## 1.2 Fundamentals of Algorithmic Problem Solving

Let us start by reiterating an important point made in the introduction to this chapter:

> We can consider algorithms to be procedural solutions to problems.

These solutions are not answers but specific instructions for getting answers. It is this emphasis on precisely defined constructive procedures that makes computer science distinct from other disciplines. In particular, this distinguishes it from theoretical mathematics whose practitioners are typically satisfied with just proving the existence of a solution to a problem and, possibly, investigating the solution's properties.

We now list and briefly discuss a sequence of steps one typically goes through in designing and analyzing an algorithm (Figure 1.2).

### Understanding the Problem

From a practical perspective, the first thing you need to do before designing an algorithm is to understand completely the problem given. Read the problem's

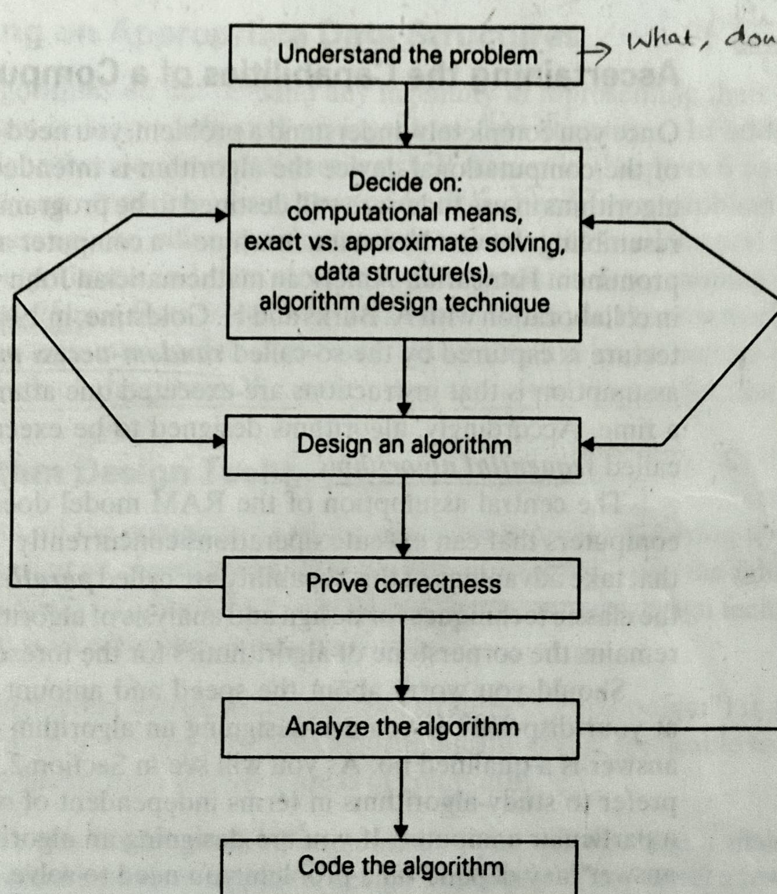



**FIGURE 1.2** Algorithm design and analysis process

description carefully and ask questions if you have any doubts about the problem, do a few small examples by hand, think about special cases, and ask questions again if needed.

There are a few types of problems that arise in computing applications quite often. We review them in the next section. If the problem in question is one of them, you might be able to use a known algorithm for solving it. Of course, it helps to understand how such an algorithm works and to know its strengths and weaknesses, especially if you have to choose among several available algorithms. But often, you will not find a readily available algorithm and will have to design your own. The sequence of steps outlined in this section should help you in this exciting but not always easy task.

An input to an algorithm specifies an ***instance*** of the problem the algorithm solves. It is very important to specify exactly the range of instances the algorithm needs to handle. (As an example, recall the variations in the range of instances for the three greatest common divisor algorithms discussed in the previous section.) If you fail to do this, your algorithm may work correctly for a majority of inputs but crash on some "boundary" value. Remember that a correct algorithm is not one that works most of the time, but one that works correctly for *all* legitimate inputs.

Do not skimp on this first step of the algorithmic problem-solving process; if you do, you will run the risk of unnecessary rework.

## Ascertaining the Capabilities of a Computational Device

Once you completely understand a problem, you need to ascertain the capabilities of the computational device the algorithm is intended for. The vast majority of algorithms in use today are still destined to be programmed for a computer closely resembling the von Neumann machine—a computer architecture outlined by the prominent Hungarian-American mathematician John von Neumann (1903–1957), in collaboration with A. Burks and H. Goldstine, in 1946. The essence of this architecture is captured by the so-called ***random-access machine*** (***RAM***). Its central assumption is that instructions are executed one after another, one operation at a time. Accordingly, algorithms designed to be executed on such machines are called ***sequential algorithms***.

The central assumption of the RAM model does not hold for some newer computers that can execute operations concurrently, i.e., in parallel. Algorithms that take advantage of this capability are called ***parallel algorithms***. Still, studying the classic techniques for design and analysis of algorithms under the RAM model remains the cornerstone of algorithmics for the foreseeable future.

Should you worry about the speed and amount of memory of a computer at your disposal? If you are designing an algorithm as a scientific exercise, the answer is a qualified no. As you will see in Section 2.1, most computer scientists prefer to study algorithms in terms independent of specification parameters for a particular computer. If you are designing an algorithm as a practical tool, the answer may depend on a problem you need to solve. Even "slow" computers of today are almost unimaginably fast. Consequently, in many situations, you need not worry about a computer being too slow for the task. There are important

problems, however, that are very complex by their nature, have to process huge volumes of data, or deal with applications where time is critical. In such situations, it is imperative to be aware of the speed and memory available on a particular computer system.

## Choosing between Exact and Approximate Problem Solving

The next principal decision is to choose between solving the problem exactly or solving it approximately. In the former case, an algorithm is called an ***exact algorithm***; in the latter case, an algorithm is called an ***approximation algorithm***. Why would one opt for an approximation algorithm? First, there are important problems that simply cannot be solved exactly for most of their instances; examples include extracting square roots, solving nonlinear equations, and evaluating definite integrals. Second, available algorithms for solving a problem exactly can be unacceptably slow because of the problem's intrinsic complexity. This happens, in particular, for many problems involving a very large number of choices; you will see examples of such difficult problems in Chapters 3, 11, and 12. Third, an approximation algorithm can be a part of a more sophisticated algorithm that solves a problem exactly.

## Deciding on Appropriate Data Structures

Some algorithms do not demand any ingenuity in representing their inputs. But others are, in fact, predicated on ingenious data structures. In addition, some of the algorithm design techniques we shall discuss in Chapters 6 and 7 depend intimately on structuring or restructuring data specifying a problem's instance. Many years ago, an influential textbook proclaimed the fundamental importance of both algorithms and data structures for computer programming by its very title: *Algorithms + Data Structures = Programs* [Wir76]. In the new world of object-oriented programming, data structures remain crucially important for both design and analysis of algorithms. We review basic data structures in Section 1.4.

## Algorithm Design Techniques

Now, with all the components of the algorithmic problem solving in place, how do you design an algorithm to solve a given problem? This is the main question this book seeks to answer by teaching you several general design techniques.

What is an algorithm design technique?

> An ***algorithm design technique*** (or "strategy" or "paradigm") is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

Check this book's table of contents and you will see that a majority of its chapters are devoted to individual design techniques. They distill a few key ideas that have proven to be useful in designing algorithms. Learning these techniques is of utmost importance for the following reasons:

First, they provide guidance for designing algorithms for new problems, i.e., problems for which there is no known satisfactory algorithm. Therefore—to use the language of a famous proverb—learning such techniques is akin to learning to fish as opposed to being given a fish caught by somebody else. It is not true, of course, that each of these general techniques will be necessarily applicable to every problem you may encounter. But taken together, they do constitute a powerful collection of tools that you will find quite handy in your studies and work.

Second, algorithms are the cornerstone of computer science. Every science is interested in classifying its principal subject, and computer science is no exception. Algorithm design techniques make it possible to classify algorithms according to an underlying design idea; therefore, they can serve as a natural way to both categorize and study algorithms.

## Methods of Specifying an Algorithm

Once you have designed an algorithm, you need to specify it in some fashion. In Section 1.1, to give you an example, we described Euclid's algorithm in words (in a free and also a step-by-step form) and in pseudocode. These are the two options that are most widely used nowadays for specifying algorithms.

Using a natural language has an obvious appeal; however, the inherent ambiguity of any natural language makes a succinct and clear description of algorithms surprisingly difficult. Nevertheless, being able to do this is an important skill that you should strive to develop in the process of learning algorithms.

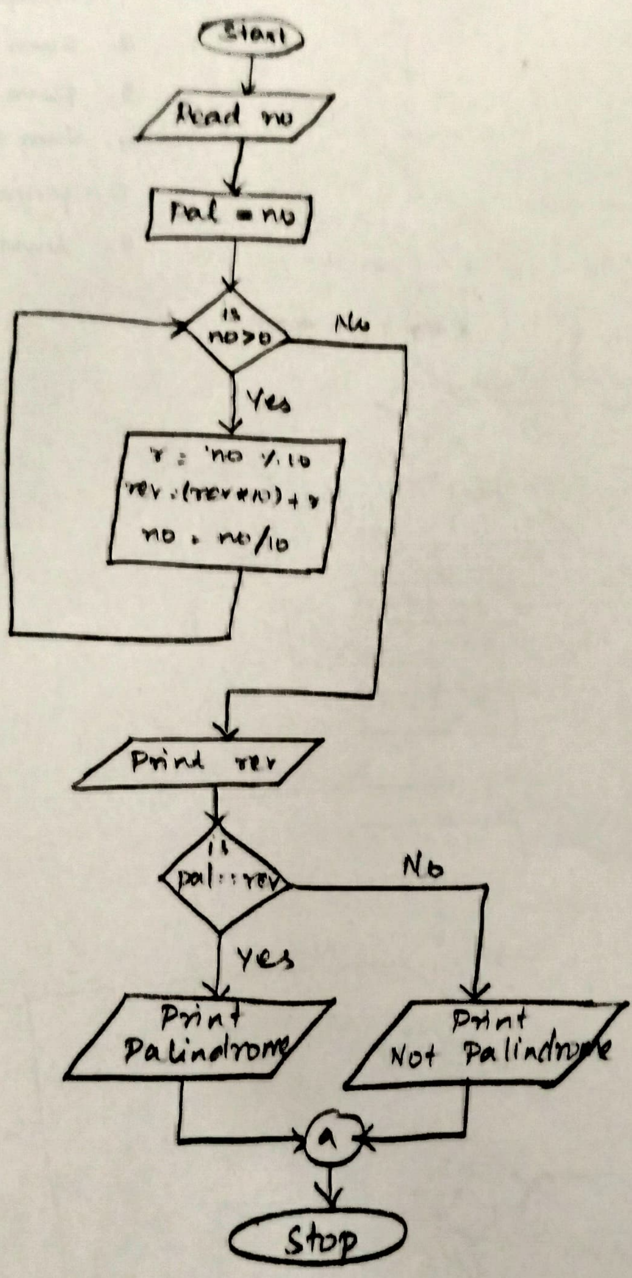
A ***pseudocode*** is a mixture of a natural language and programming language-like constructs. A pseudocode is usually more precise than a natural language, and its usage often yields more succinct algorithm descriptions. Surprisingly, computer scientists have never agreed on a single form of pseudocode, leaving textbook authors to design their own "dialects." Fortunately, these dialects are so close to each other that anyone familiar with a modern programming language should be able to understand them all.

This book's dialect was selected to cause minimal difficulty for a reader. For the sake of simplicity, we omit declarations of variables and use indentation to show the scope of such statements as **for**, **if**, and **while**. As you saw in the previous section, we use an arrow ← for the assignment operation and two slashes // for comments.

In the earlier days of computing, the dominant vehicle for specifying algorithms was a ***flowchart***, a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps. This representation technique has proved to be inconvenient for all but very simple algorithms; nowadays, it can be found only in old algorithm books.

The state of the art of computing has not yet reached a point where an algorithm's description—whether in a natural language or a pseudocode—can be fed into an electronic computer directly. Instead, it needs to be converted into a computer program written in a particular computer language. We can look at such a program as yet another way of specifying the algorithm, although it is preferable to consider it as the algorithm's implementation.

...chart for reverse the no



# Problem Solving and C Programming.

## Algorithm.

Building Blocks of Algorithm.

- Statements
- State
- Control Flow
- Functions

(i) Statements. (Action).

