Project - 9 on comp →

— x —

PSPC

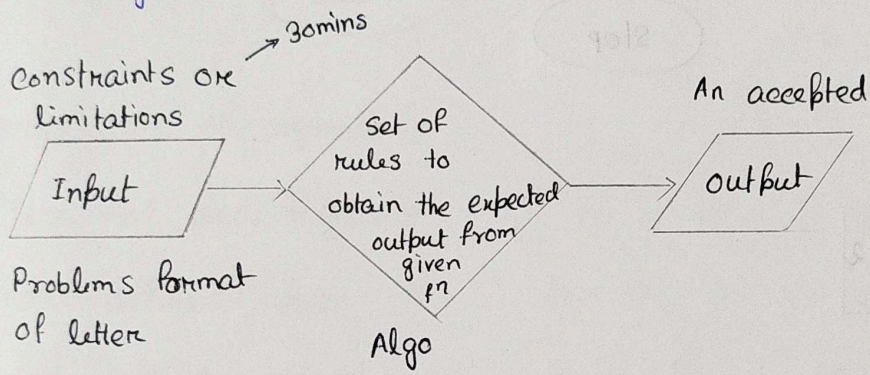
Problem solving with Programming language C.

Algorithms –

An algorithm is the list of instructions and rules that a computer needs to do to complete a task.

Algorithms are simply a series of instructions that are followed, step by step, to do something useful or solve a problem.

what is algorithm



Pseudocode –

Pseudocode is an artificial and informal language that ~~help~~ helps Programmers to develop algorithms. Pseudocode is a "text-based" detail (algorithmic) design tool.

The rule of Pseudocode are reasonably straightforward. All statements showing "dependency" are to be indented. These include while, do, for, if, switch.

Problem: Display whether a number is even or not ?

Pseudocode:

1) Take the number as input.

2) Calculate remainder n%.2 =

3) If remainder is equal to 1

Print '* The no is odd'

else

Print 'The no is even'.

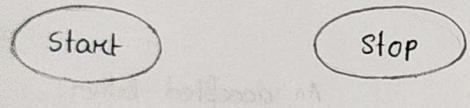
Flowchart –

A flowchart is a graphical representation of an algorithm. Programmers often use it as a programme-planning tool to solve a problem.

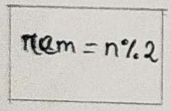
It makes use of symbols which are connected among them to indicate the flow of information and processing.

Symbols used in a flowchart:

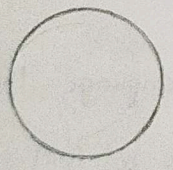
★ Terminal



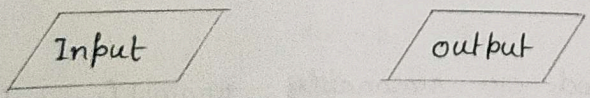
★ Processing



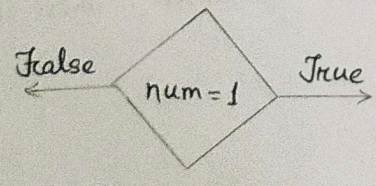
★ Connectors



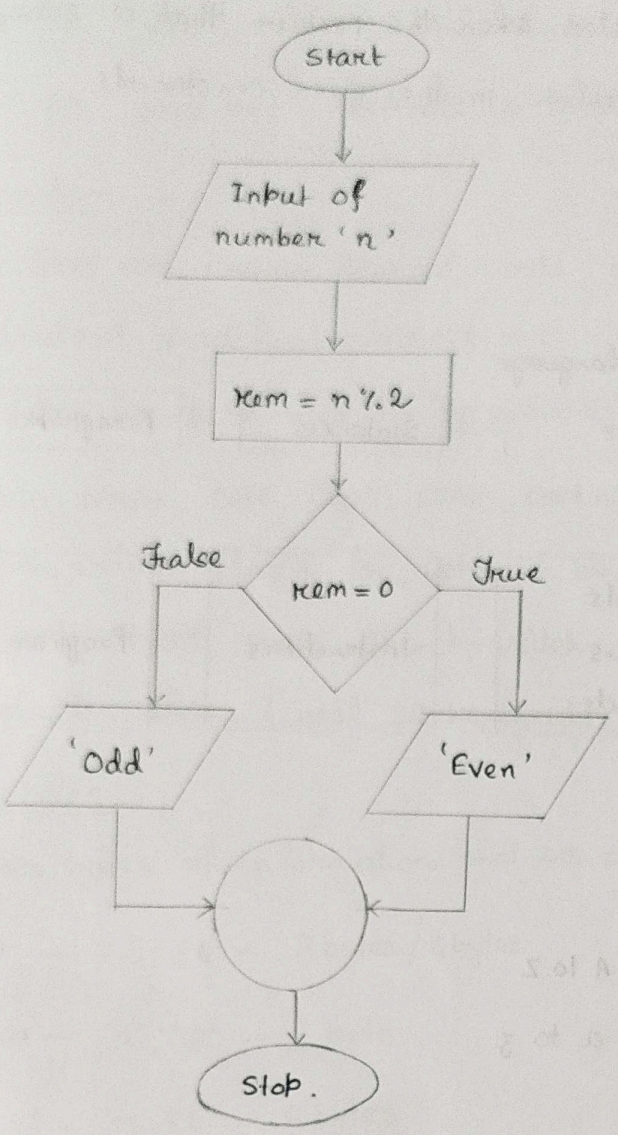
★ Input / output



★ Decision



Flowchart of odd even



Characteristics of algorithms

i) Clear and unambiguous

ii) Well-defined inputs

iii) Well-defined outputs

iv) Finite-ness

v) Feasible

vi) Language independent

In order to write an algorithm, following things are needed as Prerequisite

1) The problem that to be solved by algorithms.

2) The constraints of the problem that must be considered while solving the problem.

3) The input to be taken to solve the problem

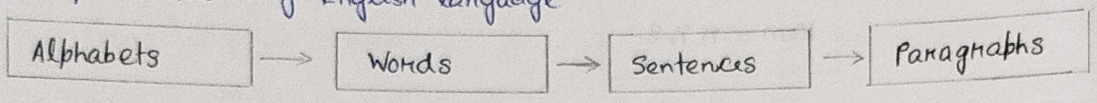
4) The output to be expected when the problem that is solved.

5) The solution to this problem, in the given constraints.

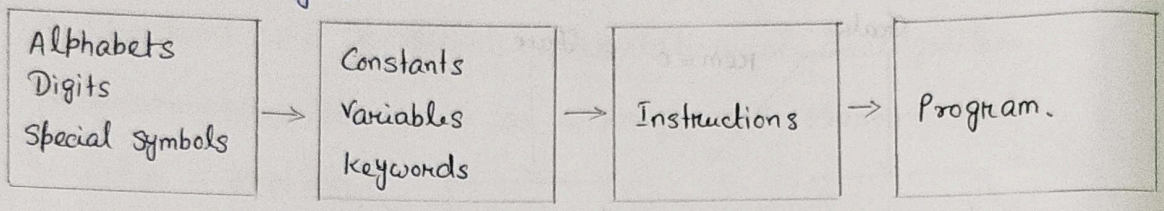
Basics of C

Introduction –

Steps in learning English language



steps in learning C



The C character set –

C uses

i) The uppercase letters A to Z

ii) The lowercase letters a to z

iii) The digits 0 to 9

iv) Certain special characters

As building blocks to form basic program elements.

+ - * / = % & # ! ? ^ " ' - \ |

< > ( ) { } [ ] : ; . , ~

Identifiers –

Identifiers are names that are given to various program elements, such as variables, function and arrays. (to identify or name various program elements)

* Identifiers consists of letters and digits, in any order, except that the first character must be a letter
* Upper-case and lowercase letters are not interchangeable (case sensitive)
* The underscore (_) character can also be included and is considered

to be a letter.

eg: x , y12 , sum_1 , _temperature , names , area , tax_rate , TABLE

1x , area rect , tax$rate → Invalid identifier names.

Keywords –

* There are certain reserved words, called keywords, that have standard, predefined meanings in C.
* They can't be used as programmer-defined identifiers.

auto, break, case, char, const, continue, default, do, double, else, enum, extern, floatn, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while. (Total 32 keywords)

Data types –

* The types of information that we will be handling in C language

Int — 76, 84 — 2bytes / 4bytes

char — 'a', '8' — 1 byte

float — 76.85 — 4 bytes

double — 76.85426821 — 8 bytes

* The basic data types can be augmented by the use of the data type qualifiers short, long, signed and unsigned.
* Every identifiers that represents a number or a character within a C program must be associated with one of the basic data types before the identifier appears in an executable statement.

Constants / Literals –

* There are four basic types of constants in C. They are integer constants, floating-point constants, character constants and string constants.
* Integer and floating-point constants represent numbers. They are often referred to collectively as numeric-type constants.

* The following rules apply to all numeric-type constants.

1) Commas and blank spaces cannot be included within the constants.

2) The constant can be preceded by a minus (-) sign if desired.

3) The value of a constant cannot exceed specified minimum and maximum bounds.

Integer constants -

0, 768, 32568....

~~36.0~~, ~~36,54~~ not integer

5000 U → unsigned int

3676 L → long int

568 UL → unsigned long int

Floating-point constants -

0. 3.56 1.

2E-8 → 2×10^{-8}

5E10 → 5×10^{10}

~~2E10.2~~ ~~2E112~~ not floating-point

Limits/Range : -3.4E-838 to 3.4E+38

character constants -

* A character constant is a single character, enclosed in apostrophes ~~(ie. single quotation marks)~~ (ie single quotation marks).

eg: 'A', '8', '$' etc..

