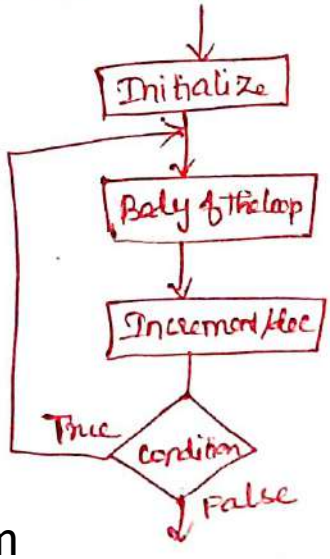
* Three steps represent the first iteration of the for loop.
* For the next iteration, steps b and c are repeated until the expression in step b evaluates to false.

Ex:

```
for(i=0; i<n; i++)
{
  printf("The numbers are %d", i);
}
```

descending order

```
for (x=10 ; x>1; x--)
{
  printf("%d", x);
}
```

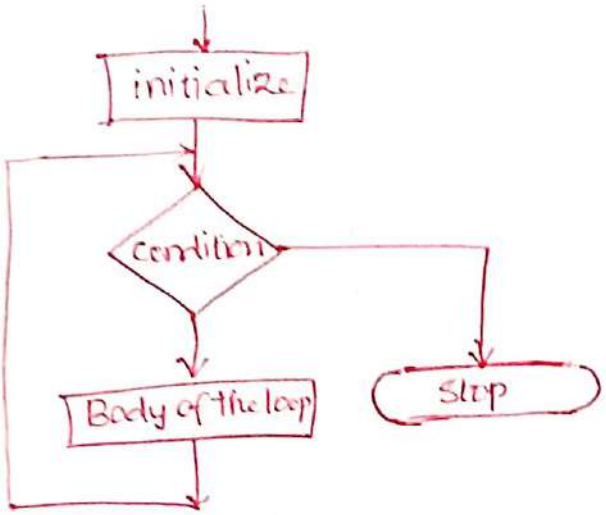


b) while statement:

General form

```
while (expression)
statement
```

```
initialize loop counter;
while (condition)
{
statement(s);
increment/decrement;
}
```



Ex:

```
int a=1, sum=0;
while (a<=10)
{ printf("%d", a);
  sum = sum+a;
  a++;
}
```

* while statement should not be terminated with a semicolon.
  - if it is terminated with a semicolon, it is treated as null statement.

* Execution:
  a) controlling expression is evaluated.
     i) if it evaluates to true, the body of the loop is executed.
     ii) if it evaluates to false, the program control is transferred to the statement present next to the while statement.
  b) After executing the body, the program control returns back to while statement.
  c) steps a & b are repeated until the condition evaluates to false.

* Initialize the loop counter before the while statement.
* manipute the loop counter is inside the body of the while statement.

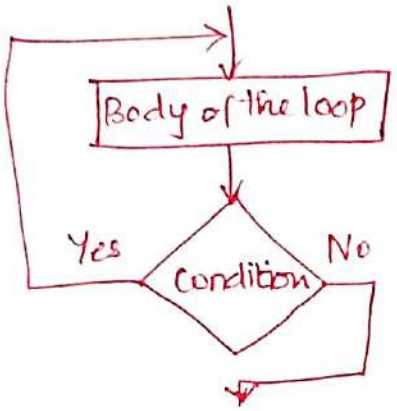
c) do-while statement:

General form:

```
do
statement
while(expression)
```

or

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



Ex:

```
do
{
  printf("Enter two numbers");
  scanf("%d%d", &a, &b);
  printf("sum=%d", a+b);
  printf("Do you want to continue
          say Y/N:");
  scanf("%c", &yes);
}while ((yes=='y') || (yes=='Y'));
```

* while is terminated with a semicolon.

* **Execution:**

a). The statement (body of the loop) is executed.

b) After the execution of body once, the condition is evaluated

i) if it evaluates to true, the body is executed again

ii) if it evaluates to false, the program control is transferred to the statement present next to the do-while statement.

* initialize the loop counter before do-while statement.

* manipulate (increment/decrement) is inside the body of the loop.

* The statement (body) is executed once, even when the do-while condition is initially false.

x —— x

1.10 PREPROCESSOR DIRECTIVES

Introduction: – Preprocessor

* Before C program is compiled, the source code is processed by a program called Preprocessor.
* The preprocessor directives starts with # symbol
  → do not require a semicolon at the end

Types of Preprocessor directives:

i) Macro substitution directives
ii) File inclusion directives
iii) Conditional compilation directives
iv) Miscellaneous directives.

i) Macro Substitution directives:

Macros:
→ a piece of code in a program which is given some name.

* Whenever the name is encountered by the compiler, the compiler replaces the name with the actual piece of code.
* A macro is defined by using:

```
#define
```

Types of macros:

a) Object-like Macros
b) Function-like Macros

a) Object-like Macros:

* An identifier that is replaced by value

```
#define x 25
```

b) Function-like Macros:

* like function call

```
#define MIN(a,b) ((a)<(b)?(a):(b))
```

ii) File Inclusion Directives:

* Tells the compiler to include a file in the source code program.

Two Types of Files:

a) Header File or Standard File
b) User defined files.

a) Header Files:

* Contains definition of pre-defined functions like printf(), scanf(), etc.

```
#include <filename>
```

Ex:

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

b) User defined files:

* Used to include some other files to your source program.


```
#include "filename"
```

Ex:

```
#include "hello.c"
```

iii) Conditional compilation directives:

* Directives which help to compile a specific portion of the program or to skip compilation of some specific part of the program based on some conditions

1) #ifdef
2) #if
3) #else
4) #elif
5) #endif
6) #ifndef

#if

* Evaluates the expression or condition

→ If condition is true, it executes the code

Syntax:

```
#if exp
    //code
#endif
```

#else

* Evaluates the expression or condition if condition of #if is false.

→ It can be used with #if, #elif, and ifndef directives.

Syntax with #if

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

Syntax with #elif

```
#if exp
    // if code
#elif exp
    // elif code
#else
    //else
#endif
```

#elif

* used to write #else and #if in one statement

#ifdef

* The macro with name as 'macro_name' is defined, then the block of statements will execute.

- If it is not defined, the compiler will simply skip the block of statements.

#endif → specifies the end of block.

Syntax:

```
#ifdef  macro_name
        Stmt 1;
        Stmt 2;
        ...
        Stmt N;
#endif
```

#ifndef

* checks if macro is not defined by #define.

→ If yes, it executes the code

Syntax:

```
#ifndef MACRO
    //code
#endif
```


iv) Miscellaneous directives:

#undef:

* To undefine a macro
  - to cancel its definition

```
#define PI 3.1415
#undef PI
```

#pragma

* To provide additional information to the compiler

1.11 COMPILATION AND LINKING PROCESSES

Executing a C Program:- steps

1. Creating the Program.
2. Compiling the program.
3. Linking the program with functions that are needed from the C library
4. Executing the program.

1. Creating the program:

* The program must be entered into a file.

Save program
↳ with **.c** extension.

ex: hello.c.

* File is created with the help of text editor.

Compiling and Linking:

2) Compilation:

* After the program is ready, it should be compiled.
ie) After the program is entered in C editor, the next is to compile the program.

Shortcut key : **Alt + F9.** or Compile option in compile Menu

* The process of converting the high level language program into machine understandable form.
- There is a possibility for errors.
- it will show errors.
ie) syntax errors.
↳ not in proper syntax.

* After the compilation, look for errors and warnings.
- warnings will not prevent the execution of the program.
- If there are errors, check the code properly.
* There should be no typing mistake
* If there is no error, go for linking the program.

3) Linking:

* process of putting together other program files and functions that are required by the program
* C language program is the collection of predefined functions.
  ↳ written in Standard 'C' header files
* Before executing a 'C' program, need to link with system library.
* can be done automatically at the time of execution

eg: for exp() function
math library.

4) Executing the Program:

* Process of running and testing the program

2 types of errors
- Logical errors
- Data errors

Shortcut key → CTRL + F9 or choose Run option from Run Menu

Steps:

1. Enter the program in a C editor
2. Save the program.
   File → Save (or) F2
   Use the extension .c
3. Compile the program (Compile → Compile) (or) Alt + F9.
4. Run the Program (Run → Run) (or) Ctrl + F9.

UNIT - II

ARRAYS AND STRINGS

Introduction to Arrays : Declaration, Initialization - One dimensional array - Two dimensional arrays - String operations : length, compare, concatenate, copy - Selection sort, linear and Binary search

UNIT – II

ARRAYS AND STRINGS

2.1 INTRODUCTION TO ARRAYS

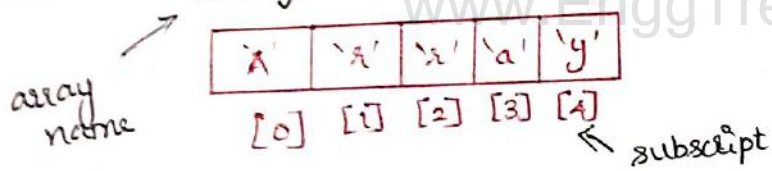
Definition:

* An array is a data structure that is used for the storage of homogeneous data ie) data of the same type.

* **Different types:**

a) **character array**

array1



b) **Integer array**

array2

1	5	8	12	7	18	11

c) **float array**

array3

1.2	5.1	12.9	10.0

d) **array of user-defined type.**

array4

'A'	1	'x'	2	'x'	3	'a'	4

* An array is a collection of similar data items, that are stored under a common name.
  - A value in an array is identified by index or subscript enclosed in square brackets with array name.
  - Each array element is referred by specifying the array name with subscript. ← position of an element eg: x[5] is sixth elt.
* Data type of an element is called element type.
* The array index starts with 0.
  ie) index of the first element of an array is 0.

* memory space required by an array can be compuled as (Size of element type) x (Number of elements in an array)

eg: char → size → 1

array1: 1 x 5 = 5 bytes

int – size – 2

array 2: 2 x 8 = 16 bytes.

* Arrays are stored in contiguous memory locations

ex: array1:

char 2000, 2001, 2002, 2003, 2004.

int: array2:

2000-2001, 2002-2003, 2004-2005 and so on.

* Needs of an array:
  - → to define a set of similar data items.
  - → faster access than dynamic memory allocation
* Classification of arrays:
  - i) One-dimensional (single dimensional) arrays
  - ii) Two-dimensional arrays
  - iii) Multi-dimensional arrays.

2.2 ONE DIMENSIONAL ARRAYS

* The collection of data items can be stored under a one variable name using only one subscript.
* The elements of an array can be accessed by using single subscript.

– type of linear array.

Declaration :

* consists of a type specifier, an identifier and a size specifier enclosed within square brackets

General form:

data_type array_variable[size specifier];

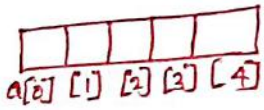
Ex:

int a[5]; ← type of data, name of an array, size of the array. → a is the array name which can hold 5 values of type integer

char name[0];

float avg[10];

int a[5];



* size specifier specifies the number of elements in an array.
- Should be a compile time constant expression of integral type
- The memory space is allocated at the compile time
- should be greater than or equal to one.

Initialization:

* The values can be initialized to an array
* An initializer is an expression that determines the initial value of an element of the array.

Two ways

i) At compile time

ii) At run time.

i) At compile time:

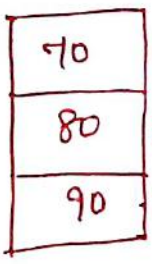
Syntax:

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

Ex: - integer array:

int marks[3] = {70,80,90};

→ List of values must be separated by commas.



ii)
marks[0] = 70;
marks[1] = 80;
marks[2] = 90;

Character array:

Ex:
```
char name[] = {'L', 'A', 'K'}
```

ii) At run time:

* The array can be explicitly initialized at run time.

Ex:

①
```
while (i<=10)
{
  if(i<5)
    sum[i]=0;
  else
    sum[i] = sum[i]+i;
}
```

②
```
int a[2];
scanf("%d%d", &a[i], &a[2]);
```


* The number of initializers should be less than or at most equal to the value of size specifier
* If the number of initializers is less than the value of the size specifier, the leading array locations equal to the number of initializers get initialized with the values of initializers. The rest of the array locations get initialized to 0 (int) or 0.0 (float) or '\0' (char).

a) Basic Initialization

```
int num[5]={3,7,12,24,25};
```

3	7	12	24	25

b) Initialization without size

```
int num[]={3,7,12,21,15};
```

3	7	12	21	15

c) Partial Initialization

```
int num[5]={3,7};
```

3	7	0	0	0

rest are filled with 0's

d) Initialization to all zeros

```
int num[5]={0};
```

0	0	0	0	0

All are filled with 0's

Operations on a Single Dimensional Array

1.Subscripting a Single Dimensional Array

The only operation allowed in an array is subscripting. Subscripting is an operation that selects an element from an array.

Eg: a[3] -> denotes the 3rd element.

2. Assigning an array to another array

A normal variable can be assigned to or initialized with another variable but an array cannot be assigned to or initialized with another array.

Eg: int a[10],b[10];

a=b; // error

Because the name of the array always refer to the address of first element of array and it is a constant.

We can do this by assigning individual elements in the array to an element in another array. Eg: a[3]=b[3]

Equating an array to another array

When we compare arrays like normal variables, result will always be false. because the name of the array always refer to the address of first element of array.

Eg: int a[10],b[10];

if(a==b) // is not possible in array

We can do this by comparing the individual elements in the arrays.Eg: if(a[2]==b[2])

One – Dimensional Array

Example Program:

//Reading,Storing and accessing elements of one dimensional array

```
#include<stdio.h>
#include<conio.h>
main()
{
        int a[10],i,n;
        printf("\nEnter the number of elements:");
        scanf("%d",&n);
        printf("\nEnter the elements of an array:");
        for(i=0;i<n;i++)
scanf("%d",&a[i]);
        printf("\nArray Elements are:");
        for(i=0;i<n;i++)
        printf("%d",a[i]);
        getch();
}
```

* To find sum of 10 non-neg. numbers entered by the user.

2.3 TWO DIMENSIONAL ARRAYS

* A two-dimensional array has its elements arranged in a rectangular grid of rows and columns.
  - The elements of a two-dimensional array can be accessed by using a row subscript and a column subscript.
  - Both the row subscript and the column subscript are required to select an element.
  - Known as a matrix.

A 2-D array

Columns →

Rows ↓

2	1	2	3	1	5	6
1	6	8	1	5	1	0
2	7	2	1	8	0	1
6	3	1	1	8	3	0

* Two dimensional arrays are used in situation where a table of values need to be stored in an array.