* Input Data
* Process Data
* Output Data

(ii) State - Transition from one process to the other

(iii) Control Flow

Execution of statements in the given order

a. Sequence
b. Selection
c. Iteration/looping

(iv) Functions

* Sub programs / Set of instructions / Block of code.
* Reusability - enhance.

```
Start

   Call Add()

Stop
```

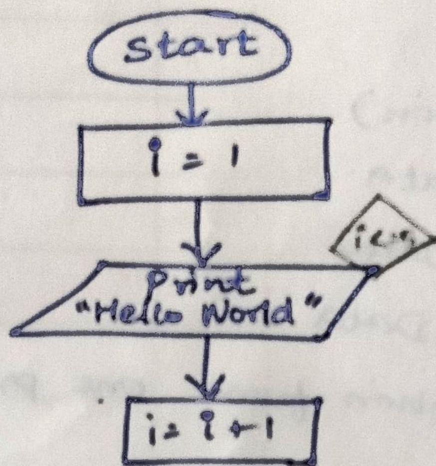
```
Add()
Start
   Read A, B
      C=A+B
   Write C
Stop.
```

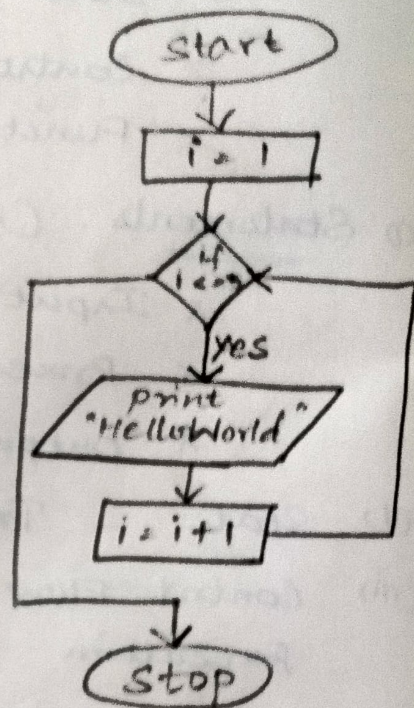
## Looping / Iteration.

* Repeat the same instruction.
* N times / Multiple times

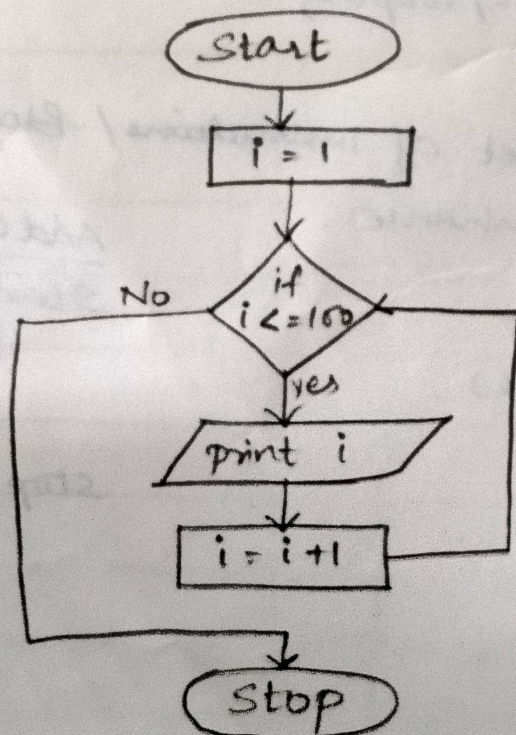
Example: Repeating a set of statements until the condition is satisfied.

1. Print "Hello World" 5 times.

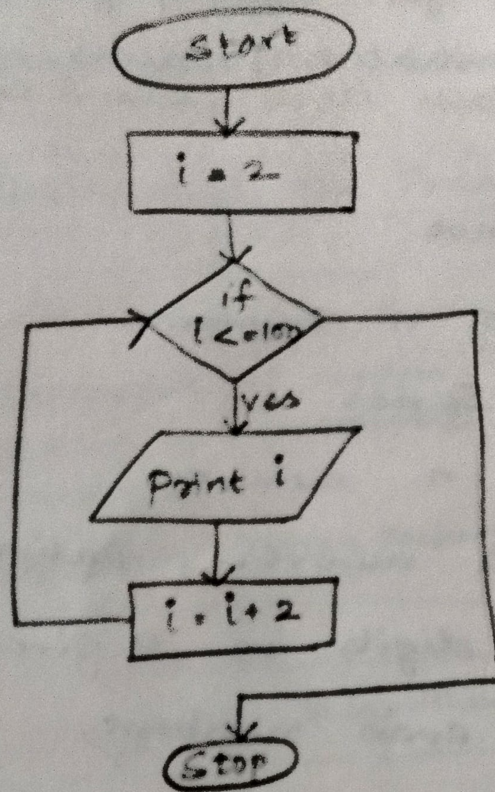


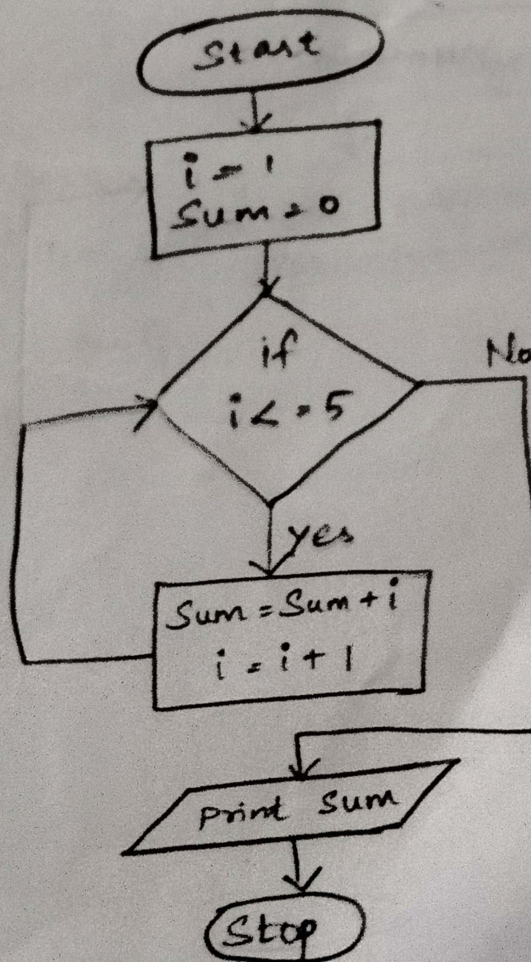

2. print All Natural Numbers from 1 to 100



3. print the even numbers from 1 to 100




4. Sum of first 5 Numbers.



Problem formulation - Algorithmic Problem Solving - Simple strategies for developing algorithms (iteration, for, while, recursion). Illustrative Problems.

Function that calls itself again & again.

Examples:

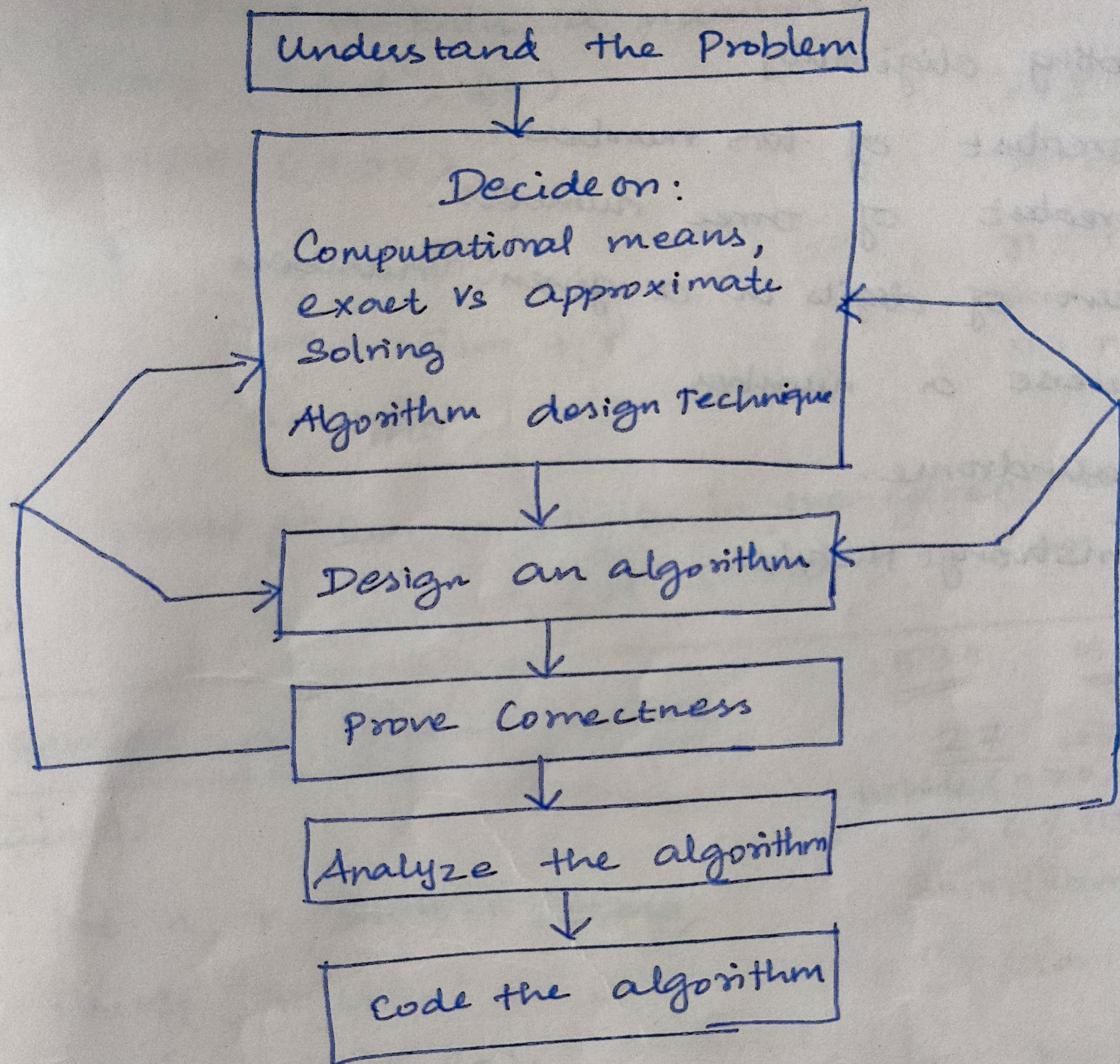
1. prime or not
2. Factorial of n number.
3. Fibonacci Series
4. Square of n numbers.
5. Reverse a number, palindrome.
6. Sum of digits in a given number.
7. Print n even numbers.
8. Sum of n even numbers, odd numbers
9. Prime number.
10. Amstrong number.

35.

```
if (n >= 0)
{n = n%10
 sum = Sum + n
 n = n/10
}
```

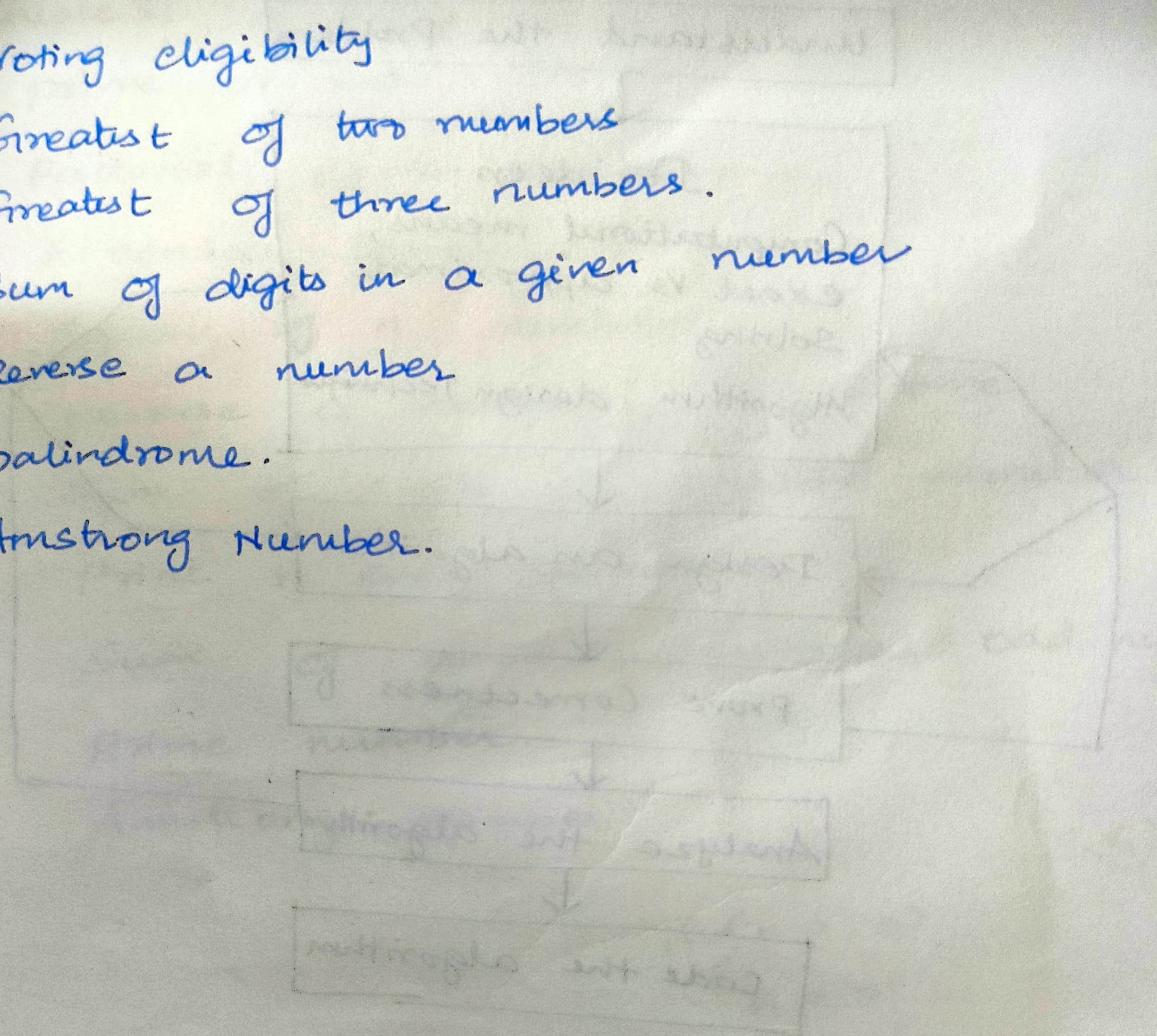
# Algorithmic Problem Solving.