* Escape sequences

'\t' → tab (horizontal)

'\n' → new line

'\b' → backspace

'\v' → tab (vertical)

'\a' → alert

\f → form feed

\" → quotation mark

\' → apostrophe

\? → question mark

\\ → backslash

\0 → null

\r → carriage return

string constants –

A string constant consists of any number of consecutive characters (including none) enclosed in double quotation marks

"String constants is written in double quote"

"line 1 \n line 2 \n line3" → line 1
line 2
line 3

Variables –

* A variable is an identifier that is used to represent some specified type of information within a ~~designed~~ designated portion of the program.
* The data item can then be accessed later in the program simply by referring to the variable name.
* The data type associated with the variable cannot change.

$x = 1$. $x = 2$ $x \neq$ 'c'

Declarations –

* A declaration associates a group of variables with a specific data type.

int y ; float a, b, c ;

* All variables must be declared before they can appear in executable statements.
* A declaration consists of a data type, followed by one or more variable names, ending with a semicolon.

Operators –

* Individual constants, variables, array elements and function references can be joined together by various operators to form expressions a%b.
* C includes a large number of operators which fall into several different catagories.
* The data items that operators act upon are called operands.

result = a + b/c + 10

Arithmetic operators :-

(+) addition (-) subtraction (*) multiplication

(/) Division (%) Remainder (must be integer)

5 % 2 = 1 5.2 % 4.8 = Error

* Among the arithmatic operators, *, / and % fall into one Precedence group and + and - fall into another group.

* Within each of the Precedence group described above, the associativity is left to right.

* An arithmatic operation between an integer and integer always yields and integer result.

* An operation between a real and real always yields a real result.

* An operation between an integer and real always yields a real result. In this operation the integer is first promoted to a real and then the operation is performed. Hence the result is real.

Unary operators -

y = -x

→ ++ & --

x = 12 y = -12

x++ = 13 x-- = 11

increment decrement

Printf ("%d", x) = 12 Printf ("%d", x++) = 13

* Unary operators have a higher precedence than arithmatic operators.

* The also associativity of the unary operators is right to left, though consecutive unary operators rarely appears in elementary programs.

Rational operator -

1) >

2) >=

3) <

4) <=

* These operators all fall ~~into~~ within the same presedence group, which is lower than the arithmatic and unary operators.

* The equality operators fall into a seperate prescedence group, beneath the relational operators.

* The associativity of these operators is left to right.

1) == → whether operand on the left is equal to operand on the right.

!= → whether operand on left is not equal to operand on right.

y = 12 == 12 y = 1 (false)

x = 12 != 12 x = 0 (True)

Logical operators –

a = 2 b = 7

1) && → output = 1, when both left operand & right operand are true or = 1.

2) || → output = 1, when either of the left or the right operand are true or = 1.

(a <= b) && (b > 5) = 1 / True.

* Each of the logical operators falls into it's own precedence group. Logical and has a higher ~~prese~~ precedence than logical or.

* Both precedence groups are lower than the group containing the equality operators.

Conditional operators –

⇒ Expression 1 ? Expression 2 : Expression 3.

Logical or relational operator

If expression 1 is True / 1 ⇒ Expression 2 is selected

False / 0 ⇒ Expression 3 is selected.

y = 6 == 10 ? 5 : -2

False, so we select -2.

Assignment operators -

1) =
2) +=
3) -=
4) *=
5) /=

$x = 10 \quad y = 2$

$x += 20 \Rightarrow x = x + 20$

$\Rightarrow x = 30$

$x -= 5 \Rightarrow x = x - 5$

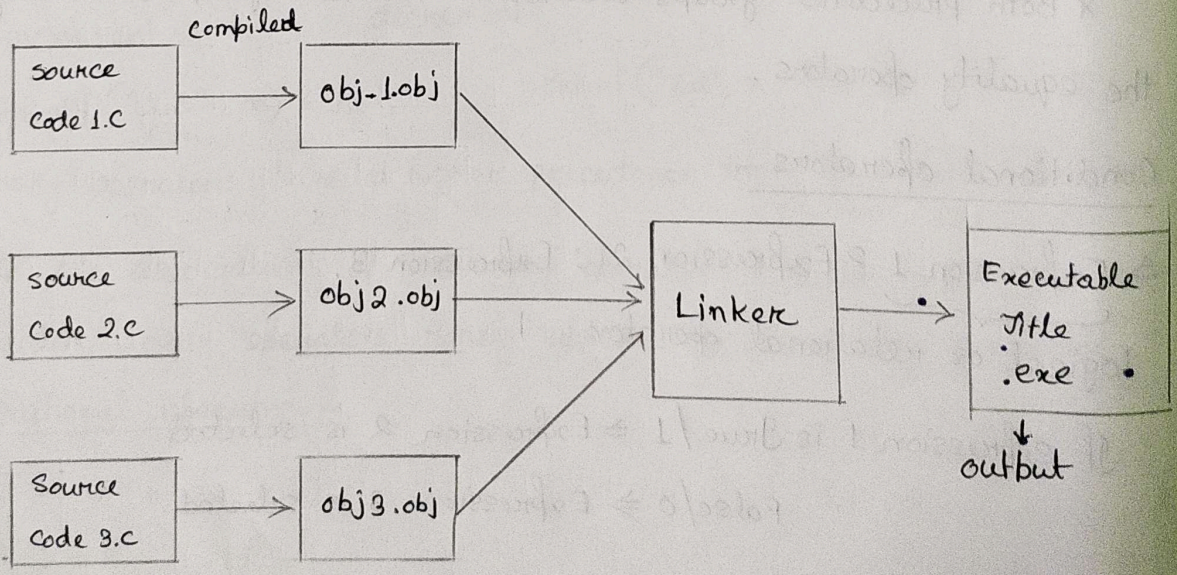
$x /= 2 \Rightarrow x = x/2$

* Assignment operators have a lower precedence than any of the other operator.

* The assignment operations have a right to left associativity.

Operator category	Operators	Associativity
Unary operators	- ++ --	R → L
Arithmatic multiply, divide and remainder	* / %	L → R
Arithmatic add and substract	+ -	L → R
Relational operators	< <= > >=	L → R
Equality operators	== !=	L → R
Logical and	&&	L → R
Logical or	||	L → R
Conditional operator	? :	R → L
Assignment operator	= += -= *= /= %=	R → L

Operator category follows priority order.



Source code -

C program is a set of instructions written in C programming language to perform a specific task. This program is called source code.

Compilation and object code -

★ Compilation is the process that the computer takes to convert a high-level programming language into a machine language that the computer can understand. The software which performs this conversion is called a compiler.

★ An object code is generated after compiling the source code.

★ The CPU cannot directly execute the object file

Linker and Executable code

Linker takes one or more object files generated by the compiler and combines them into a single executable file. Furthermore, it links the other program files and functions the program requires.

object files do not have the definition of functions used in the c-code But in -exe or executable file these definitions will be includes.

Difference between object file and executable file.

object file	Executable file.
i) A file that contains an object code that has relocatable format machine code, which is not directly executable.	i) A file that can be directly executed by the computer and is capable of performing the indicated tasks according to the encoded instructions.
ii) An intermediate file.	ii) An ~~intermediate~~ final file
iii) A compiler converts the source code to an object file.	iii) A linker links the object files with the system library and combines the object files together to create an executable file.
iv) Cannot be directly executed by the CPU.	iv) Can be directly executed by the CPU.

Syntax Error -

★ Errors that occurs when you violate the rules of writing c/c++ Syntax are known as syntax error.

★ All these errors are detected by compiler and thus are known as

Compile-time errors.

* Most frequent syntax errors are :

i) Missing paranthesis (})

ii) Printing the value of variable without declearing it.

iii) Missing semicolon like this.

Logical errors :

* On compilation and execution of a program, desired output is not obtain when certain input values are given.

* These types of errors which provides incorrect output but appears to be error free are called logical errors.

* These errors solely depends on the logical thinking of the programmer and easy to detect if we follow the line of execution and determine why the program takes that path of execution.

Conditional Branching -

Introduction -

* C language must be able to perform different sets of actions depending on the circumstances.

* C has three major decision making ~~statement~~ instructions -

i) The if statement

ii) The if-else statement, and

iii) The switch statement.

* ~~One that~~ A fourth, somewhat less important structure is the one that uses conditional operators.

1) The if statement

* The general form of if statement looks like this :

```
if (Relational, Equality or logical expression)
    Execute this statement;
if (expression)
{ ____________
  ____________ (multible ~~Ex~~ statement)
}
```

* The if statement by itself will execute a single statement, or a group statements, when the expressions following if evaluates to true.