* **DECLARATION:**
  - **Syntax:** → should have two subscripts. (row, column)

```
data_type array_name[row_size][column_size];
```

- data_type → type of the data
- array_name → name of the array
- row_size → size of the row
- column_size → size of the column

Ex:

```
int a[3][3];
```

	col 0	col 1	col 2
row 0	a[0,0]	a[0,1]	a[0,2]
row 1	a[1,0]	a[1,1]	a[1,2]
row 2	a[2,0]	a[2,1]	a[2,2]

Memory Representation:

	col 0	col 1	col 2
row 0	15	21	64
row 1	90	72	51
row 2	34	42	80

* **INITIALIZATION:**
  * The values can be initialized to the two dimensional arrays at the time of declaration

Syntax:

data_type array_name[rowsize][columnsize] = {List of values};

List of elements

Ex: int a[2][4] = {{2,1,5,6},{8,7,1,9}};

int stud[4][2] = {{6680,80},{6681,81},{6682,82},{6683,83}};

or

int stud[4][2] = {6680,80,6681,81,6682,82,6683,83};

or

int stud[][2] = {6680,80,6681,81,6682,82,6683,83};

* column size should be mentioned. rowsize is optional. Then only the compiler knows where the first row ends.

① int a[4][7] = {{2,1},{2,3,1},{5},{6,1,6,8}};

columns →

row ↓							
2	1	0	0	0	0	0	
2	3	4	0	0	0	0	
5	0	0	0	0	0	0	
6	1	6	8	0	0	0	

② int a[4][7] = {{2,1},{2,3,4}};

Column →

row ↓							
2	1	0	0	0	0	0	
2	3	4	0	0	0	0	
0	0	0	0	0	0	0	
0	0	0	0	0	0	0	

MULTI DIMENSIONAL ARRAYS:

* An array with three or more dimensions.

declaration:

Syntax:

datatype arrayname [size1][size2][size3]...[sizen];

Ex: int a[3][3][3];

Two – Dimensional Array

Example Program -1:

```
#include<stdio.h>
#include<conio.h>
main()
{
    int a[10][10],i,j,r,c;
    printf("\nEnter the row size and column size:");
    scanf("%d%d",&r,&c);
    printf("\nEnter the elements of array:");
    for(i=0;i<r;i++)
        for(j=0;j<c;j++)
            scanf("%d",&a[i][j]);
    printf("\nArray elements are:");
    for(i=0;i<r;i++)
        for(j=0;j<c;j++)
            printf("%d",a[i][j]);
    getch();
}
```

Example Program -2:

Matrix Addition:

```
#include<stdio.h>
#include<conio.h>
main()
{
    int a[3][3],b[3][3],c[3][3],i,j;
    printf("Enter the First matrix->\n");
    for(i=0;i<3;i++)
        for(j=0;j<3;j++)
        {
            scanf("%d",&a[i][j]);
        }
    printf("\nEnter the Second matrix->\n");
    for(i=0;i<3;i++)
        for(j=0;j<3;j++)
            scanf("%d",&b[i][j]);
    for(i=0;i<3;i++)
        for(j=0;j<3;j++)
            c[i][j]=a[i][j]+b[i][j];
    printf("\nThe Addition of two matrix is\n");
    for(i=0;i<3;i++)
    {
        printf("\n");
        for(j=0;j<3;j++)
            printf("%d\t",c[i][j]);
    }
    getch();
}
```

2.4 STRING OPERATIONS

2.4.1 STRING :

* String is a sequence of characters enclosed within double quotes " "
  - character is enclosed within single quote ' '

EX :

String : "A"
character : 'A'

* Every string is automatically terminated by a null character.
* C string library provides the following functions as predefined functions
  - reading
  - copying
  - comparing
  - combining
  - searching etc.

Memory space :

* Strings are stored in contiguous memory locations with terminating null character.

→ The amount of memory space required depends upon the number of characters present in the string.

- The number of bytes required is one more than the number of characters present in it.

EX: "ABC"

→ requires 4 bytes
3 bytes - string
1 byte - null character

Length of the string:

- number of characters present

Empty String:

- A string with length zero.
- written as ""
  → no characters is enclosed within double quotes.

Data Type:

* In C language, string data type is not available
  → character array is used to represent strings.

Character Array:

* Character Array is used to store a string.
  - stores elements of type "char".

Syntax: - Declaration:

```
char identifier[size] = initialization_list ;
                 ↓
          no of characters
          (length + 1)
```

→ size is optional, if initialization_list is present

a) String Declaration:

Syntax:

```
datatype str_name [size] ;
```

Ex:

```
char var[5];
```

* The variable can hold 4 characters
  - 5th space is for '\0' (null character)

b) String Initialization

* Assigning the values.

strings can be initialized in three ways:

i) By using string literal constant

```
char str[6] = "Hello";
```

ii) By using list of character initializers

```
char str[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

'\0' → end of a string.

iii) size is not mentioned

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

- size will be automatically allocated based on the number of characters.

c) Reading a string:

* Reading a single character or series of characters (string) from the input device.

Three ways:

i) scanf()
ii) getchar()
iii) gets()

i) scanf():

* formatted input function
* format string %s
* Automatically taminates the string that is read with a null character.

Ex:

```
char name[6];
scanf("%s", name);
```

ii) getchar():

* to read any alphanumeric character from the input devices.

- reads one character at a time
- unformatted function.

Syntax:

```
char var_name;
var_name = getchar();
```

(iii) get():

* string input function (unformatted)
* reads a group of characters until an enter key is pressed.

Syntax:

```
char var_name;
gets(var_name);
```


d) Printing a string:

* Printing → displaying or writing a single character or a string to the o/p devices.

Three ways:

i) printf()
ii) putchar()
iii) puts()

(i) printf():

* formatted output function to write a word or a single character to the o/p devices.
* format string %s

Syntax:

```
char var[size];
printf("%s", var);
```

EX:

```
char name[6];
scanf("%s", name);
printf("%s", name);
```

ii) putchar() :

* unformatted output function to write any alphanumeric character to the output device.
* prints one character at a time

Syntax:

```
char var;
putchar(var);
```

iii) puts() :

* unformatted output function to write any alphanumeric character to the o/p device.
* stdio.h header file is needed

Syntax:

```
char var[n];
puts (var);
```


STRING OPERATIONS

i) Length
ii) Compare
iii) Concatenate
iv) Copy

String Library Functions / String Manipulation Functions / String Standard Functions:

1.	strlen (s)	strlen returns the length of a string
2.	strcpy ()	strcpy copies one string to another
3.	strcat()	strcat combines two strings
4.	strcmp ()	compares two strings and returns an integer indicating the difference between the strings. if the strings match, then the number returned is 0.
5.	strrev()	used to reverse a string
6.	strlwr()	used to convert string to lower case
7.	strupr()	used to convert string to uppercase
8.	strset()	sets all characters in a string to a specific character
9.	strchr()	determines the first occurrence of a given character in a string
10.	strstr()	finds the first occurrence of a string in another string.
11.	strdup()	used to duplicate a string.

① Length:

* strlen():

→ used to find the length of a string.

- The function returns the length of the string as output
- The terminating null character will not be counted.

Syntax:

```
var = strlen(string);
```

Ex:
```
s1 = "Hello";
length = strlen(s1);
printf("%d", length);
```

O/P
5

---

② Compare:

* Comparing two strings.

The functions:

i) strcmp()
ii) strcmpi()
iii) strncmp()
iv) strncmpi()

i) strcmp():

→ used to compare two strings

* The function performs the comparison of two strings character by character until the corresponding characters differ or the end of the string is reached.

- returns the ASCII difference of the first mismatch characters or zero if both are same.

Syntax:

```
strcmp(string1, string2);
```

Ex:

```
char s1[5] = "Hello";
char s2[5] = "Good";
n = strcmp(s1, s2);
printf("%d", n);
```

O/P:
-1

ii) strcmpi():

* Compare two strings without case sensitivity
* i → ignore case

Syntax:

```
strcmpi(str1, str2);
```

Ex:

```
char s1[5] = "Good";
char s2[5] = "GooD";
n = strcmpi(s1, s2);
printf("%d", n);
```

O/P:
0

iii) strncmp():

* To compare a portion of two strings
* n → number of characters to be compared.

Syntax:

```
strncmp(string1, string2, n);
```

Ex:

```
char s1[20] = "Welcome";
char s2[20] = "Welcome You All";
n = strncmp(s1, s2, 3);
printf("%d", n);
```

O/P:
0

iv) strncmpi():

* to compare a portion of two strings without case sensitivity.
  - i → ignore case
  - n → no. of characters.

Syntax:

```
strncmpi(str1, str2, n);
```

---

③ Concatenate:

* To concatenate one string with another string
  - append a source string to the destination string.

Functions:

i) strcat()

ii) strncat()

i) strcat():

* concatenate one string with another string.

Syntax:

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

string1 → source, string2 → destination.

* The function appends a source string to the destination string
  - returns a pointer to the destination string as output.

Ex:

```
char s1[20] = "Good";
char s2[20] = "Morning";
strcat(s1, s2);
puts(s1);
```

s1, s2

O/P:

Good Morning

ii) strncat:

* concatenates a portion of one string with another string.

- Append atmost n characters of a source string to the destination string.

n → number of characters of the source string to be copied.

Syntax:

```
strncat(d, s, n);
```

Ex:

```
char s1[20] = "Good";
char s2[20] = "To all";
strncat(s1, s2, 2);
puts(s1);
```

O/P:

Good To

---

④ copy:

* To copy the source string to the destination string.

Functions:

i) strcpy()

ii) strncpy()

i) strcpy():

* copies the source string to the destination string

- returns a pointer to the destination string as output.

Syntax:

```
strcpy(dest, source);
       s1    s2
```

Ex:

```
char s1[20] = "welcome";
char s2[20];
strcpy(s2, s1);
puts(s2);
```

o/p:

welcome

ii) strncpy():

* to copy atmost n characters of a source string to the destination string.

Syntax:

```
strncpy(str1, str2, n);
```

Ex:

```
char s[20] = "welcome";
char d[20];
strncpy(d, s, 3);
d[3] = '\0';
puts(d)
```

o/p:

wel

Other Functions:

⑤ strrev()

→ to reverse all the characters of a string except the terminating null character

* reverses the string and returns a pointer to the reversed string as output

Syntax:

```
strrev(string);
```

Ex:

```
char s[20] = "Hello";
strrev(s);
puts(s);
```

o/p:

olleH

⑥ strlwr():

* Converts all the characters in a string to lowercase
- returns a pointer to the converted string as o/p

Syntax:

```
strlwr(str);
```

Ex:

```
char s[20] = "WELCOme";
strlwr(s);
puts(s);
```

o/p:
welcome

⑦ strupr():

* converts all the characters in a string to uppercase
- returns a pointer to the converted string as o/p.

Syntax:

```
strupr(str);
```


Ex:

```
char s[20] = "Welcome";
strupr(s);
puts(s);
```

o/p:
WELCOME

⑧ a) strset():

* sets all characters in a string to a specific character.
* input is string and a character.

Syntax:

```
strset(str, ch);
```

Ex:

```
char s[5] = "Good";
strset(s, 'H');
put(s);
```

o/p:
HHHH

b) strnset()

* sets first n characters in a string to a specific character.

- input is a string, a character and an integer value n.

Syntax:

```
strnset(string, ch, n);
```

Ex:
```
char s[10] = "Welcome";
strnset(s, 'H', 3);
puts(s)
```

o/p:
HHHcome

⑨ a) strchr()

* Searches a string for the first occurrence of a given character.

→ If a character is found, it returns a pointer to the first occurrence of the character in the given string

→ If a character is not found, it returns NULL.

Syntax:

```
strchr(string, ch);
```

b) strrchr()

* Searches a string for the last occurrence of a given character.

Syntax:

```
strrchr(string, ch);
```

Ex:
```
char s[20] = "welcome";
char *ptr;
ptr = strchr(s, 'e');
printf("%d", ptr); // first occurrence
ptr = strrchr(s, 'e');
printf("%d", ptr); // last occurrence
```

o/p:
1
6

(10) strstr() :

* finds the first occurrence of a string in another string.

- If a string s2 is found, it returns the position from where the string starts.
- If a string s2 is not found, in string s1, it returns NULL.

Syntax:

```
strstr(s1, s2);
```

Ex:

```
char s1[20] = "Welcome";
char s2[20] = "come";
char *ptr;
ptr = strstr(s1, s2);
printf("Found at %d", ptr);
```


o/p:

Found at 3

STRING ARRAYS

* A list of strings can be stored in two ways:
  a) Using an array of strings
  b) Using an array of character Pointers.

a) Array of Strings:

* list of strings can be stored by two dimensional character array.

Declaration:

```
char id[row][coln] = list;
```

EX:

```
char array[2][30];
```

Initialization:

Two ways:

i) **Using string literal constant**

```
char str[][20] = {"CIVIL", "CSE", "ECE"};
```


ii) **Using list of character initializers**

```
char str[][20] = { {'C','I','V','I','L','\0'},
                   {'C','S','E','\0'},
                   {'E','C','E','\0'} };
```

b) Array of character pointers:

* Array of strings can be stored by using an array of character pointers.

EX:

```
char *lang[20] = {"CIVIL", "CSE", "ECE"};
```

2.5 SELECTION SORT

* A sorting algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

Steps:

1. Set the first elt as min.
2. compare the remaining elts and select the smallest elt.
3. If that elt is smaller than min, then swap the two elts.
4. Again reassign the min elt from the second elt.
   - Compare the remaining elts
   - swap if min > elt.
5. The process is continued until the last elt.
6. Finally, the array will be sorted.

Ex:

arr

0	1	2	3	4
20	12	10	15	2

step 1:

* min ← arr[0] = 20
* compare the remaining elts. and select the smallest element.

12 > 10
10 < 15
10 > 2

→ select 2

* swap with min

arr

0	1	2	3	4
2	12	10	15	20

Step 2:

Now, min ← arr[1] = 12,

* compare the remaining elts & select the smallest elt.

(10) < 15

(10) < 20

— select 10

* Compare with min i.e) 12, 12 > 10.

so, swap.

0	1	2	3	4
2	10	12	15	20

Step 3

* min ← arr[2] = 12
* compare. No, smallest elt.

Step 4: min ← arr[3] = 15, No smallest elt.

Step 5: min ← arr[4] = 20, No remaining elts.

Sorted Array:

arr

0	1	2	3	4
2	10	12	15	20

Program:

```
#include<stdio.h>
main()
{
  int a[100], n, i, j, pos, swap;
  printf("Selection Sort\n");
  printf("Enter the number of elements : ");
  scanf("%d", &n);
  printf("Enter the elements : ");
  for(i=0; i<n; i++)
     scanf("%d", &a[i]);
  for(i=0; i<n-1; i++)
  {
     pos=i;
     for(j=i+1; j<n; j++)
     {
        if(a[pos] > a[j])
           pos=j;
     }
     if(pos!=i)
     {
        swap=a[i];
        a[i]=a[pos];
        a[pos]=swap;
     }
  }
  printf("Sorted Array:\n");
  for(i=0; i<n; i++)
     printf("%d\t", a[i]);
}
```


o/p:

```
Selection Sort
Enter the number of elements : 5
Enter the elements: 20 12 10 15 2
Sorted Array
2  10  12  15  20
```

2.6 SEARCHING

* Search is an operation in which a given list is searched for a particular value.
- The location of the searched element is informed
* activity of looking for a value or item in a list

Types of Search

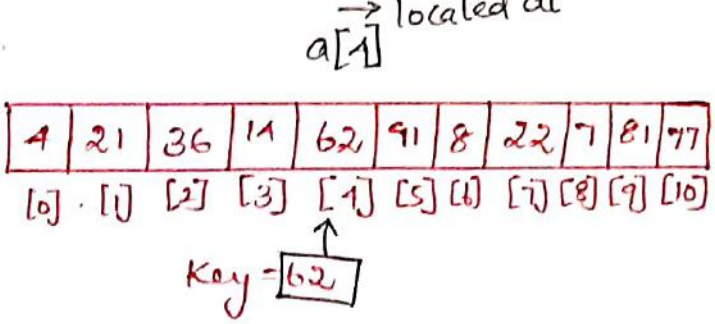
i) Linear Search
ii) Binary Search

2.6.1 (i) Linear Search: or Sequential Search:

* The search starts from the first element and continues in a sequential fashion from element to element till the desired entry is found
- Simplest search algorithm

operation: Steps:

* Traverse the array in sequence from the first element to the last.
* Each element is compared to the key.
- If the key is found in the array, the corresponding array index is returned.
- If the item is not found in the array, an invalid index, -1 is returned.
* In the worst case, the number of comparisons is proportional to the size of the array.



Program:

```
#include<stdio.h>
int main()
{
  int a[10],i,n,m,c=0;
  printf("Enter the size of an array");
 scanf("%d",&n);
printf("Enter the elements of the array");
for(i=0;i<=n-1;i++)
{
   scanf("%d",&a[i]);
}
printf("Enter the number to be search");
scanf("%d",&m);
     for(i=0;i<=n-1;i++)
     {
        if(a[i]==m)
        {
           c=1;
           break;
        }
     }
 if(c==0)
     printf("The number is not in the list");
else
    printf("The number is found");
getch();
return 0;
}
```

output:

```
Enter the size of an array : 5
Enter the elements of the array : 1 6 8 0 3
```

Enter the number to be search : 0

The number is found

Example:

0	1	2	3	4
4	6	8	0	3

Key is 0

m=0

Step 1: check i) (a[0] = 4) == (m = 0) ii) 1 ≠ 0

Step 2: check i) (a[1] = 1) == 0 ii) 1 ≠ 0

Step 3: check i) (a[2] = 8) == 0 ii) 8 ≠ 0

Step 4: check i) (a[3] = 0) == 0 ii) 0 = 0

* So set c = 1.
* Then, the number is found.

Disadvantages:

* slow
* inefficient

ex: i) 3, 2, 1, 4, 5
Key - 4

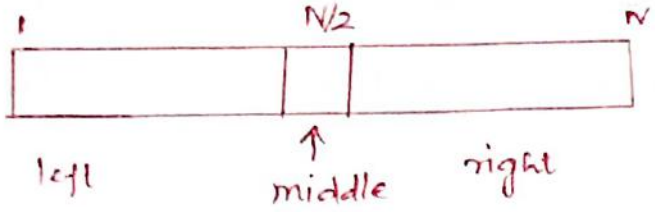
2) 1, 21, 86, 14, 63, 91, 8, 22, 7, 81, 77, 10

2.6.2

(ii) Binary Search:

* Binary search is a divide and conquer search algorithm to find out the position of a specified value within an array.
* The array must be sorted in either ascending or descending order.
* The binary search requires arrays to be sorted.

→ The list is divided into two halves separated by the middle element.

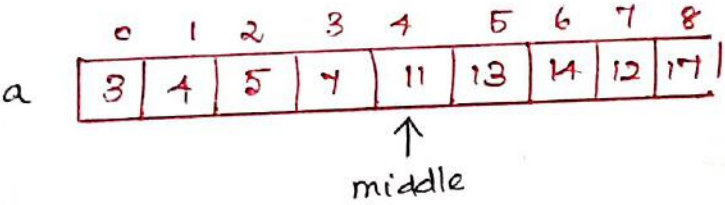


Operation:

Steps:

1. The middle element is tested for the required entry. If found, then its position is reported, else the following test is made.
2. If Key < middle, search the left half of the list, else search the right half of the list.
3. Repeat step 1 and step 2 on the selected half until the entry is found, otherwise report failure.

* In each iteration, the given list is divided into two parts.
- The search becomes limited to half the size of the list



middle value:

→ averaging the first and last indices and truncating the result.

ex:

$$\frac{0+8}{2} = \frac{8}{2} = 4.$$

ie) the content of the fourth location.

Ex: Key = 14.

Step 1:

$$\frac{0+8}{2} = 4 \qquad mid = a[4] = 11 \neq 14.$$

→ $a[4] \neq 14$

→ $14 > 11$

- The search element is present in the right half.

Step 2:

$\frac{5+8}{2} = \frac{13}{2} = 6$

a[6] = 14

Key = 14 = a[6]

- The element is found.

Ex: 1) 1, 3, 4, 5, 6
cll - 3

2) 1, 7, 8, 10, 14, 21, 22, 36, 62, 77, 81, 91
Key - 22

Program:

```
#include <stdio.h>
main()
{
  int a[10], i, n, m, c=0, l, u, mid;
  printf("Enter the size of an array");
  scanf("%d", &n);
  printf("Enter the elements in ascending order");
  for(i=0; i<n; i++)
  {
    scanf("%d", &a[i]);
  }
  printf("Enter the number to be search:");
  scanf("%d", &m);
  l=0, u=n-1;
  while(l<=u)
  {
    mid = (l+u)/2;
    if(m==a[mid])
    {
      c=1;
      break;
    }
    else if(m<a[mid])
    {
```

```
            u = mid;
        }
        else
            l = mid+1;
        }
        if (c==0)
            printf("The number is not found");
        else
            printf("The number is found");
        getch();
        return 0;
}
```

Output:

```
Enter the size of an array: 5
Enter the elements in ascending order : 4 7 8 11 21
Enter the number to be search : 11
The number is found.
```


characteristics

* list must be sorted
* faster than linear search.

UNIT – III

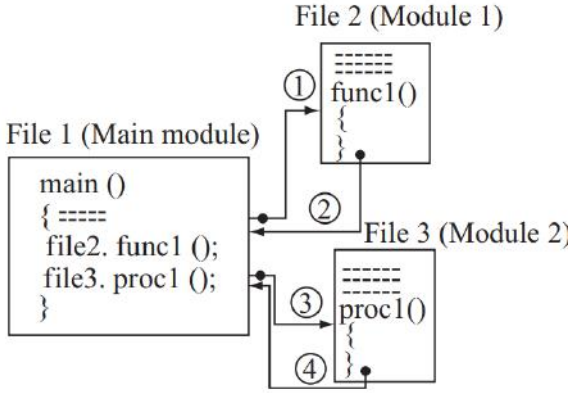
FUNCTIONS AND POINTERS

Modular Programming – Function prototype, function definition, function call, Built-in functions (string functions, math functions) – Recursion, Binary Search using recursive functions – Pointers – Pointer operators – Pointer arithmetic – Arrays and Pointers – Array of pointers – Parameters passing: Pass by value, Pass by reference

Modular Programming:

* Breaking down of a program into a group of files
  - where each file consists of a program that can be executed independently.

→ The problem is divided into different independent but related tasks.



* For each identified task,
  - a separate program (module) is written, which is a program file that can be executed independently.
* The different files of the program are integrated using a main program file.
* The main program file invokes the other files in an order that fulfills the functionality of the program.

6. Structured Programming:

* The approach to develop the software is process-centric or procedural.

→ The procedures and modules become tightly interwoven and interdependent
  - They are not re-usable

7. Examples: C, COBOL, Pascal.

3.1 FUNCTION

* A function is a self-contained program, or a subprogram of one or more statements which is used to do some particular task.
- set of instructions to perform a specific task.

Classification

Two types

i) Pre-defined Functions (Library Functions)
ii) User-defined Functions.

3.1.4.

① Pre-defined Functions / Library Functions / Built-in Functions:

* functions whose functionality has already been developed by someone and are available to the user for use.

ex: printf, scanf, sqrt(x,y); , strcpy(); strcmp(); etc.

Two aspects:

i) Declaration of library functions
ii) use of library functions.

i) Declaration of Library functions:

* A library function needs to be declared before it is called.
- available in header files.
- header files are included to access the functions.

ex:

* printf is available in stdio.h
  - so stdio.h is included before calling the printf function

Syntax:

→ To include header file

```
#include<filename.h>
```

Library Functions:

Built-in Functions

	Header file	Function
1)	stdio.h	i) getchar() ii) putchar() iii) printf() iv) scanf()
2)	string.h	i) strcat() ii) strcmp() iii) strcpy()
3)	math.h	i) acos() ii) asin() iii) atan() iv) cos() v) exp() vi) sqrt()
4)	stdlib.h	i) malloc() ii) rand()
5)	ctype.h	i) isdigit() ii) islower() iii) isupper()

String Functions

Math Functions.

ii) Use of Library Functions:

* using a function call operator .(())

ex:

strcpy();

Example:

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
void main()
{
 int a,b;
 printf("Enter the number");
 scanf("%d", &a);
 b=sqrt(a);
 printf("The square is :%d", b);
 getch();
}
```

② User-defined Functions: / Programmer-defined functions.

* Functions that are defined by the user at the time of writing a program.

- The user develops the functionality by writing the body of the function.

* Functions are used to break down a large program into a number of smaller functions.

Merits:

* Easy to locate and debug an error
* length of the program can be reduced.
* avoid coding of repeated programming.

Three aspects:

i) Function declaration / Function prototype.
ii) Function definition
iii) Function use. (function call / function invocation)

* FUNCTION DECLARATION:

* All the functions need to be declared or defined before they are used.

General form:

```
return_type function_name(parameter_list or
                          parameter_type_list);
```

Ex:

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

Rules:

* The parameter list must be separated by commas.
* The parameter names do not need to be the same in the prototype declaration & function definition.
* The types must match the types of parameters in the function definition in number & order.

* FUNCTION DEFINITION: 3.1.2

- Composing a function

* It is the process of specifying and establishing the user defined function by specifying all of its elements and characteristics.

Two parts:

i) Header of the function
ii) body of the function.

Header of the function:

General form:

return_type function_name(parameter_list)

Body of the function

- consists of a set of statements enclosed within braces.

Syntax:

```
return_type function_name(parameter_list)
  parameter declaration
{
  local variable declaration;
  :
  body of the function;
  :
  return (expression);
}
```

Example:

```
int add(int x, int y)
{
  int z;
  z=x+y;
  return(z);
}
```


* FUNCTION USE / FUNCTION CALL: 3.1.3

* The function can be called by simple specifying the name of the function, return value and parameters if present.

Syntax:

i) function_name();

ii) function_name(parameters);

iii) return_value = function_name(parameters);

Ex:

i) add();

ii) add(a,b);

iii) c = add(a,b);

Program:

```
#include<stdio.h>
#include<conio.h>
print_message();          // Function declaration
main()
{
  print_message();        // Function call
}
print_message()           // Function definition.
{
  printf("Hello");
getch();
return;
}
```

* ACCESSING A FUNCTION:

* A function can be accessed by specifying its name, followed by a list of arguments enclosed in parenthesis and separated by commas.

→ There may be several different calls to the same function from various places within a program.

*) Parameters or Arguments:

→ Parameters provide the data communication between the calling function and called function.

Two Types:

i) Actual Parameters.

ii) Formal Parameters.

i) Actual Parameters:

- Parameters transferred from calling function to called function.

ii) Formal Parameters:

- Parameters used in the called function for the values that are passed from the called function.

Example

```
main()
{
  add(x,y);
  ...
}
add(a,b)
{
  ...
  return();
}
```

calling function.

called function

// function call
x, y – actual parameters

// Function definition
a, b – formal parameters

* return statement:

→ used to return the result of the computations performed in the called function and/or to transfer the program control back to the calling function.

Two forms:

i) return; → just transfer the control to calling function
ii) return (expression); → transfers the control, returns a value to the calling function.

Ex:

i) if (x<=1)
   return(1);

ii) return(x);

iii) return (a+b*c);

* FUNCTION PROTOTYPES: 3.1.1

*Functions are classified into 4 types based on return values and arguments.

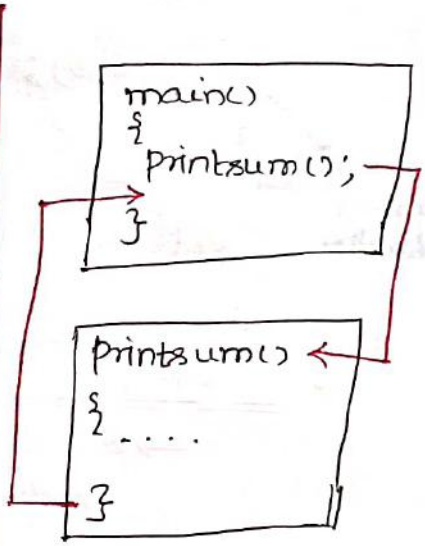
i) Function without argument and No return value
ii) Function with arguments and no return values.
iii) Function with arguments and with return values.
iv) Function without arguments and with return values

i) Function without argument and no return values:

* does not accept any input and does not return any result.
- parameter list is empty.

Ex:

```
#include<stdio.h>
void printsum(); // fn. decln.
main()
{
printsum();          // fn call
}
printsum()           // fun defn.
{
printf("sum of 2 & 3 is %d", 2+3);
}
```



Output:
Sum of 2 & 3 is 5

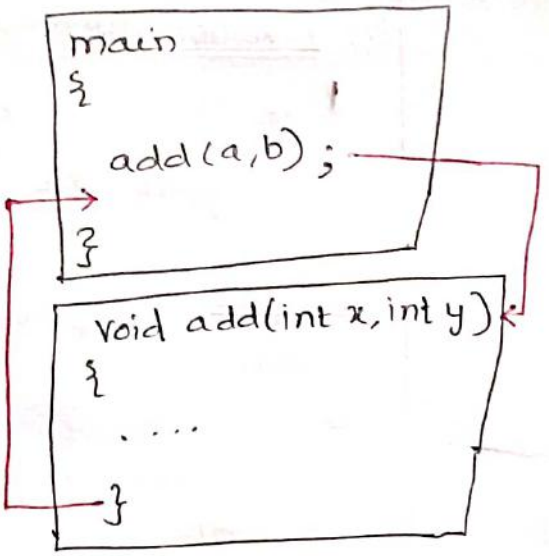
ii) Function with argument and no return values:

* A function has arguments. It receives data from the calling function.
* The calling function reads the data from input terminal and pass it to the called function.
- The program control is transferred to called function
* The execution of calling function is suspended and the called function starts the execution.
* When the execution of the called function is complete, the progeam control returns to the calling function, and the calling function resumes its execution.