* Solving problem → that require the formulation of an algorithm for the Solution.



## Example Programs.

* Even/odd, prime number
* positive/negative
* voting eligibility
* Greatest of two numbers
* Greatest of three numbers.
* Sum of digits in a given number
* Reverse a number
* palindrome.
* Amstrong Number.



## Q1. Sum of digits in a given number

```
main ()
{
    int n, sum=0, r; t =n;
    printf ("\n Enter a number");
    scanf ("%d", &n);
    while (n>0)
    {
        r = n % 10;
        sum = sum + r;
        n = n/10;
    }
    printf ("Sum of digits in the given number
            %d is %d", t, sum);
}
```

(n = 25 > 0)

sum = 0

r = n % 10;
sum = sum + r
n = n/10

---

535   5 35

## Q2. Reverse a given number.

27 ⇒ 72

```
while (n>0)
r = n % 10;
sum = (sum * 10);
n = n/10;
```

| n | r | sum |
|---|---|---|
| 27 | 7 | 70 |
| 2 | 2 | 70 + (2) |
| 0 | | |

```
main ()
{
    int n, r, sum=0, temp;
    printf ("\n Enter n");
    scanf ("%d", &n);
    temp =n;
    while (n>0)
    {
        r = n % 10;
        sum = (sum * 10) + r;
        n = n/10;
    }
    printf ("\n The reverse of the number %d is %d",
            temp, sum);
}
```

## Q3. Palindrome.

Cont. of Reverse a number.

```
if (temp == sum)
        printf ("\n The given number %d is
                        palindrome", temp);

else

    printf ("\n The given number %d is
                    not a palindrome", temp);
```

---

## Q4. Amstrong number.

$153 \to 1^3 + 5^3 + 3^3$

$\to 1 + 125 + 27$

$\to \underline{153}$

15(3).

```
main ()
{   int n, r, sum=0
    printf ("\n Enter a number");
    Scanf ("%d", &n); temp =n;
    while (n>0)
    {
        r = n%10;
        sum = sum + (r*r*r);
        n = n/10;
    }
    if (temp == sum)
        printf ("\n The given no. is
                    Amstrong number");
    else
        printf ("\n The given no. is
                    not a Amstrong No.");
}
```

```
while (n>0)
{ r = n%10;
  sum = (sum)+
        (r*r*r)
  n = n/10;
}
```

n=153

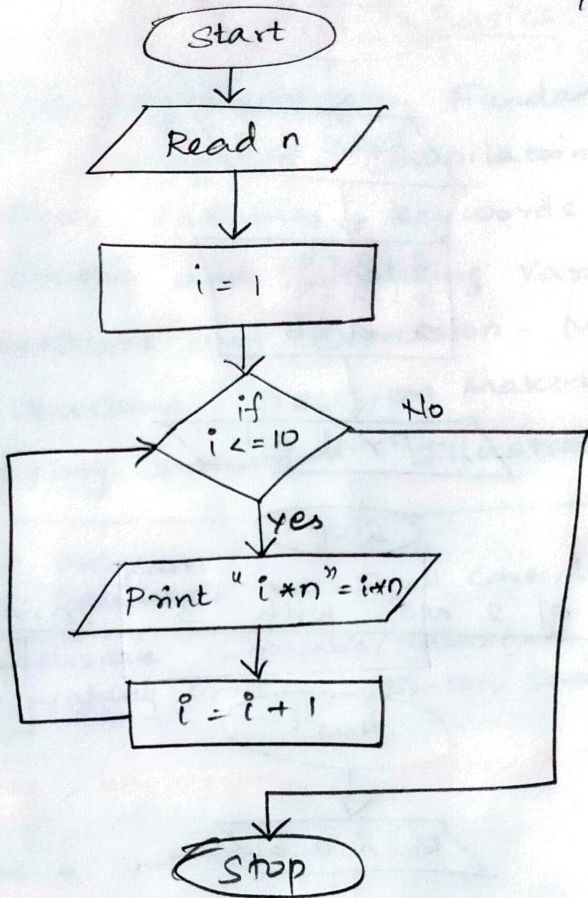
| r | sum | n |
|---|---|---|
| 3 | 27 | 15 |
| 5 | (27+125) | 1 |
| 1 | 153 | 0 |

Q5.

Q6.

Q5. Multiplication Table.

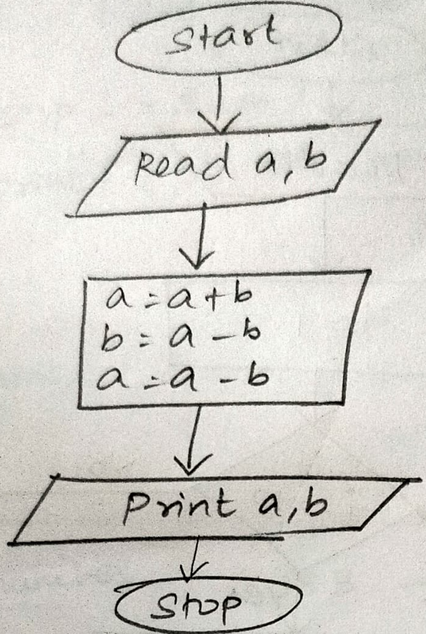
$i = 1$




Q6. Leap year or not.

## Q7. Swapping of 2 nos W/o 3rd variable

| a | b |
|---|---|
| 5 | 10 |




$a = a+b = 15$

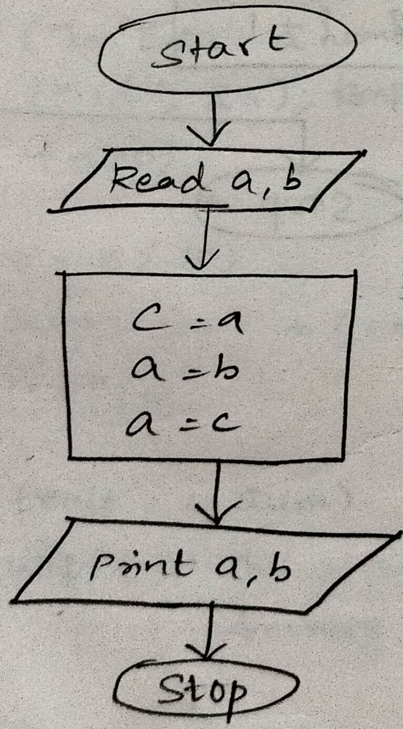
$b = a-b = 5$

$a = a-b = 10$

## Q8. Swapping of 2 nos. with 3rd variable

| a | b |
|---|---|
| 5 | 10 |

c




$c = a$

$a = b$

$a = c$

# UNIT - II.

## C Programming Basics.

Introduction to C Programming - Fundamental rules - Structure of a 'c' program - Compilation and Linking process - Constants, Variables, keywords, Identifiers, Delimiters - Declaring and Initializing Variables - Data types - Operators and Expression - Managing Input Output Operations - Decision Making and Branching - Looping statements - Illustrative programs.

---

* Structure of C Program.
* Rules < All statements are small case letter, / Semicolon / Whitespace - variable/ identifiers.
* Compiling and Linking process → Syntax, Data & Logic.

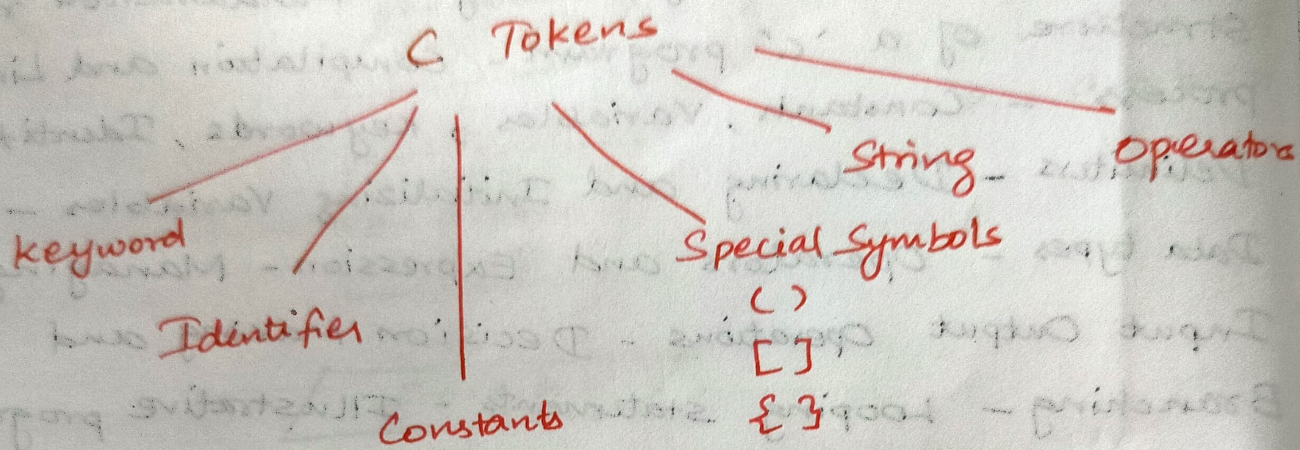
### Constants, Variables, keywords & Delimiters.

* C - programs - instructions.
* Instructions - characters.
* Character set → Characters are grouped into
  - letters
  - Digits
  - White space
  - Special characters.

punctuations ⇒ Tokens.

In paragraph - words + punctuations

In C, Tokens → Smallest individual unit




keyword - Reserved word. Fixed meaning that cannot be changed.

void, while, if, else, getch, int, char, float

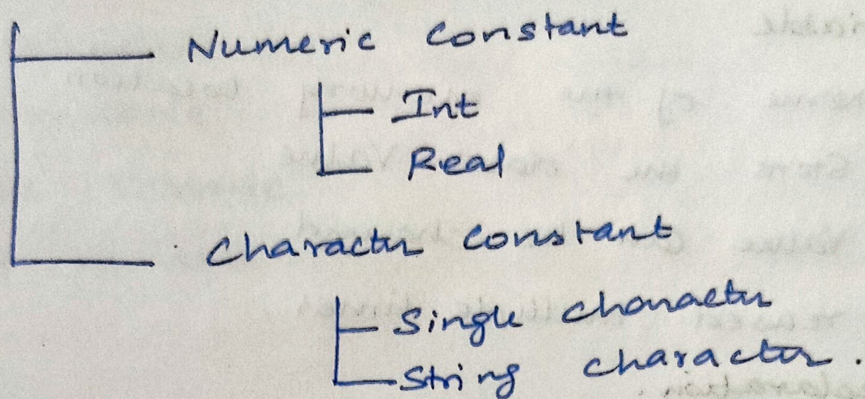
Identifier - Name of Variables, functions & array. It cannot be keywords. User defined names.

Rules:

1. characters, letters, digits, Underscore
2. First character (letter, _)
3. No whitespace
4. Not an keyword
5. only 31 characters are significant.