* It does nothing when the expression evaluates to false.

ii) If-else statement -

Using if-else statement we execute one group of statements if the expression evaluates to true and another group of statements if the expression evaluates to false.

```
If (Expression)
  {
    Multiple statements
  }
else:
  {
    Multiple statements
  }
```

Expression is True

Otherwise

iii) Nested if-else

Even < 100

```
If (no%2 == 0)
  {
    If (n < 100)
    {
        Printf(" No is even & less than 100")
    }
    else
    {
        Printf("No is even & greater than equal to 100")
    }
else
{
    if (n > 100)
    {
        Printf("No is odd and less than 100")
    }
```

The else if clause

no 0-35, 35-70, 70<

```
If (no>70)
   Printf ("Number is greater than 70")
elseif (no>35)
   Printf ("Number is greater than 35")
   else
      Printf ("Number is between 0-35")
```

Switch -

★ The control statement that allows us to make a decision from the number of choices is called a switch, or more correctly a switch-case-default, since these three keywords go together to make up the control statement.

```
switch (integer/character expression)
{
   case constant 1:   37
      do this;
   case constant 2:   17
      do this;
   case constant 3:   54
      do this;
   default:
      do this;
}
```

If this expression results in 17, then these statements are executed

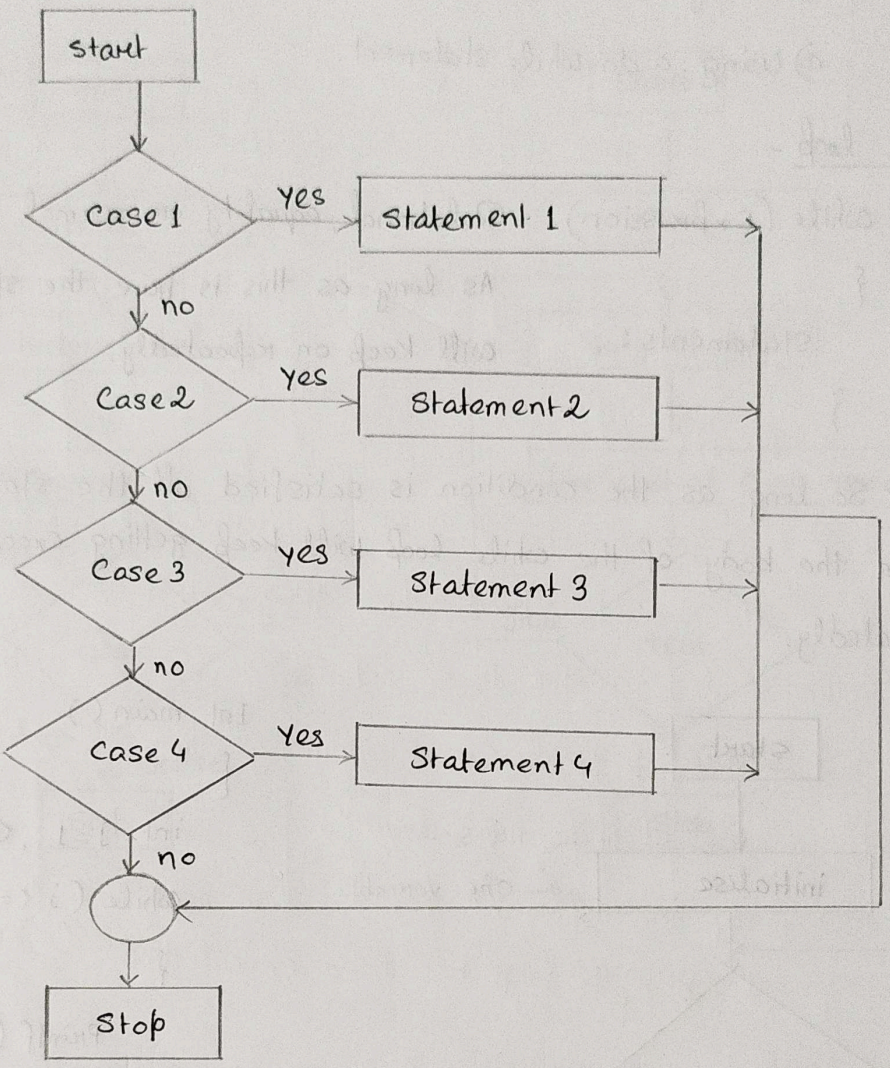
Whenever the expression in switches results in none of the cases, then default is activated.

These statements are executed.

★

```
switch (choice)
{
   Case 1:
      Statement 1;
      Break;
   Case 2:
      Statement 2;
      Break;
```

```
Case 3:
    Statement;
    break;  ——→ Tell the Program ~~Statement~~ to
                 break out of switch.
Case 4:
    Statement;
    break;
}
```



When to use if -

i) If I have only 2 or 3 conditional branching.

ii) The conditional statements operate on a range of values.

When to use switch -

i) When there is a lot of branching based on conditions.

ii) When a ~~many~~ menu type of program is ~~listed~~ utilized.

Loops -

* The ~~versatility~~ versatility of the computers lies in its ability to perform a set of instructions repeatedly.

We want a particular set of instruction to be executed repeatedly until some condition has been satisfied.

* There are three ~~more~~ methods by way of which we can repeat a part of a program. They are:

a) Using a for statement

b) Using a while statement

c) Using a do-while statement.

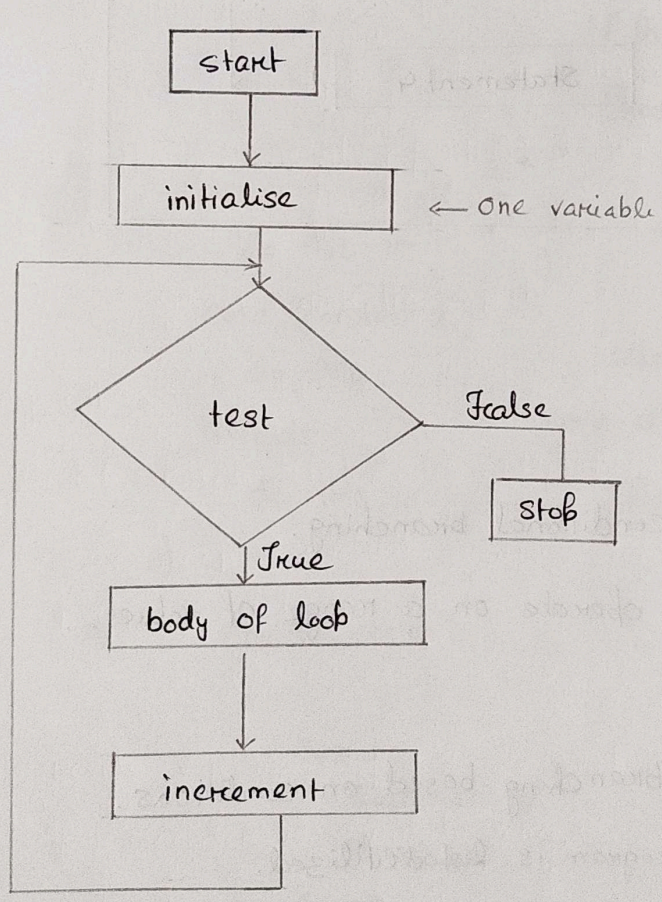
while loop -

```
while (Expression)
{
    statements;
}
```

→ Relational, equality or normal.

As long as this is true the statements will keep on repeatedly.

* So long as the condition is satisfied all the statements within the body of the while loop will keep getting executed repeatedly.



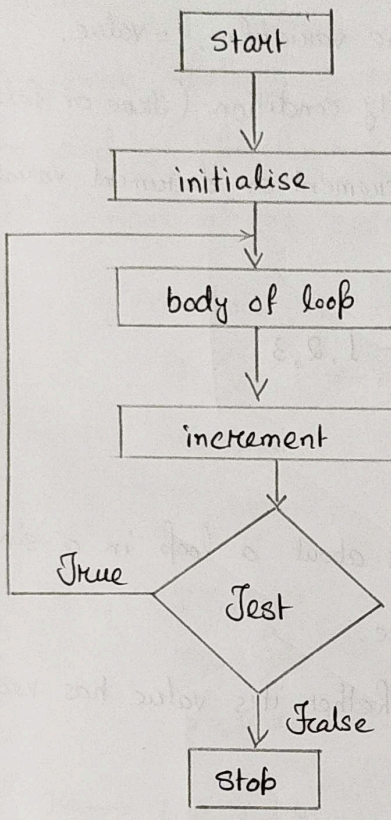

```
Int main()
{
    int j=1, check=10;
    while (j<=check)
    {
        Printf ("%d", j);
        j++;
    }
return 0
}
```

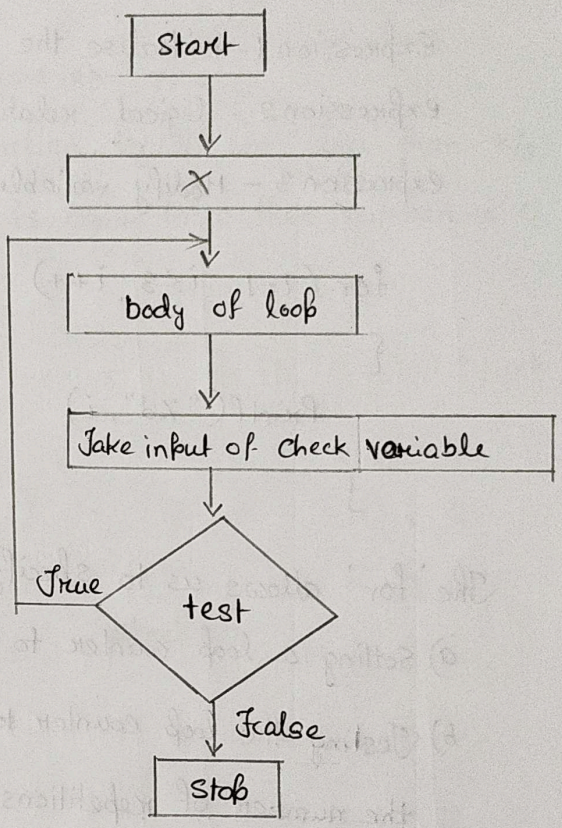
=1
2
3
4
5
6
7
8
9
10

Do-while -

```
Do
{
    statement
}
while (expression)
```

Relational, equality base or logical.






```
Int main ( )
{
    int j = 1 , check = 10 ;
    do
    {
        Printf ("%d", j) ;
        j++ ;
    }
    while (j <= 10) ;
    return 0
}
```

For loop -

The general form of the for statement is

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

expression 1 - initialise the value of some variable, i = value.

expression 2 - logical, relation & equality condition (True or false)

expression 3 - Modify variable i. usually increment or decrement variable i.

```
for (i=1, i<3, i++)
{
    Printf("%d", i)
}
```

output - 1, 2, 3

The 'for' allows us to specify three things about a loop in a single line:

a) Setting a loop counter to an initial value.

b) Testing the loop counter to determine whether it's value has reached the number of repetitions desired.

c) Increasing the value of loop counter each time the program segment within the loop has been executed.