Ex:

```
#include <stdio.h>
main()
{
 void add(int,int);
```

```
int a,b;
printf("Enter a & b");
scanf("%d %d", &a, &b);
add(a,b);
}
    void add(int x, int y)
    {
      int z;
      z=x+y;
      printf("Sum is %d", z);
      getch();
    }
```



Output:

```
Enter a & b
2 3
Sum is 5
```

iii) Function with Arguments and Return values:

* Data is transferred from calling function to called function

ie) The called function receives data from calling function and send back a value return to calling function

Ex:

```
#include<stdio.h>
main()
{
 int add(int,int);
 int a,b,c;
 printf("Enter 2 values");
 scanf("%d %d", &a, &b);
 c=add(a,b);
 printf("sum is %d", c);
}
    int add(int x, int y)
    {
      int z;
      z=x+y;
      return(z);
    }
```

```
main()
{
....
c=add(a,b);
....
}
in add(int x, int y)
{
...
...
return(z);
}
```

Output:

```
Enter 2 values
4
```

iv) Function without arguments and return values:

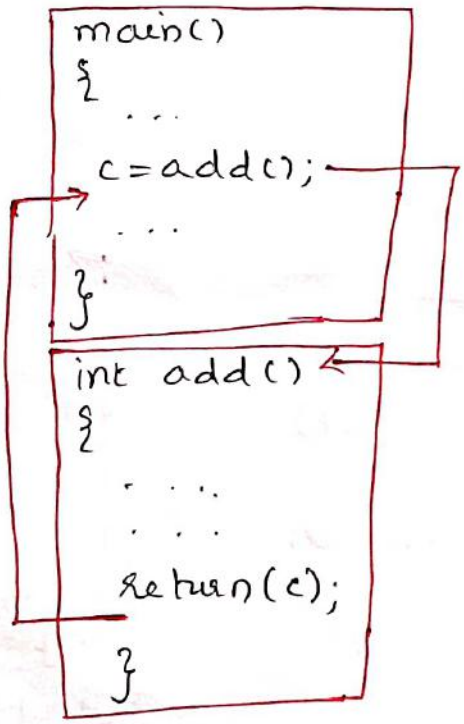
* Function has no arguments but values are returned from the called function.

Syntax:

```
main()
{
  ...
  c = fn();
  ...
}
fn()
{
  ...
  ...
  return(c);
}
```

Program:

```
#include<stdio.h>
main()
{
  int add();
  c=add();
  printf("Result is %d",c);
}
  int add()
  {
  int a,b,c ;
  printf("Enter a & b");
  scanf("%d %d",&a,&b);
  c=a+b;
  return(c);
  }
```

Output:

```
Enter a & b
5 3
Result is 8
```

PASSING ARRAYS TO FUNCTION

* Arrays can also be arguments of functions

→ When an array is passed to a function, the address of the array is passed and not the copy of the complete array.

* When a function is called with the name of the array as the argument, the address to the first element in the array is handed over to the function.

– When an array is a function argument, only the address of the array is passed to the function called.

* Modify the contents of the array.

Syntax:

```
data_type fn_name (datatype*);
main()
{
 ....
  fn_name (arr_name);
 ...
}
data_type fn_name (datatype* arr_name)
{
  ....
}
```

Ex:

```
#include <stdio.h>
main()
{
  int n, m, a[100], i;
  int max (int*, int);
  printf(" Enter the number of elements : ");
  scanf(" %d", &n);
  printf(" Enter the elements: ");
  for( i =0; i<n; i++)
  {
    scanf(" %d", &a[i]);
  }
  m = max(a,n);
  printf(" The maximum element is %d", m);
}
int max( int* arr, int num)
{
  int max_value, j;
    max_value = arr[0];
    for(j =1; j <num; j++)
    {
      if (arr[j] > max_value)
          max_value =arr[j] ;
    }
    return max_value;
}
```


Output:

```
Enter the Number of Elements in an array : 3
Enter the array elements
3
1
2
Maximum element is 3
```

3.2 RECURSION

* A recursive function is one that calls itself either directly or indirectly through another function.

→ Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied.

– used for repetitive computations in which each action is stated in terms of a previous result.

Syntax:

```
datatype fn_name()
{
    fn_name();
}
```


Example: calculating factorials.

```
long int fact(int );
main()
{ int n;
long int f;
  printf("Enter the value of n:");
  scanf("%d", &n);
  f = fact(n);
  printf("Factorial of %d is %d", n, f);
}
long int fact(int i)
{ long int f1=1;
  if(i<=1)
    return(1);
  else f1 = i * fact(i-1);
  return(f1);
}
```

o/p:
Enter the value of n: 3
Factorial of 3 is 6

Classification of Recursion:

Recursion is classified according to

1. Whether the function calls itself directly or indirectly
   a) Direct Recursion
   b) Indirect Recursion
2. Whether there is any pending operations on return from recursive call
   a) Tail Recursion
   b) Non-Tail Recursion
3. Based on the pattern of recursive call
   a) Linear Recursion
   b) Binary Recursion
   c) n-ary Recursion

1. Direct and Indirect Recursion:

a) Direct Recursion	b) Indirect Recursion
* occurs when a function calls itself. * Simpler and commonly used	* occurs when a function calls another function which in turn calls the original function

Syntax:
`func()`
`{ ...`
`func();`
`...`
`}`
	Syntax:
	`f1()`
	`{`
	`...`
	`f2();`
	`...`
	`}`
	`f2()`
	`{`
	`...`
	`f1();`
	`}`

2.

a) Tail Recursion:

* A recursion in which the last operation of a function is a recursive call.
  ie) the recursive call is the last thing done by the function
* No need to keep record of the previous state.
  ie) no pending operations to be performed on return.

→ eliminates the need to store the intermediate result.

Syntax:

```
fn()
{
  ...
  ...
  fn();
}
```

EX:

```
main()
{
  int fun(int);
  int n = 3;
  fun(3);
}
int fun(int n)
{
  if(n==0)
    return;
  else
    printf("%d", n);
  return fun(n-1);
}
```

O/P:

3 2 1

b) Non-tail Recursion:

* A recursive call is not the last thing done by the function.

ie) pending operations to be performed on return.

Syntax:

```
fn()
{
  ...
  fn();
  ...
}
```

EX:

```
main()
{
  int fun(int);
  int n=3;
  fun(3)
}
int fun(int n)
{
  if(n==0)
    return;
  fun(n-1);
  printf("%d", n);
}
```

O/P:

1 2 3

3.

a) Linear Recursion:

* A linear recursive function makes only one recursive call.

EX:

```
#include<stdio.h>
main()
{
int n,f;
printf("Enter a number");
scanf("%d",&n);
f=fact(n)
printf("Factorial is %d",f);
}
int fact(int n)
{
if(n==0)
return 1;
else return (n*fact(n-1));
}
```

O/P:

```
Enter a number:5
Factorial is 120
```

b) Binary Recursion:

* A binary recursive function calls itself twice

Syntax:

```
fn()
{
fn();
fn();
}
```

EX:

```
main()
{
int n,f;
printf("Enter a number:");
scanf("%d",&n);
f=fib(n);
printf("Fibonacci Term %d",f);
}
int fib(int n)
{
if(n==1)
return 0;
if(n==2)
return 1;
else
return fib(n-1)+fib(n-2);
}
```

0 1 1 2 3

O/P:

```
Enter a number:5
Fibonacci Term : 3
```

c). n-ary Recursion:

* Most general form of recursion
- used in Generating permutations.

3.4 POINTERS

* Pointer is a variable that holds the address of another variable.
* Every pointer variable takes the same amount of memory space. ie) 2 bytes.

Declaration of a pointer:

Syntax:

```
type_specifier *identifier;
```

EX:

```
int *p;
```

type_specifier → type of the object referred
* → punctuator read as "pointer to"
identifier → name of the pointer variable

3.4.1 POINTER OPERATORS

Operations on Pointers / Accessing the pointer Variable

i) Reference operator
ii) Dereference operator.

i) Referencing operation:

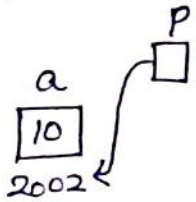
* a pointer variable is made to refer to an object with the help of reference or address of operator (&)
* Reference operator is a unary operator and the operands are of arithmetic type or pointer type.

Syntax:

```
datatype *pt_var, var1;
pt_var = &var1
```

EX:

```
int *p, a=10;
p = &a;
```



ii) Dereferencing operation:

* The object pointed or referred by a pointer can be accessed by dereference operator or indirection operator or value-at operator (*)

Syntax:

```
ptvar = &var1
var2 = *pt_var;
```

EX:

```
int *p, a, b;
p = &a;
b = *p.
```

* → used to get the value from the memory.

Assigning to a Pointer. / Initialization of Pointers.

1. A pointer can be assigned or initialized with the address of an object.

EX:
```
int *p;
p = &a;
```

2. A pointer to a type cannot be assigned or initialized with the address of an object of another type.

EX:
```
int *p;
float a;
p = &a; // not valid.
```

3. A pointer can be assigned or initialized with another pointer of the same type.

Pointers to pointers:
- pointer variable can store the address of a pointer variable.

Syntax:
```
int **pt;
```

EX:
```
int *p1, **p2, x=100;
p1 = &x;
p2 = &p1;
```

3.4.2 POINTER ARITHMETIC

* Arithmetic operations can be applied to pointers in a restricted form.

1. Addition operation:

* An expression of integer type can be added to an expression of pointer type.

Ex:

```
int *p;        // p = 2000
p = p+1;       // p = 2002

float *p;      // p = 2000
p = p+1;       // p = 2004
```

* Addition of two pointers is not allowed.
* Addition of a pointer and an integer is commutative.

ie) p+1, 1+p } same.

2. Subtraction operation:

* A pointer and an integer can be subtracted.

Ex:

```
int *p;        // p = 2000
p = p-1        // p = 1998

float *p;      // p = 2000
p = p-1        // p = 1996
```

* Subtraction of two pointers is
* Subtraction of a pointer and an integer is not commutative

ie) p-1, 1-p } not same.

3. Increment operation:

* Increment operation can be applied to an operand of pointer type

Ex:

```
int *ptr, *p;  // ptr = 2000
p = ptr++;     // ptr = 2002   p = 2000
               // ptr = 2004
```

4. Decrement operation:

* Decrement operator can be applied to an operand of pointer type.

Ex:

```
int *ptr, *p;        // ptr = 2000
p = ptr--;           // ptr = 1998
p = --ptr;           // ptr = 1996
```

5. Relational Operation:

* A pointer can be compared with the pointer of the same type or with 0.
* The result is either true or false ie) 1 or 0.

Ex:

```
int *p1, *p2, r;     // p1 = 2000  p2 = 2004
r = p1 < p2;         // r = 1
```

Illegal Pointer operations.

* Addition of two pointers is not allowed.
* Only integers can be added to pointers.
* Multiplication and division operations are not allowed.
* Bitwise operators are not allowed.
* Pointer of one type cannot be assigned to another type.

Void Pointer:

* Void is one of the basic data type
* Void means nothing.
* It is not possible to create an object of type void but it is possible to create a pointer to void.
* Such a pointer is known as void pointer and has the type void *.

Ex:

```
void *ptr;
```

Operations on void Pointers:

* A pointer to any type of an object can be assigned to a void pointer.
* Void pointers can be compared for equality and inequality.
* Void pointers cannot be dereferenced.
* Pointer arithmetic is not allowed.

Null Pointer:

* A null pointer is a pointer that does not point anywhere.
* It does not hold the address of any object.

EX:

```
int *ptr = 0;
int *ptr = NULL;
```

NULL - Symbolic constant with value 0.

3.4.3 ARRAYS AND POINTERS

* There is a strong relationship between pointers and arrays.

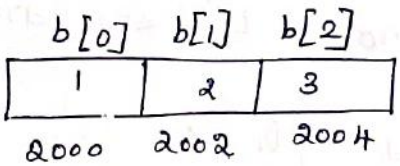
→ Any operation that can be achieved by array subscripting can also be done with pointers.

* The expression of the form E1[E2] is automatically converted into the expression of the form *(E1+E2)

ie) E1[E2] ⇒ *(E1+E2)

Ex:

```
int b[3] = {1, 2, 3};
```



* The name of the array refers to the address of the first element of the array.


```
int *x;
x = x+1;
```

* base address is incremented by 2.

Ex:

```
#include <stdio.h>
main()
{
  int a1[5] = {10, 15, 20};
  printf(" Elements of an array : %d %d %d ",
                          a1[0], a1[1], a1[2]);

  printf(" Elements of an array: %d %d %d ",
                          *(a1+0), *(a1+1), *(a1+2));
}
```

Output:

Elements of an array : 10 15 20
Elements of an array : 10 15 20.

ARRAY OF POINTERS

* An array of pointers is a collection of addresses.
* All pointers in an array must be of same type.

Syntax:

```
*a[ ] = { &v1 .... }
```

Ex:
```
#include <stdio.h>
main()
{
    int a=10, b=20, c=30;
    int *a1[ ] = { &a, &b, &c };
    printf(" Elements are %d %d %d", a, b, c);
    printf(" Elements are %d %d %d", *a1[0],
                                     *a1[1], *a1[2]);
}
```

Output:
```
Elements are 10 20 30
Elements are 10 20 30
```


Pointer to an array:

* Create a pointer that points to a complete array instead of pointing to the individual elements of an array.
- Such pointer is known as pointer to an array.

Syntax:

```
datatype (* Variable-name) [size];
```

Ex:
```
int *ptr[2];
```
↓ size of array Variable

Ex:

```
#include <stdio.h>
main()
{
  int a1[2][2] = {10,15,20,25};
  int (*ptr)[2] = a1;
  printf(" Elements in row 1 : %d %d ", a1[0][0], a1[0][1]);
  printf(" Elements in row 2: %d %d", ptr[1][0], ptr[1][1]);
}
```

Output:

```
Elements in row1 :  10  15
Elements in row2 :  20  25
```

Advantages of using pointers:

* Enables to access the memory directly.
* Increases the execution speed of the program
* Saves memory space.