Ex: marks ✓  1m ✗  m1 ✓

Constants : Fixed value. Value cannot be changed




Variables

Data Name used to ~~use~~ store the data value.

int a;

Assigning value to variable → Assignment operator

int a=10;

Delimiters. Symbols which has syntatic meaning but doesn't perform any operation

#

,

:

;

( )

[ ]

{ }

# Variables. Declaring and Initializing Variables.

| 10 |
|---|

a

## Variable

* Name of the memory location
* Store the data value
* Value can be changed
* reused multiple times.

## Declaration.

```
type  Variable_list;

int   a;        a,b,c  - Variable
float b;        int  }
char  c;        char } datatype.
                float}
```

## Initialization

```
int   a=10, b=20;
float c = 30.4;
char  a = 'z';
```

## Rules for defining Variables.

1. Variable - alphabet, digit, underscore
2. Reserved keywords must not be used.
3. No whitespace is allowed
4. It can only start with alphabet, underscore /not with digit.

```
int a, _ab, a30;   ✓
int 1a; int a b;  int char;   ✗
```

## Types of Variable in C.

1. Local Variable
2. Global Variable
3. Static Variable
4. automatic Variable
5. External Variable.

### Local Variable

A Variable that is declared inside the function / block - local Variable.

Example:

```
Void function1()
{
        int x=10; // Local Variable.
}
```

### Global Variable

A Variable that is declared outside the function/ block is called global Variable.

Any function can change the Value of the global Variable. It is available to all the functions.

```
int sum = 30;   // Global
Void add()
{
        int x = 50; //Local
}
```

Static Variable.

* Declared with "Static" keyword.
* preserve the Value even after they are out of scope.
* Static int Variable remains in memory while program is running.

Example:

```
void function()
{
    int x=10;
    static int y = 10;
    x = x+1;
    y = y+1;
    printf("%d %d", x, y);
}
```

| | int x | static int y |
|---|---|---|
| | 10 | 10 |
| function() | 11 | 11 |
| function() | 11 | 12 |

Automatic Variable.

* All variables by default - auto.
* Explicit we can declare

```
int n = 10          // local (auto)
auto int n = 10;    // auto
```

External Variable.

* extern int n;

Can be used with multiple programs.

Data Type.

* Type of data
* C supports various data types. Each type may have predefined memory requirements (storage representation).
  1. primary Datatype
  2. Userdefined Datatype
  3. Derived Datatype

primary Datatype. → Integer — signed, short, long; character; float; Void

Basic / primitive / Fundamental.

Integer — int.
- 2 bytes (16 bits)
- %d (or) %i
- 32/16 bits Compiler. → -32768 to 32767.

16th bit - Sign Indicator.

Character — char
- 1 byte (8 bit) (-128 to 127)

- %c
- char s = 'n'

float (Floating pt. & Exponential)
- float 4b (%f)
- double 8b (%lf) → 1.7E -308 to 1.7E +308 (precision)
- long double.

- 4 bytes. (3.4 E - 38 to 3.4 E + 38)

char
- char
- signed char -128 to 127
- unsigned char 0 to 255.

In 32-bit compiler

| Data Type | Size | Format Specifier |
|---|---|---|
| Short int | 2 bytes | %hd |
| Unsigned short int | 2 bytes | %hu |
| Unsigned int | 4 bytes | %u |
| int | 4 bytes | %d |
| long int | 4 bytes | %ld |
| float | 4 bytes | %f |
| double | 8 bytes | %lf |
| long double | 16 bytes | %Lf |

15 Algorithm. After plan for developing pgm by logic - correct seq & procedure of instn required to carryout the task ①

* Algorithm - logic of the program. It is the basic tool used to develop problem solving.

* " a sequence of instructions designed in such a way that if the instn are executed in specified sequence, the desired results will be obtained". (Unambiguous - (clear))

characteristics:

* Algorithm - precise & unambiguous
  - instn should not be repeated again.
  - instn should be in sequence.
  - Result produced after algorithm terminates.

Representation of Algorithms.

* Normal English
* Flowchart
* Pseudocode
* Decision table
* program.

Flowchart

* pictorial representation of an algorithm. Sequential steps are represented in flowchart using <u>standard symbols</u>.

* Layout & visual representations of the plan.

* <u>Symbol - oriented</u> design - identifies the type of stmt from the symbols used.

* Arrows ← connecting b/w 2 symbols.

* process of drawing flowchart for an algorithm is called <u>Flowcharting</u>.

* Flowcharts - Easy to Understand.
  - helps in reviewing & debugging of a pgm.
  - easy to analyze & compare various methods.

## float

* float vs double.

double ↓ 8 bytes – 15 decimal places.

float ↓ 6 decimal places

```
float temp = 98.6;
double pi = 3.14159265359;
```

## Void

* No value is present.
* represents nothing

```
Void main (void);
```

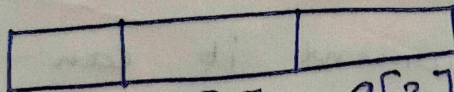
```
Sizeof (a); Sizeof (f);
```

## Derived Datatype.

* Datatype derived from built-in datatypes.
* Array, pointer, structure and union.

Array – Collection of multiple items of same data type.

```
int a1, a2, a3;
int a[3];
```

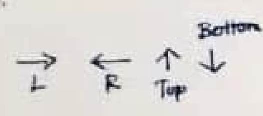


a[0] a[1] a[2]

↑ indexing starts from 0.

Symbols used.

① Flowlines




Exact sequence in which instr are to be executed. Drawn with arrow head.

② Terminal symbol

[oval shape] → begin + End
→ start (no Entering ↓)
→ stop (no ↓) Exit flow line.

③ Input/output symbol.

[parallelogram]

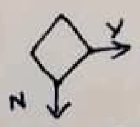
- Input (Read) & output (write)
- denotes fn of I/O devices in the pgm.

④ Process Symbol

[Rectangle]

- used for calculations & initializations of memory locations.
- Arithmetic operations, data movements.

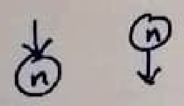
⑤ Decision Symbol




[Diamond shaped symbol]

- Indicate at a point decision is to be made b/w two alternatives.
- Yes (True), No (False).

⑥ Connectors

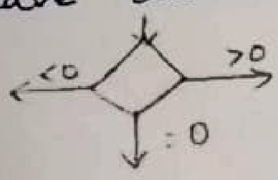



[circle & a digit/ letter placed in link to specify the link].

Example: Greatest of 2 nos, odd/Even, +ve/-ve number

Rules:

* Standard symbols should be used.
* only one should enter decision symbol, 2/3 flow lines can leave decision according to the possible answer.




* Annotation Symbol - describe data/ Computational steps more clearly

--- [This is top Secret data

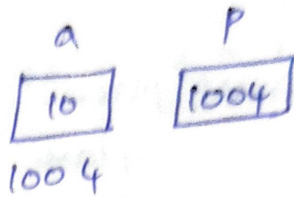
pointer - is a derived data type that keeps track of another data variables memory address

asterisk (*)

int a = 10;

int *p;

p = &a;




a = 10

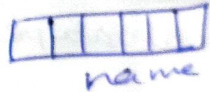
p = 1004

*p = 10.

Structure - derived datatypes which has a collection of items of various datatype.

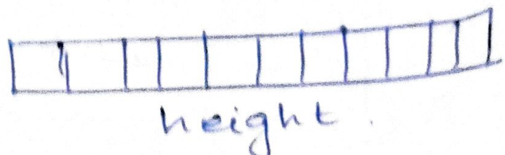
- members of structure can be accessed using . operator.

```
struct person
{
    char name[20];
    int age;
    float height;
};
```



Union - stores various datatype in same memory location.

- At a moment it can hold only one value.



# User defined Datatype

* Enumeration
* Typedef.

Typedef. → a newname could be given to a existing datatype

Ex: typedef int marks;

marks m1, m2, m3;

Enumeration. a way to define constants so that they can be used in the program. Constant integer value is assigned to variables. Value is not assigned - by default - it starts setting from 0.

```
enum identifier {variable-names};
enum month {JAN, FEB, MAR, APR, MAY};
           //default 0   1    2    3    4

enum month {JAN = 5, FEB, MAR, APR, MAY};
```

* Flow lines should not cross eachother.

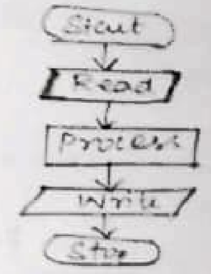
* words in Flowchart - common stmts & easy to understand.

Design structures in Flowcharts.

Design flowchart for any program & later we can combine to get the solution for complex problems, 3 designs:

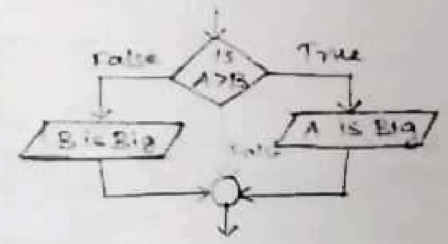
① Sequence structure.

- Simplest, series of stmts
- Sequence (same direction)
- can include any number of instn.




② Selection structure.

- Decision (make questions).
- operations done on decision made.
- 2 exists (two branches ↓ rejoined to single flow to exit the structure)



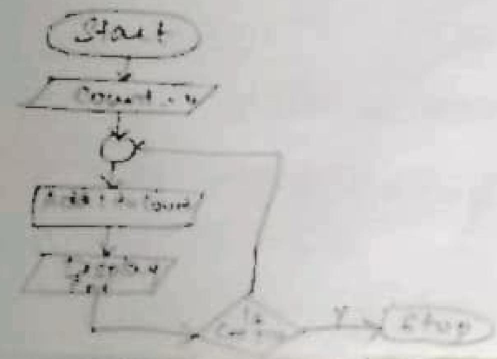

- Selection (True → one-sided selection)
  + null-sided should end with Exit.

③ Loop structure.

Execute sequence of steps in no. of times until particular condition is met

(i) Top tested loop → if Count < 10 $\xrightarrow{T}$ goes into loop
$\xrightarrow{F}$ goes to stmt after loop.

(ii) Bottom tested loop → Trailing decision loop
decision - last stmt of loop.



Algorithm. After plan for developing pgm by logic - correct seq & procedure of instn required to carryout the task

* Algorithm - logic of the program. It is the basic tool used to develop problem solving.

* "a sequence (Unambiguous - (clear)) of instructions designed in such a way that if the instn are executed in specified sequence, the desired results will be obtained".

Characteristics:

* Algorithm - precise & unambiguous
  - instn should not be repeated again.
  - instn should be in sequence.
  - Result produced after algorithm terminate.

Representation of Algorithms.

* Normal English
* Flowchart
* Pseudocode
* Decision table
* program.

Flowchart

* pictorial representation of an algorithm. Sequential steps are represented in flowchart using standard symbols.
* Layout & visual representations of the plan.
* Symbol - oriented design - identifies the type of stmt from the symbols used.
* Arrows ← Connecting b/w 2 symbols.
* process of drawing flowchart for an algorithm is c Flowcharting.
* Flowcharts - Easy to Understand.
  - helps in reviewing & debugging of a
  - easy to analyze & compare various me

Symbols used.

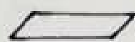
① Flowlines

Exact sequence in which instn's are to be executed. Drawn with arrow head.

② Terminal symbol

[oval shape] → begin + end.
→ start (no entering ↓)
→ stop (no ↓) exit flow line.

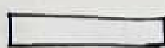
③ Input/output symbol.



[parallelogram]

- Input (Read) & output (write)
- denotes fn of I/o devices in the pgm.

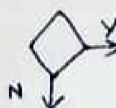
④ Process Symbol



[Rectangle]

- used for calculations & initializations of memory locations.
- Arithmetic operations, data movements.

⑤ Decision symbol




[Diamond shaped symbol]

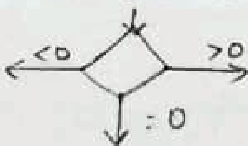
- Indicate at a point decision is to be made b/w two alternatives.
- Yes (True), No (False).

⑥ Connectors

[circle & a digit/letter placed in link to specify the link].

Rules: Example: Greatest of 2 nos, odd/even, +ve/-ve number.

* Standard symbols should be used.
* only one should enter decision symbol, 2/3 flow lines can leave decision according to the possible answer.




* Annotation Symbol - describe data/ computational steps more clearly

---- [This is to Secret dat

## Computer Software

* set of instructions grouped into programs that make the computer to function in the desired way.
* Collection of pgm to perform a task.
* instructs H/w what to do.
* pgm - set of instn stored in memory written in high level lang understandable by computer

  CPU fetches an instn, decodes it & the executes required operation.