Algorithms -

Prob 1 : Print all even numbers between 1 to 100

Some mech to evaluate or utilize all no numbers which are between 1 & 100.

```
for(i= 1 ; i<=100 ; i++)
```

Devide variable i by 2 & see what is the remainder.

If remainder is equal to zero.

Then print the numbers.

Prob 1 Print all even no between 1 to 100

1. Start

2. Initialise the variable (i=1)
3. Check for condition (To consider all numbers less than 100)
4. Check whether present no is even or not.
5. Print if even
6. Increament value of variable (i++)
7. Goto step 3
8. Stop.

Prob 2: Check whether a number is prime or not.

x and on performing module operation by all nos less than x/2 if in none of the cases remainder is equal to 0 then number is 0.

```
for (i=2 ; i <= x/2 ; i++)
    if (x % i == 0)
        check 1
        break;
```

7... 7%2≠0 7%3≠0 7%4≠0 7%5≠0 7%6≠0 ~~7%7~~ check=0

14 → 1, 2, 7, 14

14%2 = 0

%3, %4 → won't check (Break)

Prob 2: check whether a number is prime or not

1. Start
2. Take input of number & initialise the variables (i=2, check=0)
3. Check for condition (i is less than half of number)
   If true goto step 4, otherwise goto step 6.
4. If number is divisible by i, check++ & goto step 6.
5. Increase i & goto step 3.
6. If value of check is equal to zero
   Print Prime
   else Print not Prime
7. Stop.

Prob 3 : Print all prime numbers between 2 to 100.

1. ~~For~~ for loop to check all numbers between 2 & 100

```
for (i=2; i<= 100 ; i++)
```

2. for loop to check all numbers between 2 & i

```
for (j =2; j <= i/2 ; j++)
{
   if (i%2 ==0)
   {
      check ++
      break ;
   }
}
```

Prob 3 : Print all prime numbers between 2 & 100

1. Start
2. Initialise the variable (i=2)
3. Check 1st for loop (i≤ 100)

   If true goto step 4 otherwise goto step 9
4. Initialise the variable (j = 2, check = 0)
5. Check 2nd for loop (j ≤ i/2)

   If true goto step 6 otherwise goto step 8.
6. If i is divisible by j

   check ++ & goto step 8
7. Increment j (j++) & goto step 5
8. If check is equal to 0.

   Print i
9. Increment i and goto step 3.
10. Stop.

Arrays –

Introduction:

★ Often times we require the processing of multiple data item that have common characteristics (eg. a set of numerical data, represented by x1, x2, ... xn)

eg - marks, weights → It is difficult to declare or use large no of variables.

★ In such situations it is often convenient to place the data items into an array, where they will all share the same name (eg: x)

★ They must all be of the same type or the same storage class.

Defining an array –

```
Int weights [30];
```

weights [0], weights [1], weights [2], ..., weights [29]

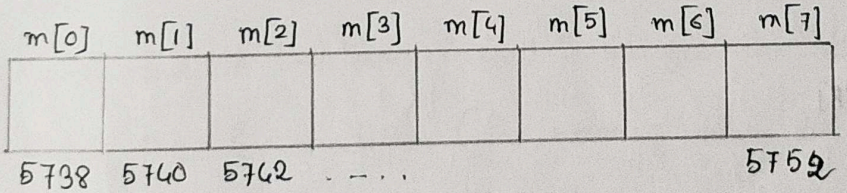
```
for (i=0, i<30, i++)
{
    Printf ("Enter the value of weight no %d", i);
    Scanf ("%d", & weights [i]);
}
```

weights [2]

```
Int weights [5] = {58, 69, 72, 83, 45}
```

Array in memory –

```
Int m [8];
```

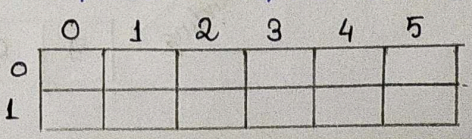


5738 + 2 × 7 = 5752

2D Array –

★ A 2 dimensional array will require two pairs of square brackets, a three dimensional array will require 3 pairs of square brackets, and so on.

```
Int MAt [2] [6];
Int Dim 3 [2] [4] [6];
```



★ for 1 dimensional array to access the values or data, we used single for loop.

★ for 2 dimensional array we will use two for loops.

1st loop compounds to the row

2nd loop compounds to the colums.

```
for (i=0; i<2 ; i++)
    for (j=0; j <6, j++)
    {Printf/scanf }
```

★ Addition of 2D matrices -

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

$c_{11} = a_{11} + b_{11}$

$c_{23} = a_{23} + b_{23}$ $\longrightarrow$ $c_{ij} = a_{ij} + b_{ij}$

★ Multiplication of 2D matrices -

$$\begin{bmatrix} a_{11} & a_{22} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

$c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31}$

$c_{23} = a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{33}$

$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + a_{i3}b_{3j}$

$$c_{ij} = \sum_{K=1}^{3} a_{ik} b_{kj}$$

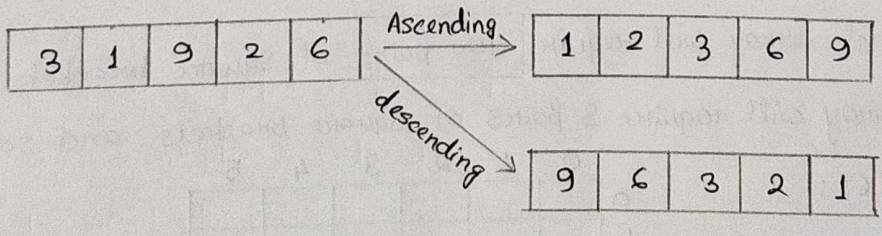
→ one 'for' loop for i

→ one 'for' loop for j

→ One 'for' loop for k

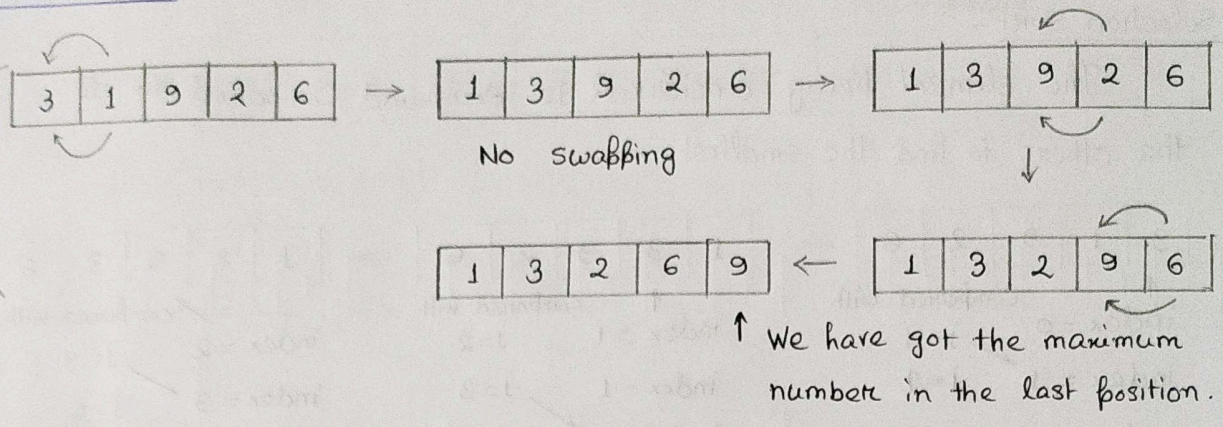
Basic Algorithms -

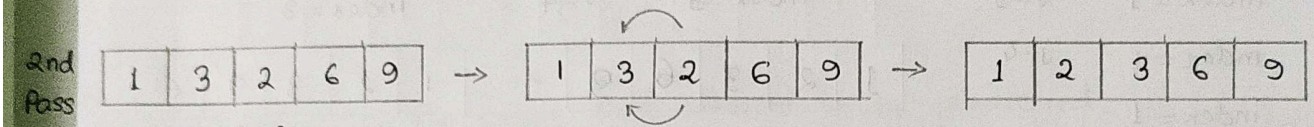
Introduction -



	a	b	temp	a=5, b=7
temp = a	5	7	5	
a = b	7	7	5	
b = temp	7	5	5	

Bubble sort -





3rd, 4th pass as well.