3.5 PARAMETERS PASSING

* Argument Passing Methods:

* Depending upon whether the values or addresses are passed as arguments to a function, the argument passing methods are classified as:

i) Pass by Value
ii) Pass by Address (Reference)

(i) PASS BY VALUE: 3.5.1

- call by value.
- passing arguments by value.

* The values of actual arguments are copied to the formal parameters of the function.
- The changes made in the values of formal parameters inside the called function are not reflected back to the calling function.

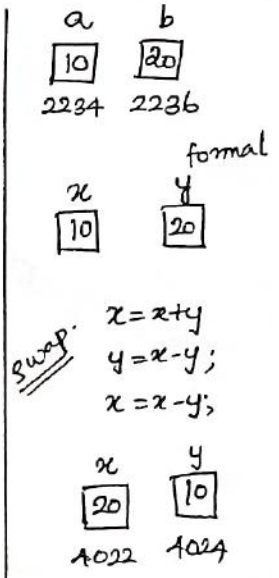
Program:

CALL BY VALUE

```
#include<stdio.h>
#include<conio.h>
main()
{
        int a,b;
        void swap(int,int);
        printf("\nENTER THE VALUE OF A & B: ");
        scanf("%d%d",&a,&b);
        printf("\nBEFORE SWAPPING: A= %d, B=%d\n",a,b);
        swap(a,b);
        printf("\nAFTER SWAPPING - In Main Function \n A= %d, B=%d\n",a,b);
        getch();
}
void swap(int x,int y)
{
        x=x+y;
        y=x-y;
        x=x-y;
        printf("\nAFTER SWAPPING - In swap function\n x=%d,y=%d\n",x,y);
}
```

OUTPUT:

```
ENTER THE VALUE OF A & B: 10   20
BEFORE SWAPPING: A= 10, B= 20
AFTER SWAPPING - In swap function x=20,y=10
AFTER SWAPPING - In Main Function  A= 10, B= 20
```



a, b → actual parameters

x, y → formal parameters

→ changes only in the swap function itself.

* On the execution of the function call, the values of actual parameters a & b are copied into the formal parameters x & y.
* Formal parameters are allocated at separate memory locations
* On returning from the called function, the formal parameters are destroyed and the access to the actual parameters gives values that are unchanged.

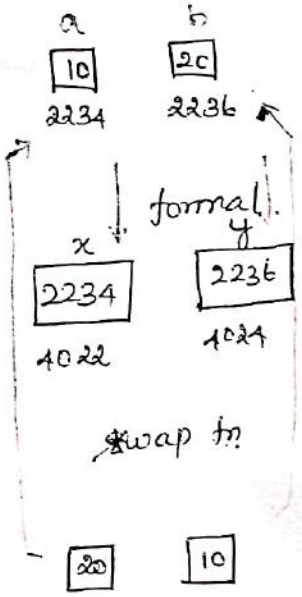
3.5.2

(ii) PASS BY REFERENCE: /call by Reference

- Call by reference.
* The addresses of the actual parameters are passed to the formal parameters of the function.
* The changes made in the values pointed to by the formal parameters in the called function are reflected back to the calling function. i.e) change in formal parameters affects the actual parameters.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
main()
{
        int a,b;
        void swap(int *,int *);
        printf("\nENTER THE VALUE OF A & B: ");
        scanf("%d%d",&a,&b);
        printf("\nBEFORE SWAPPING: A= %d, B=%d\n",a,b);
        swap(&a,&b);
        printf("\nAFTER SWAPPING - In Main Function \n A= %d, B=%d\n",a,b);
        getch();
}
void swap(int *x,int *y)
{
        int t;
        t=*x;
        *x=*y;
        *y=t;
        printf("\nAFTER SWAPPING - In swap function\n x=%d,y=%d\n",*x,*y);
}
```



OUTPUT:

```
ENTER THE VALUE OF A & B: 10   20
BEFORE SWAPPING: A= 10, B= 20
AFTER SWAPPING - In swap function x=20,y=10
AFTER SWAPPING - In Main Function A= 20, B= 10
```

UNIT – IV

STRUCTURES AND UNION

Structure – Nested structures – Pointer and structures – Array of structures – Self referential structures – Dynamic memory allocation – Singly Linked List – typedef – union – Storage classes and Visibility

4.1 STRUCTURE

INTRODUCTION

- C language provides a rich set of primitive and derived data types for the efficient storage and manipulation of data.

- Using C language new data types can be created. These data types are known as **user-defined data types** and created by using **structures, unions and enumerations**.
- Arrays are used for storage of homogeneous data. They cannot be used for storage of data of different types.
- One of the similarities between array and structure is that both contains finite number of elements. Thus array types and structure types are collectively known as aggregate types.

- **Unions** are similar to structures in all aspects except the manner in which their constituent elements are stored. In **structures**, separate memory is allocated to each element, while in unions, all the elements share the same memory.
- **Enumerations** – for defining a data type whose objects can take a limited set of values.

NEED FOR STRUCTURE DATA TYPE / USES OF STRUCTURES

- It allows grouping together of different type of elements.
- Complex data types can be handled using nesting of structures
- Structures can be used to define records to be stored in files
- It gives flexibility to programmers to define their own data types as per the requirement.
- It is also possible to create structure pointers.

STRUCTURE

- A structure is a collection of **variables of different data types** grouped under a single name.
- Structures are defined as a **collection of data items of different data types under a common name**. Structures are collection of related variables under one name.

Example:

Student: name, roll_no, marks

There are **three aspects** of working with structures.

1. Defining a structure(Creating a new type)
2. Declaring variables and constants of newly created type.
3. Using and Performing operations on objects of structure type.

STRUCTURE DEFINITION

- A **structure definition** consists of the keyword **struct followed by an** optional identifier name known as **structure tag-name** and a structure **declaration list** enclosed **within the braces**.
- The structure **declaration list** consists of declarations of one or more variables, possibly of different types. The variables declared inside the declaration list are known as **structure members** or fields.

The **general form** of structure-type definition is

```
struct structure_name
{
  type membername1;
  type membername2;
  ….....................
  ….....................
};
```

Eg:

```
struct book
{
  char title[25],author[25];
  int pages;
  float price;
};
```

- Structure definition can have an infinite number of members.

- After the definition of structure type, the keyword struct is used to declare its variables.
- A structure definition cannot contain an instance of itself. But it may contain a pointer to an instance of itself. Such a structure is known as self-referential structure.
- A structure **definition does not reserve any space** in the memory.
- It is not possible to initialize the structure members during the structure definition.

Eg:

```
struct book
{
int pages=10;   //Not valid
};
```

If a structure definition does not contain a structure tag-nam, then the created structured is unnamed. It is also known as anonymous structure type. The objects of anonymous type should be declared only at the time of structure definition.

DECLARING STRUCTURE OBJECTS/VARIABLES

- Variables and constants of the created structure type can be declared either at the time of structure definition or after the structure definition.

The **general form** of declaring structure object is

struct structure_name identifier[=initialization_list]**;**

[=initialization_list] is optional.

(or)

struct struct_name v1,v2,....vn;

where v1,v2,..vn are variables

Eg:

```
struct book b1;
struct student s1,s2,stud;
```

A structure object declaration consist of

- The keyword struct for declaring structure variables.
- The tag name of the defined structure type.
- Comma separated list of identifiers
- A terminating semicolon.

EX :

```
struct book
{
char title[20];                    //Defining a structure
int pages;
float price;
};
struct book b1,b2,b3;          //Declaring structure variable
```

- It is also possible to combine **both definition and variable declaration** in one statement.

EX:

```
struct book
{
        char title[20];                 //Defining a structure
        int pages;
        float price;
}b1,b2,b3;                              //Declaring structure variable
```

- The objects of defined structure type cannot be declared without using the keyword struct.
- The amount of memory space allocated to it is equal to the sum of the memory space required by all of its members.
- The structure members are assigned memory addresses in increasing order.
- The members of the structure object can be initialized by providing an initialization list. An initialization list is a comma separated list of initializers.

Operations on structures

The operations that can be performed on an object of structure type can be classified into two types.

1. Aggregate Operations

- operates on the entire operand as a whole.

2. Segregate Operations

- operates on the individual members of a structure object.

Aggregate Operations

There are **four aggregate operations** that can be applied on an object of a structure type.

1. Accessing members of an object of structure type
2. Assigning a structure object to a structure variable.
3. Address of a structure object.
4. Size of a structure.

Accessing members of an object of structure type

The members of a structure object can be accessed by

1. Direct Member Access operator (**. dot** operator).
2. Indirect Member Access operator (→ **arrow** operator).

Initialization of Structures

- The members of a structure can be initialized to constant values by enclosing the values to be assigned within the braces after the structure definition.

Syntax:

```
struct struct_name
{
        member1;
        member2;
        .
        .
        .

}struct_variable={contant1, constant2,....};

                        (or)

struct struct_name struct_variable={contant1, constant2,….};
```

Ex:

```
struct date
{
        int date;
        int month;
        int year;
}independence= {15,08,1947};
        or
struct date independence={15,08,1947};
```

- Initializes the member variables date, month, year of independence to 15,08,1947 respectively.

Accessing Structure members.

- The members of the structures can be accessed by using the structure variable along with the dot(.) operator.

Syntax:

variable name. member name;

Ex:

```
struct book
{
        int id;
        char name[20];
};
struct book b1;
```


For **accessing the structure members** from the above example.

b1.id;

b1.name; where 'b1' is the structure variable.

The structure can be defined either before main() as globally or inside main() locally.

Example program :

```
#include<stdio.h>
struct book                                    //structure name
{
        int id;
        char name[20];
        char author[15];
};
main()
{
        struct book b1;                                  // structure variable
        printf("\n Enter the book id, book name\n");
        scanf("%d\n%s\n",&b1.id,b1.name);
        printf("\n Book id is = %d",b1.id);               //Accessing structure member
        printf("\n Book name is = %s",b1.name);
}
```

output:

```
Enter the  book id, book name
101
Maths
Book id is = 101
 Book name is = Maths
```

4.2 NESTED STRUCTURES (STRUCTURE WITHIN A STRUCTURE)

- A structure can be declared within another structure.
- Some times it is required to keep a compound data items within another compound data item is called structure within structure or it means nesting of structures.

Syntax :

```
struct struct_name1
{
        decl 1;
        decl 2;

        ….
        decl n;
};
struct struct_name2
{
        decl 1;
        decl 2;
        struct struct_name1 variable_name1;        //structure within structure
        .....
        decl n;
};
```

Example Program :

```
#include<stdio.h>
struct date
{
        int date, month, year;
};
struct stu_data
{
        char name[20];
        struct date dob;
};
main()
{
        struct stu_data s ={“vinoth”,{01,03,82}};
        printf(“\n Name %s”,s.name);
        printf(“\n \n Date of birth : %d-%d-%d”,s.dob.date, s.dob.month, s.dob.year);
        getch();
        Return;
}
```


Output :

```
Name : Vinoth
Date of Birth : 01- 03- 82
```

4.3 POINTER AND STRUCTURES

* Assign pointers to structures
* The pointer variable that holds the address of a structure.
  → 4 bytes of memory it takes.

Declaring a pointer to a structure:

Syntax:

```
struct structname
{
    member 1;
    member 2;
    :
    member n;
};
main()
{
    struct structname *ptr, var;

    ptr = &var;
    ...
}
```


Accessing the members of the structures.

two forms:

(*ptr).member1

or

ptr → member1

EX:

ptr → regno;

Initialization:

Syntax:

(*ptr). member_name = constant;

or

ptr → member_name = constant;

EX:

ptr → regno = 47;

Example Program:

Program for printing Employee details:

```
struct employee
{
   int idno;
   float salary;
   char name[50];
};
main()
{
   struct employee *empptr, e;
   empptr = &e;
   printf(" Enter employee Id no: ");
   scanf (" %d", & empptr → idno);
   printf(" Enter employee name: ");
   scanf(" %s", empptr → name);
   printf(" Enter employee salary ");
   scanf (" %f", &empptr → salary);
   printf(" The Employee details are: ");
```

```
    printf(" ID No. is %d", emptr->idno);
    printf(" Name is %s", emptr->name);
    printf(" Salary is %f", emptr->salary);
}
```

output:

```
Enter employee Id no: 100
Enter employee name: Raju
Enter employee salary: 50000
The Employee details are:
ID No. is 100
Name is Raju
Salary is 50000
```

4.4 ARRAYS OF STRUCTURES

- The C language permits to declare an array of structure variable.
- If we want to handle more records within one structure, we need not specify the number of structure variable.
- In such cases we declare an array of structure variable to store them in one structure variables.

Syntax:

```
struct struct_name
{
        decl1;
        decl2;
        ......
        decl;
}variable_name[size];
```

Example :

```
struct marks
{
        int subject1;
        int subject2;
        int subject3;
};
main()
{
        struct marks student[3]={{95,92,89},{65,63,70},{87,76,61}};
 }
```


Differences between Array and Structure

Array	Structure
An array is a collection of related data elements of same type.	Structure can have elements of different types.
An array is derived data type	structure is a user-defined one
Any array behaves like a built-in data type	It must be declared and defined
An array can be increased or decreased	A structure element can be added if necessary.

4.5 SELF REFERENTIAL STRUCTURES

* A structure containing a member that's a pointer to the same structure type

→ one or more pointers pointing to the same type of structure as their member.

* Used in dynamic data structures such as trees, linked list etc.

Syntax:

```
struct struct_name
{
  datatype var;
  struct_name * pointer_name);
};
```


Ex:

```
struct node
{
  int data;
  struct node *next;
}
```

```
struct node
{
  int data;
  char value;
  struct node *link;
};
main()
{
  struct node obj1;
  obj1.link = NULL;
  obj1.data = 10;
  obj1.value = 20;
  struct node obj2;
  obj2.link = NULL;
  obj2.data = 30;
  obj2.value = 40;
  obj1.link = &obj2;
  printf("%d", obj1.link->data);
  printf("%d", obj1.link->value);
}
```

O/P: 30 40

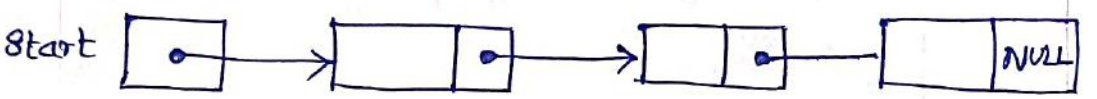
R3

* Self-referential structures are very useful in applications that involve linked data structures

Linked data structure

- each component within the structure includes a pointer indicating where the next component can be found.

* Relative order of the components can easily be changed by altering the pointers.
* Individual components can easily be added or deleted. by altering the pointers.



4.6 DYNAMIC MEMORY ALLOCATION

* The ability for a program to obtain more memory space at execution time to hold new nodes, and to release space no longer needed.

Need:

Usings Arrays:

RA

i) possibility of overflow, C does not check bounds.

ii) Wastage of space.

* **Dynamic Memory allocation:**
  * Required memory allocation at run time.

static memory allocation:

→ when fixed arrays are used.

→ memory allocated at compile time.

* **Dynamic Memory allocation** is a way to
  - defer the decision of how much memory is necessary until the program is actually running. or
  - give back memory that the program no longer needs.
* Heap is used.
  - allocate & deallocate dynamic heap memory.

Types of Functions:

i) malloc()

ii) calloc()

iii) free()

iv) realloc()

i) malloc() (Memory Allocation)

* Allocates a new area in memory of the number of bytes and stores a pointer to the allocated memory.
  - Request a contiguous block of memory of the size in the heap.

Syntax:

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

EX:

```
int *ptr;
ptr = (int*) malloc (5 * sizeof(int));
```

(sizeof(int) → 4 bytes)

* 20 bytes of memory block is dynamically allocated to ptr.

* If space is insufficient, allocation fails and returns a NULL pointer

ii) calloc() (Contiguous Allocation)

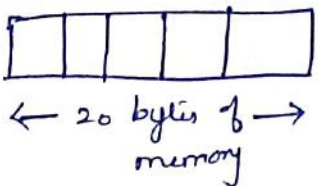
* Similar to malloc(), but initializes the memory to zero
* Dynamically allocates the specified number of blocks of memory of the specified type.

Syntax:

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

EX:

```
int *ptr = (int*) calloc (5, sizeof(int));
```



* 5 blocks of 4 bytes, each is dynamically allocated to ptr.

iii) free():

* free() is used to dynamically de-allocate the memory.

→ free() takes a pointer to a heap block earlier allocated by malloc() and returns that block to the heap for reuse.

— After free(), the client should not access any part of the block.

Syntax:

```
free(ptr);
```

* frees the space allocated in the memory pointed by ptr

EX:

```
main()
{
  int *ptr, *ptr1;
  int n, i;
  n=5;
  printf("Number of elements: %d", n);
  ptr = (int *) malloc (n* sizeof(int));
  ptr1 = (int *) calloc (n, sizeof(int));
  if (ptr==NULL || ptr1==NULL)
    { printf(" Memory not allocated");
      exit(0);
    }
  else
  {
    printf(" Memory allocated ");
    free(ptr);
    printf(" Memory Successfully freed");
  }
}
```


O/P:

```
Number of elements : 5
Memory allocated
Memory successfully freed.
```

iv) realloc() (re-allocation)

* dynamically change the memory allocation of a previously allocated memory.

→ Takes an existing heap block and tries to reallocate it to a heap block of the given size which may be larger or smaller than the original size of the block.
- It returns a pointer to the new block.

* Reallocation of memory maintains the already present value and the new block will be initialized with default garbage value.

Syntax:

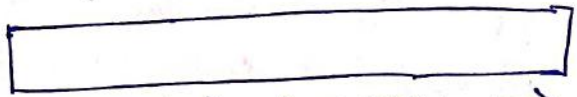
```
ptr = realloc (ptr, newsize);
```

Ex:

```
int *ptr = (int*) malloc (5 * sizeof (int));
```

ptr = 


```
ptr = realloc (ptr, 10 * sizeof (int));
```

ptr = 

stdlib.h

* To allocate memory dynamically, library functions are

malloc()
calloc()
free()
realloc()

are used.

→ These functions are defined in stdlib.h (or alloc.h) header file.

* So include these header files.

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

or

```
#include<alloc.h>
```

4.7 SINGLY LINKED LIST

Linked List:

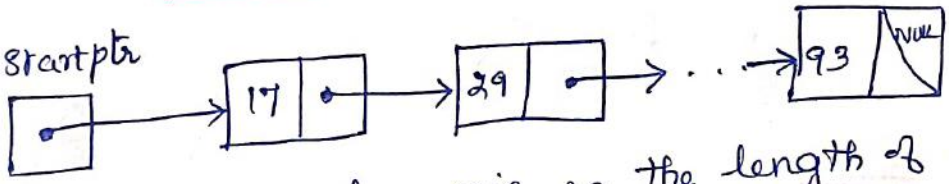
* a linear collection of self-referential structures called nodes, connected by pointer links.

→ A linked list is accessed via a pointer to the first node of the list.
- Subsequent nodes are accessed via the link pointer member stored in each node.
- The link pointer in the last node of a list is set to NULL to mark the end of the list.

* Linked list elements are not stored at a contiguous location,
- the elements are linked using pointers.

* Each structure of the list consists of two fields
  i) data item
  ii) address of the next item in the list (pointer)

Graphical Representation



* Linked lists are dynamic, so the length of a list can increase or decrease at execution time.

Types of Linked List:

i) Singly Linked List
ii) Doubly Linked List
iii) Circular Linked List.

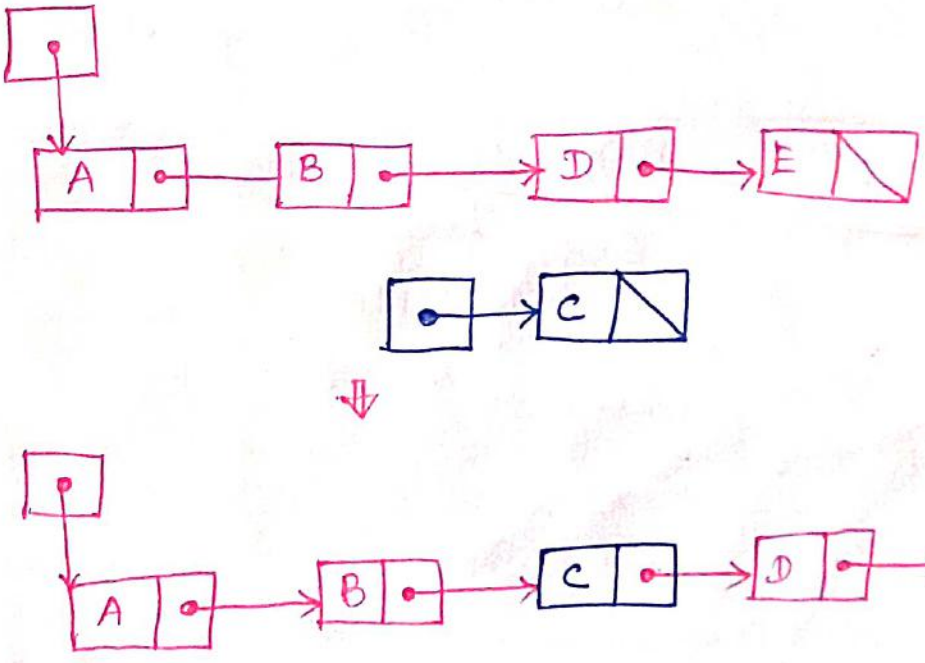
Creation of a Singly Linked List: - Structure definition

```
struct node
{
  int data;
  struct node *next;
};
main()
{
  typedef struct node *position;
  position p,L;
  L->next = NULL;
}
```

Operations :

i) Creation
ii) Traversing
iii) Insertion
iv) Deletion
v) Concatenation.
vi) Searching.

Insertion:



Insertion at the beginning:

i) Obtain space for new node
ii) Assign data
iii) Set the next field of the new node to the beginning of the list.
iv) Change the reference pointer to point to the new node

EX:

```
position insert_beg(int val, position L)
{
  position new;
  new = malloc(sizeof(struct node));
  if(new!=NULL)
  {
    new->data = val;
    new->next = L->next;
    L->next   = new;
  }
}
```

Insertion at last:

i) obtain space for new node
ii) Assign data
iii) Set the next field of the new node to NULL

EX:

```
position insert_last(int val, position L)
{
  position new, P;
  new = malloc(sizeof(struct node));
  if(new!=NULL)
  {  new->data = val;
     new->next = NULL;
```

```
    p = L->next;
    while (p->next != NULL)
        p = p->next;
    p->next = new;
}
```

Insertion at the middle:

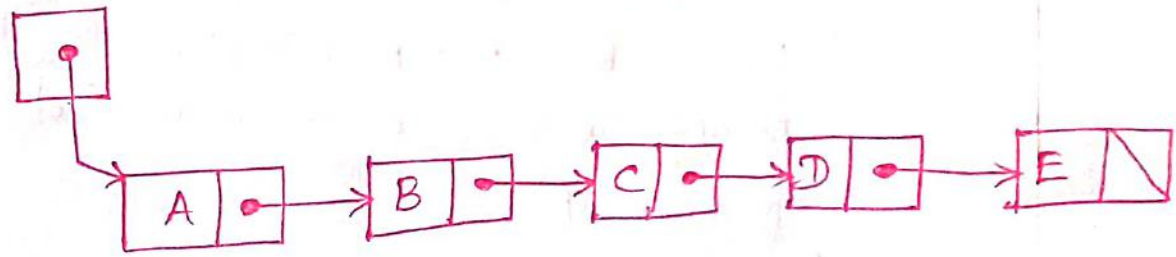
i) Obtain space for new node
ii) Assign data to the data field of the new node
iii) Get the input for after which the node has to be inserted.
iv) Move to the corresponding node
v) Set the next field of the new node to pointing to the next of the corresponding node.

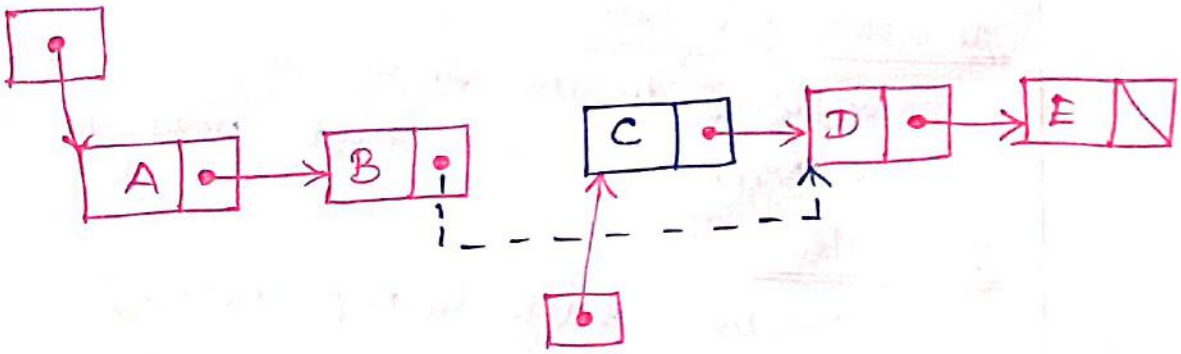
Ex:

```
position insert_mid (int val, position L)
{
    position new, p;
    int c;
    new = malloc (sizeof (struct node));
    if (new != NULL)
    {
        printf ("Enter the value after which the
                 value to be inserted ");
        scanf(" %d", &c);
        p = find (c, L);
        if (p != NULL)
        {
            new->data = val;
            new->next = p->next;
            p->next   = new;
        }
    }
}
```

```
position find (int x, position L)
{
    position p;
    p = L->next;
    while ( p! =NULL && p->data!=x)
        p = p->next;
    return p;
}
```

Deletion:





* Removing a node is the list
* Deletion can be done at
    * beginning
    * middle
    * last

Deletion at beginning:

* Make the head pointing to the second node in the list.

EX: Pgm:

```
position delete_beg(position L)
{
  position temp;
  if (L->next==NULL)
  printf("The list is empty");
  else
  {
   temp = L->next;
   L->next = temp->next;
   printf("Deleted element is %d", temp->data);
   free(temp);
  }
}
```


Deletion at last:

* Move to the last node in the list
* Make the last node next pointer to NULL.

EX: Pgm:

```
position delete_last(position L)
{
  position temp, p;
  if (L->next==NULL)
  printf("The list is empty");
  else
  {
    p = L;
    while( p->next!=NULL)
        p=p->next;
```

```
    temp = p;
    p->next = NULL;
    printf(" Deleted Element is %d", temp->data);
    free(temp);
  }
```

Deletion at middle:

* Move to the node containing the element to be deleted.
* Make the previous node next pointer to the node after that node.

Program:

```
position delete_mid (position L)
{
  position temp, p;
  int x;
  if( L->next == NULL)
  printf(" The list is empty");
  else
  {
    printf(" Enter the element to delete");
    scanf("%d", &x);
    p = find_prev (x, L);
  if( ! islast (p))
  {
    temp = p->next;
    p->next = temp->next;
    printf(" Deleted element is %d", temp->data);
    free(temp);
  }
}
```

```
position find_prev(int x, position L)
{
  position p;
  p=L;
  while (p->next!=NULL && p->next->data!=x)
        p = p->next;
  return p;
}
```

Display the List:

```
position display(position L)
{
  position p;
  p = L->next;
  printf("\n");
  while(p!=NULL)
  {
    printf(" %d ->", p->data);
    p=p->next;
  }
  printf("NULL");
}
```

4.8 TYPEDEF

* typedef keyword allows the programmer to create a new data type for an existing data type.
  → Alternate name is given to a known data type
* makes the code more portable.

Declaration:

```
typedef existingdatatype newdatatype,... ;
```

EX:

a) ① typedef int length;
length len, maxlen;

* length is type int.
* len, maxlen are regarded as int
  ie) int len, maxlen;

② typedef char lower_case;
lower_case a,b,c;

b) Array & pointers

① typedef int length;
length *lengths[];

② typedef char *string;
string p, l[50];

c) Structure:

* Complex data type like structure can use typedef.

EX:

```
typedef struct point
{
   int x;
   int y;
} ;
main()
{
 typedef struct point dot;
 dot left, right;
}
```


* When typedef is used to name a structure, the structure tag name is not necessary.

EX:

```
typedef struct
{
   float real;
   float img;
} complex;
complex u, v;
```

Program that prints x-y coordinates of the two ends of a line.