  After execution - pgm counter - next instn fetched & executed

## Types of Software.

* Application Software
* System Software.

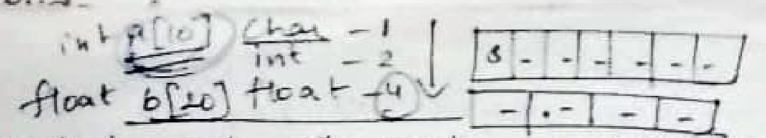
### ① System Software.

* Set of programs that governs the operation of computer system and make H/w to work.
* Controls the internal operations of computer sly.
* Types
  - OS - Controls all components (CPU, Mem, I/O devices) of computer. Ex: DOS, WINDOWS, UNIX, LINUX.
  - Language processors. Conversion of high level language to Machine language.
    - ↳ Assembler
    - ↳ Interpreter
    - ↳ Compiler
  - Utility pgms - Virus Scanner, Disk defragementer

### ② Application Software

* Set of pgms to carryout operations of specific Applt̄n.
* It is written by programmers to enable computer to perform specific task.
* Types
  - Customised Appltn S/w - based on customer requirements.
  - General Appltn S/w - General requirements for a specific task. customers can use it simultaneously.

* Example: Result preparation, Railway reservation, billing



Notes:

* OS - control all H/w parts of computer system
  - Manage resources & overall operations of computer s/y
* Function of OS - Memory Mgt
  - File Mgt
  - Security Mgt
  - Process Mgt
  - User Interface
* compiler - convert high-level lang to machine lang
  - Source code to binary code
* Interpreter - Line-by-line execute the source code.
* Assembler - convert Assembly lang to machine lang

Examples:

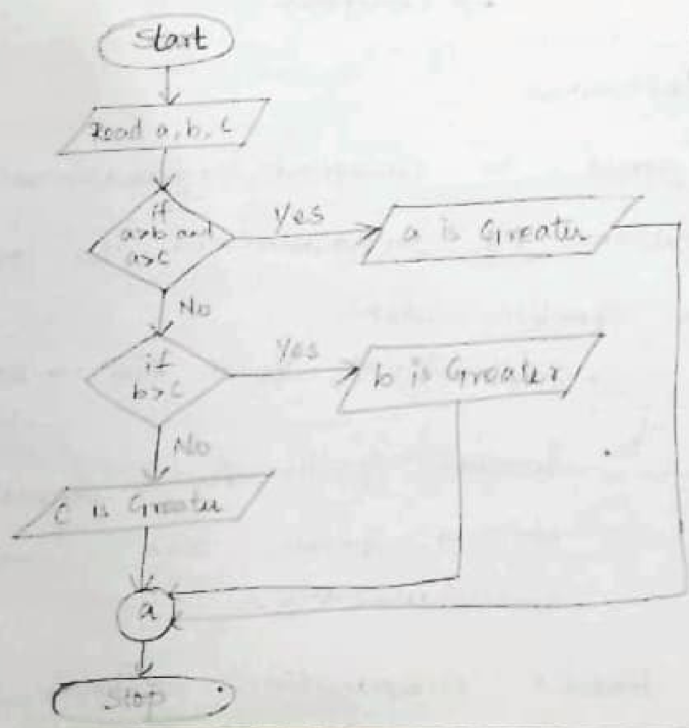
① Area of circle

② Convert Celcius to Fahrenheit. $F = (1.8 * C) + 32$

③ Greatest of 3 numbers

Internet Terminologies

1. Web page
2. Home page
3. Browser
4. URL
5. ISP
6. Download & Upload
7. online & offline
8. Hypertext
9. Web server
10. Web Site.




a>b
a>c

| a | b | c |
| --- | --- | --- |
| 5 | 10 | 15 |
| 15 | 10 | 5 |
| 5 | 15 | 10 |

# Pseudocode.

* Flowcharting Symbols were established bf structured programming
* So flowchart will not be able to handle some concepts.
* pseudocode - formal design tool with structured design
  - Visual, narrative, used for planning prog. logic

Pseudocode Also ↓ Called PDL Program Design Language
- pseudo (initiation / false).
- code (set of stmts / instn in pgm. lang).
- written in English, very clear.
- Rules:
  1. Write one stmt per line.
  2. Capitalize initial keywords.
  3. Indent to show hierarchy
  4. End multiline structure.
  5. keep stmt lang independent.

Example. To calculate student total and average.

```
READ    name, class, M1, M2, M3
        Total = M1+M2 +M3
        Average = Total/3
IF  average is greater than 75
        Rank = Distinction
ENDIF
WRITE   name, Total, Average, Rank
```

Advantages.
- Word processor
- Easily Modified
- read & understood easily
- converting pseudocode to pgm. lang is easy when compared with Flowchart.

Disadvantages.
- Not Visual (pictorial represent)
- No standardised style/ Format

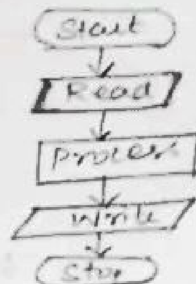
* Flow lines should not cross eachother.
* words in Flowchart - common stmts & easy to understand

Design structures in Flowcharts.

Design flowchart for any program & later we can combine to get the solution for complex problems, 3 designs:

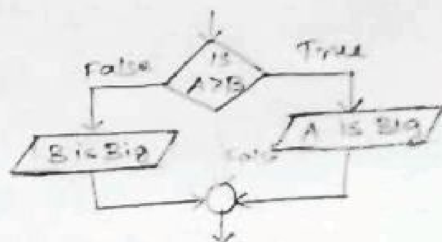
① Sequence structure.

- simplest, series of stmts
- Sequence (same direction)
- can include any number of instn.




② Selection structure.

- Decision (make questions).
- operations done on decision made.
- 2 exists (two branches ↓ rejoined to single flow to exit the structure)



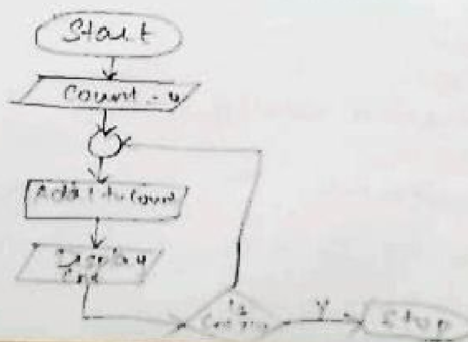

- Selection (True -> one-sided selection) + null-sided should end with Exit.

③ Loop structure.

Execute sequence of steps in no. of times until particul Condition is met

(i) Top tested loop -> if Count < 10 $\xrightarrow{T}$ goes into l
$\xrightarrow{F}$ goes to st after loop.

(ii) Bottom tested loop -> Trailing decision loop
decision - last stmt of loo



M. Shebana (1)

23CST101 - Programming for Problem Solving using C

Unit I - Introduction to Problem Solving Techniques

Fundamentals - Computer Hardware - Software - Algorithms - Building Block of Algorithms (Statements, State, Control flow & functions) - Notation (Pseudocode, flowchart, & Programming language) - Problem formulation - Algorithmic problem Solving - Simple Strategies for developing algorithm (Iteration, recursion) - Illustrative Problems.

## UNIT I

### Fundamentals

* Computer - Electronic Machine (Hardware & Software) ↗ does the human work & minimizes the workload of humans.

1. I/p - users (I/p device - keyboard, Mouse)
2. Processing - CPU (Control Processing Unit) (Brain of computer).
3. O/p - Monitor, printer.

Programming language, Machine language.

Computer Hardware

→ physical components.

→ Fuction efficiently & produce useful output only when H/w & S/w work together.

→ Internal Computer Hardware.

process/Store the instructions delivered by the program/os.

1. Mother board
2. CPU
3. RAM
4. Hard drive
5. Solid State Drive
6. optical Drive
7. GPU
8. NIC
9. Heat Sink

① Mother Board → Central hub.
holds the CPU & functions are carried out.

② CPU - brain of computer. clock speed - measures the computer performance & Efficiency.

③ RAM - Temporary Memory. Volatile Memory (stored data is cleared when computer is switched off)
Information is immediately accessible to programs.

④ Hard drive - Store Temporary & permanent data.

⑤ SSD - Non-Volatile, Safe to store. Data remains permanent even when the computer is switched off.

⑥ GPU - Extension to CPU. Graphical Data.

⑦ NIC - Connect to Network / LAN / N/w Adapter.

External Hardware Components.

* peripheral components, controls either I/p or o/p functions.

1. Mouse.
2. keyboard
3. Microphone
4. Camera
5. Memory card.

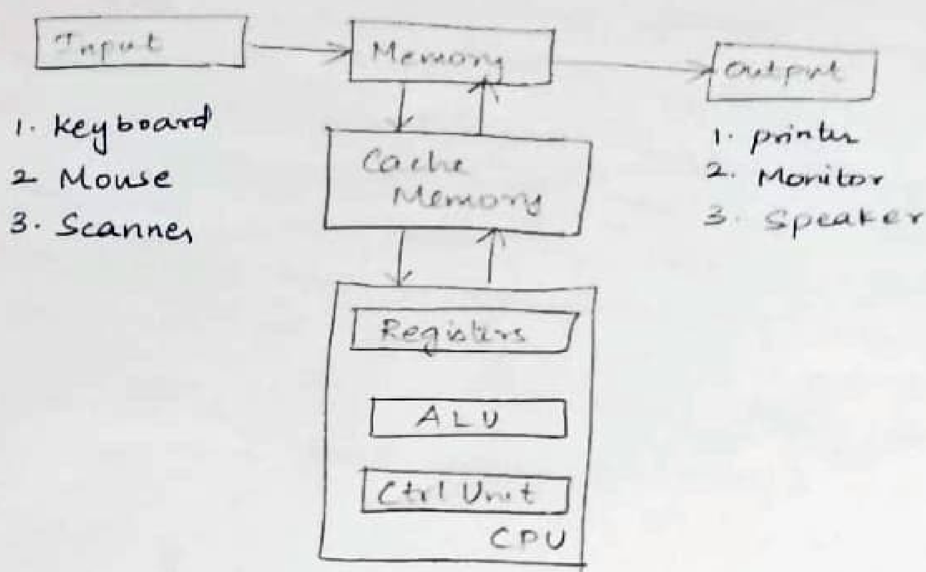
Hardware - Tangible components - run the instruction provided by the software.

Software - Intangible Components - User - H/w - interact & Command - Specific task.

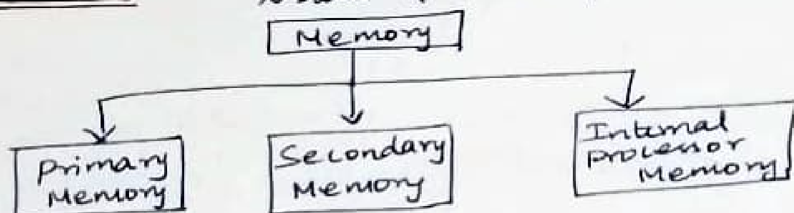
① Application Software
② System Software.

Basic Computer Organization






② Memory Unit → Store data & instruction & intermediate processing results & Final processed information.




primary Memory.

* Built-in unit of the computer
* Data is stored in Machine Understandable binary form

① ROM (Read-only Memory) - (Non-Volatile)
- permanent memory contents cannot be changed by end user.
- BIOS information - which performs POST (power-on Self Test).

② RAM - Volatile
- Temporary memory. power off contents get erased.
- Data is of no usage, contents are erased.

③ Cache Memory
- Store data & related application that was last processed by CPU
- CPU & main memory - intermediate cache is placed

(add, sub, mul, div)

Secondary Memory

* External Storage device connected to computer.
* Connected externally. Non-Volatile.

① Magnetic storage Device → read, erased & rewritten.
Floppy, Hard disk

② Optical storage Device → laser beam
CD-ROM, CD-RW, DVD

③ Magneto-optical Storage Device
Store information such as programs, Files & backup data. End user can modify the information.
Higher storage capacity → laser beams & magnets for reading & writting data to the device.
Ex: Sony minidisc

④ USB (Universal Serial Bus)
pen drive.
Compactable.
Storage Capacity is large.

③ CPU

* Brain of computer
* processing in s/y.
* Controls the Components of s/y.

Main operations

1. Fetching instructions from memory
2. Decode the inst to decide what operation to be done
3. Execute the instruction
4. Store the result in memory.

Main Components:

1. ALU
2. CU
3. Registers.

1. <u>ALU</u> (Arithmetic Logic Unit)
   ↳ Arithmetic operations (add, sub, mul, div)
   ↳ Logical operations (AND, OR, NOT)

2. Control Unit (CU)

   - Controls the flow of data & information
   - CU uses <u>program</u> counter reg for fetching the next instruction to be execution.
   - fetches instn and gives to ALU for processing.
   - CU uses <u>Status register</u> handling Conditions Such as overflow of data.