Bubble sort -

$(n-1)\left[(n-1)+(n-2)+(n-3).....\right]$

$\Rightarrow n^2 - xn - y$

$\Rightarrow$ Big order notation ie (n^2)

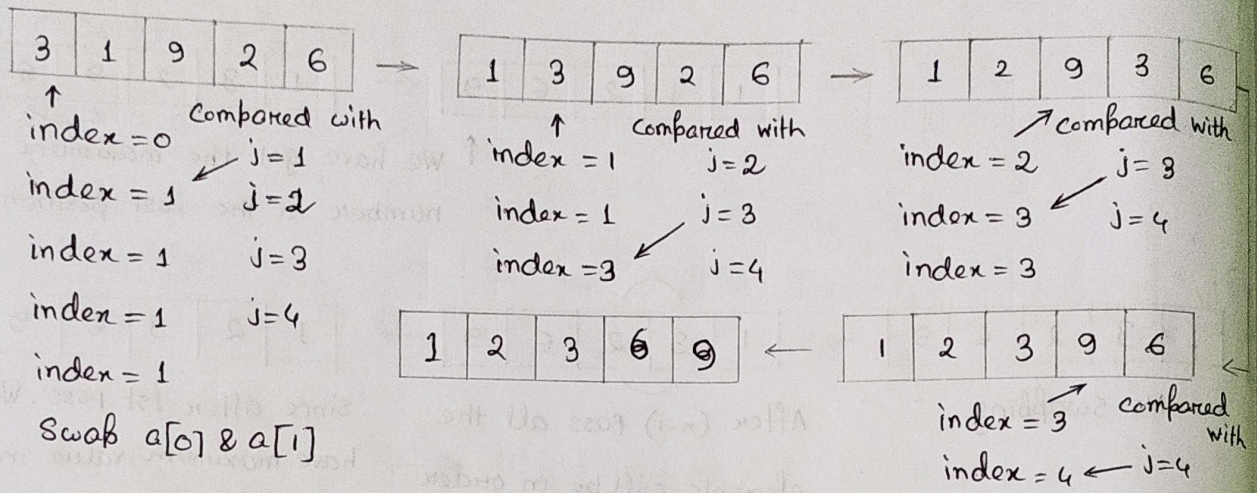
Algorithm -

1. Start
2. Take input of the array and initializes the variables $(i=0)$
3. Check the for condition (for running outer loop n-1 times) ← Passes
   If true goto step 4 otherwise goto step 9.
4. Initialize variable $(j=0)$
5. Check the for 1 condition (for running loop n-i-1 times)
   If true goto step 6 otherwise goto step 8
6. If $a[j] > a[j+1]$
   Swap the two values.

7. Increment j & goto step 5
8. Increment i & goto step 3
9. Display the sorted array.
10. Stop

Selection Sort –

The element being considered is ~~compound~~ compared to all the others to find the smallest.



On performing selection sort, we divide the array into sorted & unsorted parts.

→ One for loop to find the smallest element in the array.

→ One more for loop to move the element being considered.

Algorithm –

1. Start
2. Take input of the array & initialize the variables (i=0)
3. Check the 'for' ~~loop~~ condition for outer loop (for running loop n-1 times)
   If ~~the~~ true goto step 4 otherwise goto step 10.
4. Index = i & initialise variable (j=i) (from i to n)
5. Check the 'for' condition for inner loop (for running loop n-i times)
   If true goto step 6 otherwise goto step 8.
6. If a[index] > a[j]
   Change index to j
7. increment j and goto step 5
8. Swap ar[i] & ar[index]
9. increment i & goto step 3
10. Print the sorted array.
11. Stop.

Sequential/Linear search algorithm

* Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

Array

16	29	32	14	56	9	78	101

```
for (i=0, i<8, i++)
{
    if (arr[i] == 9)
```

Algorithm –

1. Start
2. Take input of the array and element to be searched.
3. Initialise the variables (i=0), (check=0)
4. Check for loop condition (i< length of array)
   If true goto step 5, otherwise goto step 6.
5. Check whether current element of array is equal to element to be searched.
   If yes, check=1 & goto step 6.
   NO, i=i+1 & goto step 4.
6. If check is equal to 1
   Print element found
   else
   Print element not found.
7. Stop.

Big O Notation or Complexity of an algorithm –

Big O notation or complexity of an algorithm tells or denotes about how much time or how much steps will be taken to solve a problem, in worst case scenario.

$O(n) \Rightarrow 5$ → No of elements in array = 8

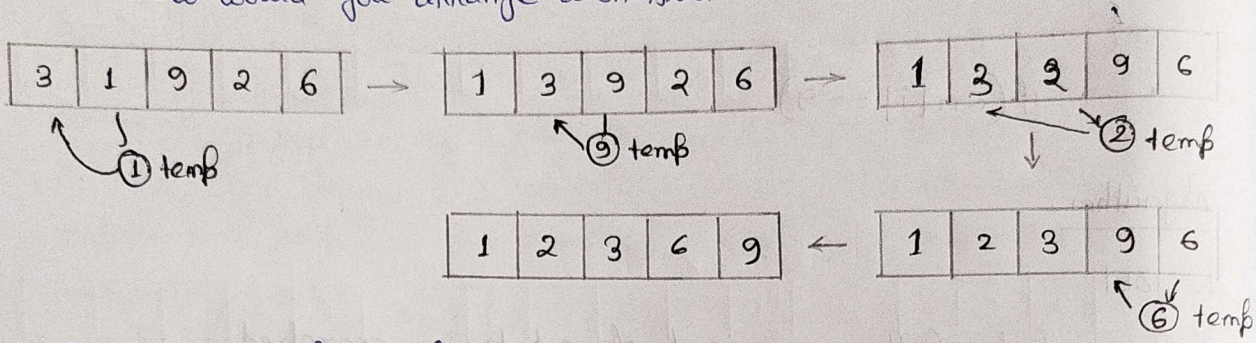
16	29	32	14	56	9	78	101

$O(n^2) \Rightarrow 25$ steps

n elements the no of steps ie. required to solve the prob is n^2.

Insertion Sort –

How would you arrange a shuffled set of playing cards.



Requirement of 2 loop

1st outer loop → to go through all locations

2nd inner loop → to go through all locations less the location of outer loop

Big O Notation –

1	3	5	6	7	8

Already sorted, only outer loop will run (n-1)

8	7	6	5	3	1

$1 \times 1 + 2 + 3 + 4 + \ldots n$

$= \frac{n(n+1)}{2} \Rightarrow \frac{n^2}{2} + \frac{n}{2}$

$O(n^2)$

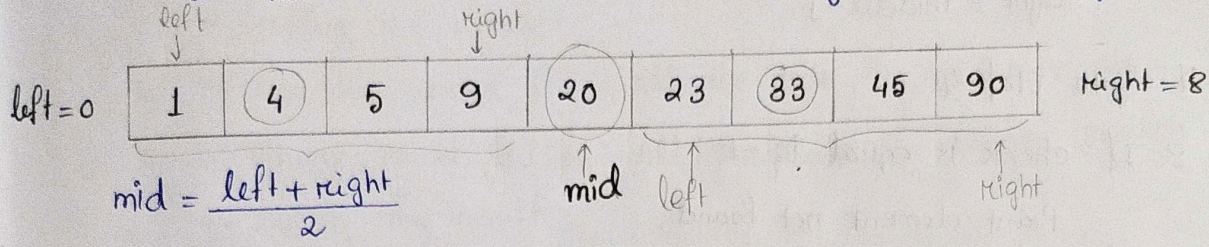
Algorithm –

1. Start
2. Take input of the array and initialise the variables (i=1)
3. Check ~~for~~ the 'for' condition for outer loop (for running loop n-1 times)
   If true goto step 4 otherwise goto step 10.
4. Store arr[i] is temp variable & initialise variable (j = i-1)
5. Check the while condition for inner loop (arr[i] > temp and j >=0)
6. If true goto step 6 otherwise goto step 8.
6. arr[j+1] = arr[j]
7. Decrement j by 1 & goto step 5.

8. arr [j+1] = temp
9. Increment i & goto step 3.
10. Print the sorted array
11. Stop

Binary Search –

The numbers should be in ascending or descending order



when right < left, stop.

Big O Notation –

$O(\log n)$

1st iteration → n

2nd iteration → $\frac{n}{2}$

3rd iteration → $\frac{n}{2 \times 2} = \frac{n}{2^2}$

. . . . . .

. . . . . .

$k+1$ iteration → $\frac{n}{2^k} = 1 \Rightarrow n = 2^k$

$\Rightarrow \log_2 n = \log_2 2^k \Rightarrow k = \log_2 n$

Binary search → $O(\log n)$

Linear search → $O(n)$

Insertion sort → $O(n^2)$

Selection sort → $O(n^2)$

Bubble sort → $O(n^2)$

Order of speed in descending manner

Algorithm –

1. Start
2. Take input of the sorted array and initialise variable (left = 0, right = n, middle, check = -1)
3. Take input of element to be searched (ele)
4. Check the while condition (As long as left <= right)

   If true goto step 5, otherwise goto step 8.

5. middle = $\frac{left + right}{2}$

6. If arr[middle] is equal else

check = middle ← Break.

else if arr[middle] < el

left = middle + 1

else

right = middle − 1

7. goto step 4

8. If check is equal to −1

Print element not found.

else

Print element found at location check.

9. Stop.

Functions –

Introduction –

→ scanf(), printf(), sqrt(), strcpy(), strcmp()

* C also allows programmers to define their own functions for carrying out various individual tasks.

* The use of programmer-defined functions allows a large program to be broken down into a number of smaller self-contained components, each of which have some unique, identifiable purpose.

Advantages –

* The use of a function avoids the need for redundant (repeated) programming of the same instructions.

```
Int main()
{
    4 statements
    Some other statements
    4 statements
    Some other statements
    4 statements
}
```

Into a function

Whenever the 4 statements are required again we will just call the function.

* The logical ~~quality~~ clearity resulting from the decomposition of a program into several concise functions, where each function represents some well defined part of the overall problem.

```
Int main ()
{
 → Input of arrays and declear variable.
 → Display the array
 → Insertion sort ( ) ← All the code for insertion sort
 → Binary search ( ) ← All the code for binary search
 → Display the result
}
```

* The use of functions also enables a programmer to build a customized library of frequently used routines or of routines containing system-dependent features

Defining a function –

* A function definition has two components : the 1st line (including the argument declearations), and the body of the function.

```
Datatype    Function_Name        (Int no, float avg)
  Int          Avg_def (
{
 .....
 .....
}
Int main ()
{
 .....
 .....
}
```

Scope of a variable –

```
Int c1 ; → global variable
Int fun1 (Arguments)
{  Int a1 ; → is not accessible in main ()
   Statements
}                          Local variable
Int main ()
{ Int b1 ; → is not accessible in fun1
```

static variable

```
Int fun2 ()
{
    Int d1 =0 ; → static
    d1++
    Printf ("%d", &d1)
}
```

1 → 1st call to func
2 → 2nd call to func
3 → 3rd call to func

Function Prototyping –

★ Many programmers prefer a 'top-down' approach, in which main appears ahead of the programmer-defined function definition.