```
#include <stdio.h>
typedef struct                    // no tag name
{
   int x;
   int y;
};
main ()
{
   typedef struct dot;
   dot left, right;
   printf(" Enter x & y coordinates of left:\n");
   scanf("%d %d", &left.x, &left.y);
   printf(" Enter x & y coordinates of right:");
   scanf("%d %d", &right.x, &right.y);

printf(" Left : x = %d, y = %d \n
        Right : x = %d, y = %d ", left.x, left.y,
                                  right.x, right.y);

}
```

output:

```
Enter x & y coordinates of left:
4 20
Enter x & y coordinates of right:
30 20
Left : x = 4  y = 20
Right : x = 30  y = 20
```

4.9 UNION

- **Union** is a collection of variables of different data types.
- **Union** is also a derived data type which is used to represent **dissimilar data items**.
- Unions are used to create user-defined types.
- Declaration and definition of union are same as structure, but use the keyword 'union' instead of 'struct'.
- The structure and union differs in terms of storage.
- In structure, a separate memory is allocated to each member, while in unions, all the members of union share the **same memory.**

Characteristics of union:

- Members of union have same memory location.
- Collection of variables of different data types.
- The keyword 'union' is used to declare a union.
- Members of the union can be accessed using the dot operator.
- Size allocated is equal to the largest data member of the union.
- Only one union member can be accessed at a time.
- The members of a union are stored in the memory in such a way that they overlap each other.

Definition and Declaration of Union

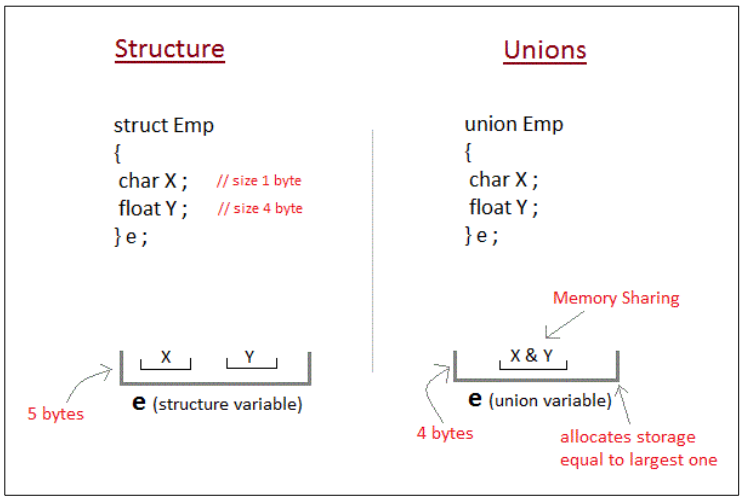
- A union **definition** consists of the keyword **union** followed by an optional **identifier name** and the union **declaration list** enclosed **within the braces**.
- A union object **declaration** consist of
  - The keyword union for declaring union variables.
  - The tag name of the defined structure type.
  - Comma separated list of identifiers
  - A terminating semicolon.

Syntax :

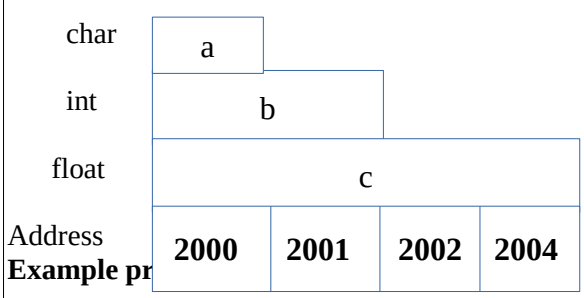

```
union union_name
{
        union member1;
        union member2;
        .....
        union member n;
};
union union_name variable;
```

Eg :

```
union numbers
{
        char a;
        int b ;
        float c;
}r;
```



Memory Allocation in Union:



Example pr

Employee details using Union.

```
#include<stdio.h>
#include<conio.h>
union employee
{
        char name[10];
        int idno;
        float salary;
}e;
main()
{
        printf("Enter the name\n");
        scanf("%s",e.name);
        printf("Enter the id number\n");
        scanf("%d",&e.idno);
        printf("Enter the salary\n");
        scanf("%f",&e.salary);

        printf("Name : %s\n",e.name);
        printf("Id number : %d\n",e.idno);
        printf("Salary : %f\n",e.salary);

        getch();
        return;
}
```

Output :


```
Enter the name
Ram
Enter the id number
101
Enter the salary
20000
Name : Ram
Id number : 101
Salary : 20000
```

Differences between Structure and Union

S.NO	Structure	Union
1	It occupies its own memory space.	It uses the same space.
2	The keyword 'struct' is used.	The keyword 'union' is used.
3	All members of a structure can be initialized.	Only the first member of a union can be initialized.
4	Each member is stored in a separate memory locations.	All members are stored in the same memory location.
5	More memory space is required.	Less memory space is required.

4.10 STORAGE CLASSES AND VISIBILITY

* The storage classes define visibility (scope) and the lifetime of any function / variable within a C program.

Storage class of a variable determines
* Where the variable would be stored
* What will be the initial value of the variables
* What is the scope of the variables.
* What is the lifetime of the variable.

Scope :
→ The area and block where the variable can be accessed.

Lifetime :
→ storage duration of the variable (global, local)

* **Classification of storage classes:**
  i) auto
  ii) extern
  iii) static
  iv) register

Syntax:

```
storage_class_specifier data_type var1,...varn;
```

EX:

```
auto int a;
extern int a;
static int a;
register int a;
```

Class	Place of Storage	Scope	Default Value	Lifetime
auto	RAM	Local	Garbage Value	Within a function
extern	RAM	Global	Zero	Till the pgm ends. declare it anywhere
Static	RAM	Local	Zero	Till the pgm ends. Retains the value
register	Register	Local	Garbage Value	Within the function

i) auto Storage Class : (Automatic variables)

* default storage class.
* declared at the start of the block.
* stored in memory.
* Memory is allocated automatically upon entry to a block and freed automatically upon exit from the block.

→ default value is a garbage value.

* Scope → local to the block in which they are defined.
* once executed, the contents and existence of the automatic variables get vanished.

EX:

```
main()
{
  auto int n=10;
  fn();
  printf("In main n is %d", n);
}
fn()
{
  int n=20;
  printf("In function n is %d", n);
}
```

O/P:

In function n is 20
In main n is 10

ii) extern Storage class (External Variables)

* May be declared outside any function block.
* Called as global variables.
* Variables are stored in memory.
* Memory is allocated when the program begins execution, and remains allocated until the program terminates.
* Scope → global.

→ If both global and auto variables have the same name in the program, first priority is given to auto variables.

Ex:

```
extern int n=10;
main()
{
  fn();
  printf("In main n is %d", n);
}
  fn()
  {
   printf("In fn n is %d", n);
}
```


O/P:

In fn n is 10
In main n is 10

iii) Static storage class

* Static variables may be internal or external depending on the place where they are declared.
* Static global → declared outside main
* Static local → declared inside main
* Variables are stored in memory.
* Initial value is zero.
* The value remains the same throughout the execution.

Ex:

```
void fn();
main()
{
 int i;
  for(i=0; i<=2; i++)
  {
   fn();
  }
}
void fn()
{
 static int count=0;
 printf("count is %d", count);
 count=count+2;
}
```

O/P:

count is 0
count is 2
count is 4

iv) register Storage class:

* Stores variables in the cpu registers instead of memory.
* Register access is faster than memory access.
  - only less variables can be stored.
* default initial value is a garbage value.
* scope - local to the block in which they are declared
* Allocates the storage upon entry to a block and the storage is freed when the block is exited.
* used in counters.

Ex:

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

O/P:

1 2 3 4 5

UNIT – V

FILE PROCESSING

Files – Types of file processing : Sequential access, Random access – Sequential access file – Random access file – Command line arguments.

5.1 FILES

* A file is a block of useful data which is available to a computer program.
  - stored on a persistent storage medium.
* Storing a file on a persistent storage medium like hard disk ensures the availability of the file for future use.
* Files are used for long-term retention of data.

→ Computers store files on secondary storage devices such as
  i) hard drives
  ii) solid-state drives
  iii) flash drives
  iv) DVDs.

* **Need:**
  * When a program is terminated, the entire data will be lost. Storing a file will preserve your data even if the program terminates.
  * Easy to move data from one computer to another.

* **Data Hierarchy:**

  i) Bit:
  - smallest data item
  - Value is 0 or 1

  ii) Byte: → 8 bits = 1 byte
  → used to store a character

iii) **Field :**

* group of characters conveying meaning

Ex: student's name

iv) **Record :**

* collection of related data fields

Ex: student's record
- contains name, address, roll number, marks and so on.

v) **File :**

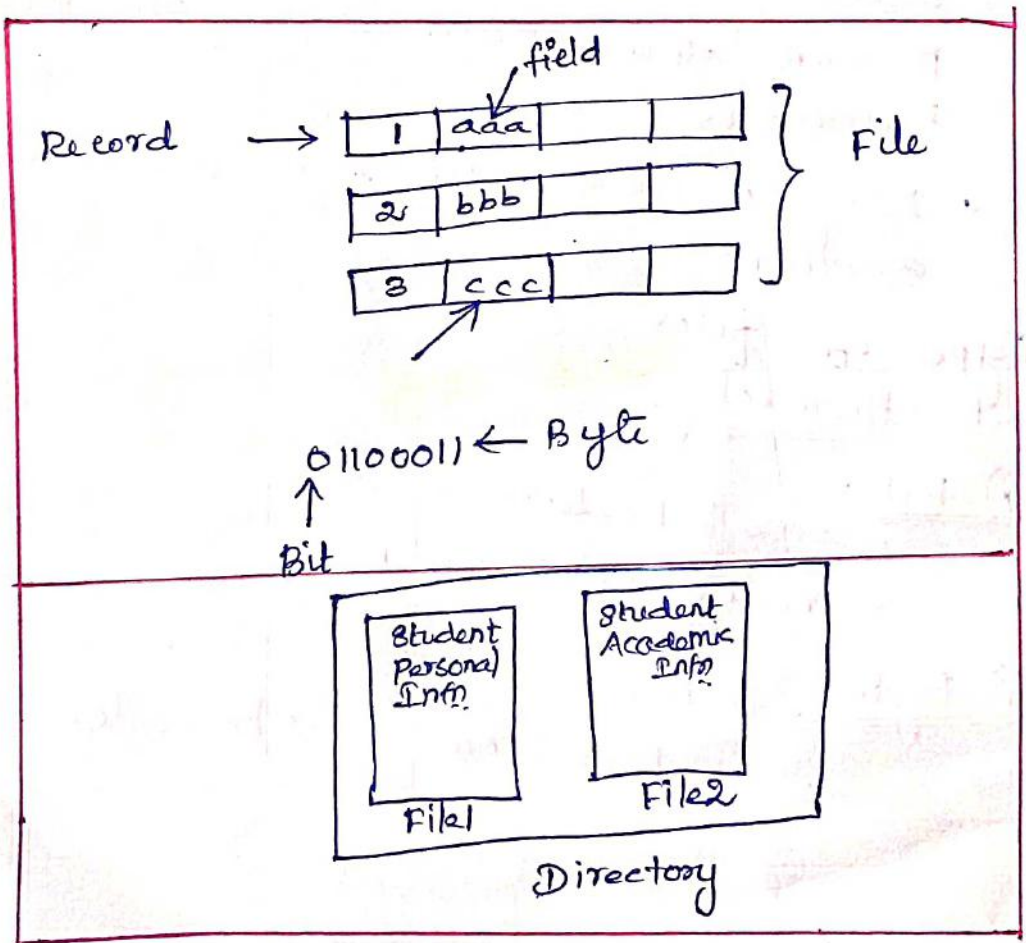
* collection of related records.

Ex:
If there are 60 students in a class, then there are 60 records.

vi) **Directory :**

* stores information of related files

→ collection of files.



A = 65
B = 66
C = 67

a = 97
b = 98
c = 99

* Types of Files:

Two types:

i) Text file

ii) Binary file.

i) Text File:

* A sequence of lines of alphabet, numerals, special characters etc.
- stored using its corresponding ASCII code
* End of a text file is denoted by a special character end-of-file marker
* A text file is also known as flat file or an ASCII file.
→ .txt files.
→ create using Notepad or text editors.
* Easy to read & maintain
* Least security
* takes bigger storage space.

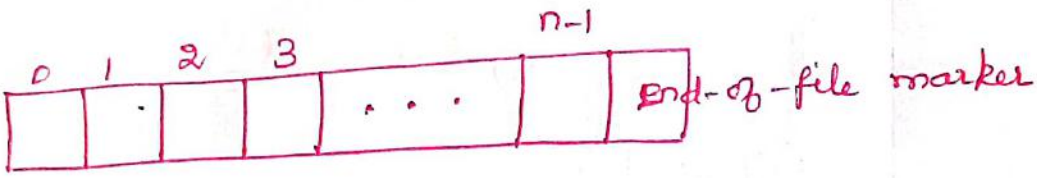
ii) Binary File:

* contains any type of data encoded in binary form. for computer storage and processing purposes
* Data is similar to the format in which the data is stored in the main memory.
* Not readable by human.
→ .bin files
* Better security than text files.

T2

(*) Files and Streams:

* C views each file simply as a sequential stream of bytes



Standard streams:

* When a file is opened, a stream is associated with it.

- Three streams are automatically opened when program execution begins.

i) standard input
   - receives i/p from the keyboard

ii) standard output
   - displays output on the screen

iii) standard error
   - displays error messages on the screen

* streams provide communication channels between files and programs.

(*) File Modes: * specifies the purpose of opening files

	Mode	Description
Text File	r	opens a text file for reading
	w	opens or create a text file for writing
	a	opens a text file in append mode
	r+	opens a text file for read/write
	w+	Create a text file for read/write
	a+	Append or create a text file for read/write

binary file	rb	Open a binary file for reading
	wb	open a binary file for writing
	ab	Append to a binary file
	rb+	open a binary file for read/write
	wb+	open a binary file for read/write
	ab+	Append a binary file for read/write

Basic File operations and Functions:

* **Creating a file:**
  → A file is created by specifying its name & mode
* **Updating a file:**
  → changing the contents of the file
* **Insertion:**
  → Insert a new record in the file
* **Deletion:**
  → Delete an existing record.
* **Retrieving from a file**
  → Extracting useful data
* **Maintaining a file**
  → Restructuring or re-organizing the file to improve the performance of the programs.
* **closing a file:**
  → The file should be closed after the file processing.

Four Actions:

* To use a file, four actions should be carried out.
  i) Delare a file pointer variable
  ii) opening a file using fopen() fn.
  iii) Process the file using functions
  iv) close the file using fclose() fn.

Functions:

S.No	Function	Description
1.	fopen()	creates a new file or open an existing file
2.	fclose()	closes a file
3.	getc()	reads a character from a file
4.	putc()	writes a character to a file
5.	fscanf()	reads a set of data from a file
6.	fprintf()	writes a set of data to a file
7.	getw()	reads an integer from a file
8.	putw()	writes an integer to a file
9.	fread()	reads an entire block from a file
10.	fwrite()	writes an entire block to a file
11.	fseek()	set the position to desire point
12.	ftell()	gives the current position in the file
13.	rewind()	set the position to the beginning point

Opening and closing a File:

Declaration of File Pointer:

* FILE is a structure declared in stdio.h file.
  → contains information used to process the file.
* file pointer → a pointer variable that points to a structure FILE.

Syntax:

```
FILE *fileptr;
```

EX:

```
FILE *fp;
```

* A number of different files may be used in a program.
  → use the file pointers. `FILE *f1, *f2...;`
* FILE establishes a buffer area
* pointer variable indicates the beginning of the buffer area.

a) Opening or creating a File:

* fopen() function is used to create a new file or to open an existing file.
* Two arguments: } character strings
  i) a filename
  ii) a file open mode

syntax:

```
FILE *fileptr;
fileptr = fopen (filename, mode);
```
→ r/w/a/r+/w+/a+ (any one)

R1

* opens an existing file.
  → If a file does not exists, a file will be created.

Ex:

```
FILE *fp;
fp = fopen("data.txt", "r");
```

* A filename can contain path information.
  → Path specifies the drive or directory where the file is located.
  → Without path:
    - located in same folder where the program is saved.

Path: `c:\\examdata\\marks.txt`
(c: → drive, examdata → directory, marks.txt → file)

R3

b) Closing a file:

* A file must be closed at the end of the program.
* fclose() function is used.

Syntax:

```
fclose(fileptr);
```

EX:

```
fclose(fp);
```

→ returns 0 on success or -1 on error.

* When a file is closed, the file's buffer is flushed.

c program constains the following statements

```
#include<stdio.h>
main()
{
   FILE *fp;
   fp = fopen("data.txt", "w");
   ...
   fclose(fp);
}
```


c) Detecting the End of a file:

* To detect end-of-file, feof() function is used.

Syntax:

```
feof(fileptr);
```

EX:

```
feof(fp);
```

RA

* The function feof() returns 0, if the end-of-file has not been reached. or returns a non-zero value, if end-of-file has been reached.

Ex: Program:

```
#include<stdio.h>
main()
{
  char temp[50];
  char fname[60];
  FILE *fp;
  printf("Enter a filename");
  scanf("%s", fname);
  fp = fopen(fname, "r");
  if(fp!=NULL)
  {
    while(!feof(fp))
    {
      fgets(temp, 50, fp);
      printf("%s", temp);
    }
  }
  else
  {printf("Error in opening file");
    exit(1);
  }
  fclose(fp);
}
```

O/P:

```
Enter a filename
first.c
#include<stdio.h>
main()
{ printf("welcome");
}
```

5.2 TYPES OF FILE PROCESSING

Two Types:

i) Sequential Access

ii) Random Access.

* C imposes no structure on a file.

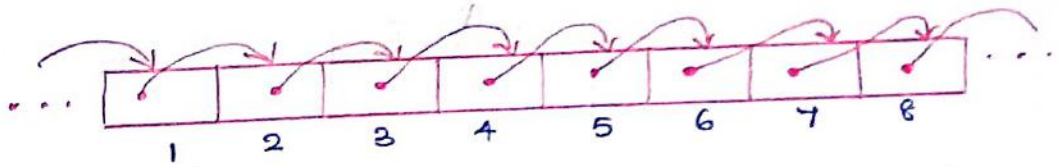
→ Need to impose record structure on a file.

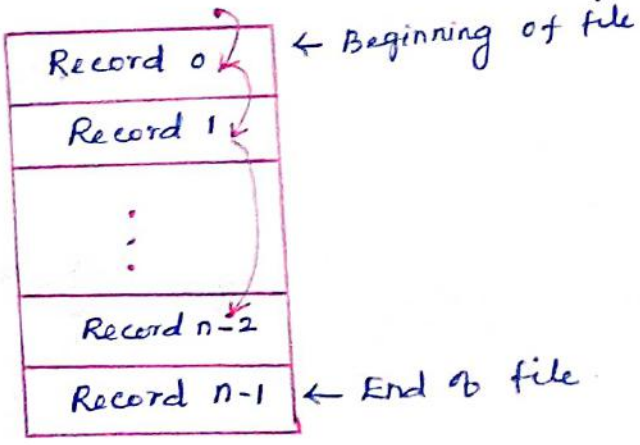
Sequential Access:

* Data is kept in sequential order.

→ To read the last record of a file, need to read all records before that record.

→ It takes more time



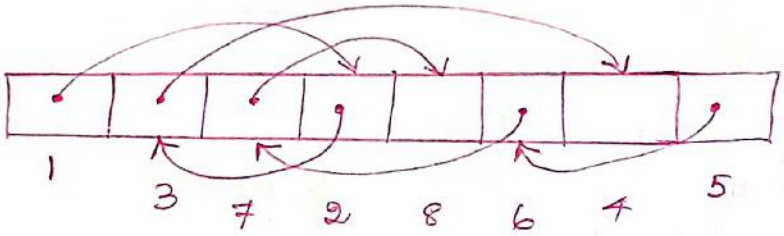


ii) Random Access

* Data can be read from, or written to, any position in a file without reading, or writing all the preceding data.

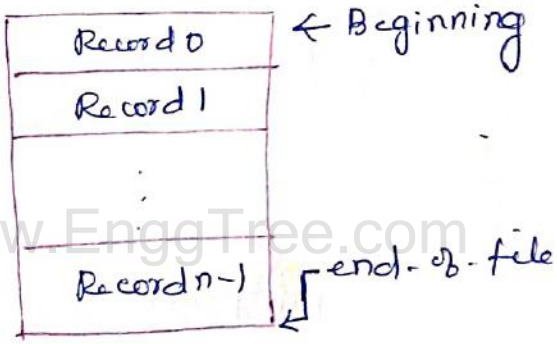
→ To read the last record, can read it directly.

* Takes less time than sequential access.



5.3 SEQUENTIAL ACCESS FILE

* A sequentially organized file stores the records in the order in which they are entered.
→ New records are added only at the end of the file.
* Sequential files can be read only sequentially, starting with the first record in the file.
→ Records numbered from 0 to n-1 stored in a sequential file.



* Once the records are stored in a file, then no modification can be made on the file.
* In case of deletion or updation of one or more records, replacing the records by creating a new file is possible.
* All the records have the same size, same field format, and every field has a fixed size.

Key:

* The records are sorted based on the value of one field or a combination of two or more fields.
  - This field is known as the key.
* Records can be sorted in either ascending or descending order.

* Declaring a pointer variable
* File operations
* File Modes.

Ex: Finding average of numbers stored in Sequential Access File.

```
#include< stdio.h>
main()
{
  FILE *f1, *f2;
  int num, i, n, sum=0, avg;
  printf("Enter the number of elements");
  scanf("%d", &n );
  f1 = fopen("data.txt", "w");
  for(i=1; i<=n; i++)
  { scanf("%d", &num);
    putw(num, f1);
  }
  fclose(f1);
  f1 = fopen("data.txt", "r");
  f2 = fopen("average.txt", "w");
  while( f1 != EOF)
  {
    num = getw(f1);
    sum = sum+num;
  }
  avg = sum/n;
  putw(avg, f2);
  fclose(f1);
  fclose(f2);
```

```
    f2 = fopen("average.txt", "r");
    printf("Average of the numbers");
    printf("%2d", avg);
    fclose(f2);
}
```

Output:

```
Enter the number of elements
5
Enter the elements
 5
 10
 15
 6
 4
Average of the numbers
8
```


Advantages:

* Simple
* Easy to handle
* can be stored on magnetic disks as well as magnetic tapes
* used for batch-oriented applications.

Disadvantages:

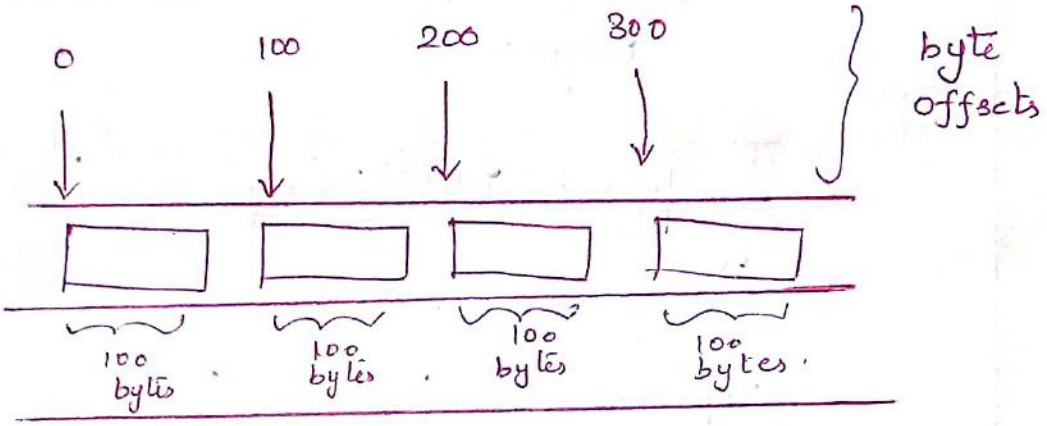
* Data can be read only sequentially.
* Does not support update operation.
* cannot be used for interactive applications.

5.4 RANDOM ACCESS FILE

R1

* Individual records that write to and read from a random-access file are normally fixed in length and may be accessed directly without searching through other records.

C's view of a Random-Access File.



Functions:

* For random access to files of records, the following functions are used
  - i) fseek()
  - ii) ftell()
  - iii) rewind()

Reading Data from a Random-Access File:

fread():

→ used to read a specified number of bytes from a file into a memory.

Syntax:

```
fread(&client, sizeof(struct clientdata), 1, fp);
```

(or)

```
fread(&var, sizeof(var), 1, filptr);
```

Ex:

```
fread &ch, sizeof(ch), 1, fp);
```

* reads number of bytes determined by sizeof() from the file referenced by fileptr, stores the data in var. and returns the number of bytes read.

1 → reads one element.

Random Access functions:

(i) fseek():

* set the position indicator anywhere in the file.

Syntax:

```
fseek(fileptr, displacement, origin);
```

Ex:

```
fseek(fp, 2, 0);
```

fileptr → file pointer associated with the file.

displacement → number of bytes skipped forwards or backwards (+ve or -ve)

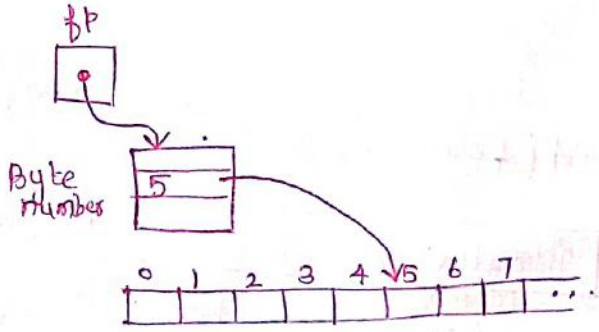
origin → position indicator's relative starting point.

R4

origin values

constant	value	Description
SEEK_SET	0	starts at the beginning of the file
SEEK_CUR	1	starts at the current location of the file
SEEK_END	2	measured from the end of the file

R1



List of common operations

fseek(fp, 0, 0)	Beginning of the file ie) moved 0 bytes forward from beginning of the file.
fseek(fp, 0, 2)	moved 0 bytes forward from end of the file
fseek(fp, N, 0)	moved N bytes forward from beginning of the file
fseek(fp, N, 1)	moved N bytes from current position of the file
fseek(fp, -N, 1)	moved N bytes backward from current position of the file
fseek(fp, -N, 2)	moved N bytes backward from end of the file

RA

ii) ftell() :

* used to determine the value of a file's position indicator.

Syntax:

```
ftell(fileptr);
```

Ex:

```
ftell(fp);
```

iii) rewind() :

* Used to set the position indicator to the beginning of the file.

syntax:

```
rewind(fileptr);
```

Ex:

```
rewind(fp);
```

Writing Data Randomly:

* Uses the combination of
  * fseek and
  * fwrite
  - to store data at specific location in the file.

fseek() → Sets the file position pointer to a specific position in the file.

fwrite() → writes the data.

Syntax:

```
fwrite(&var, sizeof(var), 1, fileptr);
```

Ex:

```
fwrite(&eb, sizeof(eb), 1, fp);
```

* writes number of bytes determined by sizeof() in the file referenced by fileptr.

Ex. Program:

Random Access File.

```
#include <stdio.h>
main()
{
   FILE *fp;
   int pos;
   fp= fopen ("file.txt", "r");
   if(!fp)
   { printf("Error in opening file\n");
     return 0;
   }
   pos = ftell(fp);
   printf("Position of the pointer: %d", pos);
   printf("Content in the file");
   char ch;
   while ( fread(&ch, sizeof(ch), 1, fp) ==1)
   {
     printf(" %c", ch);
   }
  printf(" Position of the pointer %d", ftell(fp));
  rewind(fp);
  printf("Position of the pointer", ftell(fp));
  fseek (fp, 6, 0);
  while (fread (&ch, sizeof(ch), 1, fp)==1)
 {
   printf(" %c", ch);
 }
   fseek ( fp, -6, 2);
```

```
while ( fread( &ch, sizeof (ch), 1, fp)==1)
{
  printf(" %c", ch);
}
fclose (fp);
}
```

o/p:

```
Position of the pointer : 0
content in the file :
Welcome to V V college of Engineering
Position of the pointer : 37
position of the pointer : 0          // After rewind()
e to V V College of Engineering      // forward
eering                               // backward
```

5.5 COMMAND LINE ARGUMENTS

* Any input value passed through command prompt at the time of running a program is known as command line argument.

→ main() designates the entry point of the program

* main() can be defined with formal parameters so that the program may accept command-line arguments.

→ Dos (Disk operating system) uses command line arguments.

* The program must be run from a command prompt.

→ main() allows arguments to be passed from the command line.

Syntax:

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

Two arguments:

i) argc:
- integer variable
- receives the number of command-line arguments that the user has entered.

ii) argv:
- an array of strings
- the actual command-line arguments are stored.

Compile and Run:

* Programs are compiled and run on command prompt.

Steps:

i) Open Command prompt.

ii) Follow the directory where the code saved

iii) compile : c: /Tc / Bin / Tcc pgm.c

iv) Run : c: /Tc/ Bin / pgm.c i/ps.

Ex:

```
#include <stdio.h>
main(int argc, char *argv[])
{
  int i;
  printf("Number of arguments : %d", argc);
  for(i=0; i<argc; i++)
  { printf("%s\n", argv[i]);
  }
}
```

> Tcc pgm.c

> pgm. oxford pradip manas

o/p:

```
Number of arguments : 3
oxford
pradip
manas
```

Ex 2:

```
#include<stdio.h>
main(int argc, char *argv[]
{
   FILE *fp;
   char c;
   fp = fopen(argv[1], "r");
   if (fp!=NULL)
   {
      do
        {
          putchar(c = getc(fp))
        } while (c!='\n');
   }
   else
      printf("Error. File cannot be opened");
}
```

sample.dat


```
>tcc pgm.c
>pgm sample.dat
```

o/p:

Welcome