3. Registers.

   - CPU - temporary storage unit - Registers.
   - Data, instn & intermediate results are stored in registers.

   (i) program Counter (PC)
   (ii) Instruction Register (IR) - Instn to be decoded by CU
   (iii) Accumulator - results produced by CPU
   (iv) Mem. Add. Register (MAR) - Address of next locatn in Mem. to be accessed
   (v) Mem. Buffer Register (MBR) - Storing data Sent / Received to CPU.
   (vi) Mem. Data Register (MDR) - Data & operands.

## Algorithm

* Set of instruction when executed in a sequence will get a solution.

## Representation of Algorithm

1. Normal English
2. Flowchart
3. Psendocode

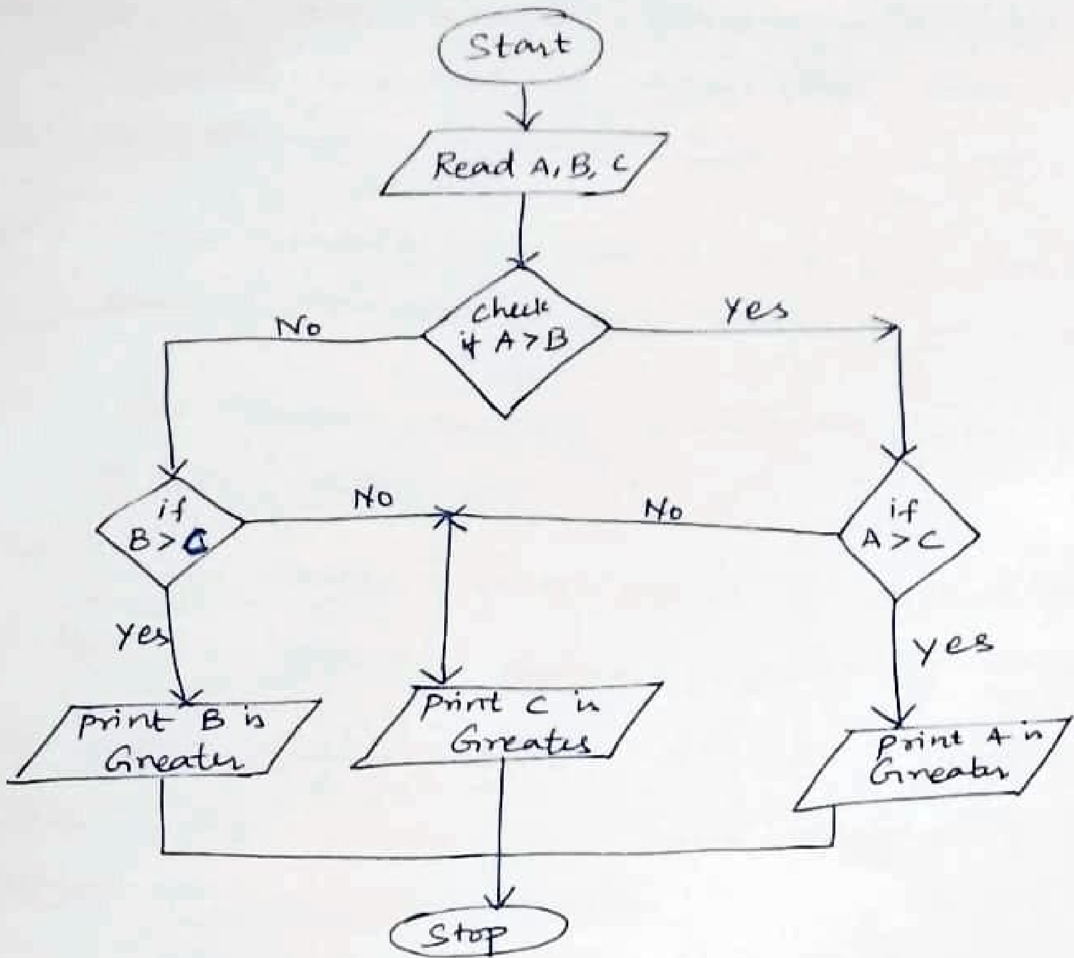
## Flowchart.

1. Flow lines
2. Terminal Symbols
3. Input / output symbol
4. Process Symbol
5. Decision Symbol
6. Connectors.

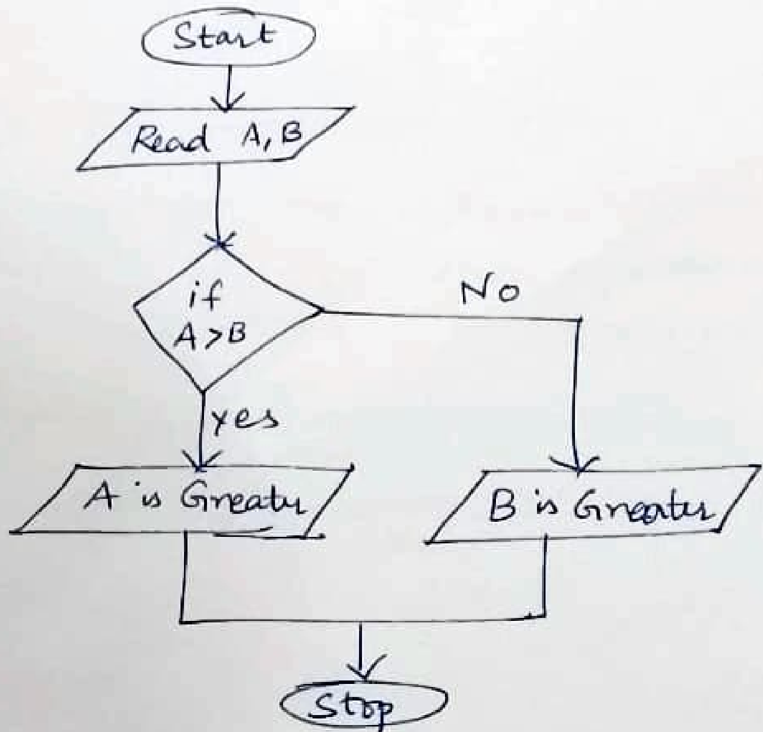
## Example:

1. Addition of two numbers
2. Greatest of two numbers, even/odd, Neg/po
3. Calculate the SI, Area of circle
4. Calculate Total + Average of Students in a c
5. Sum of 5 numbers.

## Greatest of Three Numbers.




## Greatest of two numbers.



UNIT - II.

# C PROGRAMMING BASICS.

## History of c.

* Structured, high-level, machine - independent language. (multi-platform)
* Root of modern languages is ALGOL (1960, First computer language to use block structure). ↓ used in Europe
  structured prgming
* 1967 - Martin Richards developed a lang. BCPL (Basic Combined Programming lang) primarily for writing SS.
* 1970 - ken Thompson created a language with features of BCPL called B.
* B - used to create early version of UNIX OS @ Bell lab
* B & BCPL - "Typeless" system programming languages.
* Dennis Ritchie (ALGOL, BCPL, B) 1972 ← c language
  @ Bell lab
* c uses many concepts from these languages & added up datatypes, other powerful concepts.
* Unix is associated with c. c is used in academic environments, it runs under variety of operating s/y & H/w platforms.
* 1979, c is evolved into "traditional c".
* 'the c programming language' book - Dennis Ritchie & Brain kernighan (1978).
  K&R C
* 1983 - American National Standards Institute (ANSI) appointed a technical committee to define a standard for c.
  It was approved in Dec 1989. C89
  approved by ISO in 1990.

* C++ added new features to C - to make it a Versatile and object - oriented language.

* During this period Sun Microsystems of USA created a new lang Java

* All languages are Dynamic in Nature.

Importance of C

* Robust
* Built -in functions & operators - complex pgm.
* C Compiler - capability of Assembly language
  +
  Features of high - level lang
  ↓
  Suited for writing both SS & business package.
* Pgm in C - Efficient & fast [Datatypes & operators]

* Faster than BASIC.

Ex: Increment a Variable 0 - 15000 take 1 sec in C

0 - 15000 take more than 50 sec in BASIC (Interpreter)

* 32 keywords in C, Built-in functions
* Highly portable - C pgm written in one Computer can run on another with little/no Modification
* Structured programming. - function modules
* Extends itself. C is a collection of function supported by C library

Example programs.

```
#include <stdio.h>
#include <conio.h>
main()
{
 /* printing */
 printf("C programming");
}
```

C pgm is divided into modules/functions.

functions are written by User, stored in C library.

Library fn are grouped category-wise and stored in different files known as header files.

#include <filename>

(#include) Preprocessor directive.

* #define ← preprocessor directive - Symbolic constants.

* main :

Different forms of main are:

- main ()
- int main ()
- Void main ()
- main (void) → No arguments.
- Void main (Void)
- int main (Void).

No arguments.

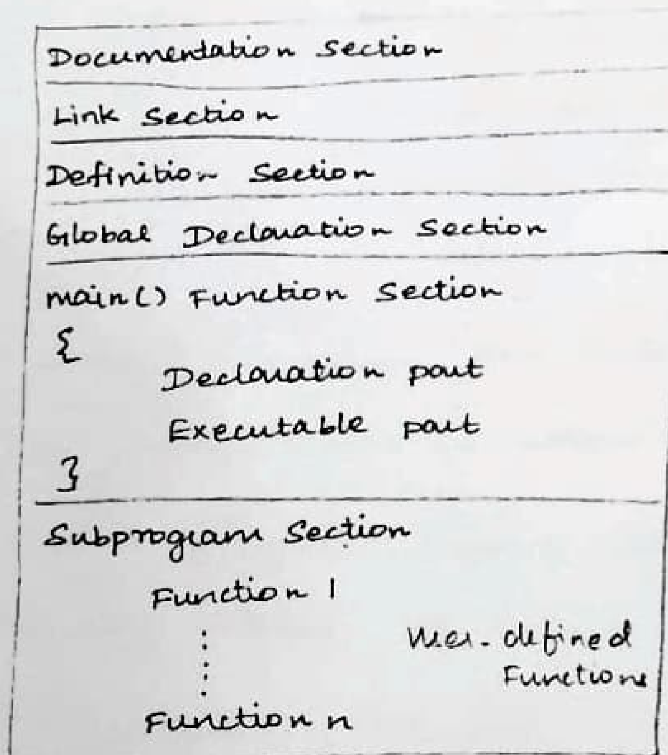
Void - doesn't return any information to o

int - returns an integer Value to OS.

return 0 ← last stm

Structure of C.

* C - group of building blocks called functions.
* Function - Subroutine that may include 1/more stmts to perform specific tasks.




C Structure.

* Documentation - Set of comment lines (Name of the pgm, author & other details). /* .... */

* Link - provides instn to the compiler to link functions from system library.

* Definition - defines all symbolic constants.

* Global Variable - Variable that can be used in more than one functions.
  (User-defined fns are also included in this section).

* Main function Section
  - Declaration part  a = 5, b = 3.
  - Executable part.  a = b * c

Subpgm → user-defined fn which are called in main fn.

Programming Rules.

* Rules to be followed are:

① All statements in c - lower case. Symbolic Constants alone uppercase letter can be used.

② Blank space inserted b/w words. Space not allowed in declaration of variable, keyword, constant & function

③ Two/Three Stmts are allowed in a single line seperated by semicolon.

Executing a C program.

* Executing a pgm written in c involves - Series of steps

① Creating a program

② Compiling a program

③ Linking the pgm with functions that are need by c library

④ Executing the program.