→ Declare the function

```
{
    body of function
}
Int main()
{
    body of main program
}
Int fun1()
{
    body
}
Int fun2()
{
    body
}
```

★ This can be confusing to the compiler, unless the compiler is first alerted to the fact that the function being accessed will be defined later in the program.

★ A function prototype resembles the 1st line of a function definition.

```
Int fun1 (Int var1, int var2);
```

Datatype — function name — Arguments

* The argument names can be omitted (though it is not a good idea to do so); however the argument data types are essential.

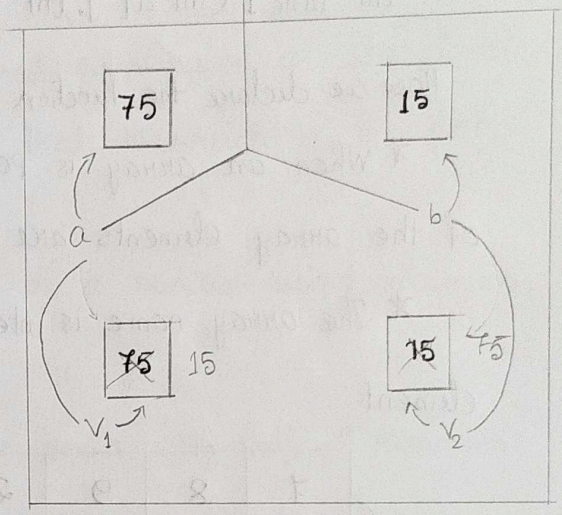
```
Int  func1 (Int , Int) ;
```

no need for variable names or argument names

Call by value -

```
Int func1 (Int v1, int v2)
{
    Source statement
}
Int main ()
{
    a = 75, b = 15;
    result = func1 (a,b)
}
```

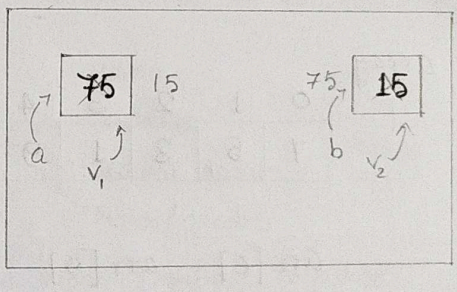
Pointing or string to a location in memory



whatever change occuring in V_1 & V_2 is not replaced in var a & b.

Call by reference -

```
Int func1 (int *v1 , int *v2)
{
    body
}
Int main ()
{
    Int a = 75 , b = 15
    result = func1 (&a , &b)
}
```



When only operation is comed out of V_1 & V_2. If of causes the value of a & b to change as well

Passing an array to an function -

* The manner in which the array is passed differs markedly, however, from that of an ordinary variable.

* To pass an array to a function, the array name must appear by itself, without brackets or subscripts, as an actual argument within the function call.

func1 (arr) ← How we call function from main()

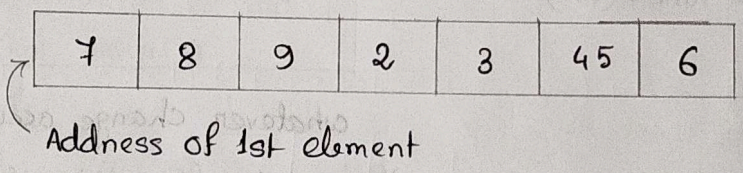
* When declaring a one dimensional array as a formal argument, the array name is written with a pair of empty square brackets.

Int func1 (int a[ ], int size)

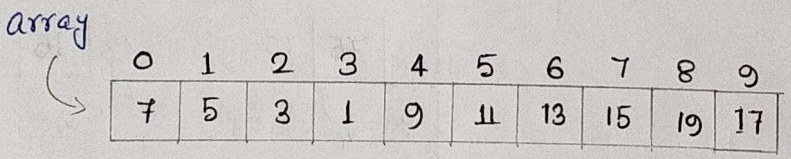
How we declare the function — Have to also send seperately the size of arr

* When an array is passed to a function, however, the values of the array elements are not passed to the function.

* The array name is interpreted as the address of the 1st array element



* If an array implement is altered within the function, the alternation will be recognised in the calling portion of the program. (No need to return the array back to main program)



length → 10
1st element → 0
last element → 9

arr[0] arr[9]

i length-1

arr[1] arr[8]

i length-i-1

temp = arr[0]

arr[0] = arr[9]

arr[9] = temp

Recursion –

Introduction –

★ A function is called 'recursive' if a statement within the body of a function calls the same function.

```
Int func1 (int val)
{
    temp = func1 (val2);
```

★ Recursion may seem strange and complicated at 1st glance.
→ After experience it will be of clear that recursion is the best methodology to solve some problems.

Q) write a program to obtain factorial of a number using recursion.

Q $n! = n \times (n-1) \times (n-2) \ldots \times 1$ ★

Q) write a program to display fibonacci series with help of recursion.

0 1 1 2 3 5 8 .....

Structures –

Introduction –

Datatypes – Int, char, float & double

★ In real world we deal with entities that are collection of things, each things having it's own attributes.

Ex of students → Have various attributes
- Name
- Address
- Class & section
- Mark / grades

A user define database.

★ A structure contains a number of data types grouped together. These data types may or may not be of the same type.

Declaring a structure –

Keyword: struct — Name of user defined database: student

```
struct student
{
    char name[20]
    char address[20]
    Int total mark;
};
```

Declare / define the basic datatype of structure.

```
Int main()
{
    Int marks, n;
    Struct student  S1, S2 ....;
}
```

Accessing structure elements -

```
Struct student
{
    char name[20]
    char address[20]
    Int total mark;
};
Int main()
{
    Struct student S1;
    Printf("Enter student name");
    Scanf("%[^\n]", S1.name)
    .....
    .....
    Printf("Total marks of the student %d", S1.total_marks);
    .....
}
```

Passing Structure to function -

Prob: We want to display grade of student based on total_marks.

1st method → To pass the variable of user defined datatypes to the function

S1 → to the function

```
Datatype of function name(Struct student s)
function
{
    Body of the function
}
within main
  function name(S1)
```

2nd method → To pass the variable relevant to total marks to the function

S1.total_marks to the function

```
Datatype of function name (Int marks)
function
{
    body of the function
}
Main program
function name (S1.total_marks)
```

Array of structure –

```
Int main()
{
    Struct student S1[5]
    int i;
    for (i=0; i<5; i++)
    {
        Printf("Input details about %d students", i+1);
        Scanf("%[^\n]", S1[i].name)
```

Pointers –

& and * operators –

& – "Address of" operator

* – "Value at address" operator

```
Input –
int i = 5;
Printf("The value stored in variable i is %d", i);
Printf("\nThe address of variable i is %d", &i);
Printf("\nThe value stored in the address of variable i is %d", *(&i));
```

output –

The value stored in variable i is 5

The ~~value~~ address of variable i is 6422044

subject to variation based on present memory status.

i = 5

&i → 68524

Scanf("%d", &i)

*(&i) = i

*68524 = 5

The value stored in the address of variable i is 5

Pointers –

Pointer is a special type of variables that is used to store memory address.

Datatype *Var_name

char *ch;

Address that is stored in ch contains a character

integeres

Input –

```
int i=5;
int *j;
j = &i;
Printf("\n the value stored in variable j is %d", j);
Printf("\n the address of variable j is %d", &j);
Printf("\n the value to which the pointer points to %d", *j);
```

output –

the value stored in variable j is 6422044

the address of variable j is 6422032

the value to which the pointer points to 5

value of i is 5 which is stored as 6422044 and value of j is 6422044 which is stored as 6422032.

Call by reference –

```
Void swap(int *n1, int *n2)
{
    int temp;
    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
int main()
{
    int var1, var2;
```

```
    Printf ("Enter two numbers");
    Scanf ("%d %d", &Var1, &Var2);
    Printf ("Before swaping : Var1 = %d and Var2 = %d", Var1, Var2);
    Swap (&Var1, &Var2);
    Printf("\n After swaping: Var1 = %d and Var2 = %d", Var1, Var2);
    Return 0;
}
```

এটা এটা করে দুবার লিখা

(Here is a error, check the code in computer)

Input –

```
int i = 55, *ii;
float f = 5.55, *ff;
Char c = 'R', *cc;
ii = &i;
ff = &f;
cc = &c;
Printf("\n the address contained in ii is %d", ii);
Printf ("\n the address contained in ff is %d", ff);
Printf ("\n the address contained in cc is %d", cc);
Printf("\n the value of to which the pointer ii points to %d", *ii);
Printf("\n the value to which the pointer ff points to %d", *ff);
Printf("\n the value to which the pointer cc Points to %d", *cc);
return 0;
```

output –

the ~~adress~~ address contained in ii is 6422020

the address contained in ff is 6422016

the address contained in cc is 6422015

the value to which the pointer ii points to 55

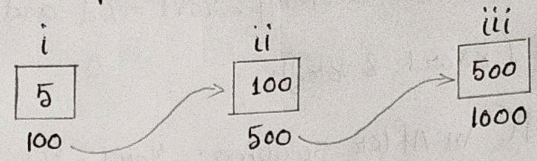
the value to which the pointer ff points to 5.550000

the value to which the pointer cc points to g

Double indirection –

```
int i = 5;
int *ii;
int **iii;
int ***iiii;
```

It is a string address of a variable which itself is a pointer which ~~itself~~ points to a value.