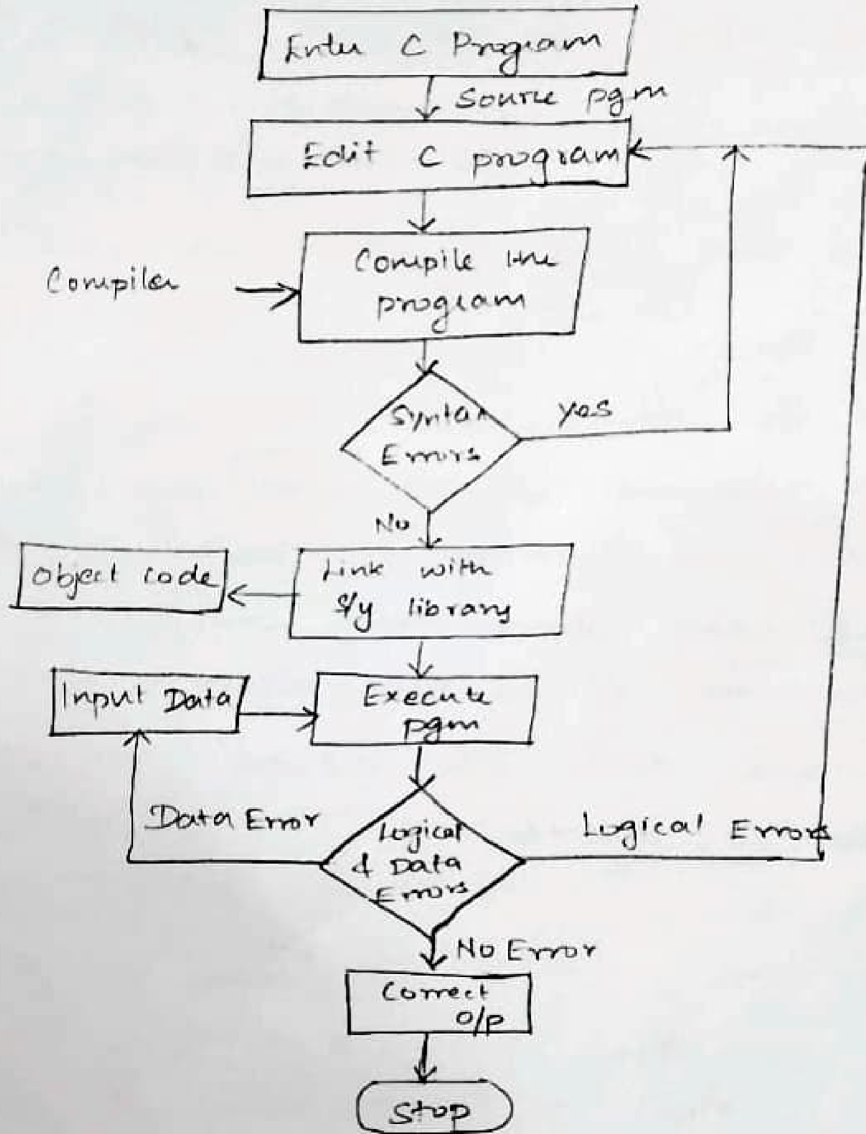



Fig: process of Compiling & running a C program.

(i) Creating - writting & Editing in c editor. (.c extension)

(ii) compiling

converts high level lang to machine language

ALT + F9 / compile option

Errors: Syntax errors - Struct are wrong.

(iii) Linking

c lang - collection of predefined functions. where functions are written in header files.

Linking with s/y library is important. This is automatically done at the time of execution.

(iv) Executing (ctrl + F9) / Run

Running & testing the pgm with sample data.

2 Errors: Logical & Data Errors.

Logical ↓ cndtn & ctrl stmts

error after som sequ. execution ← cndtn & ctrl stmts

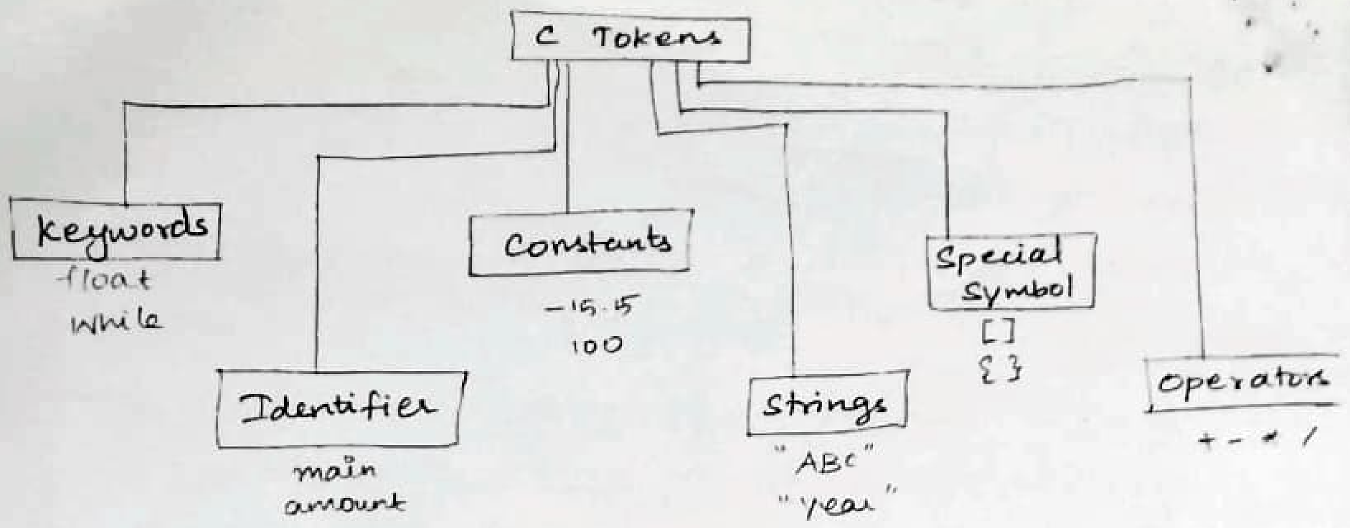
Constants, variables & Data types.

* pgm - Sequence of instn
  - Instruction ( formed using certain symbols & words according to some rigid rules (syntax rules)).

* Characters is c are grouped into:

  (i) Letters
  (ii) Digits
  (iii) Special characters
  (iv) White spaces.

* C Tokens. Smallest individual units are known as C Tokens. (words/punctation).




Identifier

* In c language, every word is classified into either a keyword / identifier.

* Names given to various pgm elements such as variables, functions & arrays.

Rules - Identifier

(i) Consists of letters & digits in any order

(ii) must begin with a letter /character / underscore (_) (considered as letter)

(iii) uppercase & lowercase letters are allowed.

(iv) Identifier can be of any length (most 'c' compiler recognizes 1st 31 characters).

(v) No space & special symbols are allowed.

(vi) Identifier cannot be a keyword.

Examples.

STDNAME, TOT_MARKS, _TEMP, Y2K.

Not allowed: 1REC, STD NAME.

Keyword.

Reserved words - Standard & predefined meaning in 'c' which cannot be changed & they are building blocks for program structure.

Examples.

auto, break, case, default, char, do, float, int, void

Variables

* Variable is an identifier which is used to represent some specific type of information within a particular portion of program.

Variable ← different values at different times during the execution.

Rules for naming the variables.

(i) Variable name - combination of 1 to 8 alphabets, digits / underscore.

(ii) first character - Alphabet / underscore (_).

(iii) Length of variable cannot extend upto 8 characters long, & some can recognize upto 31 characters long.

(iv) No commas / blank spaces allowed within a variable name.

(v) No special symbol, an underscore (_) can be used in variable name.

Variable Declaration.

* After designing variable names, Declare them in pgm & this declaration tells the compiler what variable name & what type of data it can hold.

Syntax: data_type $v_1, v_2, v_3 \ldots v_n$;

data_type ← is the type of data

$v_1, v_2, v_3 \ldots v_n$ → list of variables.

Example: int code; char sex; float price;

char name[10] (array of characters).

Initializing Variables.

* Initialization - Assignment operator (=).
* Initialization can be done while declaring variables.

Syntax: Variable = constant;/ datatype variable = const;

Example: int i = 5; char c = 's';

float f = 29.77;

User-defined variables: C provides a feature to declare a variable of type of user-defined. It allows users to define an identifier that represents existing data type & this can later be used to declare variables.

Type Declaration ⇒ Syntax: typedef data.type identifier;

(typedef → User defined type declaration.)

(identifier → identifier refers to new name given to the datatype.)

Example: typedef int marks;

marks $m_1, m_2, m_3$;

Enumerated Data type. (user-defined data type) define own datatype where its variable take value.

Syntax: enum identifier {Value1, Value2, ... Valu

(enum → User-defined enumerated Datatype)

({Value1, Value2, ...} → Enumerated Consta)

Example: enum day {mon, tue, ... sun}

enum day w_st, w_end;

w_St = mon;

w_end = sun;

[illegible] have only one value from en[illegible]

Scope of Variables.

* Availability of variables within the program.
* Two types of Scopes - Local & Global

① Local Variables.

Variables defined inside function blk / compound stmt is called local variables.

Ex:

```
function()
{
        int i, j;      → local variables
        /* body of function */
}
```

② Global/External Variables.

Variables declared before function main(). these variables are available for all functions inside the pgm.

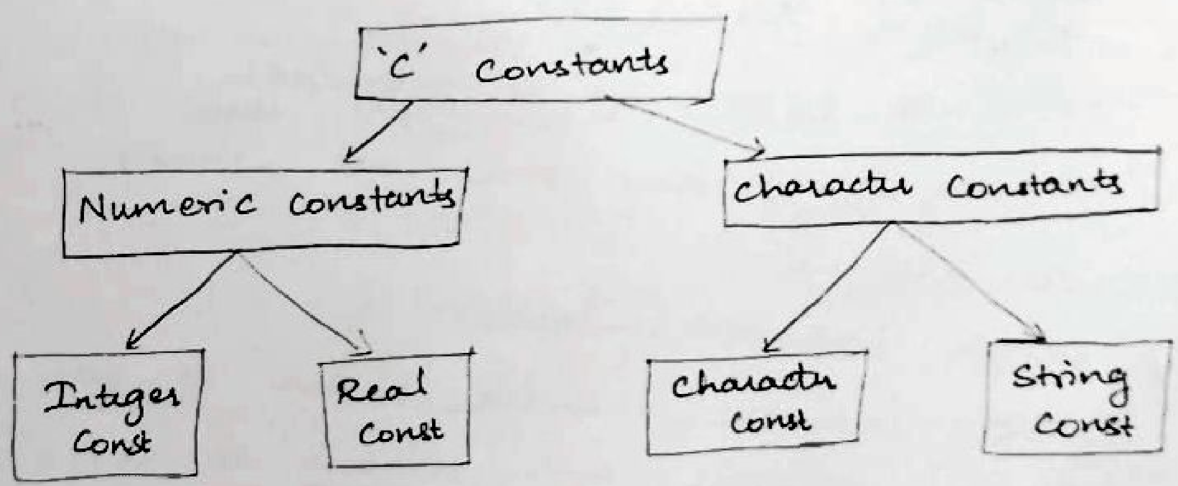
Ex:

```
int a, b=2;      → Global variables
main()
{
        ...
        fun();
}

fun()
{
        int sum;
        sum = a+b;
}
```

Constants.

Item whose value cannot be changed during execution is called constants.



... enclosed in double quotes, the

Numeric Constants.

① Integer Constant. formed with sequence of digits.

3 types:
- Decimal Number (10, 153, -321) - 0 to 9
- Octal Number (062, 0541) - 0 to 7
- Hexadecimal Number (0X4, 0X9F, 0XBCF) - 0 to 9, A, B, C, D, E, F.

Ex: marks = 90; discount = 15;

Rules:

* Decimal point is not allowed.
* Can either be +ve/-ve. -ve (Sign must be preceded) -51
* Must have atleast one digit.
* No commas / Blank spaces are allowed.
* Range (-32,768 to 32,767)

② Real constants. It is made up of sequence of numeric digits with presence of a <u>decimal point</u>.

To represent quantities that vary continuously such as distance, height, temperature, etc..

Ex: distance = 126.0;
height = 5.6;

Rules:

* Must have one digit
* Must have decimal point
* Either +ve/-ve.
* -ve (Sign must be preceded).
* No commas / Blank spaces are allowed.

Character Constants.

① Single character constants.

Single character enclosed within a pair of single inverted commas both pointing to left.

Ex: 's', 'M', '3'.

② String Constants.

Sequence of character enclosed in double quotes, the characters may be letters, numbers, special characters & blank spaces. At the end of string '\0' is automati placed.

Ex: "HI", "Nuni", "39.77", "50".

③ Declaring a Variable as constant.

When the value of some of the variables may remain constantly during execution of program, in such a situation this is done by using const keyword.

Syntax: const datatype variable = constant

keyword to declare constant

Example: const int dob = 3977;

Delimiters.

Symbols, which has some syntactic meaning & has got significance. Doesn't specify any operation.

| | | |
|---|---|---|
| # | Hash | pre-processor directive |
| , | comma | seperate list of variables. |
| : | Colon | label Delimiters |
| ; | Semicolon | stmt Delimiters |
| () | Parenthesis | used in Expression/fn. |
| {} | Curly braces | Blocking 'c' structure |
| [ ] | Square braces | used with arrays. |

...logical manipulations).

Data types. Type of data, 'c' support various data types. each type may have predefined memory requirement & storage representation.

① Primary — const, int, float, double

② Userdefined — typedef, enum

③ Derived — arrays, pointer, structures, union

④ Empty — void → has no value
→ doesn't return any value

① Primary Datatype.