Input –

```
int i=5;
int *j, **k;
j = &i;
k = &j;
Printf("the value stored in the variable i is %d", i);
Printf("\n the address of the variable i is %d", &i);
Printf("\n the value stored in the ~~variable~~ address of variable i is %d", *
Printf("\n\n\n the ~~variable~~ value stored in variable j is %d", j);
Printf("\n the address of variable j is %d", &j);
Printf("\n the value to which the pointer points to %d", *j);
Printf("\n\n\n the value stored in variable k is %d", k);
Printf("\n the address of variable k is %d", &k);
Printf("\n the value of to which the pointer k points to %d", *k);
Printf("\n the value to which the pointer to pointer k points to %d", **k
```

Output –

```
the value stored in variable i is 5
the address of variable i is 6422044
the value stored in the address of variable i is 5

the value stored in variable j is 6422044
the address of variable j is 6422032
the value to which the pointer pointer to 5.

the value stored in variable k is 6422032
the address of variable k is 6422024
the value to which the pointer k points to is 6422044
the value to which the pointer to pointer k points to is 5.
```

Pointer arithmatic

Pointer arithmatic is not allowed.

i) Addition / subtraction of 2 pointer are not allowed.

ii) Multiplication / division of a pointer & a no also not allowed.

int *j

j = j+4

= 6432 + (4) → 4x4

= 6444

c → 16

Char *c;

c = c+1;

= 154

int → size of 4 byte

float *K

K = K+1

= 136

Input -

```
int i=55,*ii;
float f=5.55,*ff;
char c='g',*cc;
ii=&i;
ff=&f;
cc=&c;
Printf("the value to which the pointer ii points to %d",*ii);
Printf("\nthe value to which the pointer ff points to %d",*ff);
Printf("\nthe value to which the pointer cc points to %d",*cc);
Printf("\nthe value to address contained in ii = %d and ii+1 = %d",ii,ii+1);
Printf("\nthe address contained in ff = %d and ff+5 = %d",ff,ff+5);
Printf("\nthe address contained in cc = %d and cc-1 = %d",cc,cc-1);
```

output -

the value to which the pointer ii points to 55

the value to which the pointer ff points to 5.550000

the value to which the pointer cc points to g

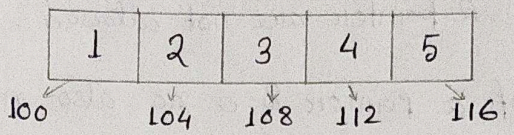
the address contained in ii = 6422020 and ii+1 = 6422024

the address contained in ff = 6422016 and ff+5 = 6422036

the address contained in cc = 6422015 and cc-1 = 6422014

Pointers and Array -

a) Array elements are always stored in contiguous memory locations.



b) A pointer when incremented always points to an immediately next location of it's type.

Input -

```
int arr[5] = {1,2,3,4,5} , *i , j=0;
i = arr ------> (arr) is also a pointer with an address stored.
                Memory location of 1st element.
while (j<5)
{
    Printf("\n the element %d has value %d and address = %d\t", j, *i);
      ++;
    j++;
}
```

output -

the element 0 has value 1 and address = 6422016

the element 1 has value 2 and address = 6422020

the element 2 has value 3 and address = 6422024

the element 3 has value 4 and address = 6422028

the element 4 has value 5 and address = 6422032.

```
int c[10];
```

c -> a pointer -> Point to 1st location in array.

Dynamic memory allocation -

arr[5] -> arr using a pointer to access element of a created array.

* It is possible to define the array as a pointer variable rather than as a conventional array.

* The use of a pointer variable to represent an array requires some type of initial memory assignment before the array elements are processed.

1 malloc → ptr_var = (Datatype *) malloc (total size) → n * size of (datatype)

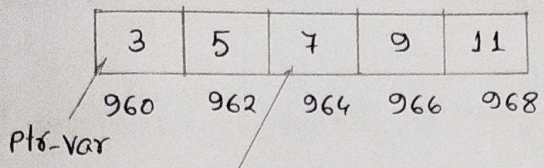
2 calloc → ptr_var = (Datatype *) calloc (n, size of (datatype))

3 free → free (ptr_var) → Deletes memory allocated.

ptr_var = (int *) malloc (total size)

↓

n * size of (int)



Ptr_var[2] → 7

*(ptr_var+2) → 7

int arr[10]

~~int arr[100]~~ int i

scanf (i)

int arr[i]

100 → is requirement of the user

int arr[100,000]

99,990

wastage of space

Memory is left untilised

Input —

```
int main()
{
    int *ptr, no, i;
    Printf("Enter the number of elements to be stored");
    Scanf("%d", &no)
     Ptr =(int *) malloc (no * sizeOf (int));
     Printf("In Enter %d elements to be Stored in the array", no);

     for (i=0; i<no; i++)
     {
          Scanf("%d", Ptr+i);
     }
```

```
Printf("In the elements and their addresses BEFORE sorting are");
for(i=0; i<no; i++)
{
    Printf("In Address of element %d = %d and its value = %d", i,
            ptr+i, *(ptr+i));
}
arrange(ptr, no);
Printf("In The elements are their address is AFTER sorting are");
for(i=0; i<no; i++),
{
    Printf("In Address of element %d = %d and its value = %d", i,
            ptr+i, *(ptr+i));
}
free(ptr);
return 0;
```

output -

Enter the number of elements to be stored 5.

Enter 5 elements to be stored in the array 9

8

7

6

5

The elements and their addresses BEFORE sorting are

Address of element 0 = 7279184 and its value = 9

Address of element 1 = 7279188 and its value = 8

Address of element 2 = 7279192 and its value = 7

Address of element 3 = 7279196 and its value = 6

Address of element 4 = 7279200 and its value = 5

The elements and their addresses AFTER sorting are

Address of element 0 = 7279184 and its value = 5

Address of element 1 = 7279188 and its value = 6

Address of element 2 = 7279192 and its value = 7

Address of element 3 = 7279196 and its value = 8

Address of element 4 = 7279200 and its value = 9

*

```
void arrange (int *ptr1, int num)
{
    int i, j, ~~num~~ temp;
    for (i=0; i<num; i++)
    {
        temp = *(ptr1+i);
        j = i-1;
        while (j>=0 && *(ptr1+j) > temp)
        {
            *(ptr1+j+1) = *(ptr1+j);
            j--;
        }
        *(ptr1+j+1) = temp;
    }
}
```

*

```
for (i=1; i<10; i++)
{
    temp = arr[i];
    j = i-1
    while (j>=0 && arr[j] > temp)
    {
        arr[j+1] = arr[j];
        j--;
    }
```

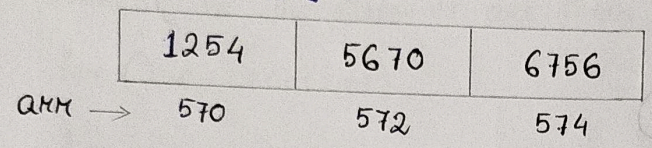
```
arr[j+1] = temp;
}
```

* An important advantage of dynamic memory allocation is the ability to reserve as much memory as may be required during program execution, and then release this memory when it is no longer needed.

Array of pointers –

Datatype *Var_name[size]

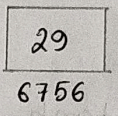
int *arr[3]



arr+2 = 574

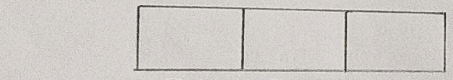
*(arr+2) = 6756

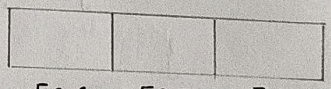
[(arr+2)] = 29

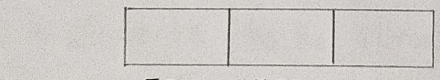


Int (*arr)[3] → A pointer which points to contiguous 3 elements.

arr+1+2

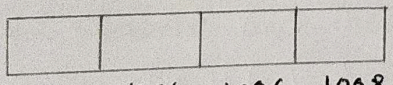


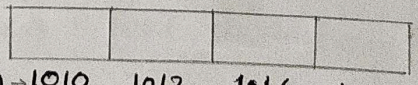


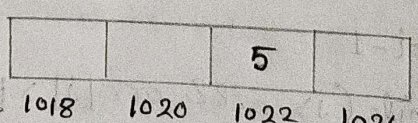


Pointers and 2D array –

a[3][4] a[i][j] → *(*(a+i)+j)