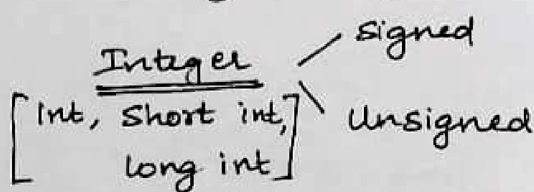
int - 2bytes (-32768 to 32767) %d or %i — 16-bit compiler; $16^{th}$ bit - sign indication.
32-bit compiler -2147483648 to 2147483647

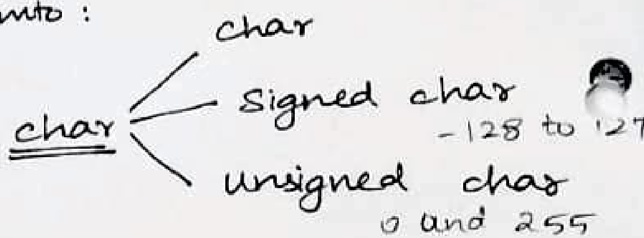
char - 1 byte (-128 to 127) %c Ex: char s = 'n';

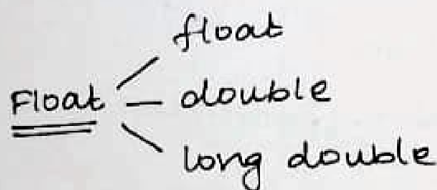
float - 4 bytes (3.4E-38 to 3.4E+38) %f or %g Ex: float f = 29.77;

double - 8 bytes (1.7E-308 to 1.7E+308) %lf Ex: double d = 291075114

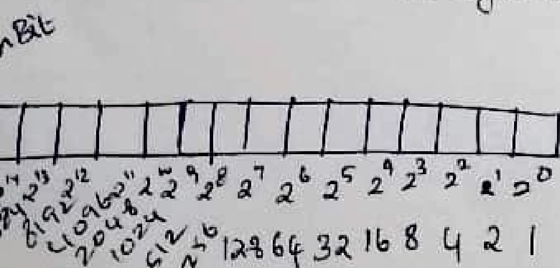
Primary Datatype is divided into:




Refer Balagurusamy Book for reference to various Datatypes




$2^0 = 1$ (0,1)

$2^{15}$ = 32,768 (Signed)

$2^{16}$ = 65536 (Unsigned)

0 - 00
1 - 01
2 - 10
3 - 11

0 - 00
1 -
2 -
3 -
4 -
5 -
6 -

# Operators & Expressions.

* Operators are used in programs to manipulate data & variables. (Mathematical / logical manipulations).
* Data item that operators act upon are called operands.
* Unary operator, Binary operator.

  Ex: a+b
  a,b - operands
  + - operator.

* Types.

| | |
|---|---|
| (i) Arithmetic | (v) Increment & Decrement |
| (ii) Relational | (vi) Conditional |
| (iii) Logical | (vii) Bitwise |
| (iv) Assignment | (viii) Special |

① Arithmetic operator.

| Operator | Meaning | Example |
|---|---|---|
| + | Addition / unary plus | 2+3 = 5 |
| - | Subtraction / Unary minus | 3 - 2 = 1 |
| * | Multiplication | 3*2 = 6 |
| / | Division | 6/3 = 2 |
| % | Modulo Division | 7%3 = 1 |

* Classification:

① Unary arithmetic - +x, -y

② Binary arithmetic - x+y

③ Integer arithmetic  a = 5  b = 4
  2 operands should be integers.

Ex: a/b = 5/4 = 1
Here the decimal part is truncated.

④ Real arithmetic - operands are real.
Ex: x = 6.0/7.0 = 0.857143.

Mixed-mode Arithmetic: one operand is real & other is integer. If any of the operand is real, result is real.

Ex: 2/5 = 0

Involves real as 1 operand:

5.0/2 = 2.5

5/2.0 = 2.5

5.0/2.0 = 2.5

— Result will be Real.

Example.

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

Output: Enter values of b, c, d: 3, 5, 10 3/10

Sum = 8 Sub = -2 mul = 15 div = 0.3

## ② Relational Operators.

* Used to compare 2/more operands. Operands - variables constants or expressions.

Ex: Compare age of 2 persons.

| | | | RESULT |
|---|---|---|---|
| < | less than | 2 < 9 | 1 (Return value) |
| > | Greater than | 2 > 9 | 0 False |
| <= | less than or equal | 2 <= 2 | 1 True |
| >= | Greater than or equal to | 2 >= 3 | 0 |
| == | Equal to | 2 == 3 | 0 |
| != | Not Equal to | 2 != 3 | 1 |

Relational operator complements:

> is complement of <=

< is complement of >=

== is complement of !=

Ex: $!(x < y) \Rightarrow x >= y.$

$!(x > y) \Rightarrow x <= y.$

## ③ Logical operators.

* Used when we want to test more than one cndtn & take a decision.

&& logical AND (exp1) && (exp2).

|| logical OR ! (exp) ⇒ if(!(c1 < c2))

! logical NOT.

## (A) Assignment operator.

* Assign a Value to an Variable, Value of a Variable to another Variable.

Syntax: Variable = expression (or) Value;

Example: x = 10;

y = a+b;

(a) Compound Assignment shorthand operator - assign a value to a variable

x += y    x = x+y

x -= y    x = x-y.

(b) Nested/Multiple Assignments.

Var1 = Var2 = ... = Varn = Single Variable or expression

i = j = k = 1;

x = y = z = (i+j+k);

## (5) Increment & Decrement operators (Unary operators)

Hence it is called

++ → Add 1 to Variable.

-- → Sub 1 from Variable

++x    pre increment

--x    pre decrement

x++    post increment

y--    post decrement.

11, 11, 10, 10

Example: a = 10    Now value of a is

a++    10    10+1 = 11

a--    11    11-1 = 10

--a    9    10-1 = 9

++a    10    9+1 = 10

(6) Conditional (Ternary) operator.

- Checks the condition & executes the statements depending on the condition.

Syntax: condition ? exp1 : exp2;  → ternary operator

↳ True: exp1 is evaluated
False: exp2 is evaluated

Example:

```
main()
{
    int a = 5, b = 3, big;
    big = a > b ? a : b;
    printf ("Big is ... %d", big);
}
```

O/p : Big is ... 5

(7) Bitwise operator. Used to manipulate data at bit level. Operate on integers only. It may not be applied to float.

| Operator | Meaning |
|---|---|
| & | Bitwise AND |
| \| | Bitwise OR |
| ^ | Bitwise XOR |
| << | Shift left |
| >> | Shift right |
| ~ | one's complement. |

Example: Bitwise AND (&).

| & | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

x = 7 = 0000 0111

y = 8 = 0000 1000

x & y = 0000 0000.

Bitwise OR

| \| | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

Bitwise XOR

| ^ | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

⑧ Special operator.

Comma operators ,

Sizeof operators Sizeof

address of ← & * → value of a variable — pointer operators

. --> Member selection operator.
→ Access the elements from the structure

(i) Comma operator. used to seperate the stmt elements such as variables, constants or expression.
Link the related expressions together.

Ex: Val=(a= 3, b=9, c=77, a+c); 3 is assigned to a
9 is assigned to b
77 is assigned to c
a+c = 80.

(ii) Sizeof() is a unary operator, returns length in bytes of the specified variable. Helps in finding the bytes of a variable.

Syntax: sizeof(var);

Example:

```
main()
{
    int a;
    printf("size of a is %d", sizeof(a));
}
```

o/p: size of a is 2.

1. Managing I/O Functions (I/P, process, O/P)
└ essential features of computer.

• 2 ways to give input to pgm
  ① Assigning data to variable
  ② I/O stmts. C I/O operations - fn call stdio)
    ├ Unformatted I/O stmts
    └ formatted I/O stmts

① Unformatted I/O statements.

* cannot specify the type of data
* I/O a single/group of characters from/to I/O devices.

| Input | output |
|---|---|
| getc() | putc() |
| getchar() | putchar() |
| gets() | puts() |

(i) getchar() single character input. char var = getchar();

Char x;
x = getchar();

(ii) putchar() single character output. putchar (char var);
displays one character at a time.

char x; putchar (x);

(iii) getc() accept a single character from standard input to a character variable

Char c = getc();

(iv) putc() display single character variable to std. O/P

putc (c);

getc() & putc() are used in file processing.

v) gets() & puts()

used to read the string, display/write string to o/p device

gets(s); puts(s);

character test functions.

check the character taken as input

isalpha(ch) isupper(ch)

isdigit(ch) tolower(ch)

islower(ch) toupper(ch)

Example:

```
char x;
printf("\n Enter any alphabet");
x=getchar();
if (islower(x))
    putchar(toupper(x));
else
    putchar(tolower(x));
```

(2) Formatted I/O stmts.

Input & output arranged in a particular format

| I/P | O/P |
| --- | --- |
| scanf() | printf() |
| fscanf() | fprintf() |

(i) Scanf()

scanf("control string", &var1, &var2 .. &varn)

character groups

% conversion character.

%c, %f, %d, %s

Example

## getchar () & putchar()

```
Void main()
{
    char a;
    printf ("\n Enter a character");
    a = getchar();
    printf ("\n Displaying that character");
    putchar(a);   putchar ('A');
}
```

## gets () & puts ()

```
main()
{
    char a[10];
    puts (" Enter string ");
    gets (a);
    puts (a);
}
```

while loop (print n numbers)

```
# include <stdio.h>
# include <conio.h>
Void main() {
int i=1, n; clrscr();
printf (" Enter n ");
scanf ("%d", &n);
while (i<=n)
{
    printf ("%d", i);
    i++;
}
getch();
}
```

do-while loop

```
# include <stdio.h>
# include <conio.h>
Void main() {
int i=1, n; clrscr()
printf (" Enter n ");
scanf (" %d", &n);
do {
    printf (" %d", i);
    i++;
} while (i<=n);
getch();
}
```

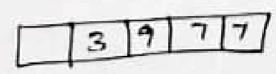
(ii) printf ()

printf ("control string", var1, ... varn);

field width

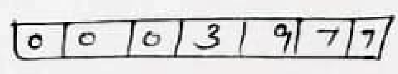
printf("%d", 3977)

%5d , 3977



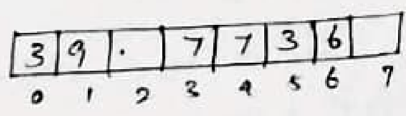
% -5d, 3977

% 07d, 3977



a = 39.7736

% 7.4f , a



% 7.2f , a

Decision Making

Control statements.

* pgm - all stmts are executed sequentially.

* No repetition cases.

* Repetition / Execution order of stmts is changed based on conditions.

"Conditional / ctrl stmts".

4 types of ctrl structures:

1. Sequential
2. Selection
3. Iteration
4. Encapsulation

① **Sequential Structure**. instn executed in a sequence

i = i+1 ; j = j+1 ;

② **Selection structure** sequence of instn are executed by deciding upon the condition

```
if (x > y)
        i = i+1;
else.
        j = j+1;
```

③ **Iteration structure** stmts are repeatedly executed

```
for (i=1; i<=5; i++)
{
      i = i+1;
}
```

④ **Encapsulation Structure** Compound structure

## Decision Making Statements :

① **if stmt**

Control the flow of execution of stmts condition

Syntax:
```
if (condition)
{
      stmts;
}
```

Example: (i) Check whether the number is less than 5

(iv) check equivalance of two nos. if (m-n == 0).

(ii) Swap two values when first no. is greater than two (2nd no). (iii) Print no. b/w 10 & 15

```
int a, b, c;
if (a > b)
{
      c = a;
      a = b;
      b = c;
}
```

| a | b |
|---|---|
| 10 | 5 |
| 5 | |
| 5 | 10 |

2 variables

a = a+b

b = a-b

a = a-b

* Two way decision making used in conjunction with C++
* Test & then take decision

Syntax:
```
If (condition)
{
        stmts ; }
else { stmts; }
```

Example: Even / odd, Leap year, Greatest of 2 nos

③ Nested if ... else stmt.

Example:
```
Void main()
{
    int a;
    printf(" Enter a number");
    Scanf ("%d", &a);
    if (a == 10)
        printf (" A Grade ");
    else
    {
        if (a == 8)
            printf (" B Grade");
        else
            printf (" C Grade"),
    } getch();
}
```

Examples: Greatest of 3 nos.

Looping & Branching.

Repetition - Set of instn in specified no. of times

↳ loop control structure.

"Block of stmts which are repeatedly executed for certain no. of times".

* Body of loop
* Control stmt

Looping stmts: Initialization
Test the ctrl stmt
Executing the body of loop
Updating Condtn Variable.