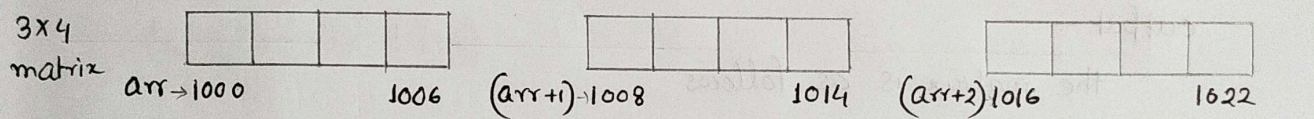




((a+2)+2) = 5

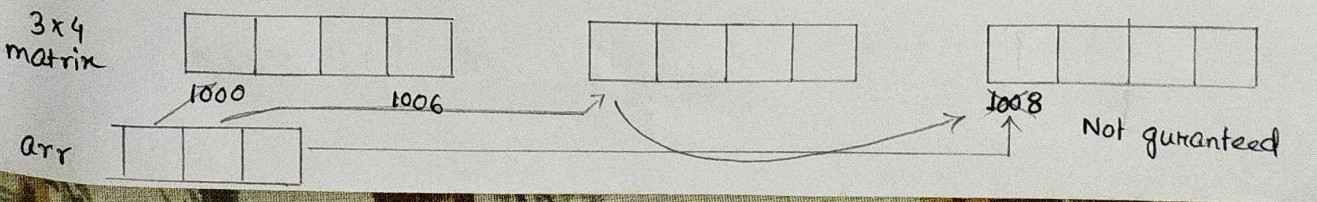
Method 1 →

```
int (*arr)[4], i, j;
Printf ("Enter the elements of the matrix");
for (i=0; i<3; i++)
{
    for (j=0; j<4; j++)
    {
        Printf ("Enter the element in row %d and column %d", i, j);
        scanf ("%d", (*(arr+i)+j));
    }
}
```



Method 2 →

```
int *arr[3], i, j;
for (i=0; i<3; i++)
{
    arr[i] = (int *) malloc (4 * sizeof (int));
}
Printf ("Enter the elements of the matrix \n");
for (i=0; i<3; i++)
{
    for (j=0; j<4; j++)
    {
        Printf ("Enter the element in row %d and column %d", i, j);
        scanf ("%d", (*(arr+i)+j));
    }
}
```



Method 1 –

Input →

```
Printf ("\n the matrix is as follows");
for (i=0; i<3; i++)
{
    Printf ("\n");
    for (j=0; j<4; j++)
    {
        Printf("%d\t", *(*(arr+i)+j));
    }
}
```

output –

the matrix is as follows

5	3	2	1
4	7	8	9
6	0	11	33

the address of element of the matrix are as follows

7410688	7410692	7410696	7410700
7410704	7410708	7410712	7410716
7410720	7410724	7410728	7410732

Method 2 –

Input –

```
Printf ("\n the address of the matrix are as follows");
for (i=0; i<3; i++)
{
    Printf ("\n");
    for (j=0; j<4; j++)
    {
        Printf ("%d\t", (*(arr+i)+j));
    }
}
```

output -

the matrix is as follows

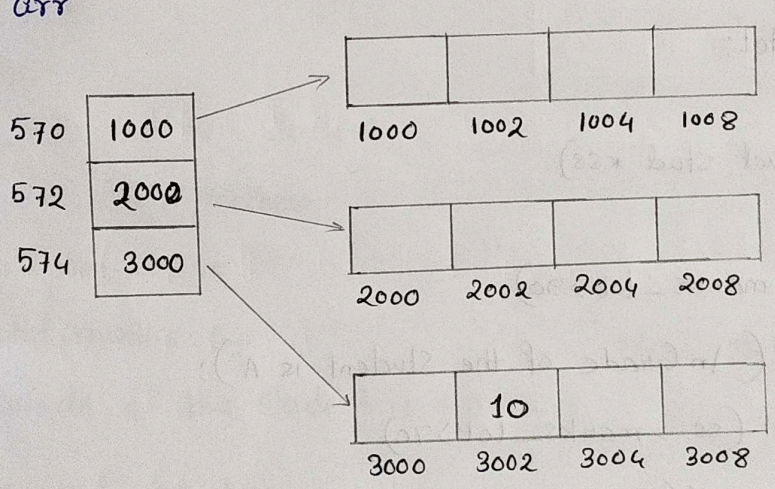
5	3	2	1
4	7	8	9
6	0	11	33

the address of elements of the matrix are as follows

1927744	1927748	1927752	1927756
1927776	1927780	1927784	1927788
1927808	1927812	1927816	1927820

Method 2 -

```
int *arr[3], i, j;
for (i=0; i<3; i++)
{
    arr[i] = (int *) malloc (4* sizeof (int));
}
```



if arr is 570 then *arr is 1000

if arr is 574 then *arr is 3000

(*arr+i)

(*(arr+1)+3) = 2008

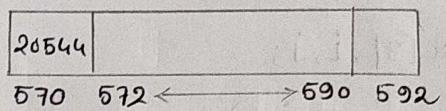
((arr+2)+1) = 10

Pointers and Structures –

Pointer to structure –

~~Struct student~~

```
{   Struct Student
    {
        Int Roll no
        Char nam[20]
        int mark
    };
    Struct student S1,*SS
     SS = &S1
```



S1.Roll no

SS → Roll no → 20544

Input –

```
Struct stud
{
    char name[20];
    int rno;
    int marks_tot;
};
void grad1 (struct stud *ss)
{
      if (ss → marks_tot > 90)
          Printf ("\n Grade of the Student is A");
          else if (ss → marks_tot > 70)
              Printf("\n Grade of the Student is B");
              else if (ss → marks_tot > 50)
                  Printf ("\n Grade of the student is C");
                  else if (ss → marks_tot > 33)
                      Printf ("\n Grade of the student is D");
                  else
                      Printf("\n Grade of the student is F");
}
```

```
int main()
{
    struct stud s1, *s;
    s = &s1;
    Printf("Enter details of 1st student : ");
    Scanf("%[^\n]", s->name);
    scanf("%d%d", &s->rno, &s->marks_tot);

    Printf("Details of 1st student\n");
    Printf("Name : %s Roll no : %d\n Total marks : %d", s->name,
            s->rno, s->marks_tot);
    grad1(s);
    return 0;
}
```

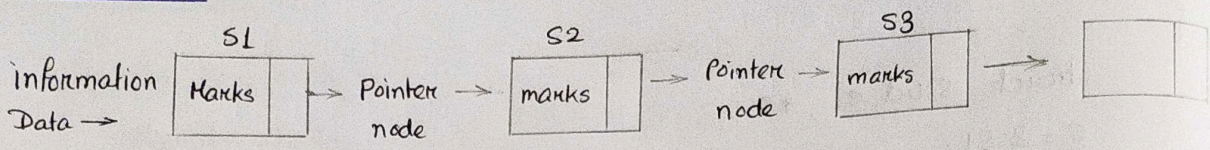
output -

```
Enter details of 1st student : Jobin Mathew
20568
82
Details of 1st Student
Name : Jobin Mathew
Roll no: 20568
Total marks: 82
Grade of the student is B
```

Self referential structures

```
struct stud
{
    Int Roll no
    char name
    Int marks
    struct stud * node
}
```

Linked list –

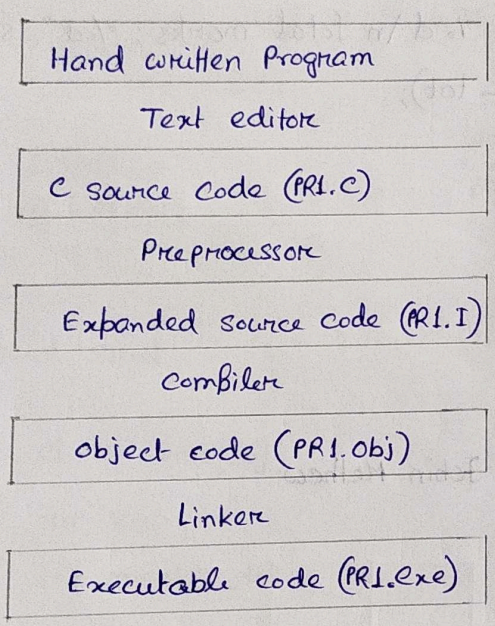


* Efficient use of memory

PreProcessor Directives –

Introduction –

* The C preprocessor is a program that processes our source program before it is passed to the compiler.



Preprocessor directives
in essence some functions
to help in processing the code.

#include
signify a preprocessor directive

* Preprocessor is such a great convenience that virtually all C Programmers rely on it.

a) Macro expansion → #Define

b) File inclusion → #include

c) Conditional compilation → #ifdef, #ifndef, #if, #else

Macro expansion –

#define CONST "Hello" + 35, 66

#define Pi = 3.14

VAR = "Hello" + 35, 66

Area = Pi * r * r

Variable – Occupying space in the memory to store some value & the value stored in future may change.

Why macro expansion instead of variables –

* The compiler can generate faster and more compact code for constants than it can for variable.

* Using a variable for what is really a constant encourages sloppy thinking.

* There is always a danger that the variable may inadvertently get altered somewhere in the program.

The constant value stored in variable may change because of ~~variable~~ mistake.

```
# define  AREA(x)   3.14*r*r
  float  AREA (int r)
   return  3.14*r*r
  ar = ~~3.1~~ AREA(1)  3.14*1*1
       AREA(2)  3.14*2*2
         :          :
       AREA(n)  3.14*n*n
```

* Usually macros make the program run faster but increase the program size, whereas functions make the program smaller & compact.

File inclusion –

#include ⟶ # include <stdlib.h> ⟶ Goto to & search the pre specified location for header files.

└⟶ # include "stdlib.h"

↳ Go & search for the file mentioned in the ~~directory~~ directory of the program

Conditional compilation –

* We can, have the compiler skip over part of a source code by inserting the preprocessing commands #ifdef & # endif

```
# define TEST
   # ifdef windows / linux
      statement 1;
      statement 2;
      statement 3;
   # endif.
```

~~# define~~

does the printing for us.

From where I left →

```
* #define INTEL
  main()
  {
      #ifdef INTEL
          Code suitable for a intel PC.
      #else
          code suitable for a motorola PC
      #endif
       Code common to both the computers
  }
```

```
*
  #define ADAPTER VGA SVGA HDMI
  #if ADAPTER == VGA
       Code for video graphics array
  #elif ADAPTER == SVGA
       Code for super video graphics array
  #else
       code for extended graphics adapter
  #endif.
```